– Diplomarbeit –

Encapsulation and Specification of Object-Oriented Runtime Components

Jan Schäfer

September 30, 2004

Author:	Jan Schäfer j_schaef@informatik.uni-kl.de
Supervisors:	Prof. Dr. Arnd Poetzsch-Heffter Dipl.-Inform. Nicole Rauch
Organization:	Universität Kaiserslautern Fachbereich Informatik AG Softwaretechnik 67663 Kaiserslautern

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kaiserslautern, 30. September 2004 .....

Encapsulation and Specification of Object-Oriented Runtime Components

Jan Schäfer

j_schaef@informatik.uni-kl.de

September 30, 2004

Revised, November 26, 2004

Dedicated to my Mother

Acknowledgments

This thesis was made possible by my supervisors Prof. Dr. Arnd Poetzsch-Heffter and Dipl.-Inform. Nicole Rauch. Prof. Dr. Arnd Poetzsch-Heffter often showed me the right direction and gave valuable advice. Many thanks goes to Nicole Rauch for her continuous feedback, her valuable corrections and her patience. Further I thank Prof. Dr. Peter Müller for the email discussion about universes, Karl-Christian Pammer and Marc Krämer for proof-reading the thesis, the developers of Linux, KDE, LaTeX, LyX, OpenOffice, Java and Eclipse for giving me the tools to write this thesis, and last but not least, I thank my fiancée for her patience and her mental support, even if I had been difficult sometimes, especially in the last weeks before the completion of the work.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

To be able to handle large software systems, these systems have to be built from smaller components. Components are reusable pieces of software consisting of a set of interacting objects. They define a strict external interface and have an internal implementation, the *representation*. The representation has to be fully encapsulated by the component and should never be directly modifiable by objects from the environment. The component's interface, the *boundary*, can consist of more than one object. These boundary objects should be able to access the representation of the component without violating the encapsulation property. This work describes how components can be properly encapsulated and still allow objects on the boundary to access the representation of their component.

"The development of reusable components requires adequate description of component behavior, and this can only be given if components are sufficiently encapsulated for their behaviors to be predictable.", was already stated more than ten years ago by Hogg et al. [38]. Today, object-oriented programming languages still lack mechanisms to sufficiently encapsulate objects. The problem lies in the nature of object-oriented languages: the combination of sharing and imperative modification of state. The sharing or *aliasing* of objects is widely accepted as the crux of the encapsulation problem. An object is *aliased* if it is referenced by more than one expression. Liskov and Guttag [48] said about 20 years ago: "Aliasing is problematic because it breaks the encapsulation necessary for building reliable software components". The treatment of aliasing has been an ongoing topic for researchers [37, 38, 54, 5, 63, 23, 60, 24, 15, 44, 17, 3, 20, 64, 14, 21, 2]. While a constant progress could be achieved, a final satisfying solution to the problem has not been found yet.

As aliasing generally cannot be forbidden, the effects of aliasing must be controlled when it is needed [38]. Aliasing is unproblematic if state changes of aliased objects are not allowed. This can be achieved in two ways: by providing a *read-only* mode [37, 34, 59, 44, 10] for references and to disallow any modifications of objects via a read-only reference, and by defining objects that cannot change their state at all, namely *immutable objects* [37, 7, 32, 63, 75, 11, 71, 29, 10].

The state of representation objects can depend on mutable objects of the environment. If this is the case, the internal state of the component can be changed by the environment without using the component's interface, even though the environment does not have a reference to a representation object. While most aliasing encapsulation approaches address representation exposure, this problem called *argument dependency* is not handled in general.

Components are used in environments which are unknown at the creation time of

the components. The encapsulation mechanism for components can therefore neither make assumptions about the objects of the environment nor the later usage of the components. This so-called *open-world analysis scope* is important when analyzing components. Components must be encapsulated, independent of the environment they are used in.

In order to be practical, an encapsulation mechanism for components should not require a change of the underlying programming language. Otherwise, these components could not be used in combination with already existing software. Theisen [75] further suggests that the mechanism should be *simple* to use, *generally applicable* to all object-oriented languages, and *efficient*, requiring no runtime-overhead. In addition, Clarke [19] demands that the techniques should be *precisely specified* and *sound*, that is, the annotations must unequivocally and correctly encapsulate components.

1.2 Goals of this Work

In this master's thesis we develop a specification technique for components written in Java [31]. Correctly specified components are guaranteed to be encapsulated. The specification can be statically checked in a practical open-world analysis scope. That is, the analysis assumes that unchecked classes are added later to the system, and that these classes do not conform to the specification. However, as our approach uses built-in visibility constraints of Java, we only need to assume that classes of components reside in packages that are *sealed*, that is, no classes can be added later to theses packages. We further demand that all classes of these packages, and all classes that are accessible by component classes are checked by the analysis. This assures that checked components can be used in unchecked environments while preserving their encapsulation. Besides representation exposure our approach handles argument dependency as well. That is, representation objects cannot depend on the mutable state of external objects. We present our approach in a subset of unmodified Java.

This work has four major goals:

1. The definition of a component model and the notion of an encapsulated and independent component.
2. The development of a specification technique to specify components in Java and to verify that correctly specified components are indeed encapsulated and independent.
3. The implementation of a tool to check the specification against the actual source code.
4. The application of the specification technique to the ProofContainer of the Jive system [51].

1.3 Outline

The remainder of this work is structured as follows.

In Chapter **2** we review object-oriented programming in general and the Java programming language in particular. Thereafter we focus on the problem of aliasing and present existing proposals to its solution appearing in the literature.

Chapter **3** presents our component model. After describing the model of an access graph, we define the notion of a component within such a graph and give a definition of encapsulation and independence. In addition, existing aliasing management techniques and typical programming patterns are reviewed by means of our model.

In Chapter **4** we show the implementation of our component model in Java. We present a class-level ownership that allows classes to own objects which is similar to *type universes* [57]. *Rep types* are used to indicate representation objects in the source code. In conjunction with a set of rules that can be statically checked, we can ensure that representation objects are encapsulated by their owner class. We propose *autonomous classes* that do not access any mutable static field, to permit static variables, which we disallowed before. After that we enable components with different classes on the boundary by presenting *boundary classes*. A boundary class can access all representation objects of its bounded class, but is still accessible by the environment, hence allowing to express programming patterns like iterators. *Rep classes*, which are a variant of *confined types* [76], are presented to allow components to act as containers and to store references to external objects. Finally, we show how an object-level protection can be achieved by giving additional rules.

Chapter **5** presents *immutable types*. Instances of immutable types never change their state after their construction. After giving a definition of *immutable objects*, we present a set of statically checkable rules like in Chapter 4 that have to be followed by the source code to guarantee the immutability of all instances of a Java class. We explored the practicality of our rules by applying them to the Boolean and String classes of the java.lang package.

We discuss the extension of our approach by well-known aliasing treatment methods in Chapter **6**. We examine *borrowed expressions* that cannot be statically aliased, *fresh expressions* that are not aliased at all and *read-only expressions* that cannot be used to modify the object that they refer to. While all extensions improve the expressiveness of our approach, none of them can be checked without restriction in an open-world scope.

Chapter **7** presents JACPOCK$_{proto}$, a prototype of a Java tool to check Java programs for conformance with the rules presented in this work. JACPOCK$_{proto}$ can be found on the CD that is delivered with this thesis.

In Chapter **8** we apply our approach to the Proof Container component of Jive, a tool to verify object-oriented programs. We show that the Proof Container in the current implementation is neither encapsulated nor independent and propose changes to the source code that assure these properties.

Finally, Chapter **9** concludes and discusses future directions of our work.

Chapter 2

Background

2.1 Object-Oriented Programming

Object-oriented programming is based on the concept of *objects*. Objects have *fields* and *methods*, where the values of the fields form the *local state* of the object and the methods perform state transformations on objects. In *class-based* languages, an object is created by using *classes*. The objects that are created from a class are also called *instances* of their class. A class defines the field and method names of its instances as well as the code of the methods. In *typed* programming languages, the class also defines the types of the fields and the types of the formal parameters and result values of the methods. An important concept of object-oriented languages is *inheritance*. If a class is the *subclass* of another class it *inherits* all fields and methods of that class. In addition, the class can define new fields and methods as well as *override* methods of its *superclass*. A mechanism called *polymorphism* allows that a subclass can be used in all places where its superclass is expected. Polymorphism and method overriding lead to a mechanism called *dynamic binding*. Dynamic binding means that the actual method that is called on an object does not depend on its *declared type* but on its *actual type*. For example, let A,B be classes, where B is a subclass of A. Let m be a method of A that is overridden by B. If a variable x is declared with class A, but its value is a reference to an object of class B, a method invocation of method m on x will result in an invocation of the method defined in class B instead of class A.

Our Requirements

Our approach only works in object-oriented languages that meet certain requirements. Like Noble et al. [63] we expect some form of memory management for objects. That is, it must not be possible to delete an object while references to that object still exist. Otherwise referenced objects could be deleted, resulting in *dangling references* Liskov and Guttag [49]. Even worse, an aliased object could be deleted, and that object's memory could be reallocated to a new object. The new object would then be aliased by the retained pointers to the nominally deleted object [63], implying that no guarantees at all could be made about aliasing. Java's garbage collection mechanism, for example, provides such a memory management for objects.

Furthermore, we require type safety [49]. Type safety means that type mismatches cannot occur at runtime, and if they occur at runtime, they *always* result in an error. For example, using an integer as a pointer must not be allowed, and array accesses must be checked to be within bounds. C^{++} [74], for example, is a programming language that is not type safe.

2.2 Java

In this section we briefly describe features of Java that are important for the remaining text.

2.2.1 General

Java Gosling et al. [31] is a strongly typed class-based object-oriented language with garbage collection. Java is a type-safe language like described above. Java source-code is compiled to *byte-code* rather than native code. The byte-code does not run natively on a target platform, but is executed by the Java Runtime Environment. This guarantees that a Java program, once it is compiled, runs without modification on any platform where a Java Runtime Environment is installed. Objects in Java cannot be deleted manually, but are deleted by Java's garbage collector when an object is not referenced anymore. So Java provides memory management for objects.

2.2.2 Type System

Java distinguishes *primitive types* and *reference types*. Primitive types are basic types like integer or boolean, reference types can be either *array types*, *class types* or *interface types*. The difference between primitive types and reference types is that values of primitive types are copied by value, whereas values of reference types are copied by reference. Thus, primitive types are *value types* (see Section 2.5.4), and instances of reference types are objects [50]. All value types are predefined, that is, the user cannot define own value types. In contrast, C# [52] allows the definition of new value types.

2.2.3 Packages

Classes and interfaces are grouped into *packages*. Packages serve two purposes [49]:

- As a naming mechanism to give classes and interfaces a unique name to avoid naming conflicts.
- As an encapsulation mechanism.

The second issue is of more interest for our work. Types that are defined in the same package can share information that is not accessible from outside the package. This is achieved by *access modifiers*.

2.2.4 Access Modifiers

Java has a number of *access modifiers* to restrict the visibility of types, fields and methods. There are four different kinds of visibilities:

- public - visible to all classes.
- protected - visible to all classes of the same package and to all subclasses.
- default (*package private*) - visible to all classes of the same package.
- private - only visible to the defining class.

Types can either be public or package private. If a type is declared to be package private it can only be used within the type's defining package. However, objects of package private types can still be used outside of the package if a supertype is a *public* type. As Object is a supertype of all types, objects of package private types can generally be used outside of the package. However, it is possible to encapsulate objects of package private types within a package boundary if certain constraints are kept (see Section 2.5.9).

Fields and methods can be declared with all four visibilities. Classes can only access entities that are visible to them. However, visibility only protects the names of entities, not their contents. That is, a private field can only be accessed by its defining class, but the value of a private field is not protected at all. If a private field, for example, holds a reference to an object, a public method of its defining class can return that reference without any restrictions.

Access modes in Java only provide protection on a *per-class* basis, which means that any object can retrieve a private reference from any other object of the same class [63]. However, an object a can only access an object b if a has a reference to b, so object-based protection can be achieved if it is guaranteed that no object can obtain a reference to another object of the same class.

2.2.5 Static Fields and Methods

Fields and methods in Java can be declared to be static. Static entities are accessed by the class name of the declaring class. Or to cite Clarke [19]: "A static field or method is a global variable or function defined within the text of a class". No reference is needed to access a static entity, so static entities are globally accessible, depending on their access modifier. Static fields and methods are needed to allow patterns like the Singleton pattern [30], but especially mutable static fields are in general considered to be bad programming practice.

2.2.6 The Final Modifier

Final Fields. Fields can be declared as final. The value of a final variable cannot be changed after it has been initialized. However, if the variable holds a reference, only the reference is protected, the referenced object is not protected at all. That is, the state of mutable objects that are referenced by a final variable can be modified without restriction.

Final Classes. Classes can also be declared as final, but it has a completely different meaning: a final class cannot be subclassed anymore. That is, it is impossible for other classes to inherit from a final class.

Final Methods. Methods that are declared as final cannot be overridden anymore by subclasses.

2.2.7 Nested Classes

It is possible in Java to declare classes within the scope of another class. These classes are called *nested class*es, and they can be nested within other nested classes as well. There are four different kinds of nested classes:

- *Static member classes*
- *Non-static member classes*
- *Local classes*
- *Anonymous Classes*

All nested classes have in common that the private members of their enclosing class are visible to them. Except static member classes, the other nested classes are so-called *inner classes*. An instance of an inner classes can only be created in connection with an instance of its enclosing class (its enclosing instance). Instances of inner class have always an implicit reference to its enclosing instance. In contrast, static member classes are not implicitly connected with an instance of their enclosing class, and can be used independently.

2.2.8 Other features

Java has additional features such as exceptions, threads, reflection and native methods. However, we will not discuss these features further as they are not of interest for this work.

2.3 Analysis of Software Components

According to Porat et al. [71] the scope of program analysis can be generally divided into

1. *Closed-world analysis scope* where all symbolic references refer to analyzed classes, and all possible targets of runtime invocations are defined in analyzed classes.
2. *Open-world analysis scope* (or *component analysis*) where the analyzed component is an arbitrary collection of classes.

Components are in general implemented without any prior knowledge of the client classes that use the components. So analysis of components can only make assumptions about the classes known by the component. To be practical a component analysis technique should

1. Be an open-world analysis.
2. Not require a change of the underlying programming language.

If the analysis scope is closed-world, it is not possible to ensure that checked components are correct in unchecked environments. If the analysis needs an adaption of the underlying programming language, Java in our case, the analysis cannot be used for existing programs, and programs written in an adapted programming language cannot be used in conjunction with programs written in the unchanged language.

A good example for an analysis technique that fulfills both requirements are *confined types* [76] (cf. Section 2.5.9), another example is the immutability analysis of Porat et al. [71]. An example for a closed-world analysis is JML [46].

Our goal is to achieve an open-world analysis technique which does not need a modification of the underlying programming language. If we assume that the checked packages are always *sealed*[1], we can guarantee that our analysis is correct.

2.4 Aliasing

Aliasing is endemic in object-oriented programming. Noble et al. [63].

"An object is *aliased* if there is more than one pointer referring to that object", Hogg et al. [38]. They also described a vivid example to illustrate aliasing: "For many centuries astronomers used the distinct terms 'evening star' and 'morning star' without realizing that both referred to one object, the planet Venus", that is, they had two names for the same object. Translated to programming languages, there would have been two different variables that refer to the same object.

Aliasing or *sharing* of objects is crucial in object-oriented languages. Sharing guarantees a consistent view to objects. If there are copies of an object, these copies could be out of date, resulting in an inconsistent system. Sharing leads to problems if there is more than one object that can actually *modify* the shared object. Or to cite Noble et al. [63]: "The problem is not the presence of aliases, but the visibility of non-local changes caused by aliases.". There can be an arbitrary number of *observers* as long as there is only one *mutator*. In addition, all observers must be aware of the existence of a mutator. Otherwise, observers might make assumptions of a shared object that could be misleading. Even more than one mutator for an object can be allowed, as long as both mutators are aware of the fact that they do not have exclusive write access, and that the shared object could be arbitrarily modified by other objects.

In current object-oriented programming languages it is generally neither possible to determine whether an object is shared, nor to determine the mutators of an object. This is the reason why verifying object-oriented programs is a nearly impossible task.

2.4.1 Static vs. Dynamic Aliases

Aliases can be divided into *static*[2] and *dynamic* aliases [37, 38]. A static alias is held by a heap-allocated instance variable, a dynamic alias is held by a stack-allocated variable, that is, a method parameter, a local variable or the result of a method call. Some authors [5, 24] argue that dynamic aliases are not as severe as static aliases and could be allowed, but as already described by Müller [56], dynamic aliases lead to serious problems in modular verification, for example.

Dynamic aliases have the advantage that they disappear when their scope is left, static aliases can remain until the program terminates. But even dynamic aliases can exist until the end of the program. Local variables in Java's `main` method for example are alive until the end of every Java program. This leads to the conclusion that dynamic aliases are as serious as static aliases in general, but can be allowed if their scope is kept under control. We illustrate this with two examples: in the first example dynamic aliases should not be allowed, in the other they can be allowed.

[1] A package contained in a JAR file may be marked sealed, in this case it must be entirely contained in that JAR file [9].

[2] Static aliases should not be confused with `static` variables in Java.

Example for bad dynamic aliases

The initialization problem (see Section 3.4.3) is a good example for bad dynamic aliases. Imagine a class A that has a constructor that takes an array as argument (see Listing 2.1).

```
1  class A {
2      int[] b;
3      A(int[] c) {
4          b = c;
5      }
6  }
7  class Violator {
8      void violate() {
9          int[] c = new int[5];
10         A a = new A(c);
11         c[0] = 5;
12     }
13 }
```

Listing 2.1: Example for bad dynamic aliasing.

A would like to use the array as its internal representation. This requires that the array is not aliased by external objects when it is passed to the constructor, because otherwise the internal array could be modified by external objects. In this case it is not sufficient to forbid static aliases. The violate method of the Violator class contains a local variable c holding a reference to an int array. c is passed as argument to the constructor of A. After the construction of a, the field a.b is effectively aliased by the local variable c. The content of the internal array of a can now be changed via the dynamic alias c. The problem is that the scope of the dynamic alias is not controlled by class A.

Example for good dynamic aliases

Now imagine that class A creates the array itself, but wants the array to be filled by a static method Util.fill (see Listing 2.2). It has to pass the array to Util.fill as parameter in order to achieve this. If fill would store the array reference in a static alias the array could be modified after the method has been left. But it is not critical if fill dynamically aliases the array, because class A can be sure that all these aliases are no longer existing after the method call has returned. The scope of the dynamic aliases is controlled by A and thus can be allowed.

```
1  class A {
2      int[] b;
3      A() {
4          b = new int[5];
5          Util.fill(b);
6      }
7  }
8  class Util {
9      static fill(int[] c) {
10         c[0] = 5;
11     }
12 }
```

Listing 2.2: Example for good dynamic aliasing.

2.5 Alias Management

2.5.1 The Geneva Convention on the Treatment of Aliasing

The Geneva convention on the treatment of object aliasing [38] divides approaches to the solution of the aliasing problem into four categories:

- *Alias Detection* – Determining aliasing with compile-time or run-time techniques on a not further annotated program.
- *Alias Advertisement* – Annotating programs with aliasing information to enhance the locality of alias analysis.
- *Alias Prevention* – Preventing aliasing by using constructs that disallow aliasing in a statically checkable fashion.
- *Alias Control* – Isolating the effects of aliasing.

In the following we will discuss different approaches of alias treatment that appear in the literature. We annotate the approaches with letters to indicate whether an approach does **D**etection, **A**dvertisement, **P**revention or **C**ontrol of aliasing.

2.5.2 Unique Variables [A,P]

"A *unique variable* is either empty or its value is the sole reference to an object", Boyland [15]. Unique variables have been discussed for more than a decade [78, 37, 8, 54, 5, 63, 18, 15, 17, 22]. Unique variables hold references to objects that are guaranteed to have no alias in the entire system. That is, the reference to an object referred to by a unique variable is only contained in one variable at a time, thus that object can always be modified without affecting other parts of the program [37]. Another term for *unique* is *free* [63]. There are different approaches to achieve the

uniqueness property: destructive reads [78, 37, 8, 54, 22], swapping [35] and copy assignment [5]. Destructive reads return the value of the source variable and set it to null afterwards. Swapping exchanges the contents of two variables with one operation. Copy assignment makes a deep copy of the object referred to by a unique variable. All these approaches require programming language support. Especially destructive reads are very unintuitive for programmers and could lead to a new bug pattern [4] of accessing nullified unique variables.

Borrowing To increase the usability of unique variables most approaches [37, 54, 15, 17, 22] offer a *borrowing* mechanism that temporarily violates the uniqueness constraint. Unique references can be *borrowed* as parameters of method invocations, as long as it is guaranteed that no static alias of the borrowed reference is made by the method. The borrow mode adds an additional mode to the language and further complicates the uniqueness approaches.

Boyland [15] proposes *alias burying*, a static checking algorithm for unique variables that does not need any programming language support. It only requires that when a unique variable is read, all other aliases have to be *dead*, i.e. never be used again. This approach is much better than destructive reads, as accessing nullified unique variables is impossible. However, in order to be modular, the uniqueness analysis depends on an effects analysis [16].

Clarke and Wrigstad [22] identified three problems with uniqueness approaches: abstraction, orthogonality and definition. If the implementation of a unique object has to be changed in a way that affects the uniqueness property, that change can force changes in the object's interface (abstraction problem). A borrowed reference cannot be treated like a usual non-unique variable (orthogonality problem). Finally, the borrowing mechanism can lead to two active references to one unique object (definitional problem). In their solution they combine uniqueness and ownership types [23] to solve the abstraction problem. They allow arbitrary references from internal objects to the owner object, but only allow one reference from external objects. The orthogonality and the definitional problem are solved by a destructive borrowing, which needs a destructive read again.

To summarize, unique variables could be a useful addition to programming languages, but it seems to be too difficult to easily integrate them into existing languages without changing the semantics of the assignment operator. This, however, would be an inconvenient extension and is possibly not worth the price. Only *alias burying* [15] could be investigated further.

2.5.3 Free Variables [A,P]

Free variables [37, 47, 63] are a "lightweight" version of unique variables. Other terms are *virgin* [47] or *fresh*. Like unique variables, objects referred to by free variables are not aliased. But the freedom is only guaranteed at a certain program position rather than at all program positions. For example, one can specify a method parameter to be free to be sure that when the method is called the object referred to by that parameter is not aliased. Within the method body arbitrary aliases can be made of the object. The advantage of the free mode is that the semantics of the assignment operator need not to be altered. If full uniqueness is not available, the free mode is the only way to solve the initialization problem (see Section 3.4.3).

2.5.4 Value Types [P]

Value Types [50, 40, 5, 6, 52] prevent aliasing by replacing the copy-by-reference semantics with a copy-by-value semantics for objects of value types (*value objects*). Primitive types like integers, for example, are value types that exist in most object-oriented programming languages. To enable user-defined value types, the semantics of the assignment operator has to be changed. Instead of simply copying the reference, the whole object has to be copied. There are two variants of the copy operation: *shallow* copy [40, 52] and *deep* copy [5, 6]. A shallow copy only copies the instance variables of an object but not the referenced objects. A deep copy recursively copies the whole object and all its referenced objects.

Value types have two drawbacks:

1. A different assignment semantics.
2. The performance penalty of the copy assignment.

The first issue is a severe problem, because it requires support of the underlying programming language. User defined value types are not possible in Java, for example, without modifying the Java virtual machine. The second issue is only a problem for complex value types and a deep copy semantics. In C# [52], for example, value types can be allocated on the stack, and if their size is small they can be even more efficient than reference types. The deep copy can be implemented efficiently if the compiler only copies the object when it is actually changed, and otherwise only copies the reference [5, 6].

An interesting point is that a variable holding a reference to a value object is always a unique variable as described above. So value types can be seen as a special case of unique variables, because all variables of value types are unique, but not all unique variables are of a value type.

2.5.5 Immutable Types [C]

Immutable types or *functional types* [37, 7, 32, 63, 75, 11, 71, 29, 10] are types whose instances (so-called *immutable objects*) cannot be changed after their construction. "An immutable object need not be protected from aliasing, because the effects of this aliasing will never be seen by any holder of an alias", Hogg [37]. Immutable objects can be shared arbitrarily, and it is not necessary for the programmer to pay attention to aliasing, that is, immutable objects are referentially transparent [63]. Immutable types can be statically checked in an open-world scope [71]. They are a natural concept that is already implicitly widely used [71]. The String class as well as wrapper classes like Boolean and Integer of the java.lang package, are immutable classes, for example. They have a strong invariant and are very useful in a lot of different applications. However, immutable types cannot be applied in all cases. As immutable objects cannot change their state, a new object has to be created for every modification. Nevertheless, they are a very useful aliasing control technique, and we have adopted that idea in our component model. We discuss immutable types further in Chapter 5.

2.5.6 Read-Only Mode [A,C]

Variables can be declared with a *read-only* mode [37, 34, 59, 44, 10] in order to offer read access to an object, but prevent state changes via that alias. The read-only mode

is always defined in conjunction with some definition of *side-effect free* methods. These methods are annotated with a special keyword, and they are the only methods that can be invoked on read-only references. The definition of side-effect free, however, varies from publication to publication. Recently, Birka and Ernst [10] proposed a promising read-only mode for Java.

C++ const mode. In C++ [74] it is possible to declare variables with the const keyword. The local state of an object cannot be changed by the reference of a const variable. So it is not possible to change the value of instance variables of objects referred to by const variables, and on const variables just const methods can be invoked. A const method is a method that is annotated with the const keyword, and which is not allowed to contain assignments to instance variables of the receiver object. However, the const mode in C++ has two major drawbacks: The const'ness can be "cast away", and const only protects the local state of an object and not the transitive state. That is, the values of instance variables are protected, but objects that are referenced by them can be modified without restriction.

Even though the const mode does not *enforce* anything, it is widely used and accepted by C++ programmers. So a read-only mode is a useful concept in general. In addition, some aliasing problems can only be solved with a read-only concept.

Java's final variables. In Java [31], variables can be declared as final. The value of a final variable cannot be changed after it has been initialized. However, if the variable holds a reference, only the reference is protected, the referenced object is not protected at all.

2.5.7 Full Alias Encapsulation

Full alias encapsulation was mainly proposed by Hogg with *islands* [37] and Almeida with *balloons* [5]. Full alias encapsulation means that certain objects are encapsulated by a boundary. Objects outside that boundary cannot obtain a reference to objects inside the boundary, and objects inside the boundary are not allowed to reference outside objects. Islands achieve this by using variable mode annotations to ensure the encapsulation. Balloons ensure the encapsulation just by program analysis. Both approaches need a new assignment semantics which means a change of the underlying language. Islands need a destructive read and balloons need a copy-by-value semantics. Full alias encapsulation might be useful in certain situations, but it is too restrictive to be used in general. We now describe both approaches in more detail.

2.5.7.1 Islands [A,P,C]

Islands [37] were one of the first proposals of alias control. Islands use aliasing mode annotations to control aliasing. There are three aliasing modes: *read*, *unique* and *free*. A *read* variable may not be assigned to. The *unique* mode indicates that the object to which it refers has only one static reference in the entire system. The mode *free* indicates that *no* other references to the variable exist anywhere in the system. An *island* is the transitive closure of a set of objects accessible from a *bridge* object. A bridge is the only access point to the objects that make up the island. That is, islands fully encapsulate their representation. Neither can objects within the island reference

objects from the outside, nor can external objects refer objects of the inside. Iterators, for example, cannot be implemented with islands. Another drawback of the island proposal is the need of an unusual "destructive read" operation. Islands distinguish between static and dynamic aliases. Objects of an island's internal representation can be dynamically aliased by external objects, provided that these aliases are read-only and thus cannot change the state of the representation objects. The islands paper also defines side-effect free functions. It is also possible to define immutable objects with islands, as an island with a bridge class that only has side-effect free functions is immutable.

2.5.7.2 Balloon Types [D,P]

Like islands, *balloon types* [5] is a proposal for full alias encapsulation. Every type is classified to be either a *balloon* or a *non-balloon*. Objects of a balloon type are not shareable by state variables of objects, and all the state that is reachable by a balloon is encapsulated, in the sense that no part of it can be referenced by state variables of any external object. That is, only static references are taken into account, dynamic references are allowed without any restrictions. Only one new keyword is introduced to declare balloon types. However, balloon types need a change of the target language, because a new assignment semantics for balloon types is needed, as objects of balloon types cannot be copied by reference to instance variables. Rather than copying the reference, balloon types are copied by value. To avoid the overhead of always making a deep-copy of a balloon, Almeida proposes optimization strategies that compilers could implement, using the knowledge of the balloon invariant. The balloon invariant is enforced by a non-trivial static analysis for every balloon type. Besides the 'normal' balloon type, additional specializations are proposed. *Opaque balloon types* have the additional invariant that besides static references, no dynamic references to the internal state can exist. Also proposed are *value types* that are a variant of opaque balloon types that are only copied by value and thus have the same semantics as primitive types.

2.5.8 Object Ownership [A,P]

Object ownership means that objects can *own* other objects. Owned objects can not be directly accessed by outside objects. Object ownership [23, 60, 12, to just name a few] has been investigated for about 10 years now. It has been used to prevent data races [12] and deadlocks [13], to allow safe upgrades of persistent object stores [14] and to allow modular verification [57, 56, 61]. This could be achieved because ownership types are able to restrict the aliasing of objects.

2.5.8.1 Ownership Types in Eiffel

Kent and Maung [41] presented an ownership notion for Eiffel. In their model every object has an owner, which can be Void. They use the keywords private and protected to indicate ownership of attributes. private attributes are owned by the object in which they are declared. protected attributes are owned by the owner of the object in which they are declared. They also allow to have multiple owners of an object with a share keyword. With an owner function it is possible to obtain the owner of an object and attach it to an attribute. Their system is a runtime-system and needs an extension of

the Eiffel language. An exception is thrown if ownership is not respected during the runtime of a program. They propose to use the ownership during development and to omit it for productive systems.

2.5.8.2 Flexible Alias Protection

Flexible alias protection (FAP) [63] is a conceptual framework that annotates types with mode declarations. FAP knows five annotations: *rep*, *arg*, *free*, *val* and *var*. Basically these are only three modes. *var* is just the ordinary reference semantics without any restrictions, and *val* is an annotation to indicate a *value type* which can be automatically attached, because it directly depends on the type of the expression. A *rep* expression refers to an object which is part of another object's representation. Objects referred to by *rep* expressions can never be exported from the object to which they belong. An *arg* expression refers to an object which is an argument of an aggregate object. It only provides access to the immutable interface of the object to which it refers. A *free* expression holds the only reference to an object in the system.

FAP also defines the term *clean* which means free of side-effects. It defines the notions of a *clean method* that is only made up of *clean expressions*, where a *clean* expression either reads a variable of mode *arg* or *val*, or calls a *clean* method. A *clean interface* provides access to the immutable properties of an otherwise mutable object, and a *clean object* implements an immutable type. FAP also states that immutable types should be identified by annotations on their declaration.

2.5.8.3 Ownership Types

Ownership types have been introduced by Clarke et al. [23] as a formalization of the core of flexible alias protection. Ownership types only regard the *rep* annotation of FAP, thus formalizing ownership, but ignoring the annotations *arg*, *free* and *val*. Besides rep, the keyword owner was introduced which allows the typing of this, and enables self-referencing object structures to be owned by a single object. They also introduced owner context parameters for types, which offers the possibility of propagating ownership information. It is possible, for example, to declare a linked list whose link objects are owned by the linked list object, but whose data objects are owned by an arbitrary different object. The core language of this formalization is without subtyping and inheritance.

In [24] ownership types have been formalized by extending an imperative object calculus of Abadi and Cardelli [1]. That formalization contains subtyping and inheritance.

Boyapati and Rinard [12] changed the notion of ownership types by not using the rep keyword, but by directly naming the owner. Owners can be this, world or an ownership context parameter. A linked list implementation with parameterized ownership types is shown in Listing 2.3. As can be seen in that code example, parameterized ownership types need a significant annotation effort and decrease the readability of the code. Boyapati and Rinard [12] states, however, that most of the annotations can be eliminated by a combination of inference and well-chosen defaults.

All ownership type systems have in common that every object has an owner, which is either another object, or a special owner that represents the entire system, namely world or root. This *ownership relation* forms a tree rooted at world. An object x can

```
1  public class LinkedList<ListOwner,DataOwner> {
2      Node<this,DataOwner> head;
3      Node<this,DataOwner> tail;
4      ...
5      void add(Object<DataOwner> value) {
6          Node<this,DataOwner> newNode =
7              new Node<this,DataOwner>(value);
8          tail.next = newNode;
9          tail = newNode;
10     }
11     Object<DataOwner> getLast() {
12         return tail.data;
13     }
14 }
15 class Node<NodeOwner,DataOwner> {
16     Node<NodeOwner,DataOwner> next;
17     Object<DataOwner> value;
18     Node(Object<DataOwner> v) {
19         value = v;
20     }
21 }
```

Listing 2.3: A linked list implemented with ownership types.

access an object owned by o iff $o \equiv x$, or o is an ancestor of x in the ownership tree. That is, an object x can access all objects it owns and can access its owner object, and all objects the owner object can access. This is a surprising property as one would not think, for example, that a node object of a linked list can access more objects than the linked list itself.

The ownership relation is not directly related to the reference relation. That is, an object can own an object, even though it does not have a reference to it. A linked list, for example, can own all its node objects, even though it only has a reference to the first and the last node of the list.

Ownership types enforce a property called *owners-as-dominators*. It states that all access paths from an object a to an object b must go through the owner of object b. This forbids the efficient implementation of iterators [62], for example, because an iterator cannot access the node objects owned by its linked list. This limitation was tried to be fixed by two approaches: by allowing dynamic aliases to access owned objects without any restriction [20], and by allowing inner classes to access the owning objects of their outer class [19, 14]. Dynamic aliases can be as bad as static aliases as already described above, and dynamic aliases cannot be used in all cases. Event callbacks, for example, cannot be implemented by dynamic aliases [2]. The somewhat ad-hoc solution [2] of the inner class approach, on the other hand, is better than the dynamic aliases approach, but forces the implementation of e.g. iterators to inner classes, which can be limiting.

Ownership types provide an object-level protection. That is, even objects of the same class cannot access each other's owning objects. This protection, however, can be limiting in practice, because it does not allow the efficient implementation of e.g. binary methods. An equals method, for example, cannot be implemented by comparing the internal objects for equality, it can only use the public methods of the parameter object. A *read-only* mode like used by the *universes* approach [59] could perhaps be applied to ownership types as well to solve this problem.

2.5.8.4 Universes

Universes [60, 58, 59, 55] enforce a hierarchical partitioning of the object store into so-called *universes*. The universe type system provides alias encapsulation and dependency control. Each object in a program execution *owns* a universe and is called the *owner* of that universe. In addition, every object *belongs* to exactly one universe, which can be the *standard universe*. So each object of the standard universe owns a universe. Objects belonging to these *child universes* are again owners of universes and so forth. That is, universes form a tree rooted at the standard universe. Universes exist in two variants: *object-universes* [60] and *type-universes* [57].

Object-universes. Object-universes are owned by objects. The invariant of object-universes guarantees that if an object x holds a direct reference to object y then one of the following conditions holds:

1. x and y belong to the same universe.
2. y is owned by x.
3. The reference is read-only.

where a read-only reference cannot be used to modify the referenced object. One consequence of the universe invariant is that objects cannot reference their owner object with a read-write reference. This means, for example, that the node objects of a linked list cannot access their linked list in a mutable way.

Type-universes. Type-universes, or *guarded* universes are owned by types rather than objects. The owner or *guard* type of a universe owns the objects belonging to the type's universe. The invariant of type universes is weaker than that of object-universes: *If an object x reaches object y by a chain of references, then the universe of the type of x is equal to or encloses the universe of y's type.*

Read-Only Mode. Universes integrated a read-only mode into their type system. Read-only references cannot be used to perform field updates or invocations of methods that have side-effects. Read-only references enable the possibility of accessing the representation of another object without being able to modify it. This allows the implementation of read-only iterators and read-only binary methods.

Universes have been presented in a language without static variables. Lately [61], universes have been integrated into JML [46].

2.5.8.5 Ownership Domains

Ownership domains [2] decouple encapsulation policy from mechanism of ownership. It is possible to specify multiple ownership domains for each object allowing a fine-grained control of aliasing. Furthermore the permitted aliasing between each pair of domains can be specified. In contrast to the owners-as-dominators property, ownership domains have a link soundness property that says: if an object *o* refers to object *o'* and *o'* is in domain *d*, then *o* has permission to access domain *d*. With ownership domains it is possible to express idioms like iterators and event callbacks without restricting their implementation to inner classes. To achieve an object level encapsulation, clients have to prefix domain names by final variables. It has to be investigated if this restriction is acceptable in practice.

2.5.9 Confined Types [P]

Confined Types [76] restrict aliasing of certain objects within the boundary of a Java package. An object of a confined class cannot be referenced by classes not belonging to the same package as the confined class. To be confined, a class has to fulfill certain constraints that are statically checkable. The most restricting constraint is that confined classes are not allowed to be public. That is, confined classes cannot be reused in other packages. Confined Types have been developed with security in mind, and they make it easy to define certain classes to be encapsulated within their package. However, confined types are too restrictive to be generally applicable, as they are not allowed to be public and thus cannot be reused in other packages, which also make it impossible to use confined types in collection classes that are defined in other packages.

In [33] the tool `Kacheck/J` was introduced. `Kacheck/J` infers confined types from not further annotated programs and is useful to identify confined types in existing code. Applied to a large collection of classes it was shown that confined types are

a natural concept that is already implicitly used in large software systems. In [25] confinement was applied to Enterprise Java Beans.

A recent work on confined types [79] formalizes confinedness with an operational semantics and a static type system based on Featherweight Java [39]. That paper also applies confined types to generic classes that allow confined collection types and confined classes that are public and thus can be reused. That extension needs so called *confinability tags* that define whether a type is confined or not. And even more recently [72] confined types have been formalized with Generic Featherweight Java [39] as basis.

2.5.10 Summary

All concepts mentioned above have been discussed for years. However, no concept alone can solve all encapsulation problems. Only a useful combination of some of these concepts can solve them. Each concept has different requirements to the analysis and the underlying language. These requirements are summarized in Table 2.1. It is shown whether a concept needs a change of the Java language in order to be able to be implemented, and whether a concept can be checked in an open-world scope. Note that most approaches can be checked in an open-world scope if they are integrated into the programming language. If a read-only mode, for example, is integrated into the programming language, it can be checked in an open-world scope. However, as it is not supported by Java it can only be checked in a closed-world scope. It is nevertheless compatible to Java as all annotations can be written within comments. Approaches like unique variables really need a change of Java as they require a destructive read, for example.

Concept	*Java Compatible*	*Open-World*
Unique Variables	no/yes	yes/no
Free Variables	yes	no
Value Types	no	yes
Immutable Types	yes	yes
Read-Only Mode	yes	no
Islands	no	yes
Balloons	no	no
Ownership Types	yes	no
Universes	subset	partly
Ownership Domains	yes	no
Confined Types	yes	yes

Table 2.1: Requirements of alias management approaches.

As we are aiming at an open-world analysis for Java programs, we can only consider three concepts:

- Universes
- Confined Types
- Immutable Types

Universes have been proposed in a language without static variables. So actually this does not require a change of Java, but is rather a restriction to a subset of Java. In addition, the universe approach can only partly be checked in an open-world scope, because of the read-only mode of universes. However, without the read-only mode universes can be checked in an open-world scope. All other approaches need either a change of the Java virtual machine or can only be checked in a closed-world scope.

We decided to adopt all three concepts. As we cannot force classes outside the analysis scope to not use static variables, we have to extend the universe approach in order to allow static variables in general. In addition, we adopt universes without a read-only mode.

2.6 The Problem of Encapsulation

> *The big lie of object-oriented programming is that objects provide encapsulation.*
> John Hogg [37].

Our goal is to encapsulate components. But what are encapsulated components? What needs to be encapsulated, why does it have to be encapsulated, and how should it be encapsulated? First of all it depends on the application domain. Among others the following domains need encapsulation:

1. *Modular Verification* needs encapsulation in order to reason locally about objects. If an object could be aliased by any object not visible in the verification context, it is impossible to reason locally about the state of that object. So the modifiable state of objects has to be encapsulated.

2. *Software Design* needs encapsulation in order to properly divide software into independent modules. This is needed in order to locally change a module without affecting arbitrary parts of the whole software system. There are two things that have to be encapsulated. The first is *information hiding*. That is, the implementation of modules should be hidden behind a strict external interface. Clients can only use the interface to interact with the module. The second is aliasing. Because, even if a client only uses the external interface, references to internal objects might leave the encapsulation boundary and enable the client to modify the state of internal objects, circumventing the external interface. “Aliasing is problematic because it breaks the encapsulation necessary for building reliable software components”, Liskov and Guttag [48].

3. *Security* needs encapsulation to ensure that certain objects never leave a certain security boundary, because sensitive internal data should *never* be available to certain parts of a software system.

All three domains have a different notion of encapsulation. All have in common that they define a certain *boundary*. Objects are divided into *internal* objects, objects which are within the boundary, and *outside objects*, those objects which are outside the boundary. And all domains have in common that references to internal objects should be prevented from leaving that boundary. In Chapter 3 we give a precise definition of our understanding of an encapsulated component.

Chapter 3

A Component Model

In this chapter we describe our component model. The model is very similar to the encapsulation model described by Noble et al. [64]. We took their *access graph* as basis of our model. Instead of defining an encapsulation function, we define the notion of an encapsulated component instead, but are using similar terms. In addition, we present a definition of independency, which was not given by [64].

3.1 Access Graph

An *access graph* $\mathcal{G} =_{def} \langle \mathcal{N}, \mathcal{E} \rangle$, is a 2-tuple consisting of a set of nodes $\mathcal{N}$ and a set of directed edges $\mathcal{E}$. Nodes and edges of the graph can be decorated. The access graph models the state of an object-oriented program execution. The nodes represent individual objects, and the edges have different meanings, which we describe later. Before that we introduce some notations:

3.1.1 Notations

Let $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ be an access graph and $a, b, c \in \mathcal{N}$,

- $\{a\}\triangleleft \stackrel{def}{=} \{c \mid (a,c) \in \mathcal{E}\}$, *is the set of all nodes that have edges beginning from a. That is, it represents the set of references and method calls*
- $\triangleright\{a\} \stackrel{def}{=} \{c \mid (c,a) \in \mathcal{E}\}$, *is the set of all nodes that have edges ending at a.*
- $\{a\}\lhd\!\lhd = \{a\}\triangleleft \cup \bigcup_{c \in \{a\}\triangleleft} \{c\}\lhd\!\lhd$, *is the transitive closure of $\{a\}\triangleleft$, that is, all nodes that can be reached from a.*
- $\rhd\!\rhd\{a\} = \triangleright\{a\} \cup \bigcup_{c \in \triangleright\{a\}} \rhd\!\rhd\{c\}$, *is the transitive closure of $\triangleright\{a\}$, that is, all nodes that can reach a.*
- $paths(a,b) \stackrel{def}{=} \{c_1, ..., c_n \mid c_1 = a, c_n = b, \forall i : 1 \leq i < n : (c_i, c_{i+1}) \in \mathcal{E}\}$, *is the set of all paths (sequences of nodes) from node a to node b.*

3.1.2 Edges

We distinguish three kinds of edges:

1. *Static references* held by instance variables. We write $a \longrightarrow_s b$ to denote that there is a static reference from object a to object b. The set of static references is named $\mathcal{E}_s \subseteq \mathcal{E}$.

2. *Dynamic references* held by local variables or formal parameters. We write $a \longrightarrow_d b$ to denote that there is a dynamic reference from object a to object b. The set of dynamic references is named $\mathcal{E}_d \subseteq \mathcal{E}$.

3. *Method invocations* that are either invoked on static or dynamic references. We write $a \longrightarrow_m b$ to denote that there is a method invocation of object a on object b. The set of method invocations is named $\mathcal{E}_m \subseteq \mathcal{E}$.

All sets of edges are pairwise disjunct, that is $\mathcal{E}_s \cap \mathcal{E}_d = \emptyset \wedge \mathcal{E}_s \cap \mathcal{E}_m = \emptyset \wedge \mathcal{E}_d \cap \mathcal{E}_m = \emptyset$. We write $a \longrightarrow_r b$ if there is either a dynamic or static reference from a to b. If there is a not further specified edge $(a, b) \in \mathcal{E}$ from object a to object b, we write $a \longrightarrow b$ and say that *a accesses b*. $\overset{*}{\longrightarrow}$ denotes the transitive closure of the corresponding edge kind. The static references can be determined by the heap, the dynamic references and the method invocations can be determined by the stack. We do not model field accesses and field updates, as these can always be replaced by using setter and getter methods.

3.1.3 Nodes

As mentioned above, nodes are objects. However, in Java one can distinguish between three kinds of objects: real objects, class objects and values of primitive types.

Value Nodes

Values are actually no objects [50], however, to keep our model simple, we model values as objects without any methods and fields. We only model integer values. The set of value nodes is named $\mathcal{N}_v \subseteq \mathcal{N}$. For every integer value, there exits exactly one node in the whole graph. The integer value can be obtained from a value node by the function *val*. Value nodes can neither reference other objects nor invoke methods on other objects:

$$\forall a \in \mathcal{N}_v : \{a\}\triangleleft = \emptyset \tag{3.1}$$

In addition, it is impossible to call a method on a value node:

$$\forall a, b \in \mathcal{N} : a \longrightarrow_m b \Longrightarrow b \notin \mathcal{N}_v \tag{3.2}$$

Real Objects

Real objects are instances of classes. The set of real nodes is named $\mathcal{N}_o \subseteq \mathcal{N}$. If an object a invokes a method on an object b, a must also have a reference to b:

$$\forall a \in \mathcal{N} : \forall b \in \mathcal{N}_o : a \longrightarrow_m b \Longrightarrow a \longrightarrow_r b \tag{3.3}$$

We use this statement in its reverse form, that is, if an object a has no reference to another object b, a cannot invoke a method on b:

$$\forall a \in \mathcal{N} : \forall b \in \mathcal{N}_o : \neg(a \longrightarrow_r b) \Longrightarrow \neg(a \longrightarrow_m b) \tag{3.4}$$

Class Objects

Class objects hold the static variables and the static methods of a class. There exists always exactly one class object for every class in the program. The set of class nodes is named $\mathcal{N}_c \subseteq \mathcal{N}$. The statement 3.3 is not true for class objects, because any object can invoke a method on a class object without having a reference to such an object. This corresponds to Java, where a static method can be called without having any reference to the class object. In fact, there can never be a reference to a class object:

$$\forall a, b \in \mathcal{N} : a \longrightarrow_r b \Longrightarrow b \notin \mathcal{N}_c \tag{3.5}$$

Like the different sets of edges, the different sets of nodes are pairwise disjunct:

$$\mathcal{N}_c \cap \mathcal{N}_o = \emptyset \wedge \mathcal{N}_c \cap \mathcal{N}_v = \emptyset \wedge \mathcal{N}_o \cap \mathcal{N}_v = \emptyset$$

Immutable Objects

Objects whose state cannot be changed after their construction are called *immutable objects* (cf. Section 5). We call nodes which are of immutable objects *immutable nodes.* We denote the set of immutable nodes with $\mathcal{N}_{im} \subseteq \mathcal{N}$. Value nodes are always immutable, so $\mathcal{N}_v \subseteq \mathcal{N}_{im}$. The immutability cannot be shown by the structure of the access graph, but has to be shown by the construction of the access graph. So we require from an instantiation of the model that immutable nodes are only represented by immutable objects. We indicate immutable nodes in figures of access graphs by filling them with gray color.

3.2 Components

After the definition of an access graph we now define the term *component* and the properties *encapsulation* and *independence*.

Definition 3.1 (Component) *Let $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ be an access graph. A* component *$\mathcal{C}$ is a triple consisting of a set of* boundary objects *$\mathcal{B} \subseteq \mathcal{N}, \mathcal{B} \neq \emptyset$, a set of* representation objects *(*rep objects*) $\mathcal{R} \subseteq \mathcal{N} - \mathcal{B}$ and a* main object *$m \in \mathcal{B}$:*

$$\mathcal{C} \stackrel{def}{=} \langle \mathcal{B}, \mathcal{R}, m \rangle$$

We indicate a component in the access graph by drawing a boundary around the representation objects and putting the boundary objects on the boundary.

Definition 3.2 (Inside) *Let $\mathcal{C} = \langle \mathcal{B}, \mathcal{R}, m \rangle$ be a component. The* inside *$\mathcal{I}_\mathcal{C}$ of $\mathcal{C}$ is given by the objects belonging to the component:*

$$\mathcal{I}_\mathcal{C} \stackrel{def}{=} \mathcal{B} \cup \mathcal{R}$$

Definition 3.3 (Outside) *Let $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ be an access graph and $\mathcal{C}$ be a component in that graph. The* outside *$\mathcal{O}_\mathcal{C}$ of $\mathcal{C}$ is given by all nodes that are not inside $\mathcal{C}$:*

$$\mathcal{O}_\mathcal{C} \stackrel{def}{=} \mathcal{N} - \mathcal{I}_\mathcal{C}$$

We will also use the term environment *to refer to outside nodes.*

Definition 3.4 (External Objects) *Let $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ be an access graph and $\mathcal{C} = \langle \mathcal{B}, \mathcal{R}, m \rangle$ be a component in that graph. The set of* external objects *$\mathcal{X}_\mathcal{C}$ of $\mathcal{C}$ is given by:*

$$\mathcal{X}_\mathcal{C} \stackrel{def}{=} \mathcal{R}_\mathcal{C} \triangleleft \cap \mathcal{O}_\mathcal{C}$$

That is, the external objects are all outside objects that are accessed by representation objects. This implies that $\mathcal{X}_\mathcal{C} \subseteq \mathcal{O}_\mathcal{C}$.
So a component is a set of objects with different roles. A component consists of boundary objects and representation objects. The external objects are referenced by the representation objects of the component, but they do not really belong to the component. That is, the abstract state of a component can only depend on boundary and representation objects. The invariants of a component can only depend on the abstract state of a component. To ensure the invariants, the environment must only be able to modify the abstract state of a component via method invocations on boundary objects. To ensure this, it is necessary that a component has two properties:

1. Representation objects have to be encapsulated within the component. That is, the environment must not be able to invoke a method on a representation object that causes any state modification. In other words, *representation exposure* has to be prevented.

2. Representation objects must not depend on mutable state of external objects. That is, no representation object must contain any code that depends on mutable results of method calls on an external object. A result of a method is mutable iff the result of that method is not always the same when supplied with identical arguments. In other words, *argument dependency* has to be prevented.

3.2.1 Encapsulated Components

We now define the notion of an encapsulated component that prevents representation exposure:

Definition 3.5 (Encapsulated Component) *Let $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ be an access graph and $\mathcal{C} = \langle \mathcal{B}, \mathcal{R}, m \rangle$ be a component in that graph. $\mathcal{C}$ is called* encapsulated *iff the following condition is satisfied:*

$$encap(\mathcal{C}) \stackrel{def}{=} \triangleright \mathcal{R}_\mathcal{C} \subseteq \mathcal{I}_\mathcal{C}$$

That is, all edges ending at a representation node have to come from boundary nodes or other representation nodes. For convenience we give some equivalent statements:

$$encap(\mathcal{C}) \iff \mathcal{O}_\mathcal{C} \triangleleft \cap \mathcal{R}_\mathcal{C} = \emptyset$$

That is, no outside object can directly access a representation object. Or a little more complex:

$$encap(\mathcal{C}) \implies \forall o \in \mathcal{O}_\mathcal{C} : \forall r \in \mathcal{R}_\mathcal{C} : \forall p \in paths(o, r) : \exists b \in \mathcal{B}_\mathcal{C} : b \in p$$

That is, all paths from outside objects to representation objects must pass through boundary objects.

The definition of an encapsulated component is more than is needed to prevent representation exposure, because an encapsulated component forbids *any* access to representation objects, even though a read-only access could be allowed. However, a read-only access needs the definition of methods that do not modify state, so we omit it here. Certainly, our model can be extended accordingly.

Example

Figure 3.1 shows an example access graph. The graph shows a component c consisting of representation objects $\mathcal{R}_c = \{a, b\}$ and boundary objects $\mathcal{B}_c = \{m, c, d\}$ where m is the main object of the component. g is the only external object, $\mathcal{X}_c = \{g\}$, denoted by a dashed edge $b \dashrightarrow g$. $\mathcal{O}_c = \{e, f, g, h\}$ are the objects of the component's environment. Encapsulated components forbid the edge from h to b ($h \nrightarrow b$).

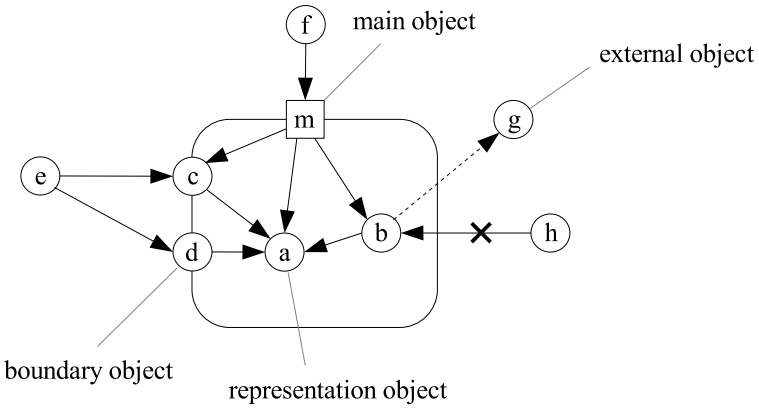

Figure 3.1: An access graph showing an encapsulated component.

Nested Components

Encapsulated components can be nested. That is, one component can be completely contained in another component. In Figure 3.2 two encapsulated components n and m are shown. Component n is nested within component m. From the perspective of n, the objects of m are just objects of the environment. So $\mathcal{R}_n = \{s, r\}$, $\mathcal{B}_n = \{n, d\}$, $\mathcal{O}_n = \{a, b, e, f, g, h, i\}$ and $\mathcal{X}_n = \{b, i\}$. Objects of m have to respect the same constraints like all other outside objects of n. So there cannot be any reference from objects of m to a representation object of n ($a \nrightarrow r$). However, all inside objects of n are representation objects of m: $\mathcal{I}_n = \mathcal{R}_m = \{a, b, d, s, n, r\}$. As all objects of n are encapsulated within m, outside objects of m cannot access any object of the nested component n ($e \nrightarrow d$, $h \nrightarrow n$, $g \nrightarrow r$).

3.2.2 Independent Components

The definition of an encapsulated component $\mathcal{C}$ does not restrict the accesses from representation objects $\mathcal{R}_\mathcal{C}$ to external objects $\mathcal{X}_\mathcal{C}$. However, if the state of representation objects depend on external objects, the state of these objects can be changed indirectly by outside objects, without using methods of boundary objects.

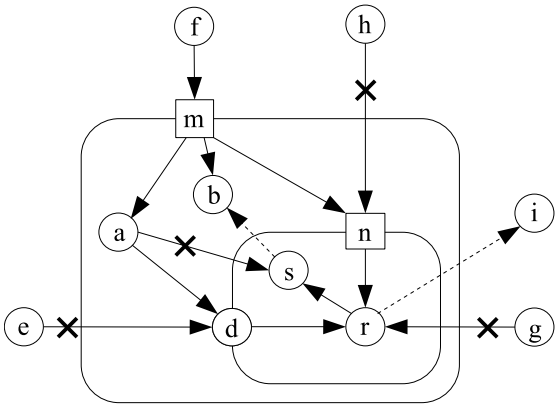

Figure 3.2: A component nested inside another component.

Definition 3.6 (Independent Component) *Let $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ be an access graph and $\mathcal{C} = \langle \mathcal{B}, \mathcal{R}, m \rangle$ a component in that graph. $\mathcal{C}$ is called* independent *iff the following condition is satisfied:*

$$indep(\mathcal{C}) \stackrel{def}{=} \forall r \in \mathcal{R}_\mathcal{C}, \forall x \in \mathcal{X}_\mathcal{C} : r \longrightarrow_m x \Longrightarrow x \in \mathcal{N}_{im}$$

That is, representation objects of independent components can only call methods on external objects iff these objects are immutable. Note that this does not restrict references to external objects, only method calls are forbidden. If we disallow *any* access, including references, from representation objects to mutable outside objects, we can obviously achieve independence:

$$\mathcal{X}_\mathcal{C} \subseteq \mathcal{N}_{im} \Longrightarrow indep(\mathcal{C}) \tag{3.6}$$

The definition of an independent component is more than is needed to prevent argument dependency, because it can be allowed for representation objects to call *immutable methods* on mutable objects. This is what Flexible Alias Protection [63] proposed. They defined so-called *clean* methods which always return an immutable result value. Even otherwise mutable objects can have clean methods. Representation objects are allowed in their definition to call clean methods on mutable objects. However, to keep our model simple, we did not adopt this idea. It has to be investigated in the future if such an extension is worth the more complex model.

Example

Figure 3.3 shows an independent component. Representation object a can access g, as g is an immutable object. The access from a to h, $a \not\rightarrow h$, however, is forbidden for independent components.

3.3 Related Work of Alias Management

In this section we present related work of alias encapsulation by means of the access graph and our component model. [64] present a more comprehensive comparison with additional concepts, like *unique* variables, for example.

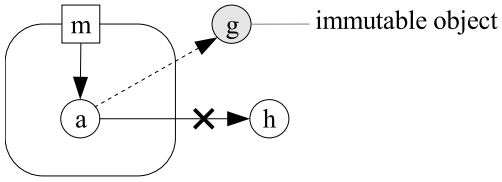

Figure 3.3: An independent component.

3.3.1 Full Alias Encapsulation

Full alias encapsulation means that references into the component as well as references out of the component are forbidden (see Figure 3.4). *Balloons* [5] and *islands* [37] are often seen as full encapsulation techniques. However, this is not true, because *balloons* allow dynamic references into the component, and *islands* allow *read-only* references to cross the component boundary. Only *opaque balloons* can be seen as a full alias encapsulation approach as these ones also forbid dynamic references.

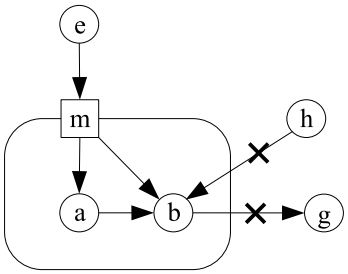

Figure 3.4: Full Alias Encapsulation

3.3.2 Owners-As-Dominators

Ownership types that enforce the *owners-as-dominators* property only allow one object on the boundary which is the main or *owner* object. References to representation objects are forbidden, but representation objects can access arbitrary objects from the outside (see Figure 3.5).

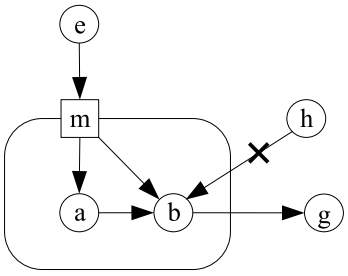

Figure 3.5: Owners-As-Dominators

3.3.3 Static Encapsulation

A variant of ownership types has been presented by Clarke et al. [24] which allows access to representation objects from outside objects by *dynamic references*. The only restriction is that these references must be allocated on the stack. References from representation objects to outside objects are not restricted at all (see Figure 3.6).

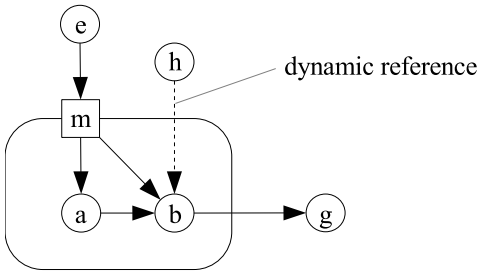

Figure 3.6: Static Encapsulation.

3.3.4 Universes

Universes [60] provide, besides representation encapsulation, dependency control. That is, representation objects are only allowed to reference objects from the environment by *read-only* references and invariants of the component must not depend on read-only references. In addition, read-only references from outside objects to representation objects are allowed. Objects on the boundary besides the main object of a component cannot be defined (see Figure 3.7).

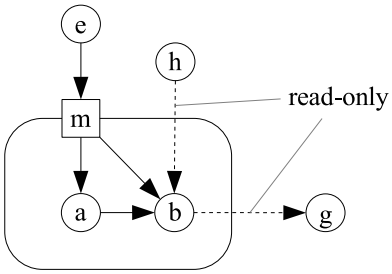

Figure 3.7: Universes

3.3.5 Inner Classes as Boundary Objects

Ownership types have been extended to allow more than one object on the boundary [19, 14]. These objects must be instances of *inner classes* of the main object's class. References from representation objects to outside objects are not restricted at all (see Figure 3.8).

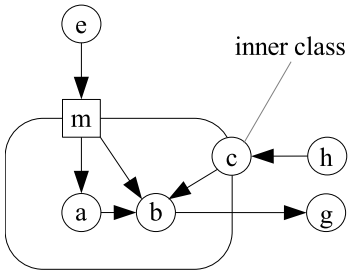

Figure 3.8: Boundary objects with inner classes.

3.3.6 Ownership Domains

Ownership domains [2] support the definition of more than one object on the boundary of a component. This can be done by defining a *public domain* and by *linking* that domain to the *owner domain* of the main object. All objects of that public domain can then access the representation of the main object and are accessible by the environment. Public domains, however, have to be attached by a final variable that references the main object. Ownership domains provide no dependency control, so representation objects can depend on mutable objects of the environment.

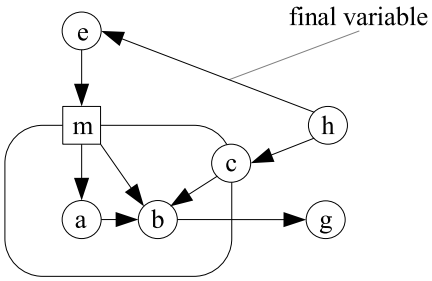

Figure 3.9: Ownership domains.

3.4 Typical Programming Patterns

This section describes programming patterns that appear in practice by means of our component model. These programming patterns show typical situations that complicate the encapsulation of components. We describe the problems and give solutions.

3.4.1 Containers

An encapsulated component should act as a container and should store references to possibly mutable outside objects. The outside objects do not belong to the representation of the component, and thus the abstract state of the container should not depend on the state of these objects. That is, the invariants of the container may neither depend on mutable result values of method invocations on outside objects nor on mutable fields of outside objects. The component has to distinguish between internal

and outside objects. In order to be reusable the type of the outside object should not be fixed (see Figure 3.10).

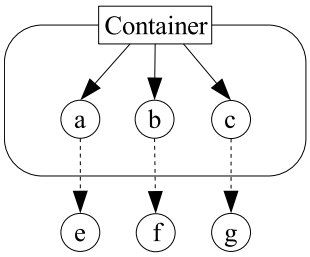

Figure 3.10: A component that acts as a container.

Solution

This problem cannot be solved with ownership types, because they do not provide any dependency control. That is, references from representation objects to outside objects are allowed, but are not restricted at all. So representation objects can depend on arbitrary mutable state. Universes [60] on the other hand only allow read-only references to outside objects, and class invariants may not depend on read-only references. Flexible Alias Protection (see Section 2.5.8.2) uses the *arg* mode to indicate argument objects. *arg* expressions only provide access to the immutable interface of the objects to which they refer. The immutable interface of an object is formed by the *clean* methods of that object. Where a clean method is guaranteed to only read immutable values.

3.4.2 Linked Data Structures

Objects which are not referenced by the main object should be representation objects of the component. The problem is that the reference relation is different to the ownership relation (see Figure 3.11).

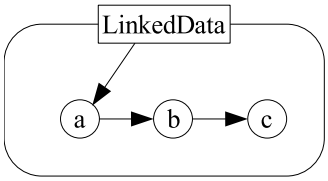

Figure 3.11: A component with representation objects that are not referenced by the main object.

Solution

This problem was solved by ownership types with the introduction of an owner keyword. In the example above, a would reference b with a reference annotated with the owner keyword.

3.4.3 Initializing Components

A representation object of a component should be created by the environment. As soon as the object has been passed to the component and becomes a representation object, there must not be any references from outside objects to the representation object anymore. Otherwise the state of that object can be modified by the environment without using the component's interface.

Figure 3.12 illustrates the situation. It shows two program states. In both states there are the three objects *a,e,m*. *m* is the main object of a component, and *e* is an outside object of *m*. In the first state, *e* references *a*, and *a* is an outside object. In the second state *a* has become a representation object of *m*, and *e* has no reference to *a* anymore.

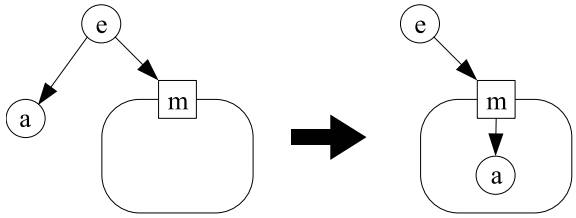

Figure 3.12: Initializing Components.

Example

Detlefs, Leino, and Nelson [28] discuss that problem with a reader and a lexer. We have modified their example and use classes from the JDK instead, it is essentially the same, however. Let us look at the java.io.Reader class of the JDK. It offers some rudimentary methods to access a stream. In Listing 3.1 we show a "light" version of the full Reader class.

```
1 public abstract class Reader {
2     public int read();
3     public void close();
4     ...
5 }
```

Listing 3.1: The java.io.Reader class.

The Reader class has a read method and a close method. The read method reads the next byte of the stream, the close method closes the stream. It is important that the read method is not called after the close method has been called. The Reader class is abstract and has to be inherited by a concrete implementation. As reading single bytes can be inconvenient there is the BufferedReader class. BufferedReader takes a

Reader class as argument and offers the method readLine that returns the next line of the stream as a String object (see Listing 3.2).

```
1 public class BufferedReader {
2     public BufferedReader(Reader reader) {...}
3     public String readLine() {...}
4 }
```

Listing 3.2: The BufferedReader class.

A client creates a BufferedReader instance and passes a certain Reader instance as parameter to the constructor of BufferedReader. If, however, the client saves an alias to the created Reader object, the client could call the close method of the Reader object without knowledge of the BufferedReader object (see Listing 3.3).

```
1 ...
2 Reader reader = new FileReader(foo.bar);
3 BufferedReader bufReader = new BufferedReader(reader);
4 reader.close(); // breaks invariant of bufReader
5 bufReader.readLine(); // error
6 ...
```

Listing 3.3: Client misusing the BufferedReader.

One could argue that the constructor of the BufferedReader could create the Reader instance itself. But this would restrict the BufferedReader to a certain Reader implementation limiting the reusability of the BufferedReader class. This example is especially good, because the Reader class cannot be made immutable, because an implementation could depend on an underlying system stream.

Solution

To solve the reader problem there *must* be some notion of a *not aliased variable*. There is no other solution. How this alias prevention is realized does not matter, but it must be guaranteed that there exists no alias to the object passed as argument. One could think of passing a factory object [30] as argument to the BufferedReader constructor instead of the concrete Reader instance. However, this is not a solution, because one must guarantee that the result of the factory method is unaliased, which only defers the responsibility. So either the variable is *unique* in the sense of Section 2.5.2 or *free* in the sense of Section 2.5.3.

3.4.4 The Iterator Pattern

A component has objects on the boundary besides the main object. These objects must access the internal representation to be efficiently implemented, but should not break the encapsulation of the component. A typical example is an iterator on a linked list. Noble [62] gives a comprehensive overview about different implementations of iterators. He comes to the conclusion that efficient iterators on linked lists can only be implemented if they have direct access to the internal node objects of the linked list.

Example

```
1  public class AddList {
2     Node head = new Node();
3     Node tail = head;
4     public void add(Object o) {
5        tail.next = new Node();
6        tail = tail.next;
7        tail.data = o;
8     }
9     public Iterator iterator() {
10       return new Iterator(head.next);
11    }
12 }
13 class Node {
14    Node next;
15    Object data;
16 }
17 public class Iterator {
18    Node current;
19    public hasNext() { current != null; }
20    public Object next() {
21       Object result = current.data;
22       current = current.next;
23       return result;
24    }
25    public void setCurrentData(Object data) {
26       current.data = data;
27    }
28 }
```

Listing 3.4: A linked list implementation.

We illustrate the problem with a simple linked list class called AddList which can

be seen in Listing 3.4. A node of the linked list is represented by the Node class. AddList has two Node fields: one for the head of the list, and one for the tail. It is only possible to add new objects to the linked list with the add method, or to traverse the list with an iterator. It is clear that the Node objects forming the list must be encapsulated within the AddList object, otherwise the linked list could be arbitrarily changed. One could remove an element from the list, for example, which is not possible via the interface of the AddList class. The Iterator object, however, must have write access to the internal Node objects to be able to change the data field of Node objects, but Iterator objects must also be accessible from the outside. This example, as easy as it seems, is one of the most challenging examples for ownership types. There has not been a satisfactory solution to this problem for ownership types so far. In fact it is still an unsolved problem. Figure 3.13 shows an access graph of an AddList example instance.

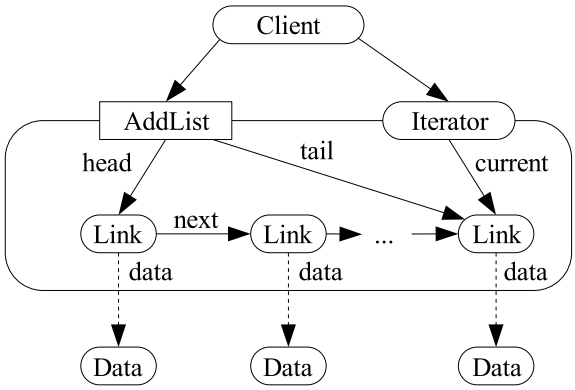

Figure 3.13: Access graph of an AddList instance.

Solution

With ownership types the only known way to solve this problem is by implementing the iterator with an inner class [19, 14]. Universes only allow the implementation of read-only iterators, as read-only access to representation objects is granted for all objects. The recent ownership domains [2] allow the definition of public domains. An iterator is put into a public domain of a linked list and is *linked* to the *owned* domain of the linked list. So the iterator has write access to the owned domain of the linked list and is nevertheless accessible by the environment. However, in order to achieve object-level protection, the client has to store the reference to the linked list in a final variable.

Chapter 4

Components in Java

In this chapter we present an approach to specify our component model in Java. We use a similar technique like Vitek and Bokowski [76] used to describe *confined types*, by presenting informal rules which must be satisfied by the source code. These rules are easily understood by programmers, and they can be immediately applied to the source code. In addition, we wrote a prototypical tool to check source code for conformance with these rules. This checker is presented in Chapter 7.

4.1 Preparations

4.1.1 Open-World Analysis

As already outlined in the introduction, we aim at a specification technique for components that is checkable in an open-world scope. In an open-world scope the set of classes which are analyzed is only a subset of all classes of the system. In addition, it is not possible to make any assumption about classes which are not covered by the analysis scope. In contrast, in a closed-world scope all classes of the system are within the scope, and no classes can be added later. If a component is checked in an open-world analysis scope, it can be used in an unchecked environment, and all invariants can still be guaranteed. Therefore we develop an open-world analysis rather than a closed-world analysis. In addition, our analysis is modular, that is, parts of a program can be checked independently of each other. A modular analysis, however, is in general not an open-world analysis as it generally assumes that all program parts are checked. JML [46] is an example for such a modular analysis that is not an open-world analysis.

4.1.2 Java Subset

We use Java as our base language. We do not require any changes of Java in order to achieve our approach, but to keep the presentation of our rules simple, we only regard a subset of Java. We omit the following features of Java:

- Threads
- Dynamic class loading
- Reflection
- Exceptions
- Native methods
- Nested Classes

- Arrays

At the end of this chapter we discuss the effects of these features to our rules and describe how the rules can be extended to handle them.

4.1.3 Object Ownership

Object ownership (see Section 2.5.8) can statically enforce the ownership of objects. Except universes, all approaches that have been proposed so far can only be checked in a closed-world scope, or need a change of the underlying programming language. Confined types (see Section 2.5.9) are encapsulated inside packages in an open-world analysis scope. We use the universe idea in our approach and enhance it later with the idea of confined types. In addition, we adopted immutable types (see Section 2.5.5) in our approach.

4.1.4 Object-Level vs. Class-Level Ownership

Most ownership type systems provide an object-level ownership. Single objects can own other objects, and objects of the same class cannot access one another's owning objects. Our approach only enforces a class-level ownership which does not forbid the access to objects which are owned by instances of the same class. This leads to a weaker invariant than that of object-level approaches, but it is still strong enough for most applications, especially when used in conjunction with program verification [55]. In Figure 4.1 two objects m and n of the same class M are shown. A class-level ownership does not forbid the access from m to c, whereas an object-level ownership does not allow this edge. So in fact, instances of the same class share a common representation (see Figure 4.2).

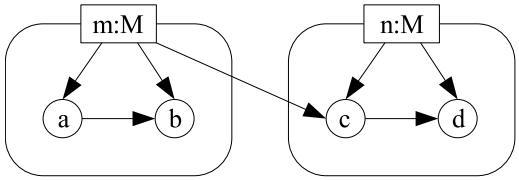

Figure 4.1: Objects of the same class can access one another's representation.

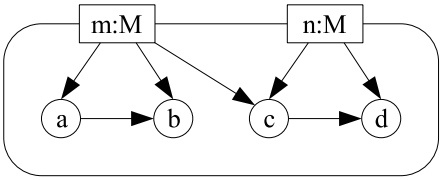

Figure 4.2: Objects of the same class share a common representation.

4.1.5 Overview of Our Approach

Our approach is similar to type-universes. Instead of types we only regard classes. That is, objects cannot be owned by interfaces, and subtypes do not own objects of their supertype. In the remainder of this chapter we describe the implementation of our component model in Java. Universes have been proposed in a language without static variables. Our first contribution is to remove this restriction and permit static variables in general. The second contribution is to allow more than one class to own a common representation. This enables the implementation of iterators that have write access to their iterating list. We integrate confined types into our approach by defining *rep classes*. Instances of such classes are always owned by a single class, and cannot be owned by other classes. Finally, we show how an object-level ownership can be achieved with our approach.

4.2 Class Components

A component, in our component model defined above, is a triple consisting of a main object, a set of boundary objects and a set of representation objects. To represent the component model in Java, all three kinds of objects must be identified. Objects in Java can be distinguished by classes. So the straightforward way to describe a component is by using classes. A component could consist of a main class, a set of boundary classes and a set of representation classes. This solution is too limiting, because it is impossible to reuse classes which are not specially written for the component. However, it is often the case that certain classes are only written for a single purpose and are not reusable elsewhere. So we keep this idea in mind, but begin with a different approach.

For the moment we only consider components with one boundary object. Later we generalize our approach to components with more than one boundary object. In addition, we begin with a class-level protection. Like the universes approach we ignore static variables for the moment.

4.2.1 Classes as Components

In principle, every object in our approach forms a component, where the object itself is the main object of the component. Without annotating a Java program, every object represents a component with an empty set of representation objects. To identify the representation objects of a component, we annotate fields that refer to representation objects with the keyword `rep`[1]. All objects that are referred to by rep-annotated fields are representation objects of the component. The encapsulation is a deep one, that is, all objects that are referred to by representation objects are representation objects as well (transitively). To clarify this property consider the following situation: Let A be a class and f be a rep-annotated field of that class. Let a be an instance of A and o be an object that is referred to by $a.f$. Thus, o is a representation object of a. Now, let p be an object that is referred to by o. The kind of the reference does not matter, that is, p could be referred to by a local variable, or by a field of o, for example. As o is a representation object of a, p is also a representation object of a (Figure 4.3.a). However, p can additionally be a representation object of o if it is

[1] In Java programs this keyword is enclosed within a comment so that the standard Java compiler can compile annotated programs. In Java 1.5 the built-in annotation possibilities can be used.

referred to by a rep-annotated field of o (Figure 4.3.b). In no case can p be an outside object of a (Figure 4.3.c). This is a fundamental difference to ownership types by

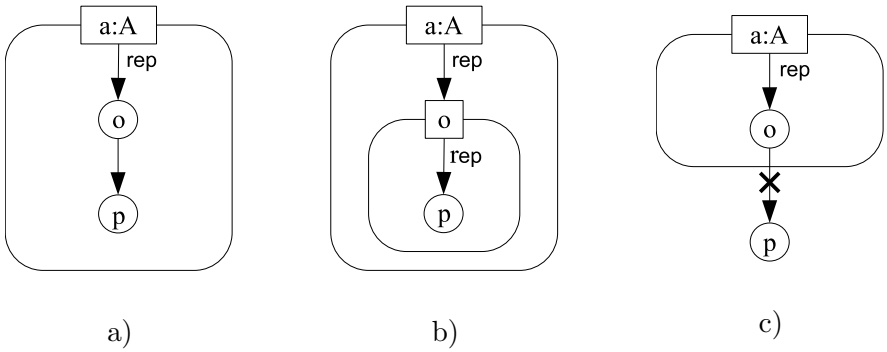

Figure 4.3: a) and b) are possible, c) is not.

Clarke et al. [23], because with ownership types all three cases are possible. The third case would even be the default case, that is, p is referenced by an ordinary expression in o. The second case would be the same like in our approach, and in the first case o had to reference p by an *owner* reference. So where the third case is the default case in ownership types, it is impossible with our approach. This shows that our approach is less flexible than ownership types regarding this situation, but unlike ownership types we provide dependency control. Ownership types can be seen as a *local* or *shallow* ownership, while our approach is a *transitive* or *deep* ownership.

Another property has to be pointed out. As all objects referenced by rep objects are rep objects as well, a rep object cannot have a reference to the main object of a component (see Figure 4.4), as the main object itself is not a rep object. This is a fact that was not forbidden by our definition of an encapsulated component, but is a result of our approach. We later partly fix this by introducing rep classes. However, this is one of the main drawbacks of our approach. Ownership types do not forbid this reference, as they do not restrict references from rep objects at all.

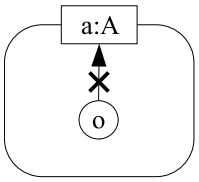

Figure 4.4: rep objects cannot reference the main object.

To ensure the encapsulation of representation objects, objects of the environment must not be able to obtain an alias to a representation object. We achieve this by not only annotating fields, but by allowing to annotate all type occurrences with the rep keyword. So actually, we extend the type system of Java. Every type can be annotated with the rep keyword to indicate that objects of these types are representation objects. So besides types of fields, the result type and parameter types of methods, the types of local variable declarations and new expressions can be annotated with the rep keyword. All possible annotations can be seen in Listing 4.1.

```
1  class Example {
2     rep Vector v;          // field declaration
3     private Example() {
4        rep Vector newV;  // local variable declaration
5        newV = new rep Vector();  // new expression
6        v = newV;
7     }
8     private rep Vector getV() {  // result type
9        return v;
10    }
11    private void setV(rep Vector aV) {
   // parameter type
12       v = aV;
13    }
14 }
```

Listing 4.1: A rep annotated source code.

Definition 4.1 (Rep Type) *A type is called* rep type *iff it is annotated with the* rep *keyword, otherwise it is called* ground type*.*

With the definition of a rep type we can now define *rep expressions.*

Definition 4.2 (Rep Expression) *An expression is called* rep expression *iff it is one of the following:*

1. *A local variable whose declared type is a rep type.*
2. *A formal parameter whose declared type is a rep type.*
3. *A field access of a field whose declared type is a rep type.*
4. *A* new *expression with rep type.*
5. *An invocation of a method with a rep type as result type.*
6. *A method invocation or a field access on a rep expression.*

Otherwise it is called ground expression*.*

Properties 1-5 can be directly deduced from the rep type definition, whereas the last property might seem confusing at first sight: A field access or method invocation on a rep expression always yields a rep expression. This behavior is independent of the type of the field or the result type of the method. That is, it is not possible to gain a ground expression from a rep expression.

Rep expressions are not assignment compatible with their corresponding ground expressions. That is, a rep expression cannot be assigned to a ground expression and

vice versa. For example, a variable with declared rep type rep T cannot be assigned to a variable with declared ground type T. Note that assignment does not only mean assignment by an assignment statement, but also includes the hidden assignments of passing a parameter to a method and returning the result of a method. We formulate this property in a rule:

R1	Rep expressions are only assignment compatible with other rep expressions.

Besides this, rep types declared in different classes are incompatible. That is, an expression of type rep T declared in a class A is not assignment compatible with an expression of type rep T declared in a class B. This can be illustrated by implicitly parameterizing rep types with the class in which they occur, so that rep types declared in different classes are syntactically different. A rep type rep T in a class A would then be declared as rep<A> T, for example. We omitted this in our approach, because of the resulting annotation overhead. It is, however, important to keep this in mind. Instead of supplying rep types with class names, we prevent the confusion of rep types declared in different classes by defining the following two rules:

R2	Fields of rep types must be private.
R3	Methods and constructors with rep types in their declarations must be private.

With these rules a class will never "see" a rep type of another class. So from the perspective of a single class there exist only rep types of the class itself and ground types.

Primitive types and immutable types cannot be rep types. Immutable types are presented in Chapter 5. Instances of an immutable type never change their states after their construction. Primitive types and immutable types cannot be annotated with the rep keyword, and if a field access or method invocation on a rep expression yields a value of a primitive or immutable type, it is implicitly casted to its ground type. Thus, immutable objects never belong to the representation of any component. In addition, this and super are never of rep types. This fact is important for the next rule.

Now there is still one problem left. This problem arises from the last point of the rep expression definition. The result of a method invocation on a rep expression is always a rep expression. Without further restriction it is not forbidden to turn a ground expression into a rep expression. That is, it is possible to indirectly assign a ground expression to a rep expression, which violates the first rule (R1). To illustrate the problem we give an example. In Listing 4.2 two classes are shown. Class A contains the method makeRep that takes a Vector as parameter and immediately returns it. Class B has a field repV with a declared rep type rep Vector and a method makeRep. The makeRep method creates a new instance of class A and assigns it to the local variable a. After that it calls the method makeRep on a with a ground expression as parameter. As a is a rep expression, the result of the method invocation is a rep expression as well and thus can be assigned to the rep field of A. So actually a ground expression was assigned to a rep expression, leading to a violation of rule R1.

```
1  class A {
2      Vector makeRep(Vector v) {
3          return v;
4      }
5  }
6  class B {
7      private rep Vector repV;
8      void makeRep(Vector v) {
9          rep A a = new rep A();
10         repV = a.makeRep(v);
11     }
12 }
```

Listing 4.2: Turning a ground type into a rep type.

To prevent such situations, no ground expressions may be passed to objects referred to by rep expressions. We formulate this in rule R4. Mutable parameters are parameters which are neither of primitive nor of immutable types.

R4	Mutable actual parameters of method invocations on rep expressions and rep constructor calls must be rep expressions.

Note that objects referred to by rep expressions can create new objects and return references to them as rep expressions. This is unproblematic, however, because there exists no ground type reference to these objects. We now define the term *rep object*.

Definition 4.3 (Rep Object) *Let A be a class. An object o is called* rep object of A *iff one of the following conditions is satisfied:*

1. *o is referenced by a rep expression occurring in A.*
2. *o is referenced by a rep object of A.*

The class A is called the owner *of o.*

Note that the definition of a rep object is w.r.t. a *class* and not an *object*. This is, because our rules do not prevent that instances of the same class confuse their rep objects. Nevertheless, we sometimes say that an object a is the *owner* of object o if a is an instance of a class which is the owner of o.

Classes that follow the rules R1-R4 can encapsulate private fields by annotating them with the rep keyword. That is, all objects referred to by rep-annotated fields are encapsulated by the declaring class. Before we formulate this property, we first define the *outside objects* of a class.

Definition 4.4 (Outside Object) *Let A,B be classes with $B \neq A$. An instance b of B is called* outside object of A *iff one of the following conditions is satisfied:*

1. *b is referenced by a ground expression occurring in A.*
2. *b is referenced by an object that is neither an instance of A nor a rep object of A.*

The following lemma states that an object can either be outside or rep but never both.

Lemma 4.1 (Rep Object Lifetime) *Let A be a class. An outside object of A can never become a rep object of A and vice versa. That is, the construction of an object determines the owner of the object for its whole lifetime.*

Proof: We show the lemma by induction on program states S_i. Let A be a class and a be an instance of A. Without loss of generality, assume that a is the only instance of A, otherwise apply the following justifications to all instances of A.

Induction Base: In the initial state S_0 there are no objects, hence no rep object can be an outside object and vice versa.

Induction Hypothesis: In state S_n no rep object is an outside object.

Induction Step: We show that in S_{n+1} no rep object can be an outside object. We show that

1. An outside object in S_n cannot become a rep object in S_{n+1}.
2. A rep object of S_n cannot become an outside object in S_{n+1}.

case 1: An outside object can never become a rep object. Let $B, B \neq A$ be a class. Let b be an instance of B and let b be an outside object but no rep object of A. We show that b cannot be a rep object of A in S_{n+1}. As b is an outside object of A, there are two cases:

1. b is referenced by a ground expression x occurring in A. To become a rep object of A in S_{n+1}, b must either be referenced by a rep expression occurring in A, or b must be referenced by a rep object of A.

   a) Suppose that in S_{n+1}, b is referenced by a rep expression y occurring in A. So there must have been an assignment of x to y. As such an assignment is forbidden by the first rule (R1), this is impossible.

   b) Suppose that in S_{n+1}, b is referenced by a rep object p of A. That is, x must have been passed to p by a. There are two cases:

      i. a has assigned x to a field of p. This is impossible, because according to the definition of a rep expression, a field access on a rep expression is a rep expression as well, thus this would be an assignment of a ground expression to a rep expression, which is forbidden by R1.

      ii. a has invoked a method on p with x as argument. This is not possible either, because R4 states that actual parameters of method invocations on rep expressions have to be rep expressions as well.

2. b is referenced by some object c which is neither an instance of A nor a rep object of A. Let C be the class of c. c is an outside object of A. To become a rep object of A in S_{n+1}, b must either be referenced by a rep expression occurring in A, or b must be referenced by a rep object of A. Let x be the reference to b stored in c.

   a) Suppose that in S_{n+1}, b is referenced by a rep expression y occurring in A. This leads to the following two cases:

      i. c has passed x to a. This could have been done in two ways:

         A. c has assigned x to a field f of a. f cannot be private, because $C \neq A$. As f is not private the type of f cannot be a rep type (R2) and so $a.f$ is not a rep expression.

         B. c has invoked a method m on a with x as argument. m cannot be private, because $C \neq A$. As m is not private the type of the formal parameter cannot be a rep type (R3). So the actual parameter cannot be a rep expression (R1).

      ii. a has obtained x from c. This could have been done in two ways:

         A. a has read a field f of c, which had x as value. As $C \neq A$, f cannot be private and the type of f has to be a ground type (R2). As c is not a rep object, the field access is a ground expression, and the resulting value cannot be assigned to a rep expression (R1).

         B. a has invoked a method m on c, which had x as result. As $C \neq A$, m cannot be private and the result type of m has to be a ground type (R3). As c is not a rep object, the method invocation is a ground expression and its result cannot be assigned to a rep expression (R1).

   b) In S_{n+1}, b is referenced by a rep object p of A. There are the following cases:

      i. c has passed x to p. In order to do this, c had to have a reference to p. However, this is not possible due to the inductive hypothesis that no outside object is a rep object. If c had a reference to p, p would be referenced by an outside object, and thus would be an outside object itself.

      ii. p has obtained x from c. In that case, p had to reference c. This is not possible, however, because c would then be a rep object of A, which is forbidden by the induction hypothesis.

case 2: A rep object can never become an outside object. Let o be a rep object of A. We show that o can not be an outside object of A in S_{n+1}. As o is a rep object there are the following two cases:

1. o is referenced by a rep expression x occurring in A. To become an outside object, o must either be referenced by a ground expression declared in A, or it must be referenced by an outside object.

   a) In S_{n+1}, o is referenced by a ground expression y in A. So there had been an assignment from x to y. This is not possible because of R1.

b) In S_{n+1}, o is referenced by an outside object e of A. Either a has passed x to e, or e has obtained x from a.

i. a has passed x to e. a could have done this in two ways:

A. a has assigned x to a field f of e. f must be a non-private field, so the type of f can only be a ground type (R2). As e is not a rep object, the field access is a ground expression, and thus the assignment is not possible (R1).

B. a has invoked a method m on e with x as argument. m must be a non-private method, and thus all parameters of m must be ground types (R3). As e is not a rep object, the argument to the method invocation has to be a ground expression, thus the invocation is not possible (R1).

ii. e has obtained x from a. e could have done this in two ways:

A. x was stored in a field f of a and e has read that field. f has to be non-private and thus the type of the field has to be a ground type (R2), so x could not have been stored in f (R1).

B. x was the result of a method m that e has invoked on a. m has to be non-private, and thus the result type of m has to be a ground type (R3). So x could not be the result of m (R1).

2. o is referenced by a rep object p of A. In order to become an outside object, o must either be referenced by a ground expression declared in A, or it must be referenced by an outside object. Let x be the reference to o stored in p.

a) In S_{n+1}, o is referenced by a ground expression y in A. So a had to obtain x from p.

i. a reads a field f of p which contains x. A field read on a rep expression is a rep expression as well, so the result cannot be assigned to y (R1).

ii. a invokes a method m on p, which returns x as result. A method invocation on a rep expression is a rep expression as well, and so the result cannot be assigned to y (R1).

b) In S_{n+1}, o is referenced by an outside object e of A. Either p has passed x to e, or e has obtained x from p.

i. p has passed x to e. In order to do this, p must have had a reference to e. This would mean that e is a rep object, which is a contradiction to the inductive hypothesis.

ii. e has obtained x from p. In order to do this, e must have had a reference to p. This would mean that p is an outside object, which is a contradiction to the induction hypothesis.

□

Lemma 4.2 (Rep Object Creation) *A rep object of a class A is always created by a method or constructor of A itself, or by a rep object of A.*

Proof: This is an immediate result of Lemma 4.1. If an object o would be created by an object e which is neither an instance of A nor a rep object of A, o would immediately be an outside object of A and could not become a rep object of A anymore.

□

Theorem 4.1 (Encapsulation Invariant) *Let A be a class and o be a rep object of A. Let x be an object that has a direct reference to o. Then x can only be one of the following:*

- *An instance of A.*
- *A rep object of A.*

Proof: The invariant is an immediate consequence from Lemma 4.1 and Lemma 4.2.

□

Note that in order to achieve the invariant, only class A must be checked to conform to rules R1-R4, no other class needs to be checked.

4.2.2 Example

In Listing 4.3 a code example is shown. There is a mutable class Point[2] that represents a two-dimensional point, and a class Circle that represents a circle with a center and a radius. Objects referred to by the center field should be encapsulated within Circle instances, so that outside objects cannot change the center of a circle without using the interface of Circle. To achieve this, we annotate the type of the center field with the rep keyword. We argue that this makes it impossible for outside objects to get an alias to the center object.

```
1 class Point {
2     int x;
3     int y;
4 }
5 class Circle {
6     private rep Point center;
7     private int radius;
8     ...
9 }
```

Listing 4.3: Example of a rep field.

To show the functionality of our rules, we now extend the Circle class with methods that try to expose an alias of the center field to outside objects. In Listing 4.4 methods

[2] Actually, this is a bad design, because the Point class should be immutable. However, for demonstration purposes we made it mutable. Ironically, the java.awt package even contains a mutable Point class.

are shown that normally would make the reference to the center object accessible to outside objects, but which are prevented by the rep constraints. For every statement, we indicate the rule which prevents the leakage in parentheses behind the statement.

```
1  class Circle {
2      ...
3      Circle(Point p) { center = p; }              // (R1)
4      Circle(rep Point p) { center = p; }          // (R3)
5      Point getCenter() { return center; }         // (R1)
6      void setCenter(Point p) { center = p; }      // (R1)
7      void setCenter(rep Point p) { center = p; } // (R3)
8      rep Point packageP = center;                 // (R2)
9      ...
10 }
```

Listing 4.4: Trying to expose an alias to a rep field.

R4 has not been needed in that example, because the Point class has no methods with mutable parameters. To see R4 in action we extend the Point class by a method that takes a java.util.Vector as parameter and stores this in that Vector (see Listing 4.5). However, R4 prevents the call of the addMe method with the Vector, because it is not of a rep type.

```
1  class Point {
2      int x;
3      int y;
4      addMe(Vector v) {
5          v.add(this);
6      }
7  }
8  class Circle {
9      ...
10     addCenter(Vector v) {
11         center.addMe(v); // (R4)
12         v.add(center);   // (R1)
13     }
14 }
```

Listing 4.5: Point stores this in a Vector.

4.3 Autonomous Classes

Our approach so far only works in the absence of static variables. Universes have also been presented in a language without static variables. There is a good reason for that limitation. To illustrate the problem we add a new method to the Point class that assigns this to a global static variable. This assignment is *not* prevented by our rules. The situation is shown in Listing 4.6.

```
1  class Global {
2      public static Point point;
3  }
4
5  class Point {
6      ...
7      void makeGlobal() {
8          Global.point = this;
9      }
10     ...
11 }
```

Listing 4.6: Class Point stores this in a global static variable.

In order to solve this problem we have to restrict the access to static fields by classes that are used as rep types. To do this, we introduce the notion of an *autonomous class*.

Definition 4.5 (Constant Static Field) *A* static *field is called* constant *iff it is declared* final*, and it is either of a primitive type or of an immutable type.*

Definition 4.6 (Autonomous Method) *A method (constructor) is called* autonomous *iff*

1. *It only accesses static fields that are constant.*
2. *It only calls autonomous methods and constructors.*

Both read and write access to non-constant static fields is disallowed for autonomous methods. If read access would be allowed, an alias to a mutable globally accessible object could be created by a rep object. Note that static methods can also be autonomous as long as they do not access any non-constant static fields. Such methods only depend on their arguments, which applies to the majority of static methods.

Definition 4.7 (Autonomous Class) *A class is called* autonomous *iff*

1. *It only has autonomous methods and autonomous constructors.*
2. *It inherits from* Object *or an autonomous class.*

We assume in this definition that Java's Object class is an autonomous class. As nearly all methods of Object are native methods, this depends on the implementation of the JDK, and so we cannot verify this. Concluding from the documentation of the Object class, however, we can assume that all methods are autonomous.

Armed with the definition of an autonomous class we now allow static fields by formulating an additional constraint:

R5	Rep types can only be autonomous classes.

With this constraint we are now able to allow static fields. However, the definition of an autonomous class holds a problem: what if a class inherits from an autonomous class and is not autonomous itself? That could be a problem, because this cannot be checked by an open-world analysis. However, as instances of rep types must always be created by objects of the component itself (Lemma 4.2), the actual types of rep objects are always known during compile time. Thus, subtypes of autonomous classes that are not autonomous as well and that are not in the scope of the analysis cannot influence checked components.

4.3.1 Extended Proof

We now extend our proof of Lemma 4.1 by regarding static fields and rule R5. Before doing this, we extend the definition of an outside object:

Definition 4.8 (Outside Object – Extended by Static Fields) *Let A be a class. Let B be another class, $B \neq A$. An instance b of B is called* outside object of A *iff one of the following conditions is satisfied:*

1. *b is referenced by a ground expression occurring in A.*
2. *b is referenced by an object that is neither an instance of A nor a rep object of A.*
3. *b is referenced by a static field, which is not a rep field of A.*

Extended proof of Lemma 4.1*:* As the first two properties have not been changed, Lemma 4.1 still holds for them. Therefore these already proven parts (cf. page 44) are omitted, and it is sufficient to prove the compliance with the third property.

case 1 is extended by a third case:

3. b is referenced by a static field f. There are the following two cases:
   a) Suppose that in S_{n+1}, b is referenced by a rep expression y occurring in A. So there must have been an assignment in A that had assigned f to y. As f must be of a ground type, such an assignment is forbidden (R1).
   b) Suppose that in S_{n+1}, b is referenced by a rep object p of A. So p had to read field f which is not possible because of R5.

case 2 is extended by the cases 1.c and 2.c:

1. c) Suppose that in S_{n+1}, o is referenced by a static field f, which is not a rep field of A. So a had to assign x to f. As f must be of a ground type (R2), this assignment is not possible (R1).
2. c) Suppose that in S_{n+1}, o is referenced by a static field f, which is not a rep field of A. So p had to read f. This is not possible, however, because of rule R5.

□

4.3.2 Static autonomous methods

The definition of autonomous methods allows the relaxation of the rep rules. A static autonomous method can only access its formal parameters and constant static fields. If such a method is invoked by a component, and all actual mutable parameters are rep expressions, such a method can neither introduce aliases from the environment to rep objects, nor from rep objects to objects of the environment. Thus, such a method invocation cannot break the encapsulation invariant of a component. We formulate this fact in a rule:

R6	Rep expressions can be passed as arguments to static autonomous methods, provided that all mutable actual parameters are rep expressions.

To prevent that it is possible with this rule to turn a rep expression into a ground expression, we have to extend the definition of a rep expression accordingly:

- A mutable result of the invocation of an autonomous static method, where all mutable actual parameters are rep expressions, is a rep expression as well.

With this definition we allow to pass rep expressions to static utility methods with mutable parameters. The methods of the java.util.Collections class are good examples for such methods.

4.4 Limitations so far

Our approach so far has similar limitations like the universes approach. Universes have an additional read-only mode for types, which increases the expressiveness of the universes approach. However, a read-only mode cannot be checked in an open-world scope, and so we did not adopt it. Instead we use the concept of immutable types. In the following, we indicate some problems that cannot be solved with our approach so far (see Section 3.4 for a more detailed description of the mentioned problems):

1. *Initializing components.* That is, it is impossible to create a rep object outside of the component. Müller [55] proposed that every object must have a clone operation in order to produce a full copy of an object which can then belong to a different universe. This is not a real solution to the problem, however. As already mentioned in Section 3.4.3, the initialization problem can only be solved

by some notion of a unique variable. We discuss this extension in Section 6.3, but the initialization problem is impossible to be solved in an open-world scope without language support.

2. *Container components.* That is, references to mutable ground objects cannot be stored in rep objects. Universes solve this problem by allowing read-only references to point into different universes. However, that solution is not perfect, because a container class that stores read-only references cannot be used to store rep references. Müller [55] proposes to store ownership information at runtime, to allow the casting of read-only references to rep references. We instead allow *immutable* ground objects to be referenced by rep objects. This problem could be solved by a parameterized type system perhaps. We leave its examination to future work.

3. *The iterator problem.* Let A be a class and o be a rep object of that class. It is impossible to give a class B access to o and still allow B to be accessible by outside objects. For example, a linked list cannot own its link objects, and still allow an iterator object to access the link objects, while being accessible by outside objects. Müller [55] partly solves this by allowing read-only access to objects belonging to a different universe. However, this solution does not allow write access, and it gives the read-only privilege to all objects, while it would be enough to give it to the boundary objects. So that approach effectively breaks information hiding.

Since the first problem is impossible to solve within our requirements, and the second problem can perhaps be solved by a parameterized type system, we concentrate on solving the third problem: Giving certain objects that are not the owner object write access to rep objects and still allow them to be accessible by objects of the environment.

4.5 Package Components

Packages are a built-in encapsulation mechanism in Java. So it is a straight-forward approach to use this mechanism to define the component boundary. We call components with the package as boundary *package components*. That is, a component is represented by a whole package. All public classes declared in the package are boundary classes, and all classes of the package share their representation objects. In order to allow an open-world analysis, it has to be ensured that it is impossible to add new classes to the analyzed package later on. This can be done by a Java mechanism called *sealing*.

4.5.1 Sealing of Packages in Java

Biberstein et al. [9] gives a comprehensive description of the sealing feature in connection with an open-world analysis. Packages that are stored in a Java Archive (JAR) can be marked to be *sealed*. Such packages have to be completely contained in that JAR file. Trying to add a new class to a sealed package results in the throw of a java.lang.SecurityException by the Java Virtual Machine. Except two packages, all packages of the JDK 1.2, for example, are sealed [9]. We assume in the following that all packages that are in the analysis scope are sealed. Biberstein et al. [9] proposed a

classification of types resulting from sealed packages, which we adopt and use in our work, too. They define four kinds of types:

1. *Polymorphic* types are types which can have an arbitrary number of subtypes. So the dynamic type of a reference with a declared polymorphic type cannot be known during compile time in general.

2. *Oligomorphic* types are types which have a known set of subtypes. For example, package-private classes of a sealed package can only have subtypes in the same package, and thus all subtypes are known at compile-time as long as no subtype is polymorphic. Even public classes can be oligomorphic if they only have package-private constructors and an explicit package-private default constructor.

3. *Monomorphic* types are types whose references can only take objects of one dynamic type. Such types cannot be subtyped at all. Classes that are either final or that only have private constructors and an explicit private default constructor [11] are monomorphic, for example. The java.lang.String class is an example for a monomorphic class.

4. *Nilmorphic* types are types without any concrete subtypes. A package-private abstract class in a sealed package without any subtypes is nilmorphic, for example. Such classes can be used as a collection of constants and static methods. The java.lang.Math class is an example for a nilmorphic class.

4.5.2 Package Components

To achieve package components, rep types declared in classes of the same package must be assignment compatible. In order to do this we change the constraints R2 and R3 to allow package-private rep fields and methods. We mark the entries with a star ($\star$) to indicate that these are changed constraints.

R2*	Fields of rep types must be private or *package-private*.
R3*	Methods and constructors with rep types in their declarations must be private or *package-private*.

Package components offer the possibility of having different classes that own the same representation objects. So an iterator on a linked list can be implemented by putting the iterator and the linked list into the same package. The iterator can now access the link fields of the list class and is still accessible by classes outside of the package. This approach is very similar to confined types [76] (cf. Section 2.5.9). However, our approach allows objects of classes of different packages to be encapsulated, while confined types can only confine classes of the same package. We achieve this with the drawback of a higher annotation effort, though. We use package components as basis for the next section and give a more general solution.

4.6 Boundary Classes

Using a package as encapsulation boundary is unsatisfactory, because this boundary is often too wide. It should be possible to define components with a narrower boundary.

For example, a linked list perhaps only needs to give its iterator class access to its representation, all other classes of the package should not be able to access it. To create a separate package only for the linked list class and its iterator would certainly be an overkill. Before going ahead we leave the Java world for a second and look into the world of C^{++}.

C^{++} [74] has no package mechanism, but a *friend* mechanism. A class can declare, within its body, certain other classes to be its *friends*, allowing these classes to access its private fields and methods. This offers a more fine-grained protection than packages in Java. We adopt this technique for our purposes. A class can be specified to be the *boundary class* of a component.

Definition 4.9 (Boundary Class) *Let A,B be classes from the same package. B is called* boundary class *of A iff the class declaration of B is annotated with* boundof A*, or if it is inherited from a boundary class of A. A is called the* bounded class *of B.*

1. *A class cannot be the boundary class of another boundary class.*
2. *A boundary class can only have one bounded class.*
3. *A boundary class that is inherited from another boundary class cannot redefine the bounded class.*

Note that if a class inherits from a boundary class that lives in a different package, it is *not* a boundary class! Boundary and bounded class must reside in the same package. The boundary class and its bounded class belong to the same component and both classes share the same representation objects. A class can have an arbitrary number of boundary classes, but a boundary class can only be the boundary class of exactly one class. If class B is a boundary class of class A and a is an instance of A and b is an instance of B we also say that b is a *boundary object* of a.

We now extend our rules to allow boundary classes. The rules are shown in Table 4.1. Note that *all* classes of a package have to follow these constraints. Also note that rules R2 and R3 are the changed rules from package components.

R1	Rep expressions are only assignment compatible with other rep expressions.
R2	Fields of rep types must be private or package-private.
R3	Methods and constructors with rep types in their declarations must be private or package-private.
R4	Mutable actual parameters of method invocations on rep expressions and rep constructor calls must be rep expressions.
R5	Rep types must be autonomous classes.
R6	Rep expressions can be passed as arguments to static autonomous methods, provided that all mutable actual parameters are rep expressions.
R7*	Classes can only access rep fields and rep methods of classes belonging to the same component.

Table 4.1: Rules for rep expressions in the presence of boundary classes.

The new rule R7 can be checked in an open-world scope, because the set of classes that have to be checked is bounded by the package. We now give a new encapsulation invariant that is extended by boundary objects.

Theorem 4.2 (Encapsulation Invariant – Extended with Boundary Objects) *Let A be a class and o be a rep object of A. Let x be an object that has a direct reference to o. Then x can only be one of the following:*

- *An instance of A.*
- *A rep object of A.*
- *A boundary object of A.*

Proof: The proof is done by giving a new proof of the rep object lifetime lemma. This proof is analogous to the previous proof, but all applications of rules R2 and R3 have to be extended by rule R7 to prevent the access of classes from the same package to rep fields and rep methods. In addition, the proof has to be extended by instances of boundary classes, which is analogous to the proof of instances of the main class.

□

4.6.1 Example

To illustrate the application of boundary classes, we show an implementation of a linked list of String objects. The implementation is shown in Listing 4.7. The nodes are represented by Node classes. They have to be encapsulated by the linked list and are therefore annotated with the rep keyword. In addition, a StringListIterator class is a boundary class of the StringList class and can access its rep fields. We used a list of Strings as example, because String is an immutable class. A general linked list cannot be implemented with this approach, as already mentioned above.

4.6.2 Relation to the Component Model

A component in our component model of Chapter 3 is defined as a triple $\mathcal{C} = \langle \mathcal{B}, \mathcal{R}, m \rangle$ consisting of a set of boundary objects $\mathcal{B}$, a set of representation objects $\mathcal{R}$ and a main object m. We can now map our component specification in Java to this model by defining these objects. As we defined components by means of classes, every instance of a class belongs to the boundary objects. In addition, all instances of boundary classes belong to the boundary objects of their bounded class. The set of representation objects is given by all rep objects of a class, that is, all objects that are transitively referenced by rep expressions in boundary objects. The main object of a component can be any boundary object.

Encapsulation

The encapsulation property of components is defined on page 26 (Definition 3.5). It states that a component is encapsulated iff all edges ending at representation nodes are coming from internal nodes: $encap(\mathcal{C}) = \triangleright\mathcal{R}_\mathcal{C} \subseteq \mathcal{I}_\mathcal{C}$. Translated to Java this means that only objects of the component can directly reference representation objects. This fact, however, is exactly formulated in Theorem 4.2. So components of our approach are encapsulated in our component model.

```
1  class Node {
2     Node next;
3     String data;
4  }
5
6  class StringList {
7     rep Node head;
8     rep Node tail;
9     public StringList() {
10       head = new rep Node();
11       tail = head;
12    }
13    public void add(String s) { ... }
14    public StringListIterator iterator() {
15       return new StringListIterator(this);
16    }
17 }
18
19 class StringListIterator boundof StringList {
20    rep Node current;
21    StringListIterator(StringList list) {
22       // allowed because StringListIterator is a
23       // boundary class of StringList
24       current = list.head.next;
25    }
26    public boolean hasNext() { return current != null; }
27    public String next() {
28       // allowed because String is immutable
29       String result = current.data;
30       current = current.next;
31       return result;
32    }
33 }
```

Listing 4.7: A string list having an iterator as boundary class.

Independence

The definition of an independent component (Definition 3.6 on page 28) states that a component is independent iff no representation object calls any method on mutable objects. Our approach ensures that components are independent according to that definition. We achieve this by using implication 3.6: $\mathcal{X}_\mathcal{C} \subseteq \mathcal{N}_{im} \Longrightarrow indep(\mathcal{C})$. Because we forbid any references from rep objects to mutable external objects, our approach ensures that all external objects are immutable ($\mathcal{X}_\mathcal{C} \subseteq \mathcal{N}_{im}$). Hence, we achieve the independence property.

To clarify the whole concept, we show the application of the rules by means of an example.

4.7 Rep Classes

It is often the case that classes are written only to be used by a certain other class. The Node class of Listing 4.7, for example, could be written only to be part of the StringList implementation. If objects of such classes should additionally be encapsulated, one could hard-wire the encapsulation property to these classes. We introduce *rep classes*.

Definition 4.10 (Rep Class) *Let A,R be classes of the same package. R is called* rep class *of A if the class declaration of R is annotated with* repof A*, or it is inherited from a rep class of A. A is called the* owner class *of R.*

1. *A rep class must not be declared* public*.*
2. *A rep class must be inherited from* Object *or a rep class.*
3. *The owner class of a rep class can be neither a boundary class nor a rep class.*
4. *A rep class that is inherited from another rep class cannot redefine the owner class.*

A rep class always belongs to the representation of exactly one other class, its owner class. Like boundary classes, rep classes and their owner classes belong to the same component, and thus they share the same representation objects. However, instances of rep classes are *not* rep objects like defined above. To distinguish instances of rep classes from rep objects, we call objects of rep classes *rep class objects*. In contrast to rep objects, rep class objects *can* access the main object and the boundary objects of its component, and rep classes must not be autonomous. Note that in contrast to rep objects, objects referred to by rep class objects are not automatically rep objects as well. Also note that rep classes are *not* rep types, and an expression whose type is a rep class is *not* a rep expression. The rules for rep classes are shown in Table 4.2.

RC1 and RC2 are obvious, because in an open-world scope we cannot assume that classes of other packages follow the rules. So we have to use the Java built-in visibility mechanism and forbid that rep classes appear in public and protected fields and methods. One might wonder why this is needed, because a rep class itself has to be declared as package-private. That is, the rep class type is invisible to all classes outside the package. However, if a public method, for example, had a rep class as result type, a class from another package could assign the result of that method to a variable that is

RC1	Fields with rep classes as declared type must be private or package-private.
RC2	Methods and constructors with rep classes in their declarations must be private or package-private.
RC3	A rep class can only be used by classes belonging to the same component as the rep class.
RC4	Widening of references from a rep class to Object is forbidden in assignments, method call arguments, return statements, and explicit casts.

Table 4.2: Rules for rep classes.

declared as Object. Thus, outside objects could obtain references to rep class objects. To prevent this we need RC1 and RC2.

RC3 prevents classes of the same package from using a rep class of another component. For example, if a class R is the rep class of a class M, only M, rep classes of M and boundary classes of M are allowed to access R.

RC4 ensures that the type information that a type is a rep class cannot be lost. For example, if a variable x has a rep class as declared type, it is not allowed that this variable is assigned to a variable y that has Object as declared type. Otherwise y could be returned by a public method, because the information that y has a reference to a rep class object is lost. So expressions of rep class types cannot be used where Object is expected.

Rep classes can be seen as a more fine-grained variant of confined types [76]. In fact the rules for rep classes resemble the rules for confined types. Like confined types, rep classes must be in the same package like their user classes, and thus cannot be reused in other packages. Both concepts need the same annotation effort, of only one keyword. But there are differences, too. We did not adopt the possibility of inheriting rep classes from ordinary classes (except Object), because this would mean to define the notion of an *anonymous* method like it is done for confined types. We left this idea as future work. Confined types can be used by all classes belonging to the same package, while rep classes can only be used by classes of the same component.

Figure 4.5 shows a component with a rep class object. Class R is a rep class of class A. a is an instance of A, and r is an instance of R. A number of properties can be seen. In contrast to rep objects, r *can* reference the main object a as well as boundary object b. In addition, r can reference outside object e, where the reference is not restricted at all. The outside object f cannot reference r, because r is encapsulated. And finally, the rep object p cannot reference r, as r is not a rep object. Object o is a rep object, because it is referenced by a rep expression of r.

The question is, does a rep class object really belong to the representation of a component? On the one hand, rep class objects are encapsulated in a way that forbids outside objects to reference them; on the other hand, rep class objects can reference outside objects without any restriction. The answer is yes and no. The whole transitive state of a rep class object does clearly not belong to the representation, because it can include outside objects. However, the local state is protected. This opens another question: how do rep class objects fit into our component model? Not very well, because they neither belong to the boundary nor to the representation of a component. In order to describe these objects, our model has to be extended by a third kind of

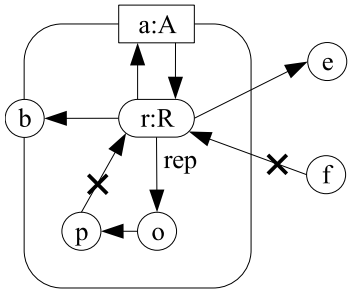

Figure 4.5: A component with a rep class object.

component objects. Objects that cannot be referenced by the environment, but which can reference objects of the environment without restriction. We left this extension as future work. An interesting point is that ownership types (cf. Section 2.5.8.3) exactly describe these kinds of objects.

Solving the Container Problem

With rep classes it is now possible to partly solve the container problem, because rep class objects can reference outside objects. We show this by means of an example. We use a similar example like in the previous section, but instead of a linked list of String objects, we show the implementation of a general linked list class. The code of the example is shown in Listing 4.8. The Node class is now declared as a rep class of the LinkedList class. Note that the rep keyword could be completely removed from the implementation.

4.8 Object-Level Protection

Our approach only offers a class-level protection. That is, two instances of the same component class can access each other's representation in general. Ownership type systems like described by Clarke et al. [23], Müller and Poetzsch-Heffter [60] provide an object-level protection. Although they are using a type system that enforces the ownership at compile-time, they are able to achieve object-level protection. The object-level protection is attained by a simple trick, which, however, is also the reason for the restrictiveness of these approaches: they only allow access to rep fields and rep methods on this. This is a very conservative rule that even forbids the implementation of binary methods like an equals method to compare two objects (see Listing 4.9). This restriction is too strong for our approach, because it forbids boundary objects to access any rep field of their bounded object, for example. The universes approach [60] additionally allows read-only access to rep fields of other objects, enabling read-only binary methods and read-only iterators. As already mentioned above, the read-only mode is no choice for us, so we cannot use this approach either.

Ownership domains [2] also achieve an object-level protection. Besides allowing access on fields of this they allow different objects of the same domain to access rep fields if the domain is attached by a final variable. This ensures that an object cannot switch between different domains and can only belong to one domain during its whole lifetime.

```
1  class Node repof LinkedList{
2      Node next;
3      Object data;
4  }
5  class LinkedList {
6      Node head;
7      Node tail;
8      public LinkedList() {
9          head = new Node();
10         tail = head;
11     }
12     ...
13 }
14 class LinkedListIterator boundof LinkedList {
15     Node current;
16     ...
17 }
```

Listing 4.8: A linked list with a Node class as rep class.

4.8.1 Rules to Ensure Object-Level Protection

We adopted the technique of using final variables. Object-level protection for our approach can be achieved by the rules presented in Table 4.3.

OL1 forbids static fields to be of rep types. This restriction is also used by ownership types, and it clearly makes no sense in an object-level protection to store references to rep objects in static fields.

OL2 restricts the accessibility of rep members (rep fields and rep methods). Clearly, access to rep members on this must be allowed. However, this would forbid boundary objects to access rep members of their bounded object, for example. So in addition, we allow that rep members can be accessed on objects that are referred to by final fields of this. So a boundary object can store its bounded object in a final field and can then access the rep members of its bounded object. This ensures that a boundary object is connected to a single bounded object during its whole lifetime. However, if a boundary object would have two final fields with two different bounded objects, it could access the rep members of two different components. The rules OL2-OL5 avoid such situations and assure that final fields of boundary objects and rep objects can only point to objects of the same component. For simplicity, we ignore the fact that a final field can be directly initialized in its declaration and assume that final fields can only be initialized by a constructor.

OL3 is important to prevent that final fields are initialized by static variables. So it is only possible to initialize final fields by constructor parameters.

OL4 prevents that more than one formal parameter can reference an object of a component class. However, if a an object of a component class has non-final fields

```
1 public class C {
2     rep Object o;
3     public boolean equals(C c) {
4         // not allowed, because c != this
5         this.o.equals(c.o);
6     }
7 }
```

Listing 4.9: Binary method example.

OL1	static fields must not be of rep types.
OL2	Rep fields and rep methods can only be accessed on this, on objects referred to by final fields of this or by formal constructor parameters.
OL3	All constructors of component classes must be autonomous.
OL4	Each constructor of rep and boundary classes can only have at most one parameter that is of a class of the same component.
OL5	Constructors of rep and boundary classes can only assign formal parameters, or final fields of formal parameters to final fields of this.
OL6	Constructors of the main class of a component must not have any parameter that is of a class of the same component.
OL7	The constructor of a main component class must not create instances of its own class.

Table 4.3: Rules to achieve an object-level protection.

with references to objects of other component classes, these fields could be assigned to final fields of this. This is prevented by rule OL5.

OL6 forbids constructors of main classes of a component to take any class of the same component as parameter. This makes sense, because the main object of a component is the first created object of that component, so there cannot exist any other objects of that component.

OL7 ensures that a main object of a component cannot create another main object of the same component within its constructor and assigns it to a final field. Note that this does not forbid main objects to create boundary objects and rep class objects of their components and assign them to final fields.

All these rules guarantee that different instances of the same component can access one another's representation objects. These rules seem to be too conservative and might be too constraining in practice. Nevertheless, they offer the possibility of implementing iterators. However, like ownership types they prevent the efficient implementation of binary methods.

4.8.2 Example

Recall the StringList implementation of Listing 4.7. This implementation already follows the new rules, and it need not be changed.

4.9 Extending the Java Subset

In this subsection we describe how Java features that have been ignored so far affect our approach.

4.9.1 Arrays

Rep Expressions

Arrays which have been declared with a rep type are called *rep arrays*. To allow arrays, we extend the definition of a rep expression by the following point:

- Array accesses on rep arrays are rep expressions as well.

The question is whether the component type of an array can be declared as rep type, but not the array itself. We show this by means of an example. In Listing 4.10 a class with two array fields can be seen. Field a is an array which is rep itself, but has the ground type Point as component type. Field b is not rep-annotated itself, but has rep type rep Point as component type. All accesses on field a always are of rep type rep Point, so in fact the element type of a is also of type rep Point. As the type of b contains a rep type, it can be seen as a rep type itself, and we could extend the definition of a rep type accordingly. However, this means that a and b are effectively of the same types. So we simply disallow the possibility of annotating the component type of an array as rep type.

Rep Classes

We call arrays of rep classes *rep class arrays*. Arrays of confined types are confined as well, so this should apply to rep class arrays as well. As the rep class always appears

```
1 public class ArrayExample {
2     rep Point[] a;
3     (rep Point)[] b;
4 }
```

Listing 4.10: Rep-annotated arrays.

in the declaration of a rep class array, all rep class rules apply to rep class arrays, too. In addition, rule RC4 must be extended to prevent the widening of rep class arrays to Object arrays.

4.9.2 Nested Classes

Nested classes in Java (cf. Subsection 2.2.7) can always access the private fields and methods of their outer classes. Inner classes even have an implicit reference to their outer class. This fact was already used to realize boundary objects for ownership types [14, 19]. There are two possibilities to integrate nested classes into our approach:

- Nested classes always belong to the component of their outer class.
- Nested classes are treated like any other class of the package and cannot access rep expressions of their outer class in general.

We decided that nested classes belong to the component of their outer class. That is, nested classes share the representation objects with their outer class. So all non-private nested classes are implicit boundary classes of their outer class. A nested classes can be declared as rep class of its outer class in which case only classes of the same component can access that class. To achieve an object-level protection, inner classes cannot take any constructor parameter of a class of the same component, as they already have an implicit reference to the main object of the component.

4.9.3 Exceptions

Rep Objects

Imagine that a rep object throws an exception that is not caught by a component class, but caught by an outer class. If that exception has a reference to a rep object, the outer class can obtain a reference to the rep object, breaking encapsulation. So exceptions are a problem for our approach. A solution is to forbid that exceptions can reference rep objects. This is assured by the following rules:

R_{Exc^1}	Rep Objects must not pass references that point to mutable objects to objects whose classes inherit from java.lang.Throwable.
R_{Exc^2}	Classes which inherit from java.lang.Throwable cannot be rep types.

An object *a passes* a reference x to object b if a invokes a method on b with x as argument, assigns x to a field of b or uses x as argument for the construction of b.

Rep Classes

Like confined types, rep classes must not inherit from java.lang.Throwable. As rep classes can only inherit from Object or a rep class, this is forbidden already by our rules.

4.9.4 Threads

Threads do not directly affect our approach. As encapsulated components, however, are not thread-safe in general, the state of components can be changed in a way that breaks the invariants of the component, even though only methods of boundary objects have been used to access the component. Thread-safe components are a research topic that could be investigated in the future.

4.9.5 Dynamic class loading

With dynamic class loading it is possible to load classes that are not known at compile time. So rep objects could load classes which are not autonomous and are able to store references to rep objects into static fields, and classes of a package containing a component could load classes which do not follow our rules. As we cannot control dynamic class loading, we have to forbid it in these cases::

R_{DCL^1}	Rep Objects must not use dynamic class loading.
R_{DCL^2}	No class of a package containing a component must use dynamic class loading.

4.9.6 Reflection

As reflection offers the possibility to access the private fields of any class, reflection cannot be handled by our approach. Outside objects can generally use reflection to access the private rep fields of components. It is, however, possible to install a SecurityManager that forbids such accesses. If such a SecurityManager is installed, components are encapsulated by our approach even in the presence of reflection.

4.9.7 Native methods

Native methods cannot be checked by our approach. So a rep object may call a native method that is not autonomous. To handle native methods we have to ensure that native methods with mutable parameters cannot be called:

R_{native^1}	Rep objects must not call native methods with mutable parameters.
R_{native^2}	Rep expressions and references to rep class objects must not be passed to native methods.

Instead of forbidding native methods at all, one can also annotate a native method with an autonomous keyword and assume that the implementation of annotated methods does not access any mutable static fields. Otherwise it is not even possible to use methods like the System.arraycopy method, for example.

4.9.8 Generics

In the upcoming Java 1.5 [53] *generics* are introduced into the language. Generics offer a wide set of new possibilities. The effects of generics to our approach have to be investigated in the future. As confined types have profited from generics [79, 72], at least rep classes will also profit from them. Generics can play an important role in solving the container problem.

4.10 Conclusion

4.10.1 Summary

We have presented a realization of our component model in Java. We used the universes idea that all objects referenced by rep objects are rep objects as well. This has the limitation that rep objects cannot reference boundary objects of a component. We presented autonomous classes to allow static variables in general, but forbid rep objects to access mutable static variables. To allow idioms like iterators we introduced boundary classes. A boundary class shares its representation with its bounded class and is accessible by the environment. We adopted confined types to our approach by defining rep classes. We showed how an object-level protection can be enforced if additional rules are followed. Finally, we discussed the effects of certain Java features to our approach.

While only mentioned at the beginning of this chapter, immutable types are not affected by rep annotations. That is, immutable types are *immune* against rep annotations. References to immutable objects can be passed to rep objects and passed from the component to the environment without any restriction. This increases the expressiveness of our approach, because immutable objects can be shared arbitrarily between components and the environment. Immutable objects are also demanded by our component model of Chapter 3. They are needed to represent immutable nodes. We describe how to achieve immutable types in Java in Chapter 5.

4.10.2 Critique

The main problem of our approach is that rep objects cannot access boundary objects of the component. We partly solved this by introducing rep classes. However, rep objects cannot access rep classes either, so this does not offer a real solution. In addition, rep objects cannot reference any mutable outside objects. This forbids the implementation of container classes with rep objects that store references to mutable outside objects. These restrictions have to be examined in practice to show their limitations. The upcoming generics of Java 1.5 could play an important role to the solution of the problem. We left that examination as future work.

4.10.3 Checking of the Rules

Our rules can be checked in an open-world scope. That is, classes can be added later to the system without being able to break the encapsulation and independency property of checked components. We only demand that packages containing checked component classes are sealed, so that it is not possible to add classes to *these* packages later. All rules can be statically checked, that is, they require no runtime overhead. We do not

require a modification of the Java language, and all annotations can be embedded into comments so that the standard Java compiler can compile annotated classes.

Chapter 5

Immutable Types

Favor Immutability. Joshua Bloch.

An *immutable type* is a type whose instances cannot be modified. Immutable types are widely accepted to be very useful [37, 7, 32, 75, 71, 9, 11, 10]. Other names for immutable types are *value types* [63] and *functional types* [7, 29]. Examples in Java for immutable types are classes in the java.lang package, e.g. String or wrapper classes like Character or Integer. However, these classes are only documented to be immutable, they are not checked by the compiler to really be immutable and are treated by the compiler like any other class.

In this chapter we describe immutable types. We begin with a general section about the advantages and disadvantages of immutable types. After that we define the notion of an immutable object. Then we present rules that ensures the immutability of objects in Java, followed by a section which applies these rules to practical examples. At last we discuss related work to this topic.

5.1 General

5.1.1 Naming Conventions: Immutable vs. Functional vs. Value

We use the term *immutable type* instead of *value type*, because in other contexts [50, 40, 5, 6, 52] value types have a different semantics: value types are types whose objects (*value objects*) are copied by value instead by reference. Value objects, however, are in general not immutable. The term value type is still a good description for immutable types, because immutable objects can be *treated* like values. Another term we could have used is *functional type*. An immutable type is a functional type because no method makes an (visible) imperative change to the object. If a different object is required, a new one is created rather than the object is changed in-place. This is a functional behavior, but most object-oriented programming languages do not support tail-call optimizations, which means that recursive method calls can lead to a stack overflow. Hence, immutable objects cannot always be used in a functional way [29].

5.1.2 Advantages

Bloch [11] names the following advantages of immutable objects:

1. Immutable objects are simple.

2. Immutable objects are inherently thread-safe; they require no synchronization.

3. Immutable objects can be shared freely.

4. Immutable objects can share their internals.

While all points are important in certain application domains, we are mainly interested in the third point: immutable objects can be shared freely. This property is also called *referential transparency* [63].

Referential Transparency

Referential transparency is an important property of immutable objects. It states that if an object is aliased by more than one object, none of the referring objects will ever notice the existence of the other aliases. Immutable objects are referentially transparent, because the visible state of an immutable object cannot change. It does not matter how many aliases to an immutable object exist because, as Noble et al. [63] describe it, "any aliasing is completely invisible to the programmer" . This is also the reason why immutable objects are not required to be encapsulated.

5.1.3 Disadvantages

Bloch also mentions one real disadvantage: immutable classes require a separate object for each distinct value. This restricts the application of immutable classes as there are classes that cannot be implemented in such a manner in a convenient way. For example, an immutable implementation of a class representing a file of a file system would be very unwieldy. Another problem is a possible performance hit for objects that are frequently changed. That is the reason why there is a mutable StringBuffer class besides the immutable String class in the JDK, because adding a single character to a String object always requires the creation of a new String object. Nevertheless, the advantages of immutable classes outweigh their disadvantages in general, and Bloch gives the advice to always implement classes to be immutable, unless there is a very good reason not to do so.

5.1.4 Usage

Biberstein et al. [9] have found 661 immutable classes in the the Java run time library, which are 19% of all concrete classes. This shows that immutable classes are already a concept that is implicitly used. We think the number of immutable classes will further increase once programmers see the advantages of immutable classes and have a tool that automatically checks them.

5.2 Definition of Immutability

Before we mention the constraints to enforce immutability, we first give a definition of immutability. The question is, what is an immutable object? The answer to this question is more difficult than it appears at first sight.

We said above that the crucial point of immutable objects is their referential transparency. That is, any aliasing of immutable objects should be completely invisible to the programmer. We will now give different definitions of immutability and will check them against the property of referential transparency.

Definition 5.1 (Immutable Object – Try 1) *An object x is called* immutable *iff the local state of x never changes after its construction.*

This definition is clearly wrong. The local state is only determined by the values of the instance variables, but not by the state of objects referred to by instance variables. So the definition must also include the state of referring objects:

Definition 5.2 (Immutable Object – Try 2) *An object x is called* immutable *iff the transitive state of x never changes after its construction.*

This definition is better already, but still not correct. This definition is used by Porat et al. [71], for example. In most of the cases, this definition leads to a correct immutable class, but there are cases in which this definition does not work. If a class has a method whose result depends on a mutable static field, instances of such a class cannot be regarded to be immutable. In Listing 5.1 an immutable class Immutable is shown that follows Definition 5.2. It has a method getValue that returns the value of a public static field. Instances of this class are referentially transparent, the local state is immutable, as it is empty, but the result of method getValue is not constant, and thus the class cannot be regarded to be immutable.

```
1 class Global {
2     public int a = 5;
3 }
4 class Immutable {
5     public int getValue() {
6         return Global.a;
7     }
8 }
```

Listing 5.1: Immutable class that follows Definition 5.2, but depends on a static field.

Recall the definition of a constant static field (Definition 4.5): *A* static *field is called* constant *if it is declared* final*, and it is either of a primitive type or of an immutable type.*

Definition 5.3 (Dependency on Mutable Global State) *A method* depends on mutable global state *iff it contains a field access of a non-constant* static *field or calls a* static *method that depends on mutable global state.*

With these definitions we now give a comprehensive definition of an immutable object:

Definition 5.4 (Immutable Object) *An object is called* immutable *iff*

1. *Its transitive state never changes after its construction.*

2. No method depends on mutable global state.

We use this definition of immutable objects in the following.

Definition 5.5 (Immutable Class) *A class is called* immutable *iff all its instances are immutable objects.*

5.3 Immutability Rules

We now present rules that ensure the immutability of classes in Java. We introduce the rules step-by-step to make it easier to follow the reasons for the rules.

5.3.1 Basic Rules

The local state of an object can easily be protected from changes by declaring all fields as final. If the types of all fields are primitive types, or immutable types as well, the transitive state of the object cannot be modified after its construction. To enforce the second condition of immutable objects, the independence of mutable global state, an immutable class must be autonomous as defined in Section 4.3. So we define the following three rules:

I1	All fields must be final.
I2	All fields must be either of primitive types or of immutable types.
I3	All methods and constructors must be autonomous.

Classes that follow the rules of I1-I3 seem to be immutable at the first glance, but they are not. An important point is missing: inheritance. An immutable class can inherit from a mutable class or can be inherited by a mutable class, hence breaking its immutability. To fix this problem, we add two rules:

I4	The superclass must be Object or an immutable class.
I5	All subclasses must be immutable classes as well.

We assume with this definition that Object is an immutable class. As the implementation of Object depends on the JDK implementation, we cannot verify this. However, following the documentation of Object, we can assume that it is immutable if we disregard I5. However, Object is not an immutable class, because this would mean that all classes had to be immutable as Object is the superclass of all classes in Java. Note that I5 cannot be checked in an open-world scope in general. In an open-world scope a class can only be immutable if it is an oligomorphic or monomorphic class (cf. Section 4.5.1).

5.3.2 Mutable Fields

The rules so far restrict the types of fields to be either primitive types or immutable types. This ensures that the transitive state of the immutable object cannot be changed. This constraint is very restrictive, as it is not possible to use any collection class from the JDK, for example. Allowing mutable fields, however, complicates the immutability assurance considerably. The following two points have to be taken into account:

1. After the construction of an immutable object, no modifying method may be called on any mutable object that is referenced by the immutable object.
2. Aliases to mutable objects that are transitively referenced by the immutable object must not be obtained by objects outside the immutable object.

The first condition requires a notion of a method that does not modify an object. The second condition can be solved by claiming that all mutable fields must be of rep types as defined in Section 4.2.1. First, we define the notion of a *read-only* method.

Definition 5.6 (Read-Only Method) *A method is called* read-only *iff all of the following conditions are satisfied:*

1. *The method does not contain any field updates.*
2. *The method does not call methods that are not read-only.*
3. *Each method that overrides a read-only method must be read-only as well.*

Otherwise the method is called read-write*.*

Definition 5.7 (Object Modification) *An object is* modified *iff one of the following conditions is satisfied:*

1. *It is the target of a field update.*
2. *A read-write method is invoked on the object.*

We can now extend our set of rules. First, we allow fields to be of mutable types, but require that these types are rep types. So we change rule I2 accordingly. The star (*) indicates that this rule is not new but modified:

I2*	All fields of mutable types must be private and of rep types.

This assures that mutable objects referred to by fields of immutable classes are not accessible by other classes. Note that an immutable class has to follow all rules that are defined in Chapter 4 regarding rep expressions. In addition, we do not allow boundary classes or rep classes of immutable classes and demand that rep types may not appear in package-private fields or methods. We shortly summarize the rules for rep expressions:

- A rep expression is only assignment compatible with other rep expressions.
- Fields, methods and constructors with rep types in their declarations have to be private.
- Arguments of method invocations on rep expressions or arguments of rep constructors must be rep expressions as well.
- Rep types must be autonomous classes.

To disallow modifications of mutable rep objects we give the following rule:

I6	No method may modify objects referred to by rep expressions.

These constraints enforce the immutability of Java classes. However, there is a loophole that still offers the possibility of modifying an immutable object after its construction. The problem lies in the constructors. In order to be practical one cannot forbid read-write method calls on rep expressions within constructors. This offers, however, also the possibility that the constructor modifies rep objects of other immutable objects of the same class. See Listing 5.2 for such a code example. To patch this hole, we add the following constraint to the set of our rules:

I7	Constructors may neither contain rep parameters nor parameters with the same type as its declaring class.

Provided that a class follows the rules I1-I7, it is assured that objects of that class are immutable (without justification). That is, the transitive state of instances of such classes never changes after their construction, and no method depends on global mutable state.

```
1 class Immutable {
2     private rep int[] values;
3     public Immutable(Immutable copy) {
4         copy.values[0]=5;
5     }
6 }
```

Listing 5.2: Constructor loophole.

5.3.3 Making the Definition More Practical

The rules that we have presented so far guarantee that an object is immutable. That is, the state of instances of a class that follows the rules I1-I7 cannot be changed after their construction. However, the rules are too conservative for a lot of practical examples. Listing 5.3 shows an implementation of an immutable StringList class. It uses an internal String array to store its entries and contains the two methods add and removeLast. Both methods do not make an imperative change to the object itself, but return a new instance of the StringList class. This implementation is immutable, but it does not follow our rules. We can identify three violations. In line 8 the constructor has a parameter with a rep type which is forbidden by rule I7, and in line 18 and 20 there are field updates on a rep expression which is forbidden by rule I6. Both violations are actually unproblematic, because the constructor does not modify the parameter array, and the field updates are done on newly created arrays. So rep parameters can be

```
1  public immutable StringList {
2      private final rep String[] values;
3      private final int size;
4      public StringList() {
5          values = new rep String[0];
6          size = 0;
7      }
8      private StringList(rep String[] v, int s) { // (I7)
9          values = v;
10         size = s;
11     }
12     public String get(int i) {
13         return values[i];
14     }
15     public StringList add(String s) {
16         rep String[] newValues = new rep String[size+1];
17         for (int i=0;i<size;i++) {
18             newValues[i] = values[i]; // (I6)
19         }
20         newValues[size] = s; // (I6)
21         return new StringList(newValues,size+1);
22     }
23     public StringList removeLast() {
24         return new StringList(values,size-1);
25     }
26 }
```

Listing 5.3: An implementation of an immutable StringList class.

allowed as long as these parameters are not modified, and rep objects can be modified if these objects are newly created objects and are not referred to by any field.

Before defining new rules, we give some definitions.

Definition 5.8 (Self Expression) *An expression is called* self expression *iff it is one of the following:*

1. *The implicit* this *parameter.*
2. *A field access on a self expression.*
3. *A method invocation on a self expression.*

We call a self expression of a rep type a self rep expression*, and we call objects referred to by self (rep) expressions* self (rep) objects.

Definition 5.9 (Local Expression) *An expression is called* local expression *iff it is one of the following:*

1. *A local variable.*
2. *A* new *expression.*
3. *A field access on a local expression.*
4. *A method invocation on a local expression.*

Otherwise it is called non-local *expression. We call a local expression of a rep type a* local rep expression*, and we call objects referred to by local (rep) expressions* local (rep) objects*.*

Definition 5.10 (Parameter Expression) *An expression is called* parameter expression *iff it is one of the following:*

1. *A formal parameter.*
2. *A field access on a parameter expression.*
3. *A method invocation on a parameter expression.*

We call a parameter expression of a rep type a parameter rep expression*, and we call objects referred to by parameter (rep) expressions* parameter (rep) objects*.*

We now define an additional rule I8, revise rule I6, and replace the previous rule I7 by a new definition:

I6*	Rep objects can only be modified if they are local.
I7*	Parameter rep expressions and self rep expressions must not be assigned to local rep expressions, passed as arguments to method invocations on local rep expressions or passed as arguments to constructors.
I8	If a local rep expression had been assigned to a self rep expression or had been passed to a constructor of an immutable class, there must not be any modifications to local rep objects afterwards.

Rule I6 has been considerably weakened. Where the previous definition forbids any modifications on rep objects within methods, rep objects can now be modified without any restriction as long as they are referred to by local rep expressions. That is, besides within constructors, rep objects can also be modified within ordinary methods. This offers a wide range of possibilities that have not been possible with the previous definition. However, I7 and I8 prevent that modifications on rep objects can affect the state of an immutable object. I7 prevents that neither a self rep expression nor a parameter rep expression can become a local rep expression. I8 ensures that if a local rep expression has become a parameter or self rep expression, the local alias cannot be used anymore. Note that we have removed the restriction that parameters of constructors cannot be of rep types or the type of the declared class. The constructor loophole described above, however, is still patched as parameter rep objects cannot be modified at all. Note that primitive types and immutable types are never rep expressions, so especially rule I7 do not apply to expressions with such types. The StringList example of Listing 5.3 is following these new rules.

5.3.4 Lazy Initialization

The rules so far demand that all fields have to be final. This constraint prevents an important design pattern for immutable types: *lazy initialization* [30]. Lazy initialization means that values are only calculated when they are really requested, and once calculated they are cached in a field. The pattern always consists of a field, the *lazy field*, and a corresponding method, the *lazy method*. The lazy field is initialized with a constant value when the object is constructed. Fields of reference types are always initialized with null. The lazy method checks if the lazy field has its initial value. If this is the case, the method performs its calculation, assigns the result to the lazy field and returns its value. Otherwise it immediately returns the value of the lazy field. The lazy method is the only method that directly accesses the lazy field. All other methods are only accessing the lazy method. If a method would directly access the lazy field without using the lazy method, it was possible that it reads the initial value of the field. The calculation of the lazy method may only depend on immutable values, because otherwise the result of the method would not always be the same and could not be cached in the lazy field. This implies that a lazy method can never have parameters. A method that is typically implemented as a lazy method is the hashCode method (see Listing 5.4).

Lazy initialization is in particular important for immutable types. As the transitive state of an immutable object does not change after its construction, all methods that have to perform calculations can be implemented as lazy methods. Lazy initialization, however, is only useful if the price of the calculation outweighs the costs of an additional field.

Definition 5.11 (Lazy Field) *A field f is called* lazy *iff the following conditions are satisfied:*

1. *f is declared as* private.
2. *All initializations of f must be with the same constant value v. If f is of a reference type, v has to be* null.
3. *Exactly one method directly accesses f, this method has to be a lazy method.*

```
1  class HashExample {
2      private int hash = 0;
3      ...
4      public int hashCode() {
5          if (hash == 0) {
6              hash = ...;
7          }
8          return hash;
9      }
10 }
```

Listing 5.4: The lazy initialization pattern by means of the hashCode method.

Definition 5.12 (Lazy Method) *Let f be a lazy field. A method is called* lazy *iff all of the following conditions are satisfied:*

1. *It has no parameters.*
2. *It contains exactly one field update. This field update is performed on f.*
3. *The field update can only be reached if f is equal to its initialized value v.*
4. *It returns the value of f.*

We now change rule I1 to allow lazy fields:

I1*	All fields must be final *or lazy.*

So all fields that are not final have to be lazy fields. That means that exactly one lazy method has to exist for each lazy field of the class. Note that a lazy field has to be private, per definition.

5.4 Applying Theory to Practice

You may now say: "Hey, defining conservative rules is easy, but are they not too limiting for the practice?". This is a legitimate question. To show that our rules can be used in practice, we applied them to real-life examples. We chose two classes which are widely used and which are documented to be immutable: java.lang.Boolean and java.lang.String. We begin with the simpler java.lang.Boolean. For convenience we have reprinted the complete list of our rules in Table 5.1.

I1	All fields must be final or lazy.
I2	All fields of mutable types must be private and of rep types.
I3	All methods and constructors must be autonomous.
I4	The super class must be Object or another immutable class.
I5	All subclasses must be immutable classes as well.
I6	Rep objects can only be modified if they are local.
I7	Parameter rep expressions and self rep expressions cannot be assigned to local rep expressions, passed as arguments to method invocations on local rep expressions or passed as arguments to constructors.
I8	If a local rep expression had been assigned to a self rep expression or had been passed to a constructor of an immutable class, there must not be any modifications to local rep objects afterwards.

Table 5.1: The complete list of immutability rules.

5.4.1 java.lang.Boolean

Listing 5.5 shows the source code of the Boolean class. We simplified the implementation by omitting the parts which are not of interest for us. It is obvious at first sight that Boolean does not conform to our rules. The value field declaration in line 4 is not final which violates rule R1. However, this is not a fault of our rules, but a fault of the implementation, because there is no reason why the field should not be final. The only assignment appears in the constructor, and thus the field can be final without any drawback. All other rules are satisfied. In particular rule I5 is satisfied, because the class is declared as final, hence there are no subclasses at all. So our definition can show the immutability of the java.lang.Boolean class, if the suggested modification is applied.

```
1  public final class Boolean {
2     public static final Boolean TRUE = new Boolean(true);
3     public static final Boolean FALSE= new Boolean(false);
4     private boolean value;
5     public Boolean(boolean v) { value = v; }
6     public boolean booleanValue() {
7        return value;
8     }
9     public String toString() {
10       return value ? "true" : "false";
11    }
12    public int hashCode() {
13       return value ? 1231 : 1237;
14    }
15 }
```

Listing 5.5: The java.lang.Boolean class.

5.4.2 java.lang.String

Listing 5.6 shows a simplified implementation of java.lang.String. It shows those parts of the implementation that are crucial for the analysis while the uninteresting parts are omitted. In addition, we already annotated the code with the rep keyword where needed. The String class is an excellent example of a practical implementation of an immutable class. It shows typical problems that occur in practice. To our knowledge, there is no immutability definition that could prove the immutability of that class. And to anticipate it, we cannot either. However, we come very far with our approach, and it should be practical enough for most implementations of immutable classes.

We go through the source code step-by-step to examine it. We start with the field declarations:

```
2     private rep char value[];
3     private int count;
4     private int offset;
5     private int hash = 0;
```

Listing 5.7: Field declarations of the String class.

As can be seen again, no field is final, and thus rule R1 is violated again. Except for the hash field, the fields are only assigned to inside constructors, thus they can be final. The hash field is a lazy field as described above, so it cannot be final. We continue the analysis, by examining the first constructor:

```
6     public String() {
7         value = new rep char[0];
8     }
```

Listing 5.8: String default constructor.

Obviously the default constructor conforms to our rules.

```
9     public String(char value[]) {
10        this.count = value.length;
11        this.value = new rep char[count];
12        System.arraycopy(value, 0, this.value, 0, count);
13    }
```

Listing 5.9: String constructor with char array as parameter.

This is a public constructor that takes a char array as parameter. The purpose of this constructor is to create a String instance from the content of the char array.

```
1  public final class String {
2     private rep char value[];
3     private int count;
4     private int offset;
5     private int hash = 0;
6     public String() {
7        value = new rep char[0];
8     }
9     public String(char value[]) {
10       this.count = value.length;
11       this.value = new rep char[count];
12       System.arraycopy(value, 0, this.value, 0, count);
13    }
14    public String(String original) {
15       this.count = original.count;
16       if (original.value.length > this.count) {
17           this.value = new rep char[this.count];
18           System.arraycopy(original.value,
19              original.offset,this.value,0,this.count);
20       } else {
21           this.value = original.value;
22       }
23       this.value = original.value;
24    }
25    public String (StringBuffer buffer) {
26       buffer.setShared();
27       this.value = buffer.getValue();
28       this.count = buffer.length();
29    }
30    public char charAt(int index) {
31       return value[index];
32    }
33    public int hashCode() {
34       int h = hash;
35       if (h == 0) {
36          int off = 0;
37          int len = count;
38          for (int i = 0; i < len; i++) {
39              h = 31*h + value[off++];
40          }
41          hash = h;
42       }
43       return hash;
44    }
45 }
```

Listing 5.6: Simplified implementation of java.lang.String

As the passed array may be aliased elsewhere, it is not possible to assign it directly to the value field. Instead a new array is created, and the content is copied by the System.arraycopy method. This is a problem for our rules, because it is not possible to pass a rep expression to a public static method. In addition, the System.arraycopy method is a native method. The way of solving this problem is by making an exception and allowing rep expressions to be passed to the System.arraycopy method. So we have to assume that the System.arraycopy method does not create aliases of the argument arrays. With this exception the constructor follows our rules. So we can go ahead and look at the next constructor:

```
14    public String(String original) {
15        this.count = original.count;
16        if (original.value.length > this.count) {
17            this.value = new rep char[this.count];
18            System.arraycopy(original.value,
19                original.offset,this.value,0,this.count);
20        } else {
21            this.value = original.value;
22        }
23    }
```

Listing 5.10: String copy constructor.

This is a copy constructor. It takes a String instance as parameter and copies the contents of that instance. If the size array of the original String is bigger than the actual number of characters, a new fitting array is created and the characters are copied, otherwise only the reference to the char array is copied. The constructor contains three assignments to fields. All assignments are possible, because all expressions are rep expressions, and it is not forbidden by the rules to assign a parameter rep expression to a self rep expression inside a constructor. However, the method System.arraycopy gives us a problem again. As parameter rep expressions may not be modified, it must be assured that the System.arraycopy method does not modify its first array parameter. Again, we have to make an exception to allow the usage of the System.arraycopy method. A general solution would be to annotate method parameters with a read-only mode and allow the handing over of parameter rep expressions as read-only arguments to method invocations. Except for the method call the constructor conforms to our rules. Now let us examine the most problematic piece of code:

```
24    public String (StringBuffer buffer) {
25        buffer.setShared();
26        this.value = buffer.getValue();
27        this.count = buffer.length();
28    }
```

Listing 5.11: String constructor with a StringBuffer parameter.

This constructor takes a StringBuffer instance and assigns the result of the getValue method of the StringBuffer to its array field. It is clear that this does not apply to our rules, because the getValue method returns a ground expression which cannot be assigned to a rep expression. To allow this, it must be ensured that the result of the getValue method is not aliased at all. So one may imagine that the result of the getValue method can be declared to be *unique* or *free* as described in Section 2.5.2, and that unique expressions can be assigned to rep expressions. We discuss this possibility in Chapter 6, but in this case even that is not possible. It may be noticed by the reader that before calling the getValue method, the setShared method of the StringBuffer is called. Let us look at the implementation and documentation of the getValue and setShared methods of StringBuffer:

```
// The following two methods are needed
// by String to efficiently convert a
// StringBuffer into a String.
// They are not public.
// They shouldn't be called by anyone but String.
final void setShared() { shared = true; }
final char[] getValue() { return value; }
```

Listing 5.12: StringBuffer methods.

So the getValue method in fact returns the *internal* array of the StringBuffer. And to prevent that the internal array is modified by the StringBuffer afterwards, the setShared method has to be called before. It is clear that this implementation cannot be handled by our approach, and we did not further invest any effort to solve this problem. We now go back to the remainder of the code:

```
29    public char charAt(int index) {
30        return value[index];
31    }
```

Listing 5.13: String's charAt method.

This method conforms to our rules, so we go to the last method:

```
32    public int hashCode() {
33        int h = hash;
34        if (h == 0) {
35            int off = 0;
36            int len = count;
37            for (int i = 0; i < len; i++) {
38                h = 31*h + value[off++];
```

```
39              }
40              hash = h;
41          }
42          return hash;
43      }
```

Listing 5.14: String's hashCode method.

This is the lazy method that belongs to the lazy field hash. Instead of working with the hash field directly, the String developers are using a local variable. This is probably done to make the method thread-safe without using a synchronized block. However, the method still meets our requirements for lazy methods. It has only one field update, and that field is the lazy hash field. The field update can only be reached if the value of the field is equal to the initial value. The method has no parameters and returns the value of the lazy field.

A last point has to be mentioned. The String class is declared to be final and thus it cannot have any subclasses, which meets rule I5.

5.4.3 Summary

To summarize, we have come very far with our immutability definition, and except one method the code of the highly optimized String implementation follows our rules. In order to achieve this, we had to make an exception to the rules to allow the usage of the System.arraycopy method. As this is a somewhat ad-hoc solution, one may think of a more general solution for such situations. We discuss this in Chapter 6. Nevertheless, we showed that our definition is not too conservative and may effectively be useful in practice.

5.5 Related Work

This section describes and discusses previous work on immutable types.

5.5.1 Immutability based on a Read-Only Mode

Birka and Ernst [10] define immutable types by means of a read-only mode. Immutable types by their definition are classes whose fields are all read-only and final. This seems to be a valid definition at the first glance, but it is not. With this definition it is possible to change the state of immutable types after their construction. This can happen because mutable objects referred to by the read-only fields can be aliased by read-write variables. The alias can appear by passing a mutable object as parameter to the constructor of the immutable type.

In Listing 5.15 a code example is given. The class Immutable contains one int array that is declared final and readonly. The constructor takes a read-only int array as parameter and assigns it to the data field. The Violator initializes the immutable class with an aliased array and can change the state of the immutable object by using the mutable alias to the data array.

```
1  public class Immutable {
2     final readonly int[] data;
3     public Immutable(readonly int[] d) {
4        data = d;
5     }
6     public int get(int i) { return data[i]; }
7  }
8  public class Violator {
9     public Violator() {
10       int data[] = new int[5];
11       data[0] = 1;
12       Immutable imm = new Immutable(data);
13       int x = imm.get(0); // x = 1
14       data[0] = 2;
15       x = imm.get(0); // x = 2
16    }
17 }
```

Listing 5.15: An immutable class with a final readonly field.

5.5.2 Immutable Types in JAC

Theisen [75] defines immutable types as follows:

> "An Immutable class is a pure readonly class where each constructor parameter must be a simple value type, itself immutable or readimmutable. It is declared by annotating a class with an immutable keyword. Subclasses of immutable classes must also be immutable."

where a pure readonly class is a class with all methods being readonly, and a readonly method is defined as:

> "Methods declared readonly may only change the state of objects explicitly passed in as a non-readonly parameter. A readonly method is declared by prepending its parameter list with the keyword readonly."

However, this definition misses an important point: it does not mention static variables. That is, the constructor of an immutable type can assign a static readonly field to an internal field. Listing 5.16 shows a code example. Following the definition of JAC the class Immutable is an immutable class, as Immutable is a pure readonly class. However, the Violator class can modify the internal Vector of Immutable as it is aliased by a static variable.

```
1  immutable class Immutable {
2      private Vector v;
3      public Immutable() {
4          v = Violator.v;
5      };
6  }
7  class Violator {
8      public static readonly Vector v;
9      public Violator {
10         v = new Vector();
11         Immutable imm = new Immutable();
12         // modifying internal Vector via an alias
13         v.add(new Integer(0));
14     }
15 }
```

Listing 5.16: An immutable class defined by JAC.

5.5.3 Immutable Types with Immutable Fields

Porat et al. [71] focuses on detecting the mutability of static variables in order to avoid isolation problems. They define a class as immutable if all its fields are immutable. A field is immutable if it is not modified after its initialization point. Immutability is tested with an open-world analysis. That is, the code is divided into code that lies within the analysis scope and code that is outside the analysis scope. A field is marked as mutable if one of the four conditions is true:

1. The value of the field
   a) Is modified within the analysis scope.
   b) May be modified from outside the analysis scope.
2. The object referenced by the field
   a) Is modified within the analysis scope.
   b) May be modified from outside the analysis scope.

This definition allows, for example, to have package-private fields to be immutable, as long as no class within the package modifies that field. However, immutable classes by their definition can contain methods whose results depend on mutable static fields, and thus the visible state of immutable objects is not immutable in general. With their definition it is not possible to define immutable classes that have fields of mutable types.

5.5.4 Immutable Types in Effective Java

Bloch [11] defines five rules to make an object immutable:

1. Don't provide any methods that modify the object.
2. Ensure that no methods may be overridden.
3. Make all fields final.
4. Make all fields private.
5. Ensure exclusive access to any mutable components.

These rules guarantee that an object is immutable, however, Bloch does not describe *how* the exclusive access to mutable components should be ensured. In addition, immutable types can contain methods that depend on mutable static fields.

Chapter 6

Extensions

In this chapter we discuss possible extensions to our approach. First, we summarize the problems that could not be solved with our approach so far. Then we present solutions to the mentioned problems and discuss the consequences of their integration into our approach.

6.1 Problems So Far

Our approach so far has some limitations. Among others these limitations are described in the following.

1. A rep object cannot reference any mutable external object. According to our independency definition on page 28, representation objects of independent components can reference mutable external objects, as long as they do not invoke any methods on them. We achieved independence by forbidding even the referencing of external objects. Thus, it is not possible to implement containers that store references to mutable external objects in rep objects.

2. A rep object cannot be created by the environment. That is, all rep objects must be created by the component and all classes of rep objects must be known during the creation time of the component. This limits the reusability of components as it is not possible to initialize a component with mutable objects whose classes are not known by the component.

3. The environment cannot obtain any reference to rep objects, where it would be sufficient to forbid modifying references. That is, it should be allowed to pass rep object references to the environment, provided that it is not possible to modify the rep objects via these references.

4. Rep expressions cannot be passed to static methods in conjunction with mutable ground expressions.

We now give extensions to our approach that solve a part of these problems. These extensions, however, cannot be checked in an open-world scope without restrictions.

6.2 Borrowed Expressions

Borrowed expressions have often been presented in conjunction with unique expressions (cf. Section 2.5.2). However, borrowed expressions may exist independently. Hogg et al. [38] mentions an *uncaptured* qualifier that can be compared to a borrowed

expression. We now define the notion of a *borrowed type* and a *borrowed expression* similar to the definitions of a rep type and rep expression in Chapter 4:

Definition 6.1 (Borrowed Type) *A mutable type is called* borrowed *iff it is annotated with the* borrowed *keyword.*

Definition 6.2 (Borrowed Expression) *An expression is called* borrowed *iff it is one of the following:*

1. *A local variable whose declared type is a borrowed type.*
2. *A formal parameter whose declared type is a borrowed type.*
3. *A method invocation or field access on a borrowed expression.*

Note that it is neither possible to create a borrowed expression with a new expression nor to declare a field with a borrowed type, nor is it possible to declare the result type of a method to be a borrowed type. We now give rules that apply to borrowed expressions. Note that an assignment does not only mean an assignment statement, but also the hidden assignment of an argument of a method invocation to the formal parameter of that method. The rules for borrowed expressions are shown in Table 6.1.

B1	A borrowed expression can only be assigned to other borrowed expressions.
B2	Only autonomous methods can be invoked on borrowed expressions.
B3	Arguments to method invocations on borrowed expressions have to be either of primitive types or of immutable types.
B4	Fields on borrowed expressions can only be updated if the type of the field is either a primitive type or an immutable type.
B5	Methods that override methods with borrowed types in their declaration cannot remove the borrowed declaration.

Table 6.1: Rules for borrowed expressions.

So if a reference to an object o is passed as argument to a method whose corresponding formal parameter is of a borrowed type, it is guaranteed that the method does neither create a static alias to o nor to any mutable object that is transitively referenced by o. In addition, all aliases made by the method are gone after the method has been left. So a method can give clients the assurance that it does not create any static alias of certain formal parameters. Note that rule B5 cannot be checked in an open-world scope. In an open-world scope only borrowed declarations in methods of monomorphic or oligomorphic types (cf. Section 4.5.1) or in static methods can be considered.

Influences on Rep Expressions

Borrowed references can be used in conjunction with rep expression to give rep expressions more flexibility. We can change the definition of the first rule of rep expressions to allow the assignment to borrowed expressions:

R1*	Rep expressions are only assignment compatible with other rep expressions, *except that rep expressions can be assigned to borrowed expressions.*

This allows, for example, that rep expressions can be passed as arguments in conjunction with mutable ground expressions to public static methods as long as the corresponding formal parameters are borrowed expressions. Hence, it could be a general solution to the System.arraycopy problem that appears in Section 5.4.2 to show the immutability of the String class. The System.arraycopy method could to declared like shown in Listing 6.1. However, System.arraycopy cannot be implemented to follow the borrowed restrictions. The method copies the contents from the first array to the second array. If the component type is a mutable type, however, it has to assign a value to a mutable field of a borrowed expression which is forbidden by rule B4. So this works only if the component types of the arrays are primitive types or immutable types.

```
1    public static native void arraycopy(
2        borrowed Object src, int srcPos,
3        borrowed Object dest, int destPos, int length);
```

Listing 6.1: System.arraycopy with borrowed parameters.

6.3 Fresh Expressions

As already mentioned in Section 3.4.3, the initialization problem can only be solved with some notion of *unique* or *free*. As unique requires a change of the underlying language, we adopt the free mode. We call it *fresh* and define it as follows:

Definition 6.3 (Fresh Type) *A mutable type is called* fresh *iff it is annotated with the* fresh *keyword.*

Definition 6.4 (Fresh Expression) *An expression is called* fresh *iff it is one of the following:*

1. *A local variable whose declared type is a fresh type.* (fresh variable)
2. *A formal parameter whose declared type is a fresh type.* (fresh parameter)
3. *A* new *expression, where all actual parameters are fresh expressions.*
4. *A method invocation of a method whose declared result type is a fresh type.*

We call objects that are referred to by fresh expressions fresh objects.

F1	Only fresh expressions can be assigned to fresh expressions.
F2	Only autonomous methods can be invoked on fresh expressions.
F3	Mutable arguments of method invocations on fresh expressions have to be fresh expressions as well.
F3	Only fresh expressions, primitive values or references to immutable objects may be assigned to fields of fresh expressions.
F4	When a fresh expression was assigned to any expression or passed as argument to any method invocation, it must not be used afterwards anymore, except it was passed to a method as a borrowed argument.
F5	Methods that override methods with a fresh type in its declaration cannot remove the fresh declaration.
F6	It is not allowed to read mutable fields or use mutable results of method invocations on fresh expressions.

Table 6.2: Rules for fresh expressions.

Note that the declared type of fields cannot be a fresh type.

The rules for fresh expressions are shown in Table 6.2. Note that F1 does not forbid the assignment of fresh expressions to normal expressions, it only forbids the opposite of assigning a normal expression to a fresh expression. F4 ensures that if a fresh expression was assigned to *any* expression, which includes other fresh expressions, the fresh expression cannot be used anymore. This is a similar restriction like that of *alias burying* [15], where all aliases to a unique variable have to be dead, that is, never be used again when the unique variable is read. F2 ensures that a method of a fresh expression does not assign this to a static variable. F3 and F4 ensure that fresh objects cannot get references to non-fresh mutable objects. F6 ensures that mutable objects transitively references by fresh expressions can be aliased.

With fresh expressions it is now possible to solve the initialization problem. However, without built-in language support the rules for fresh expressions can only be checked in a closed-world scope. In an open-world scope one cannot guarantee that the caller of a method with a fresh formal parameter really does not use its passed argument after the method invocation anymore. It is also not possible to be sure that a passed in argument was really a fresh expression without checking the code of the method caller.

However, in certain cases the fresh mode can be checked in an open-world scope. If a static method, for example, specifies its result as fresh, the result of such a method can be assured to be fresh in an open-world scope, as a static method cannot be overridden[1]. Another example would be the call of methods on objects of monomorphic or oligomorphic types.

The fresh mode is similar to the *free* mode [37], *virginity* [47] and *alias burying* [15]. The fresh mode can be used to solve the initialization problem (see Section 3.4.3) and to specify factory methods in a way that allows to guarantee unaliased result values. In addition it is useful for initializing immutable objects with mutable objects.

[1] In fact, a static method in Java can be redefined by subtypes. However, if the static method is accessed by the name of its defining class, it is assured that the desired method is invoked.

Influences on Rep Expressions

Like borrowed expressions, fresh expressions can be a useful extension to rep expressions. We can change rule R1 in the following way:

R1*	Rep expressions are only assignment compatible with other rep expressions, except that rep expressions can be assigned to borrowed expressions, *and that fresh expressions can be assigned to rep expressions.*

That is, fresh expressions can be assigned to rep expressions. This makes sense, because if an object is not aliased at all, it can become a rep object. This extension allows that result values of public static methods, for example, can be assigned to rep expressions, provided that the result is fresh. In a closed-world scope this extension even allows that rep objects of components can be created by objects of the environment.

6.4 Read-Only Expressions

A read-only mode has been discussed in a lot of papers before (cf. Section 2.5.6), and we do not present anything new here. However, read-only is not always defined in the same way, so we give a definition:

Definition 6.5 (Read-Only Type) *A type is called* read-only *iff it is annotated with the keyword* readonly.

Definition 6.6 (Read-Only Expression) *An expression is called* read-only *iff it is one of the following:*

1. *A local variable whose declared type is read-only.*
2. *A formal parameter whose declared type is read-only.*
3. *The result of a method whose result type is read-only.*
4. *A field whose declared type is read-only.*
5. *A method invocation or field access on a read-only expression.*

We call objects that are referred to by read-only expressions read-only objects.

RO1	Read-only expressions can only be assigned to other read-only expressions.
RO2	Only self-read-only methods can be invoked on read-only expressions.
RO3	Field updates on read-only expressions are forbidden.
RO4	Methods that override methods with a read-only type in their declaration cannot remove the read-only declaration.

Table 6.3: Rules for read-only expressions.

Table 6.3 shows the rules for read-only expressions. The notion of a self-read-only method is defined below. RO1 prevents that read-only expressions can become normal expressions, but it allows that any expression can become a read-only expression. RO2 and RO3 prevent any modifications of read-only objects. RO4 prevents subtypes from ignoring the read-only declarations of supertypes. We now define the notion of a read-only method.

We already defined a read-only method on page 71. That definition is rather restrictive as it even forbids field updates on objects referred to by formal parameters and local variables. In addition, methods that follow that definition are not side-effect free according to Hogg [37]. It is possible, for example, that such a read-only method can return a mutable field of the receiver object that enables the caller of the method to modify the object afterwards. That had not been a problem so far, because read-only methods have only played a role in conjunction with immutable objects, so far.

With the aid of a read-only mode it is possible to only protect the receiver object of a method. So we call such methods self-read-only methods, as formal parameters, local variables and static variables are not protected at all. This definition is similar to that of Birka and Ernst [10] and Kniesel and Theisen [44]:

Definition 6.7 (Self-Read-Only Method) *A method is called* self-read-only *iff it is annotated with the* selfreadonly *keyword.*

- *The implicit receiver parameter* this *of a self-read-only method is per definition a read-only expression.*
- *Self-read-only methods can only be overridden by methods that are self-read-only as well.*

That is, a self-read-only method may modify its parameter objects without any restriction and can also modify any global state, but a self-read-only method cannot change its receiver object. Note that if a self-read-only method returns a mutable field of its receiver object, the result is a read-only expression. Thus, it is not possible to obtain a read-write reference from a self-read-only method to mutable objects that belong to the transitive state of its receiver object.

Influences on Rep Expressions

Like the expression kinds before, a read-only expression can be a useful extension to rep expressions. Like for the other expressions we can extend the first rule R1 of rep expressions:

R1*	Rep expressions are only assignment compatible with other rep expressions, except that rep expressions can be assigned to borrowed expressions *and read-only expressions*, and that fresh expressions can be assigned to rep expressions.

That is, a rep expression can be assigned to a read-only expression. This means that rep objects can be accessed by objects of the environment, but it is assured that these accesses cannot change the state of the rep object. This is identical to the universe approach [60]. However, this can only be checked in a closed-world scope. In an

open-world scope objects of the environment can ignore the readonly declaration and can call methods that are not self-read-only. As with borrowed and fresh expressions, read-only expressions can be checked in an open-world scope in certain cases. A static method, for example, could specify its parameters as read-only to guarantee to the client that it does not modify the parameter object. If, in addition, the static method is autonomous, the method can be checked to conform to the read-only rules. Hence, an immutable object can pass a rep object to such a method and can be sure that it is not modified by the method. The first parameter of the System.arraycopy method, for example, could be specified to be read-only. In addition, the second array parameter could be borrowed like above (see Listing 6.2). Now, the constructor of an immutable object can copy the contents of another immutable object's array into its own array. However, as already said above, the implementation of System.arraycopy can only follow the rules if the component type of the array is an immutable type.

```
1    public static native void arraycopy(
2        readonly Object src, int srcPos,
3        borrowed Object dest, int destPos, int length);
```

Listing 6.2: System.arraycopy with read-only parameters.

6.5 Conclusion

In this chapter we have presented ideas to solve some limitations of our approach. Except for the container problem we could present solutions to the mentioned problems. However, these solutions cannot be checked generally in an open-world scope. So without language support they can only be used in a closed-world scope. We left the examination of the container problem to future work. It might be solvable with a parameterized type system like the *generics* of the upcoming Java 1.5. Another possibility is perhaps to introduce *external expressions* which cannot be used to invoke *mutable methods,* similar to the *arg* mode of Flexible Alias Protection (cf. Section 2.5.8.2).

Chapter 7

Jacpock$_{proto}$

We have implemented a Java checker to verify that annotated Java source code conforms to our rules. Due to the limited time of this thesis, we only implemented a prototype rather than a productive tool. It is left to future work to create a productive checking tool for our approach. We call the tool JACPOCK$_{proto}$ for Java Component Checker.

7.1 General

JACPOCK$_{proto}$ is a command line tool written in Java. It takes a set of annotated Java files as input, checks the annotations against the source code and prints the results to the standard output. The implementation is only a proof-of-concept implementation, without any optimization in mind. So the tool might be slow or run out of memory for a large set of classes.

7.2 System Design

JACPOCK$_{proto}$ consists of four components: the Main component that coordinates the actions, the Front-End to read annotated Java files, the Class Repository that stores all collectable information about classes and the actual Checker component. The system design can be seen in Figure 7.1. Besides the four components, there is a set of immutable classes that represent the abstract syntax tree (AST) of parsed Java files.

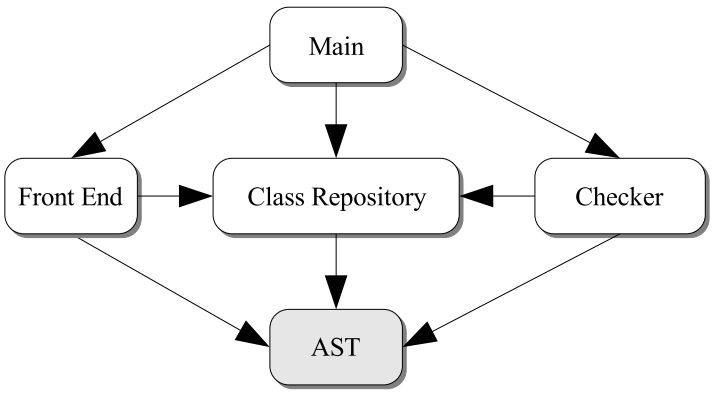

Figure 7.1: The system design of JACPOCK$_{proto}$.

7.2.1 The Front-End

The front-end consists of a scanner, a parser and the classes for the abstract syntax tree, which are also used by other parts of the program. The Java classes of the front-end are all generated by different tools. The scanner is generated by JFlex [43] from the scanner description file jacpock.flex, which is derived from the Java 1.4 scanner description file of the JFlex distribution. The parser is generated by CUP [27] from the parser description file jacpock.cup, derived from the Java 1.4 parser description file of the JFlex distribution. The classes representing the AST are generated by Katja [73] from the AST description file jacpock.katja. The front-end design is shown in Figure 7.2.

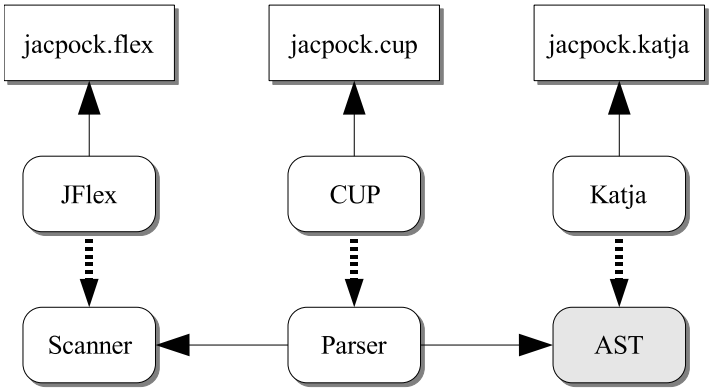

Figure 7.2: The front-end of JACPOCK$_{proto}$.

7.2.2 The Class Repository

The Class Repository stores informations about classes. It consists of a hash table that stores for every full qualified classname the complete AST of that class. The class repository is filled at start-up with the classes of the predefined packages java.util and java.lang. The user can specify additional directories that should be included in the class repository.

7.2.3 The Checker

The checker component consists of a set of different checkers (see Figure 7.3). For every checkable entity there is one checker:

- Rep Checker – Checks the rep rules R1-R5 (see Section 4.2 and 4.3). We have not implemented the checking of the remaining rep rules yet.
- Autonomous Checker – Checks autonomous declared methods and classes to conform to the autonomous definition (see Section 4.3).
- Read-Only Checker – Checks whether read-only declared methods apply to the read-only definition (see Definition 5.6 on page 71).

- Immutability Checker – Checks the immutability rules I1-I8 (see Table 5.1 on page 77). The checking of lazy methods and lazy fields has not been implemented yet. The Immutability checker uses the Rep Checker to verify the rep rules and the Autonomous Checker to ensure that immutable classes are autonomous.

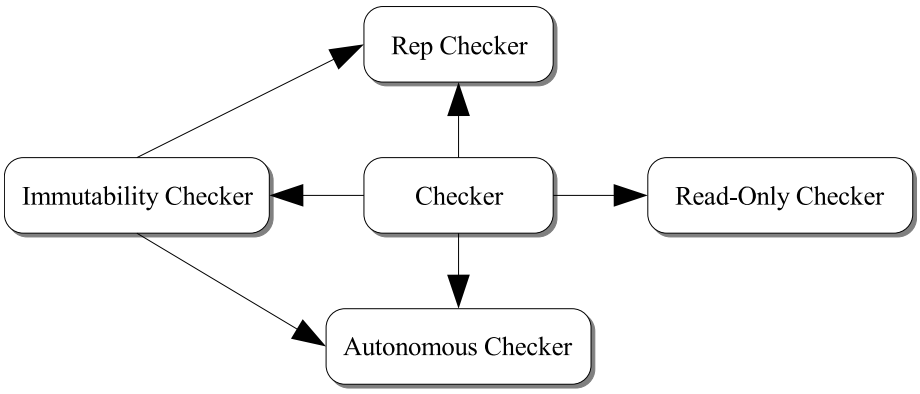

Figure 7.3: The Checker component

7.3 Annotation of Java Files

All annotations are embedded in Java comments, so that annotated files can be compiled by the standard Java compiler. In the future these annotations can be written in the new annotation syntax of Java 1.5. It will then be possible to prepend words with the @ sign to define own annotations. These annotations even appear in the compiled Java class files. However, until now we are using comments. The following annotations can be made:

- /*rep*/ – written before a declared type to indicate a rep type.
- /*immutable*/ – written before the class keyword to indicate an immutable class.
- /*autonomous*/ – written before the class keyword to indicate an autonomous class, or written before the result type of a method to indicate an autonomous method.
- /*readonly*/ – written before the result type of a method to indicate an readonly method.

Note that neither methods of immutable classes nor of autonomous classes have to be annotated with the /*autonomous*/ keyword as these methods are always autonomous. In Listing 7.1 all annotations are shown.

7.4 Checking

The checking consists of three phases:

```
1  public /*immutable*/ /*autonomous*/ class Example {
2      /*rep*/ int[] array;
3      /*rep*/ int integer;
4      /*rep*/ Vector vector;
5      public /*autonomous*/ void autonomousMethod() { }
6      public /*readonly*/ int size() {
7          return vector.size();
8      }
9      private /*rep*/ Vector setVector(/*rep*/ Vector v) {
10         /*rep*/ Vector localVector = new /*rep*/ Vector();
11         return vector = v;
12     }
13 }
```

Listing 7.1: All possible annotations.

- Phase 1: Filling the Class Repository with class information. Classes from the packages java.lang and java.util, and classes from user defined packages are parsed and put into the Class Repository. These classes are not checked, that is, all annotations in these classes are assumed to be correct.
- Phase 2: Checking of the files given by the user.
- Phase 3: Output of the results.

On a Pentium Mobile 1.4 GHz system, the first phase takes about 5 seconds for parsing the predefined packages. This time can certainly be optimized if the files are only read on demand, instead of parsing all files at start-up. The amount of time for the second phase linearly depends on the number and the size of the analyzed files. That is, the checking scales with the size of the code that has to be analyzed. The duration of the last phase is negligible.

7.5 Usage

In the following we only describe the usage under Unix operating systems. However, the tool should run under all operating systems that fulfill the requirements.

7.5.1 Requirements

The following software is needed by JACPOCK$_{proto}$ and must exist on the system:

- At least version 1.3 of the Java runtime environment (JRE[1]).
- Java CUP [27]

[1]The JRE can be downloaded from http://java.sun.com

- JFlex [43]
- Katja[2] [73]

Note that, except the JRE, these tools are already on the CD that is delivered with this thesis and need not to be installed.

7.5.2 Installation

The CD that is delivered with this thesis contains JACPOCK$_{proto}$ as source code and as bytecode. The tool can also be downloaded from http://softech.informatik.uni-kl.de. To install the tool simply copy the whole jacpock directory to an arbitrary place on your hard disk. It can also be directly executed from CD without any installation.

7.5.3 Run Jacpock$_{proto}$

JACPOCK$_{proto}$ is used on the command line. To run JACPOCK$_{proto}$ go to the jacpock directory and type

./jacpock [options] (<file> | <dir>)

JACPOCK$_{proto}$ can either check a single Java file (<file>), or it can check whole directories recursively (<dir>). The following options are provided:

- -immutable – checks for immutability only.
- -autonomous – checks for autonomy only.
- -rep – checks rep rules only.
- -readonly – checks read-only methods only.
- -dir=<dir> – recursively adds all files of directory <dir> to the Class Repository.
- -help – shows a help message.
- -debug – prints additional debug messages.

[2]Katja can be downloaded from http://softech.informatik.uni-kl.de

Chapter 8

Case Study: The Proof Container of Jive[1]

In this chapter we describe the application of our approach to the Proof Container of the Jive system [51]. We do this for two purposes:

1. For testing our approach on a real-life example from the practice.
2. For improving the encapsulation of the Proof Container component of Jive.

The remainder of this chapter is structured as follows. We begin with the description of the Jive system. After that we concentrate on the Proof Container component of Jive. We show the implementation details of the Proof Container and illustrate it by means of our component model. Then we use our approach to encapsulate the Proof Container. We discuss the problems that we have uncovered and give solutions. Finally, we conclude the chapter with a discussion of the results.

8.1 Jive 1.0

Jive [51] is a verification tool mostly written in Java to prove properties about programs written in a subset of Java. Jive 1.0 was a prototypical implementation and is going to be replaced by Jive 2.0. We applied our approach to version 1.0 of Jive to find possible problems in the implementation that should be avoided in Jive 2.0.

Jive takes an annotated Java file as input and generates a set of proof obligations. These proof obligations can then be proven interactively in Jive to show the correctness of the implementation with respect to the specification.

8.1.1 How to Verify Programs

To verify a Java program in Jive 1.0, it has to be written in a language called Anja[2]. Jive 2.0 will support the JML [46] syntax instead. Methods can be annotated with pre- and postconditions and classes can be given invariants that have to hold for all methods of all classes. It is now possible with Jive to prove that the program code is correct w.r.t. its specification. That is, for each method it can be shown that after the execution of the method body the postcondition holds, provided that the precondition was satisfied at method entry. At startup Jive reads the Anja file and creates so-called *goals* for each pair of pre- and postconditions. In addition, a goal for every invariant and every method is created. In order to prove the correctness of a whole program,

[1] Jive stands for Java Interactive Verification Environment
[2] Anja stands for Annotated Java

all goals have to be proven. A goal is represented by a *sequent*. A sequent consists of a *Hoare triple* [36] and a set of *assumptions*. A Hoare triple consists of a pre- and a postcondition and a program part. Assumptions are needed to allow recursive methods.

To prove a goal, Jive offers a set of so-called *rules*. Rules perform correct proof steps on sequents. See [70] for a description of the underlying logic of Jive. A rule can either directly prove a sequent or can create *subsequents* that have to be proven in order to prove the parent sequent. As a sequent and its subsequents, which are created during a proof, form a tree and the sequents can be proven, this structure is also called a *proof tree*. Goals are always the root nodes of their corresponding proof tree.

As it is often possible to mechanically decide which rule has to be applied to a certain sequent, it is possible to define so-called *tactics*. Tactics can be seen as rule macros. A tactic can only modify the proof trees by using rules just like the user. This guarantees that tactics cannot perform invalid steps on the proof trees. Tactics can be written by the user and added to the Jive system. This means that in order to properly encapsulate the Proof Container, this has to be done in an open-world scope as not all classes of the system are available during the checking time.

To prove sequents for their correctness, Jive relies on an external theorem prover. This can currently be either PVS [67] or Isabelle/HOL [68].

8.1.2 Architecture

The architecture of Jive is shown in Figure 8.1. It mainly consists of the following sets of classes:

- Jive – The main component.
- Front End – Reads and parses annotated Java files and constructs an abstract syntax tree (AST).
- Proof Container – Manages the proofs.
- GUI – The graphical user interface.
- Tactics – A set of rule macros, which can be extended by the user.
- Theorem Prover Interface – A Java interface to theorem provers like Isabelle or PVS.
- Formula – A set of classes representing formulas.
- AST – The abstract syntax tree of annotated Java files.

Formula is drawn in gray color do indicate that these classes are immutable. AST is striped, because its classes *should* have been implemented to be immutable, but actually they are *not*. What cannot be seen is that the representation of the Proof Container accesses the AST and Formula classes. That is, the Proof Container can only be independent if these classes are immutable. So before we describe the Proof Container component in detail, we examine the AST and Formula classes in the following two subsections.

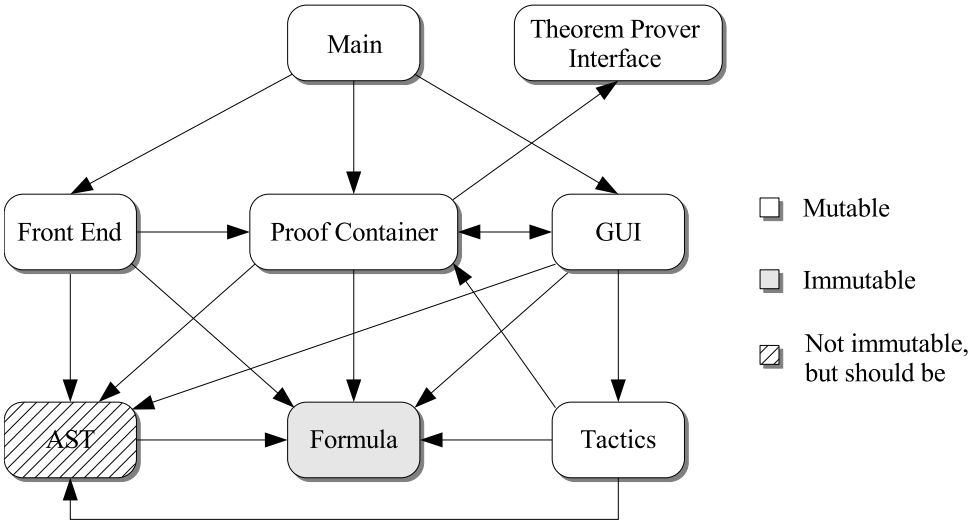

Figure 8.1: The architecture of Jive

8.1.3 The Abstract Syntax Tree

The classes that represent the AST are located in a package called jive.PC[3]. That package contains three subpackages, viz FrontEnd, PCPeer and Program. The FrontEnd package contains files related to the MAX [69] system. The PCPeer package mainly contains the class PCPeer with static methods that work on the abstract syntax tree of MAX. The Program package contains wrapper classes to abstract from the PCPeer class. The reason behind this design is that the MAX system is not written for Java but for C [42]. We now further examine the FrontEnd and Program packages.

8.1.3.1 The FrontEnd package

The AST is implemented with the aid of the MAX system [69]. MAX is a system to support programming language specification and implementation. It takes a specification file as input and generates C-Code as output. The specification file contains the description of the abstract syntax. Besides that it can contain attributes, functions and context conditions, written in an own functional language, which operates on the abstract syntax tree, but which cannot modify it. In addition to the C-Code, MAX can generate a JNI[4] file as an interface to Java. This interface is implemented by a set of public static methods residing in a single Java class. That implies that *all* classes of the Jive system can access these methods. This should not be a problem, however, as these functions cannot modify the AST. We examined that file, called pis_native.java, to ensure that it is really immutable, and surprisingly found a mutable public static field (see Listing 8.1). That is, all parts of the Jive system can potentially modify that field. It is not even final so it can be completely replaced. This problem, however, can be easily fixed by making the field private and by providing a public getter method that only returns the entries of the Hashtable.

[3] PC stands for Program Component.
[4] Java Native Interface

```
1 public class pis_native implements pis {
2     ...
3     public static java.util.Hashtable names =
4         new java.util.Hashtable();
5     ...
6 }
```

Listing 8.1: Mutable public static field in the Java interface file generated by MAX.

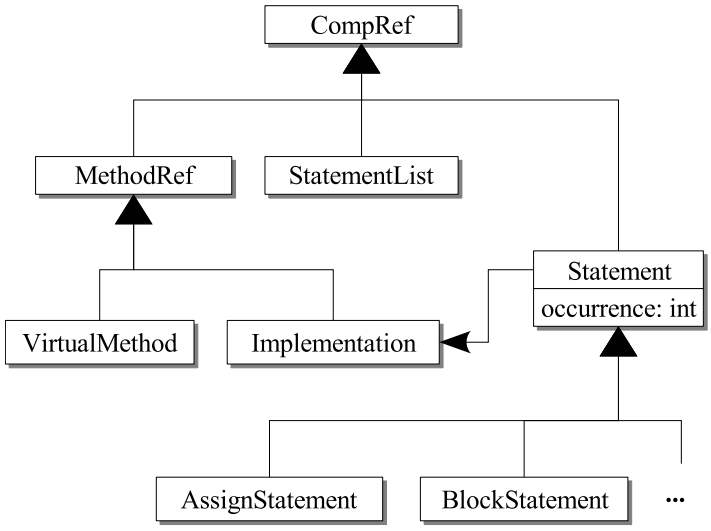

Figure 8.2: The CompRef inheritance hierarchy

8.1.3.2 The Program Package

The Java interface file of MAX is not the only problem of the abstract syntax tree implementation of the Jive system. As the Java interface file of MAX is inconvenient to use, there exist Java classes that abstract from that interface. All these classes inherit from the abstract class CompRef which represents a single occurrence in the abstract syntax tree of MAX. The inheritance hierarchy is shown in Figure 8.2. We examined these classes for their immutability and found problems in the following three classes:

- MethodRef
- StatementList
- Implementation

All classes have in common that they have an internal array field and return this field by a public method. But this is not the only problem. Listing 8.2 shows the problematic

```
1  abstract public class MethodRef extends CompRef {
2      String className;
3      String methodName;
4      String[] parameterTypes;
5
6      MethodRef(String cl, String me, String[] pt) {
7          className = cl;
8          methodName = me;
9          parameterTypes = pt;
10     }
11     ...
12     public String[] getParametertypes() {
13         return parameterTypes;
14     }
15     ...
16 }
```

Listing 8.2: The MethodRef class.

parts of the MethodRef class. It is obvious that this class is not immutable. There are a number of problems:

1. No field is declared final even though it were possible.
2. The mutable parameterTypes field is not private.
3. The parameterTypes field is returned by a public method.
4. The constructor assigns the formal parameter pt to the internal parameterTypes field.

Without breaking any code, we can declare all fields as final, solving the first problem. Making the parameterTypes field private, however, breaks code of some classes of the same package that accesses that field. We modified that code to use the public method that returns the field instead and made the field private. To follow our immutability definition the field must additionally be annotated with /*rep*/. Then the assignment in the constructor and the return statement in the public method would not be allowed anymore. However, there is a better solution. Instead of trying to protect the mutable field, we can also turn it into an immutable field. As String is already immutable, we only have to create an immutable list. As Katja generates immutable list classes, we simply defined a StringList in Katja and used the generated class instead of the String array. This required some code changes in Jive, but resulted in an immutable MethodRef class. The same changes can be applied to the Implementation and the StatementList class to make them immutable.

8.1.4 Formula

The Formula classes reside in package jive.PVC.Container.Formula. These classes have already been written with immutability in mind. So it is even more of a surprise that we found problems with these classes, too. However, these problems have not been as obvious as with the AST classes. The Formula classes are mainly generated by Katja [73].

Katja

Katja is the successor of the MAX system. It is entirely written in Java, and instead of C-code it generates Java code. Katja claims that the generated Java classes are all immutable. We examined the generated classes and discovered that they are not immutable according to our definition. Recall the last rule of the basic rules for immutable classes of Section 5.3.1: all subclasses of immutable classes must be immutable classes as well. To ensure this in an open-world scope, immutable classes have to be monomorphic or oligomorphic (cf. Section 4.5.1). However, the classes that are generated by Katja are polymorphic. That is, the number of subclasses is not restricted at all. This offers the possibility of subclassing Katja classes and breaking their immutability by overwriting methods. This problem, however, can easily be fixed by extending Katja and let it declare all generated files as final.

Katja generates two different kinds of classes: tuple classes and list classes. A tuple class has only private final fields, and all fields have Katja types as declared types. So if all Katja types are immutable, a tuple class is immutable, too. The list classes are more difficult. A list class always inherits from KatjaListImpl and delegates all method invocations to its superclass which implements the behavior. A part of the KatjaListImpl class is shown in Listing 8.3. It has only one protected List as field. In addition, it has an abstract method createInstance that takes a List as parameter. This method has to be overridden by subclasses. The list is implemented in a functional way. That is, instead of modifying the list in-place, a new instance is created instead, which can be seen by means of the addInternal method. As the KatjaListImpl class does not know its subclasses, it calls the createInstance method to create a new instance. Listing 8.4 shows a StringList class that was generated by Katja. It can be seen that a private constructor is generated that takes a List as argument and assigns it to its internal field. The createInstance method is overridden and just calls the private constructor with its formal parameter as argument. The add method just delegates the method call to the addInternal method of the KatjaListImpl class.

It is clear that the current implementation does not follow our immutability rules. The current implementation *is* not even immutable. The problem lies in the createInstance method. As the method is protected, classes of the same package as well as subclasses can call that method. For example, a class could be added to the same package in which the generated StringList is located and could call the createInstance method with an aliased List as argument. After the creation of the StringList, its internal list can be modified via the alias. So immutability of Katja classes in the current implementation can only be ensured if no classes can be added to packages of Katja generated classes.

However, immutability according to our rules can be achieved if the code from the KatjaListImpl class is moved into its subclasses. The lost code reuse is not important,

```
1  public abstract class KatjaListImpl
2     extends KatjaTermImpl
3  {
4     protected List values;
5     protected abstract KatjaListImpl
6        createInstance(List l);
7
8     protected KatjaListImpl addInternal(Object o)  {
9        List list = new ArrayList(values.size()+1);
10       list.addAll(values);
11       list.add(o);
12       return createInstance(list);
13    }
14    ...
15 }
```

Listing 8.3: The KatjaListImpl class.

```
1  public class StringList extends KatjaListImpl {
2     public Stringlist() {
3        values = new ArrayList();
4     }
5     private StringList(List listValues) {
6        values = listValues;
7     }
8     protected KatjaListImpl createInstance(List l) {
9        return new StringList(l);
10    }
11    public StringList add(String s)  {
12       return (StringList) addInternal(o);
13    }
14    ...
15 }
```

Listing 8.4: A StringList class generated by Katja.

```
1  public final class StringList extends KatjaListImpl {
2     private final rep ArrayList values;
3     public StringList() {
4        values = new rep ArrayList();
5     }
6     private StringList(rep ArrayList listValues) {
7        values = listValues;
8     }
9     public StringList add(String s)  {
10       rep ArrayList list =
11           new rep ArrayList(values.size()+1);
12       list.addAll(values);
13       list.add(o);
14       return new StringList(list);
15    }
16    ...
17 }
```

Listing 8.5: A modified StringList class that follows the immutability rules.

because the classes are generated anyway. In Listing 8.5 we show a modified StringList class that follows our immutability rules.

To sum up the Formula classes in the current Jive implementation are immutable in a closed-world scope, but Katja has to be modified in order to generate immutable classes that conform to our immutability rules.

8.1.5 Theorem Prover Interface

The Theorem Prover Interface is a Java interface to theorem provers like PVS [67] or Isabelle [68]. It starts the theorem prover in the background and offers methods to communicate with the running process. As the proofs of the Proof Container rely on this component, the correctness of the internal proofs also depends on that component. However, as we cannot check the underlying theorem prover, we cannot make any statements about it. We just have to assume that this component works as expected.

8.2 The Proof Container

The Proof Container component is responsible for the proofs. It offers an interface to manipulate proofs and stores the proofs in its internal representation. In addition, it is possible to view the internal representation by an iterator. Via the interface of the Proof Container it is not possible to create unsound proofs. However, it has to be

ensured that outside objects can only modify proofs via that interface. So the internal proofs have to be encapsulated by the Proof Container and must not be accessible by objects from the environment. In addition, proofs must not depend on any mutable state that is not encapsulated by the Proof Container. So the Proof Container has to be encapsulated and independent according to our definition of Chapter 3. We now describe the Proof Container in more detail.

8.2.1 Implementation Details

The Proof Container component is completely represented by a single package, namely jive.PVC.Container[5]. The package contains the following classes:

- The ProofContainer class, which is the main class of the component.
- The ProofTreeNode class, which represents a single node in a proof tree.
- The ProofTreeNodeIt class, which is a Proxy [30] for the ProofTreeNode class to offer the environment a read-only view to the ProofTreeNode class. It can be used to iterate through a proof tree.
- Sequent, Assumptions and Triple, which are simple classes to represent a Hoare triple [36].
- The abstract Operation class with direct subclasses Rule and Tactic and further concrete subclasses of Rule. The Rule classes represent valid proof transformations. Every rule represents a rule of the axiomatic semantics of the underlying logic of Jive and is assumed to be correctly implemented. These rules are part of the boundary of the Proof Container component and should be the only way for the environment to modify the proofs. The Tactic class can be subclassed to allo the user to create own tactics. A tactic can be seen as a macro for rules. The Operation class is used by the Views to create menu entries for the rules and tactics.

Except for the ProofTreeNode class, all classes are public. We begin with the classes representing a Hoare triple as the other classes depend on them.

8.2.1.1 Sequent, Triple, Assumptions

A Sequent (see Listing 8.6) consists of Assumptions and a Triple. Assumptions (see Listing 8.8) is simply an immutable list of Triples. The Assumptions are needed to allow recursive methods. A Triple (see Listing 8.7) is representing a Hoare triple [36] and thus consists of a pre- and a postcondition and a program part. The pre- and postconditions are represented by Formula classes, the program part is represented by the CompRef class. As all classes directly or indirectly reference the CompRef class, and that class was shown not to be immutable, all three classes are not immutable in the current implementation. If the CompRef class, however, were immutable, these classes would be immutable, too.

```
1 public final class Sequent {
2    public final Assumptions assump;
3    public final Triple triple;
4    public Sequent(Assumptions a, Triple t) {
5       assump = a; triple = t;
6    }
7 }
```

Listing 8.6: Simplified Sequent class.

```
1 public final class Triple {
2    public final Formula pre,post;
3    public final CompRef compRef;
4    public Triple(Formula p, CompRef c, Formula q) {
5       pre = p; post = q; compRef = c;
6    }
7 }
```

Listing 8.7: Simplified Triple class.

```
1 public class Assumptions {
2    private final Vector triples;
3    public Assumptions() { triples = new Vector(0); }
4    private Assumptions(Vector v) { triples = v; }
5    public Assumptions add(Triple t) {
6       new Assumptions(new Vector(triples).add(t));
7    }
8    ...
9  }
```

Listing 8.8: Simplified Assumptions class.

```
1  class ProofContainer {
2     final Vector proofTrees = new Vector();
3     private final Vector natives = new Vector(25);
4     private Vector views = new Vector();
5     public static Class[] tactics, axioms;
6     public static Class[] foperations, boperations;
7
8     public void addView(View v) { views.add(v); }
9     void nodeAdded(ProofTreeNode t) {
10       Enumeration e = views.elements();
11       while (e.hasMoreElements())
12          ((View) e.nextElement()).nodeAdded(t.getIt());
13    }
14    public Enumeration getProofTrees() {
15       return prooftrees.elements();
16    }
17    public ProofTreeNodeIt insertProofTree(Sequent s) {
18       ProofTreeNode result = new ProofTreeNode(this, s);
19       nodeAdded(result);
20       return result.getIt();
21    }
22    ...
23 }
```

Listing 8.9: A part of the ProofContainer class.

8.2.1.2 The ProofContainer Class

A part of the ProofContainer class is shown Listing 8.9. The class has three instance fields:

- proofTrees – a Vector that contains ProofTreeNode instances.
- natives – a Vector that contains Sequent instances.
- views – a Vector that contains View instances.

The views list is used to implement the Observer pattern [30]. A View instance can register itself with the ProofContainer by calling the addView method. The view is then added to the views list of the ProofContainer and is informed of any structural changes of the proof trees. The nodeAdded method, for example, calls the nodeAdded method on each View that is contained in the views list to indicate that a node was added to a proof tree. This ensures that the Views are always consistent with the ProofContainer. As the list of Views does not belong to the representation of the Proof Container, it need not be encapsulated.

The proofTrees Vector contains ProofTreeNode instances. These nodes are the roots of the proof trees. A new proof tree can be added with the insertProofTree method. The method takes a Sequent as argument and passes the argument to the ProofTreeNode constructor. This implies that the Sequent class has to be immutable. Otherwise the environment could keep an alias to a Sequent and could modify the transitive state of a ProofTreeNode.

The natives Vector contains Sequent objects representing native methods. These native methods are assumed to be correct and need not to be proven. This list has to be encapsulated by the Proof Container as it would otherwise be possible to prove methods by inserting them into the natives list.

Besides the three instance fields, the ProofContainer class has four public static Class[] fields. These fields are filled at the start-up of Jive with rule and tactic classes. Clearly, the current implementation cannot stay as it is, because any class of the system can modify these fields. So these fields have to be encapsulated, too.

8.2.1.3 The ProofTreeNode Class

A proof in Jive is represented by a tree, the so-called proof tree. The proof tree is a recursive data structure which is built of ProofTreeNodes. Every ProofTreeNode has a possibly empty list of ProofTreeNode children and a Sequent which represents a Hoare triple. In addition, every ProofTreeNode has a flag called closed that indicates whether the node is closed or not. A node is closed iff it is either proven, or it has child nodes. If a backward rule, for example, is applied to an open node, the rule closes that node and creates one or more open child nodes.

Besides these fields the ProofTreeNode has two final fields that point to a ProofContainer instance and a ProofTreeNodeIt instance. The ProofContainer instance is needed to inform the container about structural changes of the node in order to update the views accordingly.

The ProofTreeNodeIt class is a Proxy for the ProofTreeNode class. Its implementation does not offer any surprising elements so we omit it here. It has a reference to a

[5] PVC stands for Program Verification Component.

```
1  class ProofTreeNode {
2     private boolean closed = false;
3     private Sequent sequent;
4     ProofTreeNode parent;
5     private final Vector children = new Vector();
6     final ProofContainer container;
7     final ProofTreeNodeIt iter = new ProofTreeNodeIt(this);
8
9     ProofTreeNode(ProofContainer c, Sequent s) {
10       this.container = c;
11       this.sequent = s;
12    }
13    ProofTreeNode(ProofTreeNode parent, Sequent s) {
14       this.container = parent.container;
15       this.parent = parent;
16       this.sequent = s;
17    }
18    Sequent getSequent() { return sequent; }
19    ProofTreeNodeIt getIt() { return iter; }
20    ProofTreeNode getChild(int i) {
21       return (ProofTreeNode) children.get(i);
22    }
23    int getNumberOfChildren() { return children.size(); }
24    boolean hasChildren() { return ! children.isEmpty(); }
25    void addChild(ProofTreeNode node) {
26       children.add(node);
27       node.parent = this;
28       container.nodeAdded(node);
29    }
30    ...
31 }
```

Listing 8.10: Simplified ProofTreeNode class.

ProofTreeNode instance as field and is delegating method calls to that instance. That Proxy class only contains methods that do not modify the underlying ProofTreeNode instance. It is used to give the environment a read-only view to the proof trees. If Java had a built-in read-only mode it would have also been possible to give the environment a read-only reference to an internal ProofTreeNode. See Theisen [75] for a comprehensive discussion about the advantages of using a read-only mode instead of a Proxy. However, as Java has no built-in read-only mode, and as a read-only mode cannot be checked in an open-world scope without language support, we cannot use it here.

As the ProofTreeNode objects represent the state of the overall proof, they must be encapsulated by the Proof Container. It must not be possible for the environment to obtain a reference to a ProofTreeNode. The environment may only use the external interface of the Proof Container component to manipulate the proof trees, that is, either by using methods of the ProofContainer class or by using methods of Rule classes. In addition, the ProofTreeNode objects must not depend on any mutable external object.

8.2.2 Using our Component Model

Figure 8.3 shows a simplified Proof Container by means of our component model. The ProofContainer is the main object of the component. The Rules and the PTNIterators are boundary objects. The PTNIterators can be used by the environment to iterate through the proof trees. They only offer a read-only view to the ProofTreeNodes and cannot be used to modify the ProofTreeNodes. As described above, the ProofTreeNodes have references to the Sequent class which indirectly references the AST classes which are not immutable.

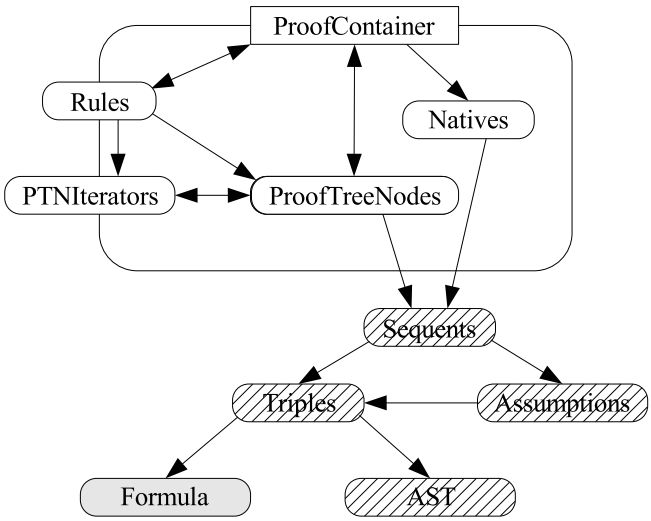

Figure 8.3: The Proof Container showed by means of our component model.

As can be immediately seen, the Proof Container is not independent in the current implementation. The internal representation, the ProofTreeNode objects, is accessing the external AST objects which are not immutable as shown above already. In our

component model it is possible for independent components that the representation objects can *reference* mutable outside objects as long as no method is called on them. However, the ProofTreeNodes are also calling methods on the AST objects and are thus breaking independency. As the Formula classes are immutable the ProofTreeNodes can access them. However, as already described above, it is not difficult to make the AST classes immutable, so the Proof Container can be made independent according to our definition.

8.2.3 Encapsulating the Proof Container

In order to encapsulate the Proof Container, we have to protect the proofTrees field, the natives field and the fields that contain the rules and tactics. As the natives Vector only contains Sequent objects which can be made immutable, we annotate that field with rep to encapsulate it. The same applies to the rule and tactic arrays. As the Class class is (assumed to be) immutable we can annotate these arrays with rep as well. The encapsulation of the proofTree field is more difficult and is shown in the following.

Encapsulating the ProofTreeNodes

The proofTree Vector contains ProofTreeNode instances. The ProofTreeNode class is not immutable, so if we annotate the proofTree field with rep, the ProofTreeNode objects have to be rep objects. However, as ProofTreeNode references the boundary classes ProofContainer and ProofTreeNodeIt, it cannot be a rep object, because rep objects can only reference immutable objects or other rep objects. There are two possible solutions to that problem:

1. Changing the implementation of the Proof Container, so that ProofTreeNode does not reference boundary classes anymore.

2. Declare the ProofTreeNode class to be a rep class of the ProofContainer class.

The second option seems to be the easier one, so we begin with it.

Using Rep Classes

In order to declare the ProofTreeNode class as a rep class, we annotate it with repof ProofContainer. The ProofContainer class as well as the ProofTreeNode itself store ProofTreeNode instances in the Vector class. The Vector class, however, has Object as element type and so the ProofTreeNode type is widened to Object when added to the Vector. This violates rule RC4 which forbids widening of rep class types to Object (see Table 4.2 on page 58). This problem can be solved by using an array instead of the Vector class, as rep class arrays are as encapsulated as their component type (see Section 4.9.1). Given these changes we can encapsulate all ProofTreeNode instances in the Proof Container component. To allow the Rule classes and the ProofTreeNodeIt class to access the ProofTreeNode class, they have to be annotated with boundof ProofContainer to declare them as boundary classes of the ProofContainer class.

Using Rep Types

Instead of using rep classes it is also possible to declare the proofTree field with a rep-annotated Vector. In this case, however, the implementation of the ProofTreeNode class has to be changed, so that the reference to the ProofContainer is not needed anymore. The reference to the ProofContainer is used to inform the views about changes. That code, however, can be moved to the places where the nodes are actually changed, namely the Rule classes. In addition, the ProofContainer reference is used by the View as the root node of all proof trees. So the parent node of root nodes is the ProofContainer itself. This design must be changed so that there is a ProofTreeNode as root node for all proof trees. With these modifications we can encapsulate the ProofTreeNode instances in the Proof Container component. Like with rep classes, the Rule and ProofContainerIt classes must be declared as boundary classes of the ProofContainer class.

8.3 Conclusion

We applied our approach to the Proof Container component of the Jive system. During that application we uncovered a number of problems. The AST classes are not immutable, but are accessed by the representation of the Proof Container. So objects of the environment are able to modify the program part of any Hoare triple. The program part of an already proven Hoare triple can be modified, producing an unsound proof. As the AST for Jive 2.0 is replaced anyway, it is important to implement it in an immutable way. While the Katja generated classes are immutable in a closed-world scope, they are not immutable in an open-world scope. So Katja has to be modified in order to generate classes that are immutable in an open-world scope.

We used our approach to encapsulate the Proof Container component. We showed how the ProofTreeNodes can be encapsulated with rep classes as well as rep types. Both approaches need modifications to the current implementation, but both can be applied.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this master's thesis we have developed a solution to encapsulate components in Java. We presented a component model based on the encapsulation model of Noble et al. [64] to formulate the notion of encapsulation and independence. We mapped this model to Java by defining a specification technique for Java programs. Our proposal is based on rep types, an idea known from universes [60]. Objects referred to by rep expressions are guaranteed to be encapsulated and independent. In contrast to universes we allow static variables, but we require that rep types may only be of autonomous classes. In addition, boundary classes can be specified to allow the implementation of programming patterns like iterators. We adopted confined types [76] and used their idea in our notion of rep classes. Instances of rep classes are encapsulated but not independent from the environment. Our approach has been introduced as class-level protection, and we showed how object-level protection can be achieved. Our rules are statically checkable in an open-world scope. This ensures that checked components can be used in unchecked environments while preserving encapsulation and independence. The rules can be applied to unmodified Java and annotated classes can be compiled by the standard Java compiler.

A second contribution that is related to components, as well as being valuable on its own, are immutable types. Instances of immutable types, immutable objects, can be shared arbitrarily between a component and its environment. Immutable objects can be used similar to primitive values and can be seen as complex value objects, even though they are not copied by value, but copied by reference. Our immutability definition is, to our knowledge, the most practical approach so far, as we allow the lazy initialization pattern [30] which was not handled by previous work. Immutable types are orthogonal to the concept of ownership, hence it should be possible to combine immutable types with other ownership approaches to increase their expressiveness.

We implemented a prototypical checking tool for a subset of our concepts to verify the correctness of specified code. We applied our approach to the Jive system [51], uncovered the problems of its current implementation and showed how the Proof Container component of Jive can be properly encapsulated by our approach.

To summarize, this thesis is a further but not the final step towards the encapsulation of components in object-oriented programming languages.

9.2 Future Work

Our approach has some limitations. It is not satisfactory that rep objects cannot access boundary objects. It has to be investigated how this limitation can be removed.

In addition, it should be possible to store references to mutable outside objects in representation objects in order to solve the container problem. The upcoming generics of Java 1.5 could play an important role to the solution of that problem. Rep classes showed that our component model may have to be extended to cover objects of components that are encapsulated, but not independent. Possible extensions to our approach like borrowed, fresh and read-only expressions have to be further investigated.

An interesting topic could be the examination of encapsulated components in multi-threaded environments. It is an interesting question what it means for a component to be thread-safe and how thread-safe components can be implemented.

Finally, our checking tool has to be further improved to cover all concepts of this thesis, and our rules should be expressed by a type system to formally prove their correctness by means of an operational semantics.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Jonathan Aldrich and Criag Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, June 2004. ISBN 3-540-22159-X.

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In OOPSLA'02 [66], pages 311–330.

[4] Eric Allen. *Bug Patterns in Java*. APress, 2002.

[5] Paulo Sérgio Almeida. Balloon Types: Controlling sharing of state in data types. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, June 1998. ISBN 3-540-63089-9.

[6] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect if Java without data races. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'00)*, pages 382–400. ACM Press, October 2000.

[7] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *SIGPLAN OOPS Messenger*, 4(4):2–27, 1993.

[8] Henry G. Baker. 'Use-Once' variables and linear objects–storage management, reflection and multi-threading. *SIGPLAN Notices*, 30(1):45–52, January 1995.

[9] Marina Biberstein, Joseph Gil, and Sara Porat. Sealing, encapsulation, and mutability. In Knudsen [45], pages 28–52. ISBN 3-540-42206-4.

[10] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In Vlissides and Schmidt [77], pages 35–49. ISBN 1-58113-831-9.

[11] Joshua Bloch. *Effective Java*. Addison-Wesley, Reading, Massachusetts, 2001.

[12] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In OOPSLA'01 [65], pages 56–69.

[13] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In OOPSLA'02 [66], pages 211–230.

[14] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 213–223. ACM Press, January 2003. ISBN 1-58113-628-5. doi: http://doi.acm.org/10.1145/604131.604156.

[15] John Boyland. Alias burying: Unique variables without destructive reads. *Software – Practice and Experience*, 31(6):533–553, May 2001.

[16] John Boyland. The interdependence of effects and uniqueness. In *3rd Workshop on Formal Techniques for Java Programs*, Budapest, Hungary, June 2001.

[17] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In Knudsen [45], pages 2–27. ISBN 3-540-42206-4.

[18] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *20th International Conference on Software Engineering (ICSE)*, pages 167–176, Los Alamitos, CA, April 1998.

[19] Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July 2001.

[20] Dave Clarke and S. Drossopoulou. Ownership, encapsulation, and the disjointness of type and effect. In OOPSLA'02 [66], pages 292–310.

[21] Dave Clarke and Tobias Wrigstad. External uniqueness. In *10th Workshop on Foundations of Object-Oriented Languages (FOOL)*, New Orleans, LA, January 2003.

[22] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In Luca Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer-Verlag, July 2003. ISBN 3-540-40531-3.

[23] Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64. ACM Press, October 1998.

[24] Dave Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In Knudsen [45], pages 53–76. ISBN 3-540-42206-4.

[25] David G. Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: Deployment-time confinement checking. In Crocker and Jr. [26], pages 374–387. ISBN 1-58113-712-5.

[26] Ron Crocker and Guy L. Steele Jr., editors. *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'03)*, October 2003. ACM Press. ISBN 1-58113-712-5.

[27] CUP. Parser generator for Java, 1999. URL `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[28] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Digital Systems Research Center, July 1998. SRC-RR-156.

[29] Matthias Felleisen. Functional objects. In Vlissides and Schmidt [77], page 267. ISBN 1-58113-831-9.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[31] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification – Second Edition*. Addison-Wesley, June 2000.

[32] Peter Grogono and Patrice Chalin. Copying, sharing, and aliasing. In Vangalur S. Alagar and Rokia Missaoui, editors, *Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering (COODBSE'94)*, pages 77–89, May 1994.

[33] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In OOPSLA'01 [65], pages 241–253.

[34] Harri Hakonen, Ville Leppänen, Timo Raita, Tapio Salakoski, and Jukka Teuhola. Improving object integrity and preventing side effects via deeply immutable references. In *Proceedings of the Sixth Fenno-Ugric Symposium on Software Technology (FUSST'99)*, pages 139–150, 1999.

[35] B.W. Harms, D.E. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17 (5):424–435, May 1991.

[36] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[37] John Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *Proceedings of the 6th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'91)*, pages 271–285. ACM Press, November 1991.

[38] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Messenger*, 3(2):11–16, 1992. ISSN 1055-6400.

[39] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001.

[40] Stuart Kent and John Howse. Value Types in Eiffel. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 19)*, pages 145–160. Prentice Hall, January 1996.

[41] Stuart Kent and Ian Maung. Encapsulation and aggregation. In *In Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS PACIFIC 95)*, Melbourne, Australia, November 1995. Prentice Hall.

[42] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[43] Gerwin Klein. JFlex - the fast scanner generator for Java, September 2003. URL `http://www.jflex.de`.

[44] Günter Kniesel and Dirk Theisen. JAC – Access right based encapsulation for Java. *Software – Practice and Experience*, 31(6):555–576, May 2001.

[45] J. Lindskov Knudsen, editor. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, June 2001. Springer-Verlag. ISBN 3-540-42206-4.

[46] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.

[47] K. Rustan M. Leino and Raymie Stata. Virginity: A contribution to the specification of object-oriented software. Technical Report 1997-001, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, April 1997. SRC Technical Note.

[48] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[49] Barbara Liskov and John Guttag. *Program Development in Java, Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.

[50] B. J. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12):70–79, December 1982. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/988164.988172.

[51] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Interner Bericht, 2000.

[52] Microsoft. *C# Language Specification*. 2001.

[53] Sun Microsystems. JDK^{TM} 5.0 Documentation. URL `http://java.sun.com/j2se/1.5.0/docs/index.html`.

[54] Naftaly H. Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer-Verlag, July 1996. ISBN 3-540-61439-7.

[55] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, Fernuniversität Hagen, 2001.

[56] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[57] Peter Müller and Arnd Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 137–159. Cambridge University Press, 2000.

[58] Peter Müller and Arnd Poetzsch-Heffter. A type system for controlling representation exposure in Java. In Drossopoulou et al., editor, *Formal Techniques for Java Programs*. Technical Report 269–5, Fernuniversität Hagen, 2000.

[59] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279–1, Fernuniversität Hagen, 2001.

[60] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programmiersprachen und Grundlagen der Programmierung, Kolloquiumsband '99*, Informatik Berichte 263–1. Fernuniversität Hagen, 2000.

[61] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in jml. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.

[62] James Noble. Iterators and encapsulation. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 431, St. Malo, France, June 2000. IEEE Computer Society. ISBN 0-7695-0731-X.

[63] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *Proceedings of the 12^{th} European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998. ISBN 3-540-64737-6.

[64] James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke. Towards a model of encapsulation. In *International Workshop on Aliasing, Confinement and Ownership (IWACO 21)*, July 2003.

[65] OOPSLA'01. *Proceedings of the 16^{th} ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'01)*, October 2001. ACM Press. ISBN 1-58113-355-9.

[66] OOPSLA'02. *Proceedings of the 17^{th} ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'02)*, November 2002. ACM Press.

[67] Sam Owre, Natarajan Shankar, John M. Rushby, and David W. J. Stringer-Calvert. PVS Language Reference, Version 2.3. Technical report, Computer Science Laboratory SRI International, September 1999.

[68] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, USA, 1994.

[69] A. Poetzsch-Heffter and T. Eisenbarth. The MAX system: A tutorial introduction. Technischer Bericht TUM-I9307, Technische Universität München, April 1993.

[70] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. Doaitse Swierstra, editor, *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999. ISBN 3-540-65699-5.

[71] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. Automatic detection of immutable fields in Java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 10. IBM Press, 2000.

[72] Alex Potanin, James Noble, David G. Clarke, and Robert Biddle. Featherweight generic confinement. In *Proceedings of the 11th International Workshop on Foundations of Object-Oriented Languages (FOOL 11)*, January 2004.

[73] Jan Schäfer. Katja: Generating order-sorted data types in Java. Internal Report, mail to: j_schaef@informatik.uni-kl.de, January 2004.

[74] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, third edition, 1997. ISBN 0–201–88954–4.

[75] Dirk Theisen. Enhanced encapsulation in oop – a practical approach. Master's thesis, University of Bonn, Computer Science Department III, Bonn, Germany, July 1999.

[76] Jan Vitek and Boris Bokowski. Confined types in Java. *Software – Practice and Experience*, 31(6):507–532, 2001.

[77] John M. Vlissides and Douglas C. Schmidt, editors. *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'04)*, October 2004. ACM Press. ISBN 1-58113-831-9.

[78] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Working Conference on Programming Concepts and Methods (PROCOMET)*, pages 347–359. North-Holland, April 1990.

[79] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for featherweight Java. In Crocker and Jr. [26], pages 135–148. ISBN 1-58113-712-5.