Identifying and checking component structures in object-oriented programs

Master's Thesis
by Max Bechtold

June 17, 2013

Abstract

Software components are an often-postulated concept to cope with the complexity and costs of large software. Components allow modularisation of software, foster reuse of functionality and in turn reduce costs of software development. By using components, one can achieve a more abstract and clearer model of software. We conceive that small or medium-size programs can also benefit from the advantages inherent to components. However, Java as a prominent object-oriented programming language provides little support to cater for software components. This thesis demonstrates how one can identify components in Java programs as types and objects with state. Analysing the interconnection of components facilitates a component-oriented view on programs by concentrating on the components as the principle program elements. We provide mechanisms to employ and validate components in Java programs in form of analysis and monitoring tools for the presented concepts. We evaluate our component constructs on the Java library and on the open-source tool Sat4J, showing the benefits of identifying components in object-oriented programs.

[German translation]

Softwarekomponenten sind ein häufig postuliertes Konzept zur Bewältigung der Komplexität und Kosten umfangreicher Software. Komponenten ermöglichen Modularisierung von Software, fördern Wiederverwendung von Funktionalität und verringern die Kosten der Softwareentwicklung. Mit Komponenten kann ein abstrakteres, klareres Modell von Software erreicht werden. Es ist unsere Überzeugung, dass auch kleine und mittelgroße Programme von den Vorteilen von Komponenten profitieren können. Java als eine prominente objektorientierte Programmiersprache bietet jedoch wenig Unterstützung für Softwarekomponenten. In dieser Arbeit zeigen wir, wie Komponenten in Java als zustandsbehaftete Typen und Objekte ausgemacht werden können. Die Analyse der Querverbindungen von Komponenten ermöglicht eine komponentenorientierte Sicht auf Programme. Wir stellen Mechanismen zum Einsatz und zur Validierung von Komponenten in Java-Programmen in Form von Analyse- und Kontrollwerkzeugen für die vorgestellten Konzepte bereit. Wir evaluieren unsere Komponentenkonstrukte anhand der Java-Typbibliothek und des quelloffenen Programms Sat4J, und zeigen dabei die Vorzüge der Identifizierung von Komponenten in objektorientierten Programmen.

Declaration

With this statement I declare that I have independently completed this master's thesis entitled "Identifying and checking component structures in object-oriented programs." The thoughts and quotations taken directly or indirectly from external sources are properly marked as such.

Place, Date: _____

Signature: _____

To Jessica

Contents

1. Introduction **8**

2. Basic concepts **10**
2.1. Types in Java . . . 10
2.1.1. Types and objects . . . 10
2.1.2. Type declarations . . . 11
2.1.3. Using types . . . 13
2.2. Facets . . . 13
2.2.1. Static facets . . . 14
2.2.2. Dynamic facets . . . 17
2.2.3. Final remarks . . . 19

3. Components and their interconnection **21**
3.1. Identifying components . . . 21
3.1.1. State through fields . . . 23
3.1.2. State through methods . . . 24
3.1.3. Classification and type hierarchy . . . 31
3.1.4. Object and class components . . . 34
3.1.5. Other component notions . . . 35
3.2. Component interconnection . . . 36
3.2.1. Referencing objects . . . 37
3.2.2. Referring to types . . . 45
3.2.3. Transitory references . . . 47
3.2.4. Links relevant for interconnection . . . 49
3.2.5. Connected components . . . 50
3.2.6. Subcomponents and dependencies . . . 53
3.2.7. Views on components . . . 59
3.2.8. Interconnection in summary . . . 64

4. Components in Practice **67**
4.1. Employing component concepts with annotations . . . 67
4.1.1. Annotations in Java . . . 68
4.1.2. Facet annotations . . . 69
4.1.3. @Transitory for parameters and variables . . . 70
4.1.4. Declaring connectivity with @Connected . . . 73
4.1.5. Distinguishing subcomponents from dependencies . . . 74
4.1.6. @GetView for methods yielding views . . . 75

4.1.7. Annotations and type hierarchy . . . 76
4.2. The Compiler example . . . 78
4.2.1. The main class Compiler . . . 79
4.2.2. Scanners produce tokens . . . 82
4.2.3. Code generators create compile units . . . 83
4.3. Static program analysis . . . 87
4.3.1. Type facet analysis . . . 88
4.3.2. Interconnection analysis . . . 91
4.4. Dynamic compliance checking . . . 95
4.4.1. Overview on AspectJ . . . 95
4.4.2. Checking connectivity of fields . . . 99
4.4.3. Verifying encapsulation of subcomponents . . . 102
4.4.4. Remarks on aspects . . . 105

5. Evaluation **106**
5.1. Components in the Java library . . . 106
5.1.1. Facet analysis of the Java library . . . 106
5.1.2. Classification of common Java types . . . 107
5.2. Analysis of Sat4J . . . 110
5.2.1. Classification of Sat4J . . . 110

6. Conclusion **114**

A. Appendix **118**

1. Introduction

Object-oriented programs exist in many sizes. They range from single class tools over medium-size programs to large scale software. The latter are often based on technologies such as component frameworks in order to cope with the complexity of developing and maintaining software. Such frameworks allow structuring of large programs into smaller units, the components, and concert their interaction. Subdividing a program into components yields a more abstract view on the functionality, providing for a clearer picture of a program and better understandability [11].

Small or medium-size programs do not usually make use of component frameworks. Such frameworks impose an overhead on programs, which have to be connected with the infrastructure of frameworks and adhere to their design and interaction rules. Also, programs must be fit into the strict harness provided by frameworks. In small and medium-size programs developers often do not follow a well-structured design by which the program can be easily separated into components. Yet, it can be beneficial for these programs to adopt the principle idea of component frameworks, in which components are the main entities of programs. Components model distinct parts of the solution that software implements and are the interacting elements of a program.

Our goal is to add the concept of components to Java as an object-oriented language. There is no such notion of components inherent to Java. In this language, objects are modelled by classes. Classes can be structured in packages, however, this is simply a syntactic means to segregate parts of a program from other parts. There is no concept in Java to distinguish classes or objects semantically. All classes and objects are equal from the perspective of the language. We conceive components as a concept not yet existing in the Java language by which the main classes and objects in a program can be identified. We expect to achieve a clearer view on programs by distinguishing components from other elements. A component-oriented model of a program allows a better understanding of the interconnection of classes and objects of a program.

The approach to defining components presented in this thesis is based on state as a differentiating property of classes and objects. Java does not define a concept of state on the level of the language. However, the terms “stateful” and “stateless” are often used to distinguish objects and other elements in object-oriented programs semantically. We provide a definition for the notion of state, which is the basic criterion for identifying components in Java programs.

As we define components as an extension of Java, we will use concepts and mecha-

nisms of this language to anchor and define our terms. To achieve a precise basis for the introduced concepts, we will frequently refer to the specification of Java. To avoid cluttering of the text, the abbreviation "JLS" will be used to cite the *Java Language Specification Java SE 7 Edition* [6]. Development of Java programs is supported by the Java SE Development Kit (JDK). In particular, the JDK provides a library of predefined classes that can be used by other programs. We evaluate our concepts on classes of the Java library,[1] and also utilise these classes in examples throughout this thesis.

The structure of this thesis is as follows: After this introduction Chapter 2 illuminates the basic concepts required to define components in Java. We explain how *types* of Java can be separated through the notion of *facets*. The third chapter uses facets as the foundation to define *components*, which we segregate from *immutable* types and objects. We also present *interconnection* as a component-oriented view on programs, facilitating a clearer model of a program than without components. In Chapter 4, we illustrate how components and related concepts can be represented in the code of programs. Furthermore, we introduce tools that aid in identification of our concepts in programs and compliance of programs with requirements of these concepts. Chapter 5 evaluates the introduced concepts with the Java library and a real-world project. We also identify components in the Java library, which therefore occur in many programs using this library. Finally, this thesis ends with a conclusion.

[1] in particular on the JDK 7 Update 9 Windows version, see `http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html#jdk-7u9-oth-JPR`

2. Basic concepts

The major distinction of object-oriented programming from other paradigms such as procedural or functional programming is that it deals with classes and objects. Classes and objects govern the dynamics of a program. They constitute data structures, implement functionality, and allow for a closer reflection of the real world in the program structure than it was possible before. We use Java to demonstrate how an object-oriented language can be extended with components in order to reason about programs in a more abstract way than possible with objects and classes. The first part of this chapter explains Java *types* as the basis of classes and objects. In the second part, we describe how types can be subdivided with the notion of *facets*.

2.1. Types in Java

The concepts devised in this thesis are based on parts of types, creating a more granular basis for defining components in Java than whole types. Before showing how types can be subdivided in the next section, we illuminate types in Java in an overview.

2.1.1. Types and objects

Java embodies the concepts of object-oriented programming by means of *class types* and *interface types* (see JLS 4.3, 4.3.1). During the *runtime* of a program, i.e. when executing it, class types can occur as *instances*, so-called *objects*. Objects can be used in programs just as the class and interface types themselves, which can be considered a special instance, the *class object*. A different set of types are the *primitive* types (JLS 4.2), which are simple value types representing numbers (type `int`) or characters (type `char`), for instance.

A special variant of objects are *arrays* (see JLS ch. 10), which are vectors of *variables*[1] with a fix order given by their *index*. Arrays are of an *array type* that can be based on a primitive type, allowing assignment of primitive values (*values*) to its variables. It can also be based on a class or interface type, meaning one can assign objects of that type to its variables. Lastly, an array type can be based on another array type, which allows assigning of array objects to array variables, i.e. arrays can be nested.

Arrays can be instantiated through an array creation or array initialization expression (JLS 15.10, 10.6), commonly denoted *array creation* in the following. Array creation can include *array initialisation*, which allows to specify values or objects assigned to variables

[1] We refrain from using the alternative term "array components" to avoid misunderstanding

directly. As arrays have no independent type, array types are not included when using the term *type*, which in the following refers to class and interface types. Despite this caveat, arrays as objects are considered when defining component concepts.

Types are organised into so-called *packages*, which are a means to structure a set of Java types. When a type *extends* a class or interface type or *implements* interface types, these become *supertypes* of the type, which in turn is a *subtype* of extended and implemented types.[2] Subtypes and supertypes that are class types are also called *subclasses* and *superclasses*, respectively.

2.1.2. Type declarations

Types declare fields and methods which can be *accessed* and *invoked*, respectively. Fields and methods are how interaction between objects and types at runtime takes place. Values or objects can be assigned to and retrieved from a field depending on its type. Methods can take parameters of certain types, perform some kind of operation, and possibly return a result that matches their *return type*[3] when concluding this operation. Constructors are similar to methods, but used to initialise instances of classes, which are instantiated by using the keyword `new`.[4] Interfaces cannot be instantiated directly, and neither classes declared `abstract`. At runtime instances of a class type `T` can be used in a program anywhere a supertype of `T` occurs, e.g. as type of fields, variables, or method parameters or return type of methods. For instance, a `T` instance can be assigned to a field whose type is a supertype of `T`.

Methods and constructors have a similar structure: Both have *signatures* consisting of a name (which, for constructors, is always the name of the declaring type) and a possibly empty parameter list, and both have a *body* (except for abstract methods in abstract classes or interfaces).

A type can also declare *static initializers* and *instance initializers* (see JLS 8.7, 8.6). These serve to initialise a type or its instances, respectively. Method and constructor bodies and initializers consist of *statements* (see JLS ch. 14) controlling the flow of execution of a program. Statements can contain other statements and *expressions* (see JLS ch. 15), which evaluate to values or objects when a program is executed. One kind of statements are local variable declarations. *Local variables* can store values or objects just as fields, but only exist until the end of the execution of the method, constructor, or initializer they are declared in.

An *exception* that occurs during the execution of a statement or evaluation of an

[2] Both notions are transitive: A supertype of a supertype of a type `T` is also `T`'s supertype (similarly for subtypes)

[3] which can be a primitive, class, interface, or array type, or `void`, meaning no value is returned (see JLS 8.4.5)

[4] which preludes a *class instance creation expression* (JLS 15.9)

expression indicates an abnormal situation. An exception will cause the program to terminate prematurely unless it is caught, i.e. it is handled appropriately. A program can also terminate with an error, e.g. when it consumes all memory available. Both exceptions and errors are modelled as objects in Java, specifically as instances of subtypes of java.lang.Throwable. Due to their special semantics we opt not to consider subtypes of Throwable. Thus, in the following objects will not be instances of Throwable. Also, class or interface types will not correspond to Throwable or a subtype thereof.

Types can also declare *member types*. These are nested in their *enclosing* type, which they are declared by and associated with. Member types can as well declare methods, fields, constructors, initializers, and members types. A type that is not a member type is called *top-level* type,[5] as it is not nested in another type.

Fields, methods, and member types are collectively referred to as *members*. If they are declared static, they are part of the type at runtime, if not, they belong to instances of the type. Thus, static members are accessed on the type, while non-static members are accessed on objects. Interface types may only declare static fields and non-static methods. Non-static member classes require an instance of the enclosing class to be instantiated, tying it closely to that instance. Depending on *accessibility* and *hiding*, members of supertypes can be inherited by subtypes. A non-static method can *override* a non-static method of a supertype under certain conditions, and the overridden method is then not inherited. These concepts are required later and explained in Sections 2.2 and 4.1.

The following small example illustrates types and their members with two type declarations A and I:

```
1  public class A implements I {
2    static int a;
3    int[] b = new int[] {1, 2, 3};
4
5    static class B {
6      static void m() {
7        B b = new B();
8      }
9    }
10
11   public void n() {...}
12 }
13
14 public interface I {
15   public void n();
16 }
```

[5] *local* and *anonymous* classes (see JLS ch. 8), which are not member types, are disregarded

The top-level class A declares a static int field a, and a non-static field b which is an int array. The array b is created with array initialisation that determines the values assigned to b's variables in order. A also declares a member type B, which is a static member class of A. B declares a static method m (where void indicates that it does not return a value) in which a local variable b is declared. As A is declared to implement I, it must implement all methods its supertype I declares (or A must be marked as abstract). The modifier public (or its absence) defines accessibility, which is explained at a later point.

2.1.3. Using types

In summary, types in Java can be used in a program in two manners: statically and dynamically. The first case concerns the type only, while the latter is about instances of a type. Some types can be used in both manners, others only in one. A small example demonstrates this idea:

```
1 String message = "%s was here";
2 message = String.format(message, "I");
3 System.out.println(message);
```

This code consisting of three statements prints text on the console. Two types are used by their name, namely String and System (from the java.lang package). The first line shows how String is used dynamically by instantiating an object of that class, using a *String literal* expression (JLS 3.10.5). In the second line the type String occurs again, but is used statically by invoking the static format method. The last line refers to the System class statically by invoking a method on its out field to print the text.

This example already shows the motivation for distinguishing static from dynamic aspects of types when extending Java with new concepts: String objects are usually considered data, whereas the String class offers functions (e.g. for formatting), i.e. deterministic methods whose output only depends on method parameters. This type therefore can be used in both static and dynamic ways, i.e. both as the type itself and through instances at runtime. Some types, such as System, can only be referred to statically, whereas others are only useful as objects.

2.2. Facets

A type can be used in two ways, as the previous section exemplified. These ways are represented by what we call *type facets* or simply *facets* in the following. A facet determines if and how a type can be used statically or dynamically, i.e. through fields, methods, and member types (if any) that are directly accessible via the type itself or its objects. As such, we distinguish the static facet from the dynamic facet of a type, i.e. the *mode* of a facet is either *static* or *dynamic*. The remainder of this section aims at rendering these concepts more precise.

2.2.1. Static facets

The static facet of a type constitutes how the type can be used by referring to any field or method whose declaration uses the modifier static. This modifier associates a field or method with a type instead of its instances. The static facet not only comprises declared fields of a type, but extends to supertypes and member types. However, care must be taken with inherited members as these must be accessible in the inheriting type:

Definition 1. A member m of a type V is *accessible* in a type X if one of the following holds:

1. m is declared public
2. m is declared protected and V is a supertype of X or resides in the same package
3. m has no access modifier (it is *package private*) and X resides in the same package as V

This definition is a subset of the definition of member accessibility as per JLS 6.6.1 as it leaves out the case that m is declared private (which is, as public and protected, an *access modifier*). That is because we exclude private members from facets, as they are meant to be used inside a type, but not outside.[6] Facets, however, manifest how types can be used in other places of a program or even in different programs, which is a common scenario when leveraging code packed in libraries. With this in mind we can proceed to define static facets.

Definition 2. The *static facet* of a type T embodied by an interface or (abstract) class encompasses

S1. every static field or method T declares that is
   a) not declared private
   b) not deprecated

S2. every field or method in the static facet of every member type M declared by T where
   a) M is not declared private
   b) M is not deprecated

S3. every field or method x in the static facet of every supertype S declared by T where
   a) S is distinct from java.lang.Object, the root of the Java type hierarchy
   b) x is accessible in T
   c) x is inherited and not hidden by T or belongs to a static member type inherited and not hidden by T

[6] which corresponds to Java's definition of accessibility of private members: these are accessible in the top-level type they are defined in

The static facet of T includes its static fields and methods, but not those declared private (as they are not accessible outside of T) or deprecated, as the corresponding code annotation @Deprecated (from the java.lang package) marks members that should not be used any more (see JLS 9.6.3.6). Due to the motivation for facets that is based on usage of types, this annotation is taken into account for defining static facets.

The above definition is transitive, as a type's static facet is based on that of member types (S2) and supertypes (S3). A type inherits members of a supertype, which contribute to the static facet. Also, static fields or methods of a member type (of arbitrary depth) of T can be accessed statically via the enclosing type, which means they can be accessed via T. Therefore, they belong to T's static facet. These two kinds of contribution are detailed in the following.

Facets implied by member types and supertypes

Static class, interface, and *Enum* types (a special class type[7]) as member types of a type T can contribute to T's static facet. All fields and methods in their static facet also belong to T's static facet. However, non-static member types must not declare any static fields or methods besides primitive or java.lang.String fields with modifier static final, so-called *constant variables* (see JLS 8.1.3). Non-static member types can only have other static fields or static methods if they are inherited from supertypes.

Inherited static fields and methods of supertypes are as well accessible via T; hence, subtyping needs to be considered when analysing the static facet of a type. However, Java exhibits some peculiarities concerning inheritance of members, which has to be taken with care. It is possible for a Java type T to hide fields and member types of a supertype, which are then not accessible through T and consequently not part of T's static facet:

Definition 3. T *hides* an accessible field f of a supertype if it declares a (static or non-static) field of the same name.

Definition 4. T *hides* an accessible member type M of a supertype if it declares a (static or non-static) member type of the same name.

Hiding does not take into account static modifiers. Irrespective of f being declared static or not, it can be hidden by a static or non-static field of the same name, which can have a different access modifier. The analogue holds for M. Being hidden, f and M can only be accessed through the supertype (see JLS 8.3, 8.5).

Hiding can occur among static methods as well if they have the same signature, i.e. their names are equal, both have equally many parameters, and their parameters have the same type (as per JLS 8.4.8.2). Thus, static methods of a supertype that are hidden

[7] Enums have a fix number of predetermined instances, see JLS 8.9

in T are not in T's static facet.

Analysing a type's static facet can be cyclic, a simple example being that a type T declares a member type N which itself declares T as its supertype by an extends or implements clause. Such a cycle is demonstrated by an example:

```
1 class A {
2   static class B extends A { }
3 }
```

Analysing A's static facet requires testing B's static facet (S2), which takes into account the static facet of B's supertype (S3), which is again A. Cycles can occur with intermingled testing of member types and supertypes and have to be avoided, in particular when the check is performed programmatically, as described in Section 4.3.

Example on static facets

The following code provides an example for the above definition. Classes A, C, and D are assumed to reside in the same package. Asterisks (*) mark members belonging to the static facet of C.

```
1  public class A {
2    static int a; *                // inherited by C
3    public static int b;           // hidden in C
4
5    static class B {               // inherited by C
6      public static int c; *       // accessible in C
7      private static B b;          // not accessible in C
8    }
9  }
10
11 class C extends A {
12   boolean b;                     // hides A.b despite incompatible types
13   static void c() {...} *
14 }
15
16 class D {
17   void a() {
18     C.c();                       // access method on C
19     int i = C.a + C.B.c;         // access field on C and its member type B
20   }
21 }
```

Any method or field in C's static facet can be accessed via the type C at runtime, as the method a in class D illustrates. As C declares a field named b, it hides the field of equal name in its superclass A. While still accessible through the expression A.b in code of class C, A's field b cannot be accessed via C from external code where C is accessible

(e.g. in class D). Hence, C's static facet comprises only the method c and the fields a (inherited from A) and C (from A.B).

Now imagine a class E residing in a different package than A:

```
1 public class A {
2   static int a; *                // inaccessible in E
3   public static int b;           // inherited by E
4
5   static class B {...}           // inaccessible in E
6 }
```

```
1 class E extends A {
2   void a() {...}
3 }
```

E does not declare static members on its own, but it inherits A's static field b. All other members of A are inaccessible outside of A's package. Consequently, E's static facet merely contains b.

The supertype Object

The reason that Object is excluded in S3 a) of the Definition 2 of static facets is that the developer of a class cannot choose whether he wants to inherit from Object and thus provide its functionality.[8] Any class C is a subclass of Object, either directly (when it does not declare any superclass) or via its finite superclass hierarchy (see JLS 8.1.4). The non-private methods of Object (which declares no fields) encompass thread synchronisation, utility methods for cloning, finalisation, and equality testing, class object access via getClass and the representation methods hashcode and toString. Every object of any Java class inherits these methods, which can therefore be used on any object.[9] They do not, however, represent the intended use of a class, which is defined by the developer through the methods declared in that class or inherited from declared superclasses (again, distinct from Object). As facets are defined from the viewpoint of using a type, inherited methods from Object are not considered for this property of types.[10]

2.2.2. Dynamic facets

Just as a type T at runtime can be used to access other static members besides those declared by T (namely any field or method inside T's static facet), objects that are

[8] It is prohibited to restrict the accessibility of methods when overriding, see JLS 8.4.8.3. The inherited or overridden (non-private) Object methods are therefore non-privately accessible on all Java objects

[9] albeit clone will throw an exception on any object whose class does not implement the interface java.lang.Cloneable (see Javadoc of Cloneable and Object.clone())

[10] unless methods of Object are overridden in a type by the developer, who then intends to make them part of his type

instances of T comprise not only the declared non-static fields and methods of T, but every field and method in T's dynamic facet.

Definition 5. The *dynamic facet* of a type T embodied by an interface or (abstract) class encompasses

D1. every non-static field or method T declares that is
   a) not declared private
   b) not deprecated

D2. every field and method x in the dynamic facet of every supertype S declared by T where
   a) S is distinct from java.lang.Object
   b) x is inherited and not hidden by T

The definition resembles that of static facets in D1, but targets non-static fields and methods of T. It deviates in D2, as there is no inherent connection between an object of type T and instances of member types of T as it is for the type T at runtime. While a member type M of T can be accessed through T by an expression T.M (in a class where M is accessible), there is no possibility for the user of a T object t to directly access instances of member types of T through t. Instead, if T is a non-static member class, i.e. an *inner class* (JLS 8.1.3), its objects are bound to an instance of T's enclosing class E (which is required for instantiating T). T instances can access E's fields and methods through the reference E.this. However, this reference can only be used inside T. Therefore, the dynamic facet of enclosing types does not contribute to T's dynamic facet.

Fields and methods can be inherited from supertypes, adding to T's dynamic facet, unless they are hidden in T (see above). As instances of inherited member types are unconnected to T objects and no different than instances of declared member types, inherited member types are not taken into consideration here.

The example below illustrates the concept of dynamic facets. Classes A, C, and E are considered to be in the same package, and the symbol * now indicates fields and methods of B's dynamic facet:

```
1  class A {                          // encloses B, irrelevant
2    int i;
3
4    class B extends C {
5      public int c; *
6      private int b;                 // hides C.b
7
8      public void m() {...} *
9
10     class D {...}                  // inner class of B, irrelevant
```

```
11   }
12 }
13
14 class C {
15   int a; *                          // accessible in B
16   protected boolean b;              // hidden in B
17   private C c;                      // not accessible in B
18 }
19
20 class E {
21   void a() {
22     B b = new A().new B();          // Require A instance for B creation
23     b.m();                          // Access method on b
24     int i = b.a + b.c;              // Access inherited and declared field of c
25   }
26 }
```

Both the enclosing class A and the inner class D are irrelevant for B's dynamic facet. Hiding has to be considered as with static facets, which restricts B's dynamic facet to the fields c and a (which is inherited from C) and the method m. All these can be used in E's method a, where a B object is created on an enclosing A instance before accessing the methods and fields of B's dynamic facet.

2.2.3. Final remarks

The definitions of static and dynamic type facets are both based on fields and methods. They exclude constructors, although these resemble methods in their structure. Despite their similarity to methods constructors are a special case. They can be invoked without an instance of a type (by a new expression), which could motivate counting them as part of the static facet. On the other hand, they can reference the object being instantiated via the reference this, and thus could also be considered a part of the dynamic facet. We disregard them in both definitions however, as they are not relevant for how an object or type is used, but merely required to initialise instances.

Note that facets do not extend over objects that are assigned to fields in the facet, i.e. they do not include the methods and fields of these objects. Doing so would leave the scope of the type and is not intended here. Facets comprise every field and method that can be accessed directly via a type or object. This is, as the previous sections elucidated, more than those fields and methods declared by a type.

Visibility of facets

Facets encompass fields and methods of a type and its member types and supertypes, taking into account hiding and accessibility of members. As such, facets exclude fields and methods with the modifier private. Nevertheless, fields and methods of the static or dynamic facet of a type have different accessibility depending on their modifier. Thus, a

facet is the maximum set of fields and methods accessible via a type T or its instances, but depending on access modifiers, only a subset (which might even be empty) is accessible in the code of a certain class that uses T.

To exemplify visibility of facets as described above, consider again C from the example on page 17. C's static facet contains a method c and a field a, both package private, and another field c from its inherited package private member B. None of these are accessible in E, which means the static facet of C appears to contain no fields or methods to E.

Empty facets

If a type T's static facet does not encompass any fields or methods, i.e. there are no fields or methods accessible through T, T does not have a static facet. We then call T's static facet *empty*. Such a type does not provide any functionality that can be accessed in a static way. An example of an empty static facet is java.lang.Object, which does not declare any static members. Being the root of the Java type hierarchy, it does not have any supertypes and cannot inherit any members that would contribute to its static facet. Similarly, the dynamic facet of a type T is *empty* if it does not encompass any fields or methods. We already exemplified such a facet with java.lang.System in Section 2.1.

If both the static and the dynamic facet of T are empty, this type is practically unusable in a program, as it does not provide any possibility of interaction (besides the methods inherited from Object). Such pathological types are of no interest for our analysis.

In order to extend the Java terminology of types and objects with a concept of components, the next chapter will use type facets as the basis for this concept.

3. Components and their interconnection

We strive to enhance Java as an object-oriented language with more semantics and means to structure programs. Java does not allow to treat types or objects differently with means of the language. Such a differentiation is the goal of this thesis. To that end we distinguish types and objects with respect to state as an externally observable property.

Section 3.1 presents our approach to identifying components as the major types and objects in a program. Focussing on components yields a clearer model of programs. Afterwards, Section 3.2 distinguishes links between components in form of object references and type referrals. References can be qualified and have different influence on coupling of components. The considerations of this chapter converge in interconnection, a component-oriented view on a program providing for better understandability of programs.

3.1. Identifying components

Java allows structuring of programs by means of packages and types, which at runtime also occur as objects, as explained above. However, there is no further notion that allows categorisation of objects and types. Instead, these are all equal on the language level. We show how a distinction of types and objects can be achieved by identifying components as the major types and objects of a program.

Components are a common means to combine and abstract from interrelated elements that fulfil a certain task – not only in computer science, but in many disciplines. Due to its widespread use, there is no absolute definition of the term. However, Szyperski et al. [16] identifies some basic characteristics of components that we will relate to that term as defined here. Our notion differs from that of Szyperski et al. who explore "software components", which can be as abstract as whole programs. We on the other hand define this concept on the lower level of individual programs, which justifies a different approach as described below.

We consider a pragmatic and user-centric approach to separate components from other types and objects of a program: If a type at runtime or object has no *state* that is obvious for a user, it is not a component. Consequently, a component is a type or object with obvious state. State is represented by certain fields of a type or object, but can also become obvious to a user when invoking methods. It is an external property of types and objects, i.e. it is apparent to and can be manipulated by the user. This coarse and

abstract understanding of state will be concretised in the following, but it is the anchoring idea for rendering the ambivalent term component more precise in the context of Java.

User here is a developer who uses a type, either as itself or through instances, in his program. It is the hypothetical person that the developer of a type has in mind when using meaningful names, e.g. for methods and fields of the class or interface he creates. It is the user to whom the developer explains the semantics and effects of his methods and the type in its whole when annotating them with *Javadoc*, Java's means for structured code documentation. Ideally, any user will have the same basic understanding of how a type is to be used, which requires adequate documentation of types.

We use state as the prime criterion for components as Java does not distinguish types and objects with respect to this property. However, typical Java programs often contain *immutable*, i.e. stateless types that serve as collections of functions, and use immutable objects that take the role of data values. Thus, a distinction between stateful and stateless types and objects can provide a categorisation of types and objects. We conceive components as the main elements of a program and deem immutable objects and types as less relevant for a model of the program in whole. The idea of components as major program elements ultimately leads to abstraction from immutable objects and types when viewing a program as a set of interconnected components.

The remainder of this section deliberates on criteria to distinguish components from immutable types and objects, and on how to apply these criteria in order to identify components and immutables. To give an idea of the different steps of reasoning and to introduce terms, we precede the considerations of this section with a brief overview on the presented concepts.

Type classification is the process of segregating components from immutables. This segregation requires a definition of *state*, which we base on facets, i.e. accessible fields and methods of types and objects. Both types and objects can have state, and both the static and dynamic facets of types are taken into account. By tying the definition of state to facets, which are based on members accessible to a user, we achieve that the state is obvious to the user.

We show how to *determine* facets by analysing methods and fields of facets for state. This analysis also takes supertypes into account. Fields and methods of a facet of a type can *imply* state, i.e. indicate that the type or its instances have state. If fields or methods of a facet imply state, then the facet *defines* a component. If no field or method of a facet implies state (and under consideration of subtypes of the analysed type), the facet *defines* an immutable. If a facet of a type defines a component, we call the type or an instance *class component* or *object component*, respectively.

3.1.1. State through fields

While state of types and objects is also obvious through methods, it is based on fields. To achieve a clearer understanding of this term, we relate state to facets. If a field f in the static or dynamic facet of a type T is *mutable*, i.e. it can be used to store values or objects, it implies state.

If f is not declared final, it is mutable. Fields marked as final can only be assigned once, which we do not consider as mutable. Without this modifier, f can freely be assigned values or objects depending on its type. If f is of an array type, it is mutable as well, even if declared final: Objects or values can be assigned to and retrieved from variables of an array assigned to f.

Fields of a class or interface type F are considered mutable as well, irrespective of the final modifier, as long as F's dynamic facet does not define an immutable object. Then, we cannot discount that objects assigned to a field of type F have state (in particular if F's dynamic facet defines a component). State of such an object assigned to f can be manipulated via f as part of the static or dynamic facet of T. Therefore, we consider f mutable too. Summarising this reasoning yields the following definition for state implied by fields:

Definition 6. A field f in T's static or dynamic facet *implies* state of T at runtime or its instances, respectively, if one of the following holds

1. f is not declared final
2. f has an array type
3. f has a class or interface type F and the dynamic facet of F does not define an immutable object

To exemplify state through fields we consider a class A as follows:

```
1 public class A {
2   int a;
3   final int[] b;
4   A friend;
5 }
```

All three fields A declares imply state: a is non-final and thus mutable, and so is the int array b, regardless of its final modifier. The field friend of type A is mutable as well: First, because it is non-final and thus allows assignment. Second, because A's dynamic facet does not define an immutable object (it actually defines a component, as will be obvious later). Thus, the state of an A object assigned to the field friend of another A object is manipulatable via the latter.

The above discussed how fields in the static or dynamic facet of a type can be used to manipulate state of the type or its instances. However, direct state manipulation through fields does not adhere to the well-known programming principle of *information hiding*. There does not seem to be a single consensus on this very general and long-existing term (with Parnas [13] one of the first to advocate it). It has been suggested by Rogers [14] that information hiding in the context of Java indeed pertains to fields. Also, the *Java Code Conventions* [15], which provide general guidelines on how to write "readable" and "clean" Java code, discourage using publicly accessible fields for classes except in "data structures, with no behavior" [15, 10.1].[1] Instead, it recommends so-called get- and set-methods for providing read and write access to fields, which do not need to be accessible then (i.e. they can be private). These methods in turn can imply state, and are subject to the following considerations.

3.1.2. State through methods

If there are no fields in the static or dynamic facet of a type,[2] or if none of them imply state as per above criteria, methods in the static or dynamic facet can imply state of a class or object. This is the case when invoking a method on a class or object affects the class or object in a way that is important to and desired by the user. The following will transform this idea into a more concrete form.

We do not consider interfaces as types at runtime when describing how methods can imply state. Interfaces may only declare non-static methods, which can only be invoked on objects. Therefore, at runtime an interface can only have state through fields, as explained above, and its methods are not part of the discussion below. Only class types can have state implied through methods.

Besides yielding a return value, calling a method can have an *effect* on a class or object in a way that an invocation is related to subsequent invocations. For instance, if the object is a data structure and allows storing objects (such as a `java.util.List l`), this storing is the effect on the object. After passing an object `o` in the invocation of a method whose purpose is to store objects (e.g. `l.add(o)`), the user wants to be able to retrieve it again later through another method (e.g. `l.get(i)` if `i` corresponds to `o`'s position in `l`). In that sense, the invocations of the methods to store and retrieve an object are related. If calling a method exhibits an effect as described above, this method implies state of the class or object.

The following provides a test to determine if classes and objects exhibit state for a user. These classes and objects are not specified any further in order to achieve a more objective rationale.

[1] which in general provide unrestricted and uncontrolled access to the data stored and do not "hide" any information

[2] which then complies to the principle of "Programming to an Interface" (cf. Gamma et al. [5]), which postulates one should not depend on implementation details such as fields

A code extension scheme

In order to find out if a method of an object or class implies state, we introduce a scheme that describes how the code of a program can be extended. Through this scheme invocation of a method always occur on unaltered classes and objects, which appear as if they had never been used before. We begin by defining the two terms *program* and *program outcome*.

Definition 7. A *program* denotes a deterministic sequence of expressions that are evaluated when the program is executed. This sequence encompasses every expression from the invocation of to the returning from the main method of the program code (see JLS 12.1.4). We assume that the program terminates, be it by reaching the end of the main method or due to an unhandled exception or error.[3] We disregard concurrent execution, i.e. the program is single-threaded.

Definition 8. A *program outcome* is an abstract property. In general, it can include every way a Java program can behave when executed, e.g. how it terminates (both successful and with an exception or error) and how it interacts with its environment (e.g. by printing text on the console or using the file system). For the considerations on state we restrict outcomes to termination (successful or not) and console output of a program. We also require that the outcome of a program is deterministic and comparable with other program outcomes. Comparability holds for our definition of an outcome as per the following definition.

Definition 9. Two programs have *equal* outcomes if the following applies:

- both produce the same console output
- both terminate successful or both terminate with an exception or error

Console output can be compared on a character basis. We also assume that exceptions or errors that lead to unsuccessful termination of a program are apparent through console output that can also be compared character-wise.

Testing objects for state

We consider an object o of type T as stateless if in any program every invocation m(..)[4] of a method m in T's dynamic facet can occur on a new instance of T without changing the program outcome. If this code adaption yields different outcome than before the adaption, we consider o as stateful.

State of o through methods can be determined as per the following code extension scheme: If o is created by an expression <init> that obtains an instance of T, we can extend the code of the program by appending every method invocation on o with a reinitialisation of o through <init>. This expression can be a new expression or a method

[3] Invoking System.exit(int) as another means to terminate a program is discounted
[4] ".." indicates a possibly empty parameter list of a method or constructor

call, but its evaluation must not influence the outcome of a program. Thus, evaluating <init> must not change how the program terminates, and no console output must occur during the evaluation of this expression.

The following illustrates how this code extension scheme can be applied in practice:

Original code:[5]

```
T o = <init>;              // Initialise object o
...
o.m(..);                   // Invoke method on o
...
```

Code extended with reinitialisation expressions:

```
T o = <init>;              // Initialise object o
...
o.m(..);                   // Invoke method on o
o = <init>;                // Reinitialise o
...
```

To understand the effect of this scheme consider a String object used in a program. Then <init> can be a call to a constructor of String, or a String literal, which has no influence on outcome:[6]

```
String s = "String";       // Initialise s (<init>="String")
int l = s.length() - 1;    // Get the last character's position
...
char c = s.charAt(l);      // Retrieve the last character
System.out.println(c);     // Print c
...
```

The above code extension rule yields the following:

```
String s = "String";       // Initialise String object s
int l = s.length() - 1;    // Get the last character's position
s = "String";              // Reinitialise s
...
char c = s.charAt(l);      // Retrieve the last character
s = "String";              // Reinitialise s
System.out.println(c);     // Print c
...
```

[5] "..." is a placeholder for arbitrary many (or no) lines of code

[6] We abstract from any exceptions that might occur and be printed on the console during the instantiation of a String object

Reinitialising the String object s does not change the outcome of the program: the same position will be found and the same text will appear on the console without influencing how the program terminates. It might be a different String object that is passed to the method println,[7] and the program will be different after all (e.g. more memory could be required due to reinitialising objects). However, it will have the same outcome, i.e. it will appear to the user to run in the same way as the original program.

The above code extract only exemplifies the described scheme, as one needs to take into account every method in the dynamic facet of String. By doing so one would conclude that code extension for String objects does not change console output or termination of a program in general. Thus, String objects do not have state obvious to the user – they are immutable data objects.

As another example consider the Java List interface from the package java.util. Objects of this interface type allow storing objects[8] in a sequential manner and retrieve them at a later point. Clearly, calling the method add which stores objects in the list is expected to have an effect on the list object. The code extension changes the outcome of the code below (LinkedList is used as a class type implementing the List interface):

```
1 List l = new LinkedList();          // Initialise l (<init>=new LinkedList())
2 l.add(new String("String"));        // Add a String to l
3 ...
4 int i = l.size();                   // Get l's size
5 String s = (String) l.get(i - 1); // Retrieve s as l's last element
6 ...
```

The above code extension rule yields the following:

```
1 List l = new LinkedList();          // Initialise l
2 l.add(new String("String"));        // Add a String to l
3 l = new LinkedList();               // Reinitialise l
4 ...
5 int i = l.size();                   // Get l's size
6 l = new LinkedList();               // Reinitialise l
7 String s = (String) l.get(i - 1); // Retrieve s as l's last element
8 l = new LinkedList();               // Reinitialise l
9 ...
```

Reinitialisation in line 3 changes the program outcome. The list l is instantiated anew through a constructor call which renders the stored element to be virtually lost: It cannot be retrieved from l again after line 3, as this reference then no longer points to the original list. Instead, assuming no other objects have been added to the original list, an exception due to an illegal index (-1) will be thrown and printed on the console.

[7] Due to a mechanism called String *interning* (see JLS 3.10.5), it is actually the same object
[8] Restricting these to a type T by using a generic list List<T> is not considered here

This exception changes the program outcome if the retrieval succeeded prior to code extension. Depending on whether the list l is used again in the remainder of this code the reinitialisation in lines 6 and 8 might be of no significance for the outcome. Nevertheless, the code extension of this program indicates that List objects in Java indeed have state.

Testing classes for state

We consider a class to have no state if the application of the code extension scheme to methods of its static facet does not change the outcome of the program. For classes we need to adapt this scheme, as there is no explicit initialisation mechanism for classes as there is for objects in form of constructors. Classes (and interfaces) are loaded by a so-called *class loader*, which initialises a type. Loading occurs automatically the first time a type is referred to in the code (see JLS 12.2, 12.4.1). There is no built-in mechanism to reinitialise types in Java.

Class and interface reinitialisation is possible through modification of the class loading process, as Jenkov [7] describes. Being rather technical and of no further importance to this thesis, it is not detailed here. Instead, we assume there is a mechanism to reinitialise classes and interfaces in form of a static method reload that is accessible on any class.[9] The only parameter of reload is the name of the class or interface to be reinitialised. After returning from that method, the class or interface will appear to the user (by accessing fields and invoking methods of its static facet) as if it were first loaded.

Below is the extension scheme for a class T. There is no <init> expression involved, as T is loaded on the first referral. Reinitialisation expressions are of the form reload(T). Every time a method is invoked on T, we append that invocation with reload(T).

Original code:

```
1 T.m(..);                  // Invoke method on T
2 ...
```

Code extended with reinitialisation expression:

```
1 T.m(..);                  // Invoke method on T
2 reload(T);                // Reinitialise T
3 ...
```

This scheme corresponds to the one we introduced for objects. We use two examples to illustrate the idea. Let us revisit the method format of String in the following example code:

```
1 String s = String.format("%s", "s");    // Invoke method on String
```

[9] One could imagine it as declared as a public method of java.lang.Object, thus inherited by any class

```
2 ...
3 int i = s.indexOf('s');                  // Retrieve index of 's'
4 s = String.format("%d", i);              // Create String from index
5 ...
```

Applying code extension with class reinitialising yields:[10]

```
1 String s = String.format("%s", "s");     // Invoke method on String
2 reload(String);                          // Reinitialise class String
3 ...
4 int i = s.indexOf('s');                  // Retrieve index of 's'
5 s = String.format("%d", i);              // Create String from index
6 reload(String);                          // Reinitialise class String
7 ...
```

Note that code extension is not applied after invoking the non-static method indexOf on s, which is a String object and not under test. Reloading the class String has no apparent effect to the user: The format method is not influenced by reinitialisation and will yield the same result for the parameters as before code extension. If s were printed on the console, it would produce the same output. Also, the program will still terminate in the same way as before; thus, its outcome remains unchanged. We conclude that the code extension scheme for the method format does not change the outcome. The same holds for any other method of String's static facet. Therefore one can deduce that this class does not have state.

Extending the code below will show that the class java.lang.System indeed exhibits state:

```
1 System.setProperty("set", "Property"); // Set a property...
2 ...
3 String s = System.getProperty("set");  // ...and retrieve it later
4 ...
```

Reinitialising System after each method invocation on this class alters the program outcome:

```
1 System.setProperty("set", "Property"); // Set a property...
2 reload(System);                        // Reinitialise class System
3 ...
4 String s = System.getProperty("set");  // ...and retrieve it later
5 reload(System);                        // Reinitialise class System
6 ...
```

[10] The correct syntax of reloading would be reload(String.class.getName()), which uses the class object of String to get the full name of that class

After reloading the class System, it is as if the value in line 1 had never been set. Unless the property "set" has been set to the same value as in line 1 by the time line 4 is executed, s is not assigned "Property". Reloading will in general change the outcome of the program, e.g. when printing the value of s on the console. System therefore is a type with state.

These examples illustrate how methods can correlate and have an effect on classes or objects. The following definitions relate state to methods.

Definition 10. A method m in the dynamic facet F of a type T *implies* state of instances of T if there is a program P such that the following holds:

1. P has a deterministic outcome.
2. P includes one or more invocations of m on a T instance o which is obtained through an expression <init>.
3. P does not include invocations of methods of F distinct from m that imply state, unless they occur during evaluation of <init> expressions.
4. P does not include assignments to fields of F, unless they occur during evaluation of <init> expressions.
5. Executing P yields outcome different from executing P extended with reinitialisation expressions.

Definition 11. A method m in the static facet F of a class type T *implies* state of T at runtime if there is a program P such that the following holds:

1. P has a deterministic outcome.
2. P includes one or more invocations of m on T.
3. P does not include invocations of methods of F distinct from m that imply state, unless they occur during evaluation of reload(T).
4. P does not include assignments to fields of F, unless they occur during evaluation of reload(T).
5. Executing P yields outcome different from executing P extended with reinitialisation expressions.

By disallowing assignments of fields and invocations of other methods implying state of T's static or dynamic facet we can relate the change of the outcome to m. However, this restriction excludes evaluation of <init> expressions, which will usually involve field assignments, in particular when <init> corresponds to a constructor invocation. It also excludes evaluation of reload(T), which can therefore also assign fields and invoke methods of T without any restriction.

Discussion

These definitions have subjective criteria, based on the outcome of individual programs and its appearance to the user. Even if they were objective, it would require enumerating possible combinations of method invocations if performed programmatically, which is impractical. Also, programs are often not deterministic with respect to evaluated expressions, as they can have indeterministic input (due to program arguments and a complex and variable execution environment). Automating this test is not the goal here, however. The above definitions merely serve as a rationale for a user that analyses methods for implication of state. This user is ideally the developer of a type, applying the concepts of this thesis to his own code. If one can devise a program that includes the invocation of a method m in question and satisfies above criteria (as done in the examples of this section), m implies state. In Section 4.3 we describe a heuristic to programmatically infer if methods imply state.

Summarising the above reasoning on state, accessible mutable fields can imply state of a class, interface, or object. Methods of the dynamic or static facet may as well imply state, which can be tested by the explained code extension scheme. State through fields and methods is the basic criterion in segregating components from immutables in both a static and dynamic sense. However, we do not define state on a type alone, but take into account its supertypes, as the following deliberates.

3.1.3. Classification and type hierarchy

When analysing the static and dynamic facet of a type and checking for state, one has to consider the type's hierarchy as well, i.e. how it relates to other types with respect to subtyping. The reason for taking into account type hierarchy is twofold: First, we deem the classification as object component, class component, immutable object, or immutable type invariant with respect to subtyping. Second, considering type hierarchy is a necessity for static program analysis.

Invariance of classification

If the static or dynamic facet of a type has already been determined, i.e. it defines either a component or immutable, we consider this classification as invariant with respect to subtyping. As instances of a given type T can be used in a program anywhere a supertype of T is required, types should be substitutable with subtypes.[11] That means, objects can be used in lieu of instances of supertypes.

State of a type or object as the prime criterion for components is an important aspect of usage. In other words, a stateful type or object will usually not behave the same as a type or object without state, even if related through type hierarchy. Thus, we do not consider an object component substitutable for an immutable object, and vice versa. Also,

[11] this requirement is often called *behavioural subtyping*

we conceive that types should relate to supertypes with respect to state. Albeit there is no such substitutability in the code for types as there is for objects, we deem state (or lack thereof) as an important external property that should be preserved by subtypes. In summary, subtypes of class components should be class components. If instances of a type are object components, instances of subtypes should too. The equivalent should hold for immutable types and objects: Subtypes of immutable types should be immutable, and if instances of a type are immutable objects, subtype instances should also be immutable objects.

Invariance of classification as object or class component is supported by the definition of facets: The static or dynamic facet F of T is based on fields or methods of T, which can be inherited by subtypes (as they are non-private). If a field or method in F implies state of T or its instances, it does so for subtypes of T or instances of subtypes unless the field or method is hidden in a subtype. This is demonstrated by the class A below:

```
1 class A {
2   int i;
3 }
4
5 class B extends A {
6   void print() {
7     System.out.println("i=" + i);
8   }
9 }
```

A defines a mutable field i which implies state of instances of A. This field is inherited by B as a subclass of A. Therefore, i also implies state of B instances. A therefore is an object component. The same holds for B, irrespective of A's classification, due to the inherited field i. This shows that state is a property that can be carried over to subtypes, and supports the requirement of invariance of classification.

Moreover, we require invariance of classification for static code analysis, due to the explained possibility of instances of a subtype occurring in a specific code location requiring an instance of a type T (e.g. as a method parameter). Without knowing that T's classification holds for subtypes, static analysis would not have been possible in the extent described in Section 4.3.

Immutability and type hierarchy

If classification as component or immutable cannot be inferred from supertypes and no fields or method of the static or dynamic facet imply state, we do not consider the type or object to be a component. Instead, without state obvious to the user, the type or object could be seen as immutable. However, one has to take subtypes into account, both known and unknown.

There are no subtypes to consider if the type is declared final, as this disallows subtyping (JLS 8.1.1.2). Otherwise, subtypes might exist, both when the classification takes place and in the future. As we conceive classification as invariant with respect to subtyping, classifying a type affects all subtypes. This might not be desirable, in particular if a non-final class type T is meant to be extended (in the intent of its developer) or if it is an interface. Subtypes of T can declare or inherit further fields and methods, and its static or dynamic facet can yield a different result when checking if a field or method implies state than the corresponding[12] facet of T.

Thus, consider the static or dynamic facet of T where no field or method in the facet implies state. To define an immutable we require that the facet also stipulates corresponding facets of subtypes to define immutables. Again, this is a subjective criterion, based on the intent of the developer of T if subtypes or instances thereof should also be immutable. If he does not perform classification himself, Javadoc of T can convey this intent and provide a basis for decision in this matter, as it does for the question whether methods imply state.

The following code provides an example for the above explanations:

```
1  interface I {
2    public void noop();
3  }
4
5  class A {
6    final int i = 0;
7  }
8
9  class B extends A implements I {
10   boolean b;
11
12   public void noop() {}
13
14   void toggle() {
15     b = !b;
16   }
17 }
```

A, B, and I illustrate a small type hierarchy in which B extends A and also implements I. A's dynamic facet does not contain methods, and its single field i is not mutable and does not imply state as per above. The dynamic facet of I consists of a single method we consider not to imply state (its name suggests no effect at all). Disregarding subtypes, both these facets would define immutables, i.e. objects of type A or B would be immutable. This would carry over to B as their mutual subtype. But B's dynamic facet

[12] The static facet of a subtype of T corresponds to T's static facet, and vice versa (similar for dynamic facets)

contains a mutable field and a method toggle, both implying state of instances of B. State is the criterion for components, thus B objects are components. A and I therefore cannot be classified as immutable, as it would not be invariant regarding B.

Consider now a modified version of the previous example, where A's field i is mutable:

```
1 class A {
2   int i = 0;
3 }
```

The field i implies state of A objects, thus A's dynamic facet defines a component. B inherits i, thus its instances are also object components. Here, we can no longer say that the dynamic facet of B's supertype I defines an immutable, as doing so would contradict the classification of B and would violate invariance of classification.

3.1.4. Object and class components

We now combine the definitions of the previous sections into a list of steps by which facets are determined, leading to the classification of types. These steps are based on either a static or dynamic type facet, respecting type hierarchy.

Let F be a non-empty facet of a type T, where the mode of F is either static or dynamic. Determining F adheres to the following procedure:

1. If the facet F' of a supertype S of T is determined, where F' corresponds to F, then F is *determined* and
   a) *defines a component* if F' defines a component.
   b) *defines an immutable* if F' defines an immutable.
2. Otherwise, if a field in F implies state, then F is determined and defines a component.
3. Otherwise, if a method in F implies state, then F is determined and defines a component.
4. Otherwise, if T is declared final or only stipulates subtype facets corresponding to F to define immutables, then F is determined and defines an immutable.
5. Otherwise, F remains *undetermined*.

If a facet is empty, it cannot be determined, and classification can only be based on the other facet of a type, unless that is empty too. By taking into account the type hierarchy and fields and methods of a facet, the procedure yields that a facet cannot be determined, or that the facet defines a component or an immutable. In the latter case, we can finally fix the terms component and immutable with respect to types and objects:

Definition 12. If T's dynamic facet

- defines a component, an instance of type T is called *object component*
- defines an immutable, an instance of type T is called *immutable object*
- is undetermined, an instance of type T is called *unclassified*

Definition 13. If T's static facet

- defines a component, T at runtime is called a *class component*
- defines an immutable, T at runtime is called an *immutable type*
- is undetermined, T at runtime is called *unclassified*

We opted against "type component" as the name for types whose static facet defines a component, although this would match the term of immutable types better. Our analysis, which is presented in the evaluation part of this thesis (Chapter 4), showed that interface types are usually immutable types as per our definition, and types with state foremostly are class types, suggesting "class component" as a more fitting name.

In the following the term component as envisioned here is related to other notions of that term.

3.1.5. Other component notions

There is no notion of "components" inherent to Java. This absence allowed us to define the components in our own way, based on state of a type or object. Our definition contrasts how Szyperski et al. [16] understand components in that they exclude "(externally) observable state" [16, 4.1.1]. State as per Szyperski et al. is defined similar to our notion (Section 3.1) as an external property, but is not necessarily manipulatable. However, this mismatch should not be overrated. Szyperski et al. focus on "software components", whereas components as defined here can be understood as "program components", suggesting a smaller scope. On the level of programs consisting of interrelated types and objects, obvious state is inevitable to implement functionality that is based on persistent data, e.g. stored in a java.util.List.

Our approach to introduce components in Java resembles the idea of ArchJava, an extension of Java devised by Aldrich et al. [1]. As here, the central concept of ArchJava are components, which Aldrich et al. define as a special kind of object. Thus, types cannot be components, unlike in our understanding of the term component. We revisit ArchJava in the next section.

Components are often connected to *component frameworks* which structure and concert large programs and foster modularity. As such, the term component in these frameworks implies more abstraction and is less granular than the one introduced here, which anchors

components in types and objects. Still, we attempt to compare the different notions to find analogies.

OSGi [11] is an established component framework for Java, dynamically managing *bundles* that can provide a number of *services* defined through interface types. Under the aspect of interaction, services can be related to components in our understanding. Through services bundles provide to and utilise from others, interaction takes place in a component-oriented program as governed by OSGi. However, bundles in this framework are segregated units that are deployed individually. Thus, bundles can as well be related to our notion of components as nodes in a component-oriented model of a program.

Spring [8] is an application framework for Java and focussed on practical business cases with prefabricated solutions. Application frameworks are therefore not as generic as component frameworks. Spring does, however, also exhibit components in the form of *beans*, which are simple classes managed by a so-called *Inversion of Control (IoC) container*. This container instantiates beans and provides references to other beans through *injection* as per the IoC principle (in contrast to objects acquiring references to other objects on their own). Although we conceive no such concept of IoC here, components in this thesis can be compared to Spring beans as the units of composition in a program. Moreover, the next section shows how references injected into beans can be distinguished and related to further concepts of this thesis.

3.2. Component interconnection

Java programs consist of a set of types organised into packages. The code of these types implements the functionality of the program through statements and expressions and allows interaction of objects and types at runtime through access of fields and invocation of methods. During execution, one can see a program as an interlinked network of objects and types at runtime. *Links* are either represented by object references or so-called qualified expression names (see JLS 6.5.6.2) that refer to a type. By concentrating on links that are external to the scope of a component, components become the nodes in the aforementioned network.

Through analysing links in the network view of a program we can determine its *interconnection* with respect to components. Interconnection is a static view on the dynamics of interacting object and class components, differentiating links between components. This distinction is achieved through concepts elucidated in this section, on which the following paragraphs provide an overview. Interconnection aims at a clearer understanding of coupling of components in a program, also employing annotations so as to render semantics of links explicitly in the code. This code annotation is detailed in Section 4.1.

As we distinguish class and object components from immutable types and objects, we also distinguish references to object components from references to immutables objects,

and referrals to class components from referrals to immutable types. This distinction allows for a clearer view on coupling in Java programs than with concepts inherent to this language, and is the first step in analysing interconnection of components in a program.

In the second step, we add a temporal aspect to links in a program and introduce criteria to distinguish references by duration, in particular *transitory* references from others. Furthermore, a reference to an object through a field represents a more durable link than one through a local variable. Also, the semantics of objects *connected* through fields can be augmented with quantitative and temporal restrictions through *connectivity*. This property helps to assess the volatility of connections at runtime from the static perspective of interconnection.

Objects referenced through fields are divided into *subcomponents* and *dependencies*. Distinguishing subcomponents as integral parts of class and object components allows for an aggregated model of a program by stepping away from Java's flat network view of interlinked objects. *Views* aim at this goal as well in that they represent another perspective on components.

The terms and definitions in this section are devised with type-wise static code analysis in mind, which the concepts in this chapter serve. This objective requires some constraints, as the idea is to analyse programs on the level of expressions, inspecting each expression individually. Thus, there is very limited context from which to derive information, which carries over to the definitions below.

3.2.1. Referencing objects

Object references in Java are pointers through which field access and method invocation on an object or access of array variables takes place. To define and analyse the notion of interconnection between components in a Java program, we aim at distinguishing *external* references and referrals to object and class components from those that are not external. Thus, we disregard references and referrals within the *scope* of an object or class component.

Acquiring references

When accessing fields storing objects or invoking methods that return objects (i.e. their return type is a class, interface, or array type), references are *acquired* by a class or object. To decide whether the access or invocation is external we must consider the target type or object, i.e. where the access or invocation occurs.

Invoking a constructor through a `new` expression is another way of acquiring a reference, specifically to the object instantiated. The same holds for a reference to a new array acquired through array creation. References through object instantiation are considered external, as the object did not exist before and thus cannot originate from the scope of

the class or object instantiating it.

While new expressions yield external references, we discount super and alternate constructor invocations (occurring at the beginning of constructors, see JLS 8.8.7.1). Such invocations do not establish new references, but only serve to initialise the current object (referenced through this). For that matter, we also disregard references to the enclosing class of the direct superclass acquired in a *primary* expression a superclass constructor is invoked on.[13] These expressions can occur in constructors, but are disregarded for acquiring of references in object components. These two exceptions for constructors are evident in the following example:

```
1 class Super {
2   class Member {}                  // Requires Super instance for creation
3 }
4
5 class Sub extends Super.Member { // Subtype of an inner class
6   Sub() {
7     (new Super()).super();         // Instantiate Super to invoke super()
8   }
9 }
```

Sub extends the inner class Member of Super. Invoking a super constructor with super() in line 7 requires a reference to an instance of the enclosing class of the supertype, namely a reference to a Super object. This reference is acquired by the primary expression (new Super()). Both acquiring of a Super instance and the subsequent invocation of a super constructor are inevitable to initialise the instance of class Sub, but do not yield external references.

By accessing an array of a class, interface, or array base type, one acquires the reference stored in the accessed variable. Here, arrays are considered as self-contained objects defining their own scope, meaning stored references are also external to where an array is accessed. Thus, arrays are taken into account for acquiring references.

The above outlines ways of acquiring references to objects, which requires a definition of the term external. Type facets as the basis of class and object components are not sufficient for specifying this term as they exclude private members. Nevertheless, facets are the grounding idea for the following definitions.

Definition 14 (Code bodies)**.** Let T be an interface or class type.

- Every static method, static initializer, or static field initializer[14] declared by T or by a member type T declares is a *static code body* of T.

[13] So-called *qualified superclass constructor invocations*, see JLS 8.8.7.1

[14] yielding an initial value or object for a field in field declarations (see JLS 8.3.2)

- Every non-static method, instance initializer, non-static field initializer, or constructor declared by T is a *dynamic code body* of T. If T is an inner class, any dynamic code body of an enclosing type of T is also a dynamic code body of T.

Code bodies are related to facets in that they distinguish static parts of a type, belonging to the type at runtime, from dynamic parts, which belong to instances of the type. This definition also parallels the Definitions 2 and 5 of static and dynamic facets, as member types and enclosing types are taken into consideration. Constructors are listed as dynamic code bodies along with non-static methods and (field) initializers, as all these can reference the current object using this. Note that interfaces cannot declare initializers, constructors, or static methods. Also, non-static methods that an interface declares are implicitly abstract, i.e. do not have a method body. As such, expressions cannot occur in a method declared by an interface. Thus, interfaces cannot acquire references through expressions in methods.

Based on code bodies we can define external access as follows.

Definition 15. Access or invocation of a field or method x from a static code body of a type T is called *external* to T in any of the following cases:

- x is not declared static
- x is not accessed or invoked on T or any of its member types

Definition 16. Access or invocation of a field or method x from a dynamic code body of a type T is called *external* to an instance of T in any of the following cases:

- x is declared static
- x is not accessed or invoked on this or super. If T is an inner class of a class E, x is also not accessed or invoked on E.this or E.super

Both definitions view methods and fields of different mode (i.e. static versus dynamic) as external, just as members of different mode of a type account for the two facets of a type. E.g. accessing a static field from a constructor is considered external, even if the same type T declares the field and the constructor. Access among static code bodies is not external (e.g. accessing a static field from a static method in the same type), and neither access among dynamic code bodies. The second requirement in each definition resembles the definitions of static and dynamic facets. Access within the static or dynamic facet is indeed considered as within the scope of the type or object, i.e. as not external. However, unlike for facets, private fields and methods are considered for external access, as they are accessible inside types and objects.

The definitions diverge with respect to supertypes: While access or invocation on super is considered within the scope of an object, static access or invocation on a supertype

is external as per above definitions. This divergence reflects the understanding that instances of a class type are tied more closely to their supertypes through the special reference super than types are. For types there is no such keyword to refer to supertypes. Types must access supertype members (that they hide and cannot access on themselves) by using the name of the supertype, just as any other type. See below for an example of a class B accessing members of the superclass A that are hidden in A:

```
1  class A {
2    static boolean s;
3    boolean b;
4  }
5
6  class B extends A {
7    int s;                // hides s of A
8    int b;                // hides b of A
9
10   static boolean getS() {
11     return A.s;         // refer to A by name
12   }
13
14   boolean getB() {
15     return super.b;     // refer to super instance using super
16   }
17 }
```

Receiving references

An object or class at runtime can *receive* an object reference through parameters of its methods, an object also through parameters of the constructor that instantiates it. We conceive references received through parameters as external, as argued in the following.

Inspecting a method signature during code analysis does not reveal the origin of a reference received through a parameter of the method. For instance, an object can call a method of its type's dynamic facet, or one of its private methods, thus the origin of passed references would be the object itself. In general, however, a method can be called from anywhere it is accessible. Therefore, we assume the origin of references passed via method parameters to be external.

As we distinguish the static and non-static parts of types with the notion of facets, parameters of private methods can also yield external references. A non-static method of a type can invoke a private static method of the same type. Then, references received through parameters of the method are external, as the method was called from an object of the type, but not the type itself.

References passed through constructor parameters in a new expression always originate from outside the scope of the object being initialised, unless a constructor is invoked from another constructor. However, the caller of a constructor and thus the origin of parameter references cannot be deduced from constructor signatures in our extent of analysis. Therefore, we also regard references through constructor parameters as external.

Obtaining references

The following consolidates acquiring and receiving of references in the definition for *obtaining* of references. Moreover, we define how we can *infer* the type of a referenced object. The type is important for static code analysis in order to decide if a reference points to an object component or not, as we consider immutable objects as less interesting for a component-oriented view on programs.

Definition 17 (References obtained by class components)**.** Let Q and T denote class, interface, or array types and e be an expression occurring in a static code body of a class component C.

C *obtains* a reference to an object o if e is one of the following:

O1. instantiation of o with a new expression that invokes a constructor of a class type Q

O2. creation of o as an array of type Q

O3. access of an array variable storing o, where the base type of the array is Q

O4. external access of a field f which stores o, where f's type is Q

O5. external invocation of a method m with return type Q, where m returns o

In addition, C can also obtain a reference to o if

O6. o is bound to a parameter of type T in a static method declared by C or one of its static member classes

Obtaining references in object components occurs similar to class components, thus the definitions parallel. However, we must now also consider references passed via parameters in constructors of object components, which adds a further case. Also, recall that the reference to an instance of the enclosing class of a superclass (used to invoke super(..)) is not considered obtained.

Definition 18 (References obtained by object components)**.** Let Q and T denote class, interface, or array types, and c be an object component that is an instance of a class C. Let e be an expression occurring in a dynamic code body of C, but not as primary expression in a qualified superclass constructor invocation.

c *obtains* a reference to an object o if e is one of the following:

O1. instantiation of o with a new expression that invokes a constructor of a class type Q

O2. creation of o as an array of type Q

O3. access of an array variable storing o, where the base type of the array is Q

O4. external access of a field f which stores o, where f's type is Q

O5. external invocation of a method m with return type Q, where m returns o

In addition, c can also obtain a reference to o if one of the following holds:

O7. o is bound to a parameter of type T in a non-static method declared by C, or by an enclosing class if C is an inner class

O8. o is bound to a parameter of type T in the constructor declared by C that initialises c

O1 to O5 are repeated from Definition 17. While O1 to O3 directly yield external references, this needs to be demanded explicitly in O4 and O5. The remaining cases describe obtaining of references through parameters. For class components parameters of static methods of static member classes are also considered,[15] just as member types were considered for static facets. Similarly, object components that are instances of an inner class obtain references through methods of their enclosing instance.

It should be noted that access to local variables in a method of T is disregarded in both definitions. References stored in a local variable may have been obtained previously in a method, constructor, or initializer.

Inferring the type of the obtained reference is similar for object and class components. Hence, we combine the explanation for both.

Definition 19. The type of an obtained reference is *inferred* as follows:

- In O1 and O2 we infer T as Q as o is an instance of type Q
- In O3 to O5
  - we infer o's type as T if e is enclosed in one or more *cast* expressions,[16] where the outermost cast expression converts to T
  - otherwise, we infer o's type as Q
- In O6 to O8 we infer o's type as T

In the following example C's dynamic facet defines a component and highlights ways to obtain references (the cases are indicated in parentheses):

[15] recall that non-static member types and interface types cannot declare static methods
[16] a cast expression converts an object to a specific type (see JLS 15.16)

```
1  class C {
2    static Object[] array = new Object[100];    // Q = Object[] (O2)
3    static int count = 0;
4    C friend;
5
6    C() {
7      array[count++] = this;
8    }
9
10   C make() {
11     friend = new C();                         // Q = C (O1)
12     return friend;
13   }
14
15   void m(C in) {                              // Q = C (O7)
16     C c = (C) array[0];                       // Q = Object (O3)
17     c = c.make();                             // Q = C (O5)
18     c = c.friend;                             // Q = C (O4)
19   }
20 }
```

In this example a C instance acquires references through non-static methods and in a static field initializer. Together with C's only constructor the methods are the dynamic code bodies of C. The parameter in of method m exemplifies O7, which is similar to references obtained through constructor parameters (O8). O2 is illustrated by the static array array, which (along with the static field count) has a static field initializer that is a static code body of C. As all instances of C are stored in array upon initialisation in the constructor, m obtains a reference to this in line 16 if called on the C instance first created. However, this cannot be deduced from the array access itself, hence, the reference is considered obtained as per O3. In line 11 an instance of C is created, representing case O1. O4 is shown in line 17, where a reference is obtained via the return value of a method that is invoked externally on another C object. Similarly, a reference is obtained through external field access on another C instance in line 18. Note that if the method or field was invoked on this, i.e. the current object, it would be an invocation within the scope of the current object. Such an invocation is not considered for obtaining of references.

In most cases of the previous example the reference type can be inferred as C directly (as evident in the comments of the example). When accessing the field array in line 16, however, the base type of the stored array is Object. Yet the inferred type is also C due to the enclosing cast expression, which converts to C.

While facets are based on supertype facets, supertypes are disregarded for obtaining of references, as static program analysis occurs on a per-type-basis. The Definitions 17 and 18 reflect this caveat as they only consider declared methods, constructors, and initializers in which references can be acquired and received. Therefore, to achieve a complete

picture of references obtained by an object or class component, one has to consider references obtained by supertypes as well. This is not trivial, as only inherited members should be taken into account. Through field and method hiding and method overriding, members of supertypes can be seen as irrelevant for a type. As such, references obtained in overridden methods or hidden fields and methods should be discounted. In the following, obtained references are considered for a single type only, disregarding supertypes.

Obtaining of references is only defined for components. In the following we use "receiving" and "acquiring" for references unless a concept only applies to obtained references.

Limitations

In practice, determining if a reference is external and does not originate from within the scope of an object or class component can be difficult. With the given definitions static code analysis will not discount all non-external references, as the following example shows for code in a non-static method of a class C (suppose C's dynamic facet defines a component):

```
1  class C {
2    C make() {
3      return null;
4    }
5
6    void m() {
7      C c = this;
8      C ref = c.make();
9    }
10
11   ...
12 }
```

In the non-static method m the current object referenced through this is assigned to c, on which the method make is invoked. Due to the criteria of external method access, this invocation is external, although it actually occurs on the current object. If make is instead invoked on this (either implicitly through make(), or explicitly with this.make()), the reference it returns would have been recognised as from within the scope of the current object. The code as it is exposes limitations one has to cope with in code analysis based on single expressions. It is also notable in the example above that obtained references can be the null reference, pointing to no object. This is not obvious from the invocation of make, however, and thus we also regard null references as obtained.

Another limit of our analysis model is that the same reference can be obtained many times. A straightforward example is an object o stored in array, which is referenced through a local variable. If o is later retrieved by accessing the array at o's specific location, its reference will be considered obtained, although it is known prior to storing

it in the array and could have already been obtained. Moreover, if o was assigned to a variable, this variable might still be accessible at the point where a reference to o is obtained a second time, as in the following example:

```
1 class C {
2   void m() {
3     C[] array = new C[1];
4     C c = new C();
5     array[0] = c;
6     C ref = array[0];
7   }
8 }
```

The obtained C instance (acquired through a new expression in line 4) is again obtained by storing and retrieving it from the array array, after which the instance is assigned to both c and ref. This example of re-encountered references shows that analysis of references is not trivial with static code analysis, in particular without tracing references and variables across expressions and putting them into relation.

3.2.2. Referring to types

Component interconnection is also constituted through static fields and methods of types accessed by class or object components. Here, it is unnecessary to distinguish object and class components, hence the definition below applies to both.

Definition 20. An object or class component *refers to*

- a class type T by externally accessing or invoking a static field or method on T or one of its member types.
- an interface type T by externally accessing a static field on T or one of its member types.

As with references, referrals are also based on external access and invocation. Thus, the notion of facets becomes apparent here too.

As per this definition, it is also considered a referral if a static field or method of a class is accessed by an instance of that class. The example below shows this special case (suppose both A's dynamic and static facet define a component):

```
1 class A {
2   static A one;
3
4   A get() {
5     return one;
6   }
7
```

```
8   static void m() {
9     System.out.println("m()");
10  }
11 }
```

The method get is part of the dynamic facet of A and accesses the static field one, thus refers to its own type A. In the static method m, the class System is referred to for printing text on the console with the method println. Line 9 also features external field access: A obtains a reference to the object assigned to the field out of System, thus also exemplifies reference obtaining as per O3 of Definition 17.

Duration of referral

Classes and interfaces at runtime can be referred to statically by using a qualified expression name, e.g. java.lang.System. When referral occurs from code of a type in another package, it requires adding an import statement to the *compilation unit* (which encompasses all type declarations in a single Java code file) of that type in order to refer to a type with its simple name, i.e. without stating its package.[17] For instance, the simple name of the class java.lang.System is System. As we strive to characterise coupling in a program also in a temporal dimension we are interested in the duration of links. However, duration of coupling between an object or class component and a type it refers to is not as easy to analyse as with references, as illustrated in the following.

While obtaining a reference to an object marks the beginning of coupling with respect to this reference, the loss of the reference (e.g. after returning from a method where the reference was obtained) signifies the end of coupling. However, the start and the end are not as obvious for referrals. One can consider the referral in a particular expression in an object or class component as both the start and the end of the coupling, thus considering the coupling short-lived. This idea seems reasonable when using a qualified name (that states the package of a type) instead of an import statement. One can as well argue that an import statement, which makes the type referable by simple name throughout all types of the compilation unit, constitutes coupling. This would imply more long-term coupling. The second interpretation seems more appropriate if the type is referred to more than once in the compilation unit. We will not offer a final judgement in this matter, but concentrate on references, which provide more possibilities in analysis of duration.

The remainder of this section focusses on references, how they can be used, and what this means in a temporal view. In particular, it is different if an object reference is assigned to a local variable, or if it is assigned to a field of an object or class component. References through local variables are covered in the next section, references through fields follow afterwards.

[17] Types in java.lang can be referred to by simple name without import statements, see JLS 7.3

3.2.3. Transitory references

The idea of transitory references is that references are not kept for long, i.e. they are either not assigned at all or only to local variables. The first case encompasses direct passing of references through method parameters and invocation of a method on a reference without assigning the reference, for instance. Both cases are illustrated in an example.

```
1  class A {
2    void pass(B b) {}
3  }
4
5  class B {
6    A like;
7
8    B(A a) {
9      doSomething(a);
10     a.pass(this);
11   }
12
13   static void doSomething(A a) {...}
14 }
```

B's constructor receives a reference to an object of class A via its parameter. However, it does not retain the reference by assigning it to a field, for instance. Instead, it uses the reference as a parameter for the static doSomething method and to invoke a method pass on it. After the constructor completes and returns, the reference to the A object will no longer be available to the instantiated B object.

A reference in a method, constructor, or initializer is transitory if it is not retained past the execution of the method, constructor, or initializer. As such, the reference must only be used in a controlled way to qualify as transitory. In particular, further *aliases* to the reference (i.e. assigning the reference to local variables, fields, or array variables, binding it to parameters of invoked methods or constructors, or returning it as a method return value) are only allowed in a very limited manner. A useful concept defined by Vitek and Bokowski [17] is alias restriction through *anonymous* methods and constructors, which is slightly simplified here.

Definition 21. A method m is *anonymous* if

- it does not create aliases to the reference this
- any method invoked on this from m is anonymous
- any method overriding m is anonymous

Definition 22. A constructor c is *anonymous* if

- it does not create aliases to the reference this

- any method invoked on this from c is anonymous
- any invoked super or alternate constructor is anonymous

Anonymous constructors are used later in this chapter. Anonymous methods preserve the transitory property of references: The method pass of class A in the last example is anonymous, as it does not create aliases to this. As anonymous methods of an object o do not create aliases to references of o, a transitory reference to o cannot be retained by invoking an anonymous method on it. Other methods of o that are not anonymous can indeed be used for that purpose, as exemplified by the modified pass method:

```
1  class A {
2    void pass(B b) {
3      b.like = this;
4    }
5  }
6
7  class B {
8    A like;
9
10   B(A a) {
11     doSomething(a);
12     a.pass(this);
13   }
14
15   static void doSomething(A a) {...}
16 }
```

Now an alias to this is created by pass, specifically through assigning the reference to the current object to a field of b that will retain this reference past the execution of its constructor. Therefore, the parameter a in B's constructor can no longer be considered transitory, as the invoked method pass is not anonymous.

We combine the above deliberations on aliases and anonymous methods into a definition of transitory references.

Definition 23. A reference to an object o is *transitory* if

T1. o is not assigned to a field

T2. o is not stored in arrays and not used in array initialisation

T3. o is not returned via a return statement of a method

T4. any method invoked on o is anonymous

T5. when passing o through a parameter, the reference to o is transitory in the invoked method or constructor

As consequence of the above definition, transitory references must not be retained through persistent aliases. T1 prohibits establishment of persistent aliases in fields. Assigning to array variables is also disallowed for transitory references, as we consider arrays as self-contained objects (T2). It is also disallowed to use o in array initialisation, which would persist the reference to o in a newly created array. If such an array is assigned to a field, the stored reference is retained there. Similarly, the array might be passed on through method or constructor parameters, and with it the stored reference, which is persistently aliased by its array variable.

If the reference to o is used in a return statement of a method, this also qualifies as persistent aliasing, as the reference will exist past the execution of the method. Hence, T3 disallows returning of transitory references. T4 guarantees that o does not establish any aliases to a reference to itself by only allowing anonymous methods to be invoked on o. Finally, T5 ensures that invoked methods and constructors preserve the transitory property of references.

The requirements for transitory references restrict their assignment to local variables, signifying a very limited availability for the referencing object or class. The next chapter illuminates a means to enhance code with explicit annotations marking transitory references, and a mechanism to check if a program adheres to the given requirements on these references.

In contrast to transitory references, assigning references to fields allows durable access and use, typically when the reference points to an object component that is necessary to retain state of or provide functionality for the referencing component. Before decomposing references via fields and discussing their semantics, we show how the above concepts influence interconnection.

3.2.4. Links relevant for interconnection

Based on the ideas discussed so far, this section states that only certain links are important for interconnection. In particular, we do not consider references to immutable objects and referrals to immutable types for this view on programs. Also, transitory references can be neglected when analysing interconnection.

As immutable objects exhibit no state obvious to the user, they cannot be manipulated, much like values of primitive types. As we discount primitive types when analysing links in the component-oriented view of a program, we also disregard immutable objects in interconnection. Immutable objects can be exchanged freely among components. Components can collectively access fields and invoke methods of shared immutable objects without any implications for the component interconnection. Similarly, immutable types can be discounted. However, there is no way of exchanging or passing types as objects can be passed by references. Instead, types can appear as a collection of methods (usually declared public) that can be used anywhere the type is accessible by its name. Mutual

use of an immutable type is transparent to referring types and objects. An example of such a utility type is the class java.lang.Math, which provides common functions related to mathematics. Therefore, immutable types can also be disregarded in interconnection.

Our analysis on interconnection is focussed on references to object components and arrays. By obtaining such references, a group of components can share state that a mutually referenced object component exhibits. Manipulation of this state through one component in the group will in general be visible to the other components of the group. For instance, consider that the state is implied through a mutable field of the shared object. A value or object assigned to such a field by one component can be directly read or altered by any other component of the group. Moreover, it can affect the other components if their state depends on the shared object. Such manipulation might be undesirable, even impair components in the group if manipulation by others is not taken into account. Therefore, obtained references to object components can induce complex entanglement in a network of components and are a critical factor when examining interconnection in an object-oriented program. Arrays have state similar to object components, as explained in Section 3.1 when relating state to fields. Shared state of a mutually referenced array is manifested by the array variables, through which assigned objects can be read and reassigned arbitrarily, allowing complex interaction of components referencing such an array. Hence, references to arrays are included in the further considerations of this chapter.

Class components that are mutually referred to imply similar entanglement as per the above reasoning. A class component is manifested through a type at runtime, which exists only once, whereas there can be many instances of a type in general.

The effects through commonly referenced object components and arrays can be mitigated when the reference is transitory, which limits sharing of objects. Transitory references can curtail the aforementioned entanglement of components. As interconnection is a long-term view on coupling in a program, transitory references can be disregarded when analysing component coupling.

The following examines references relevant for interconnection, i.e. non-transitory references to object components and arrays, and aims at a distinction of references assigned to fields of object and class components.

3.2.5. Connected components

Assigning an array or object component o to a field of an object or class component signifies a stronger coupling to o than if o is referenced transitorily or through a local variable. If the reference to o is assigned to a local variable, it can no longer be accessed via the variable after the scope of the variable is left, e.g. at the end of a method execution. In contrast, through a field o becomes accessible across multiple executions of methods (as long as no other object or the null reference is assigned to the particular

field). We consider assignment to fields as an indication of stronger coupling between o and the object or class component referencing it.

The above idea leads to the term of *connected* objects. For these we can establish further semantics through restriction of assignments. Java provides little support for limiting assignments of references to fields except through the field modifier final. This modifier requires a field to be assigned upon initialisation of an object or a type. Once assigned, the reference can no longer be changed. We introduce *connectivity* that adds a notion of coupling complexity to interconnection.

Definition 24 (Connected objects)**.** Let o be an object component or array:

- A class component C is *connected* to o if o is assigned to a field f that is accessible from a static code body of C on itself or one of its member types
- An object component c of a type T is *connected* to o if o is assigned to a field f that is accessible from a dynamic code body of T on this or super, or E.this or E.super if E is an enclosing class of T

If C or c is connected to o, o is also connected to C or c, respectively. We also say that C or c has a *connection* to o (and o to C or c) if it is connected to it.

Here, not only declared fields of an object or class component are considered, but also fields of enclosing classes for object components and of member types for class components. Also, inherited fields are incorporated, disregarding hidden fields of supertypes. Before providing an example for connected objects, we explain how connections can be subdivided. A connection can be characterised and restricted in both quantitative and temporal ways, as the following definition shows.

Definition 25. The *connectivity* of a field f with a class, interface, or array type is

- *single* if there is at most one object ever assigned to f
- *permanent* if there is always an object assigned to f
- *constant* if there is a single object that is always assigned to f
- *flexible* if it is neither single, permanent, or constant

Always refers to the runtime and encompasses the *lifetime* of an object or type owning a field. The lifetime starts with the instantiation of an object and ends when it is destroyed during *garbage collection*, which removes objects when no more references to it exist. The lifetime of types encompasses everything between initialisation (after loading) of a type and program termination.[18]

[18] Unless a type is *unloaded* when it is no longer used (see JLS 12.7)

The connectivity constant combines the quantitative and temporal criteria of single and permanent connectivity, respectively. Thus, a constant connection is both single and permanent. Constant connectivity comes close to Java's final modifier, which also requires the field to be assigned when its owning object or type is initialised. However, constant connectivity disallows the null reference to be assigned to a field, which is allowed for a field modified with final.

If the connectivity of a field is single, the coupling is of low complexity, because at most one object is connected through this field. A permanent connection means that an object or class component is always connected, which excludes a field to be null (i.e. a reference to null is assigned to it). Constant connectivity states that exactly one object is always assigned to a field, which further reduces coupling complexity. Flexible connections do not impose any restrictions on assignments to a field, which is considered a more complex coupling than with other kinds of connectivity.

Connectivity should be declared in the code, just as a final modifier restricts assignments to a field explicitly. This declaration is illuminated in the next chapter.

Example

In the example below, connections and their connectivity are illustrated. The connectivity of the fields is indicated in the corresponding comments.

```
1  class C {
2    static C instance = new C();            // Constant connectivity
3    D child = new D(this);                  // Permanent connectivity
4    C friend;                               // Flexible connectivity
5
6    D make() {
7      child.detach();
8      child = new D(this);
9      return child;
10   }
11
12   void set(C c) {
13     friend = c;
14   }
15
16   static class D {
17     static D last;                        // Flexible connectivity
18     private C owner;                      // Single connectivity
19
20     D(C c) {
21       owner = c;
22       last = this;
23     }
24
```

```
25      void detach() {
26        owner = null;
27      }
28    }
29  }
```

C is connected to an instance of itself with constant connectivity through the static field instance. Instances of class C are connected to any D instance they create when the method make (line 6) is called, as the instantiated D object is assigned to the field child (line 3). This field is already initialised with a D object; thus, its connectivity can be considered permanent. A C object can also be connected to another C object through the field friend (line 4). The connectivity of this field cannot be considered constant or permanent. The method set (line 12), which assigns friend, can be invoked with null as parameter. Moreover, friend is null after initialisation. Therefore, flexible connectivity holds for friend.

The static member class D of C declares a non-static field owner (line 18) that is initialised with the C instance passed to its constructor. The reference stored in owner is only changed once when detach is called, setting owner to null. This field exemplifies single connectivity. D is connected to its most recently created instance through the field last (line 17). Although this field is always assigned with a reference to an object, it is initialised with null and therefore exhibits flexible connectivity. As D is a member type of C, C is also connected to D objects through the field last.

Enriching field semantics with connectivity renders coupling dynamics explicitly in the code and helps in analysing coupling complexity as one aspect of interconnection in a program. In the following section we deliberate on how connections through fields can be further distinguished.

3.2.6. Subcomponents and dependencies

Object components and arrays connected to an object or class component can be distinguished further by their relevance to the component. The prime criterion to distinguish object and class components from immutable objects and types is state. An object or class component p exhibits state that is obvious to the user and that can persist across method calls, as Section 3.1 explained. Fields are an obvious means to persist state, specifically non-final fields of a primitive type that can directly be assigned values. A second possibility is through connected objects that retain state, in particular through object components or arrays connected to p.

Arrays can be used to store values or objects by means of their variables, just as non-final fields of p. An object component connected to p can manifest the state of p by its own state. For instance, it can have mutable fields that p assigns. Delegating persistence of state of p to connected arrays or object components is the motivation for dividing connected objects into *subcomponents* and *dependencies*. If a connected array

or object component *manifests* the state of p and is *encapsulated* as per criteria defined in the following, we consider it a subcomponent. We will not further determine the subjective criterion of manifested state, but instead provide an example for it. In the end the developer of a type has to decide whether connected objects manifest the state of a class or not, and if so, encapsulate them properly so as to make them subcomponents.

The following definition enumerates prerequisites of subcomponents based on the field of p to which arrays or object components are assigned. A field that manifests the state of a component can be considered to connect subcomponents if it is encapsulated. Encapsulation in this context means that a subcomponent must not be leaked from the scope of p. The following definition of encapsulation refers to obtaining of references (see Definitions 17 and 18).

Definition 26. A field f of type T of an object or class component p is *encapsulated* if

F1. T is either an array type or T's dynamic facet defines a component

F2. f is declared private

F3. f is not accessed externally

and for any object o assigned to f the following holds:

E1. The reference to o is obtained through a new expression (O1) invoking an anonymous constructor or through array creation (O2).

E2. o is not assigned to an externally accessed field.

E3. o is not stored in an array or used in array initialisation.

E4. o is not returned via a return statement of a method.

E5. Any method invoked on o is anonymous.

E6. o is not used as parameter of a constructor invoked through a new expression.

E7. o is not used as a parameter of an externally invoked method.

If f is encapsulated and its assigned objects are considered to manifest the state of p, then f *connects subcomponents* and any object o assigned to f is called *subcomponent* of p.

The restriction on the type of f in F1 ensures that connected objects are either arrays or object components, which can manifest the state of p. By requiring f to be a private field (F2), access to it from outside the scope of p is constricted. However, if p is a class component, f can still be accessed from instances of that class. Also, other instances of p's class can access f if p is an object component. Both ways of external access of f as a

form of passive leaking are subsumed and prohibited by F3.

E1 ensures that the obtained reference to a subcomponent is known exclusively by p. Exclusivity is achieved through instantiation of an object with a new expression, specifically one that invokes an anonymous constructor. Anonymity ensures that the instantiated object is not leaked by establishing an alias to this, e.g. by assigning it to a field. An exclusive object reference is also achievable through creation of an array. E2–E7 correspond to the requirements T1–T5 of transitory references (Definition 23) in that they describe how aliasing is avoided. However, the above restrictions allow subcomponents to be assigned to fields (E2) and passed via parameters of methods (E7) if the access or invocation is not external.

It should also be noted that the prohibition of leaking only holds while o is connected to p. Thus, if the connectivity of f is constant, leaking of o is disallowed for the lifetime of p. With different connectivity, more than one object (and possibly null) can be assigned to f. Once another object is assigned and becomes a subcomponent of p, o no longer manifests the state of p (assuming o is not also assigned to another field connecting subcomponents). This subtlety shows that objects become subcomponents through the field that connects them. Basing subcomponents on fields simplifies static program analysis and also enables dynamic checks which monitor fields connecting subcomponents. The next chapter shows how subcomponents can be declared and checked in practice.

Due to the required exclusivity, class components do not qualify as subcomponents. There is no possibility to guarantee referral to be exclusive. Therefore, we only consider objects as subcomponents.

The example below of two classes S and Box, whose dynamic facets both define components, demonstrates how instances of S can act as subcomponents of Box objects.

```
1  class S {
2    int value;
3
4    S(int v) {                          // Anonymous constructor
5      value = v;
6    }
7
8    boolean check(int c) {              // Anonymous method
9      return c == value;
10   }
11 }
12
13 class Box {
14   private S sub;                      // private field (F2), connects object component(F1)
15   Box last;
16
17   Box() {
```

```
18      sub = new S(0);                    // obtained new (E1)
19    }
20
21    void pass(Box other) {               // other.sub not accessed (F3)
22      if (other.compare(sub.value)) {    // sub not passed as parameter (E7)
23        sub = new S(sub.value + 1);      // sub not passed as parameter (E6)
24        last = other;
25      }
26    }                                    // sub not assigned to other.sub (E2)
27
28    boolean compare(int c) {
29      return sub.check(c);               // Invocation of anonymous method (E5)
30    }
31  }                                  // sub not stored in to array or returned in method (E3, E4)
```

The state of Box instances is represented by an int number, and the S instance connected through sub (line 14) manifests this state. Thus, the subjective criterion for subcomponents is fulfilled. Also, sub is encapsulated as per the above requirements. Thus, sub connects subcomponents, and S instances are subcomponents of the Box instance they are connected to through sub. Here, the external access (F3) to sub among different Box objects is avoided: In the method pass (line 21), the sub field of the Box object other is not accessed, although it is accessible within the type Box.

The example demonstrates how multiple objects can be subcomponents of the same object or class component through a single field, depending on its connectivity: The field sub is reassigned in line 23 with another S instance, which is acquired newly as per E2. The same holds for the assignment of sub in the Box constructor (line 18). We can derive the connectivity of sub to be permanent, as sub is never null.

Notably, the field value of class S is package private and thus is accessible from other types in the package of S. However, as a Box instance has the only reference to its subcomponent of type S, the value field of that subcomponent cannot be assigned by anyone else. Thus, the encapsulation criteria suffice to protect the state manifested by subcomponents.

Dependencies

We conclude that certain conditions must be met for a connected object to be considered a subcomponent. If an array or object component o is connected through an encapsulated field and manifests the state of an object or class component p, it becomes a subcomponent of p. Then, o is an integral part of p, which can help to form an aggregated view on a program, as described later.

If o is connected through a field that does not fulfil the requirements of encapsulation, or if o is not considered to manifest the state of p, we call o a *dependency* of p. This term

implies that there is a certain coupling between o and p due to the connection through a field. However, o is not as integral to p as if it was a subcomponent. In particular, there are no restrictions for leaking, so dependencies can be passed freely among components. Still, dependencies are arrays or object components and thus not immutable. Therefore, dependencies have to be taken into account for interconnection and play a major role in this model of a program.

The Box example (page 55) also exhibits a dependency: The Box field last is not private, i.e. not encapsulated and therefore cannot be considered to connect subcomponents. This field does not manifest the state of Box instances either, but merely connects these to other Box objects. Objects assigned to last are dependencies of a Box object. Thus, they are not integral to a Box instance and there are no restrictions concerning leaking of objects assigned to last.

Similar encapsulation techniques

The criteria that prevent leaking of subcomponents can also be found in other encapsulation techniques striving to extend Java with further means of structuring and regulation of aliasing.

Vitek and Bokowski [17] envision an encapsulation model that is based on types and packages, disallowing *confined* types to leak from the package they are declared in. This model requires the notion of anonymous methods, which are reused in this work. A confined type must not occur in code outside its package. Confined types realise encapsulation in a static way and can also be checked statically by utilising static type information in a program. Compared to our idea of encapsulation, one can relate confined types to subcomponents: A confined type must not be leaked from a package, whereas a subcomponent must not be leaked from the component it is connected to.

Balloon types introduced by Almeida [2] are based on object *clusters*. A cluster is spanned by a balloon object, which is an instance of a balloon type, and contains a set of objects. The balloon object itself belongs to this set. Furthermore, the set contains all non-balloon objects that are referenced from an object in the set. As an object that is a subcomponent must only be referenced by the component it is connected to, a balloon object must only be referenced once. Thus, both subcomponents and balloon objects are referenced exclusively. However, it is also required for a balloon object that objects inside its cluster must only be referenced by other objects inside the cluster, and thus not leak from the balloon. For simplicity, we do not demand a similar restriction for subcomponents, i.e. the objects a subcomponent references can be passed freely. Therefore, subcomponents do not span a balloon in the sense of [2].

Categorising aliases is the main mechanism for encapsulation as per Noble et al. [10], which allows fields, variables, and method and constructor parameters to be associated with *aliasing modes*. Some of the concepts in this thesis can be related to these modes.

Fields that connect subcomponents correspond to fields with the aliasing mode *representation*. The idea of representation is that objects referenced in this mode are used to represent the state of the referencing object, just as subcomponents manifest the state of a component. References of the mode representation must not be leaked from the object owning the field, which matches the requirements for subcomponents. Also, these references must not originate from outside the object, just as subcomponent references must be obtained newly (requirement E1 of Definition 26). The mode *var* pertains to mutable objects, which are objects with state in our terms. This mode implies no restrictions, i.e. references of this mode can be exchanged freely among objects. When applied to a field, this mode corresponds to the dependencies of components, i.e. connected arrays or object components without encapsulation restrictions. The mode *argument* denotes immutable objects. However, we cannot relate this mode directly to immutable objects here, which are based on types, more specifically on type facets. An aliasing mode cannot be directly associated with a type, but only with references.

The concept of *unique variables* as used by Boyland [3] parallels fields that connect subcomponents: Both must either be `null` or reference an *unshared object*, i.e. an object referenced exclusively.

Subcomponents and dependencies have counterparts in Spring [8] beans: A bean can specify other beans as *collaborators*, to which references will be injected by the IoC container managing the beans. Collaborators of a bean are not required to be referenced exclusively, thus they match the dependencies of a component. Beans can also define *inner* beans, which are always instantiated newly before injected. Thus, inner beans can be compared to subcomponents, which also must be instantiated for a component. However, Spring does not define further encapsulation requirements for inner beans like those we conceive for subcomponents.

ArchJava [1] also introduces a notion of subcomponents, which are components (i.e. objects) nested in other components, i.e. assigned to the field of another component. Subcomponents in ArchJava must not leave the scope of their component, which corresponds to our encapsulation criteria for fields connecting subcomponents. Interaction with other components in ArchJava occurs via so-called *ports*. Through a port a component is connected to one or more other components. Ports are a collection of methods that a component invokes on other components (*required* methods) and that are invoked on it by other components (*provided* methods). Ports have an identifier that can be used to invoke methods on, similar to fields.

A field of an object or class component to which dependencies are assigned corresponds to a port in ArchJava through which a component is only connected to a single other component. Moreover, this port has no provided methods: There is no inherent back link from dependencies to a component that the dependency could use to invoke methods of the component. Calling methods of other components directly and not through ports is forbidden in ArchJava. In our model, this restriction could be achieved by only allowing

links between components through fields, i.e. through subcomponents and dependencies. Thus, coupling in ArchJava programs is constrained and made explicit through ports. ArchJava uses a more restrictive, but also less flexible component model than we do.

3.2.7. Views on components

The state of object or class components of a type T can be implied by fields of the dynamic or static facet of T, respectively. As deliberated in Section 3.1.2, the state can also be implied by methods. The set of methods in a facet of T that defines a component provides an interface to the component, enabling interaction with the component. This *primary* interface is often not the only way to perceive or manipulate the state of a component. In the following, we introduce *views* on components as an alternative means to interact with object components.

The term "view" implies a different perspective on a component than through its primary interface. Thus, the state of a component that is obvious to the user can be perceived differently through views. The notion of perspective corresponds to the eponymous concept of views in relational databases: Views provide an alternative perspective on tables of a database, where the contents of tables (the data) can be considered as the state of a database. Relational views yield an extended, restricted, or alternative perspective on the data in a database, i.e. its state. The same is conceived for views on components.

Views are also related to common design patterns[19] in software engineering. In particular, the *Iterator* pattern is the grounding idea for views. It is based on object structures that allow for a sequential traversal of stored elements. A java.util.List is an example for such a structure, and we indeed consider an object of that type an object component. At least one of the methods in the dynamic facet of List implies state, as argued in Section 3.1.2.

Lists in Java provide iterators in form of objects of type java.util.Iterator. Iterators allow for a sequential perspective on the list elements as per the above pattern. As such, we can consider the iterator as a view on a list. In particular, it is a restricted view: The elements, which can be accessed randomly on the list itself through the non-static method get, can only be accessed sequentially through an Iterator object. Moreover, iterators are not required to allow modification of the underlying list.[20] An iterator disallowing modification stands for an even further restricted view on the list.

While iterators are a typical example for views, we deem the Iterator pattern as too specific for this concept. Iterators are based on data structures, which many components can be seen as. However, we do not restrict the concept of components to data structures. One can also relate views to the *Adapter* pattern, more specifically to *object adapters*,

[19] cf. Gamma et al. [5]
[20] See Javadoc of java.util.List

which provide an alternative interface to another object. Through an adapter, the operations of the primary interface of an adapted object can be enhanced or disabled. Thus, the Adapter pattern is more generic, providing a better foundation for views than the Iterator pattern. Restricted views in particular can also be compared to the *Proxy* pattern, as proxies are meant to control interaction with objects they substitute, possibly disallowing certain operations on objects.

With these patterns in mind we define the term view as a perspective on the state of an object component. This state can appear in another way, depending if the view provides an extended, restricted, or alternative perspective. The manipulation of the state of the underlying component is generally apparent through the view. This does not mean that a view must be an object component; it can as well be an immutable object. If a view is an immutable object, neither the fields nor the methods of the view imply state. However, if the view is an object component and thus exhibits state, manipulating the state of the view will in general affect the component it is based on. We will provide examples for both immutable views and those allowing for manipulation.

There are no views on class components. As class components are types at runtime, a view object would have to refer to its underlying class component by name, just as any other object or type using the class component. There are no means of the Java language we could employ to tie a view to a class component. Views can only be tied to object components, by requiring the view to be based on certain references to object components as explained below.

To achieve that a view is tied closely to its underlying object component, we require that it is returned from a method invoked on the component, as per the following definition.

Definition 27. An object o is a *view* on an object component p of type T if o is of a class type and

V1. the reference to o is returned from a method m invoked on p

V2. p is one of the following:

a) the enclosing instance of o. Then o can access p via T.this.

b) o itself if the return type of m is distinct from T. Then o can access p via this.

c) assigned to a field f of o with constant connectivity. Then o can access p via this.f.

The type of o as a view is not further specified, i.e. its dynamic facet can define both a component or an immutable.

V1 implies that p controls creation of a view o on itself, as o must be returned from a method of p. This is different from the Adapter pattern, where object adapters can

generally wrap any object, which usually is not aware of its adaption. Thus, a view originates from its component as per this requirement, which is taken into account for interconnection.

As per V2 a), `o` can be an instance of an inner class of `T`. Then `o` can access non-static fields of `p` (including private fields) manifesting `p`'s state, on which `o` provides a perspective. A view represented by an inner class is exemplified in the following example.

```
1  interface Pair {
2    public int high();
3    public int low();
4    public void add(Pair p);
5  }
6
7  class P implements Pair {
8    private int[] values = new int[2];
9
10   P(int low, int high) {
11     values[0] = low;
12     values[1] = high;
13   }
14
15   public int high() {
16     return values[1];
17   }
18
19   public int low() {
20     return values[0];
21   }
22
23   public void add(Pair p) {
24     values[0] += p.low();
25     values[1] += p.high();
26   }
27
28   Pair flip() {
29     return new V();
30   }
31
32   class V implements Pair {
33     public int high() {
34       return P.this.low();
35     }
36
37      public int low() {
38       return P.this.high();
39     }
40
41     public void add(Pair p) {
```

```
42        values[0] += p.high();
43        values[1] += p.low();
44      }
45    }
46  }
```

A Pair represents two numbers, and the classes P and V implement this interface. The dynamic facet of Pair defines a component due to the method add, which implies state, and thus Pair objects are object components. Its two methods high and low allow retrieval of each value of the pair. When invoking the method flip (line 28) of the class P, a view is returned in form of an instance of the inner class V. The V object represents an inverse perspective on the pair of values, as its high and low methods invoke low and high of P, respectively. Being an inner class, a V instance can access the private values array (line 8) of its enclosing instance. This view is tied to the original pair, i.e. the method add (line 23) invoked on the original pair changes the view. Furthermore, the view allows for manipulation through add (line 41), and adding another pair to the view reflects this change in the values array of the underlying pair.

V2 b) states that p can be o, meaning p represents a view on itself. A view as per V2 b) yields a different perspective through the requirement on the return type of the method m in which the view is created: The return type must not equal T. Thus, if p itself is returned via m, m's return type can only be a supertype S of T. Unless p is cast to a subtype (e.g. again T) after invoking m, it will in general yield a restricted perspective: Only through fields and methods of the dynamic facet of S p's state can become obvious via the obtained view if that appears as of type S. Consider the following example illustrating views through p itself:

```
1  class B {
2    int value;
3
4    void inc() {
5      value++;
6    }
7  }
8
9  class C extends B {
10   void dec() {
11     value--;
12   }
13
14   B incOnly() {
15     return this;
16   }
17 }
```

B and C instances are object components, as both their fields and methods imply state, which is represented by a number. A view obtained on a C object through the method incOnly yields a restricted perspective on the C object, as the return type of this method is B. Through methods of this type one can only increase the value the C object represents, unless a cast to C is performed.

If none of the previous cases in V2 applied, then p must be assigned to a field of o with constant connectivity as per V2 c). Constant connectivity ensures access to the component until the end of o's lifetime and allows the state of p to become obvious through o. The following code exemplifies a view based on class P from the example on pairs (page 61).

```
1  class Dice extends P {
2    Dice(int high, int low) {
3      super(high, low);
4    }
5
6    Sum getSum() {
7      return new Sum(this);
8    }
9  }
10
11 class Sum {
12   private Pair pair;
13
14   Sum(Pair p) {
15     pair = p;
16   }
17
18   int get() {
19     return pair.low() + pair.high();
20   }
21 }
```

Dice instances are object components just as P instances (due to the invariance of classification), which allow obtaining of a view on themselves through the method getSum (line 6). This view is an instance of Sum, whose dynamic facet only contains the method get (line 18), which yields a sum of the underlying pair. This method does not imply state; it provides read access only. Sum is an example of a view that does not allow for manipulation of the underlying component, and its instances are immutable objects.[21] Nevertheless, manipulating the pair a sum is based on will become apparent through this view, as the sum is tied to the pair.

Methods yielding views on a component should be declared explicitly, as perspective, which is the purpose of views, is a subjective criterion. Section 4.1 explains how to

[21] Due to the invariance of classification, we require the same for instances of subtypes of Sum

achieve declaration of views through method annotations. The next section will combine the introduced concepts including views in a model of a program based on components.

3.2.8. Interconnection in summary

Interconnection is a static perspective on a program and meant to reflect coupling on a long-term basis. Identifying components as the interacting types and objects at runtime leads to a more concise and enriched model of a program. Immutable objects and types as passive elements are of minor interest for interconnection. Furthermore, we disregard transitory references and focus on connections among components through fields. Adding these temporal aspects helps to abstract from short-term links in a program.

Connections can be decomposed, as explained above, depending on the relevance of the connected object for an object or class component. Subdividing connected objects into external dependencies and integral subcomponents allows to aggregate subcomponents in the components they are connected to. This aggregation leads to a more high-level model of a program, where only links from components to dependencies are reflected.

Aggregation of subcomponents and dependencies on other components correspond to the modularity Szyperski et al. [16] demand for components. Components should be "units of independent deployment" and "third-party composition" with "external context dependencies" [16, 4.1.1, 4.1.5]. We employ the term "user" to denote use and perception of components by others, which can be equated with composition by third-parties. On an abstract level external context dependencies correlate with components that are referenced as dependencies. Technically, an object or class component as a Java object or type requires further types and objects, namely all those it refers to or references, or types that it extends or implements as a type. Thus, components based on types are hard to dislodge from a program and will in general have many "context dependencies". "Independent deployment" can be realised in Java by packing types into libraries. Libraries usually encompass a set of packages and are thus more coarse-grained than single types components are based on. However, recall that Szyperski et al. explore "software components" and therefore deliberate on a higher level than we do by identifying components in programs.

Views obtained on components also constitute coupling in a program, unless a view is represented by an immutable object, which interconnection abstracts from. In the following we consider object components as views. Then one can conceive to merge a view with the component it is based on: Just as subcomponents are connected to their component, there is a strong coupling between a view and its underlying component. If the reference to an object component is already accounted for in interconnection, i.e. it is not transitory, one might want to abstract from views. Views obtained on a component can be seen as lightweight proxies for the component. However, we require that the link to the view is as strong as that to the component, as obvious from the following definition.

Definition 28 (Abstraction from views)**.** Consider an object or class component p and a view v on an object component o. If p has a reference to v that is not transitory this reference can be disregarded in interconnection if

- v is assigned to a field of p with constant connectivity
- o is assigned to a field of p with constant connectivity

Abstraction from views is portrayed in the following example based on the Pair example (page 61):

```
1  class D {
2    final P pair;
3    final Pair inverse;
4
5    D(P p) {
6      pair = p;
7      inverse = pair.flip();
8    }
9
10   Pair get() {
11     return inverse;
12   }
13
14   ...
15 }
```

Here, constant connectivity is guaranteed through the modifier final of the fields pair and inverse and their initialisation in the constructor. In interconnection, only the connection between D instances and P objects through pair is represented, but not the connection to the view connected through inverse. If the method get is invoked externally, meaning one obtains a reference to the view connected through inverse, then this reference has to be considered independently: A Pair object is an object component, and its reference is subject to the above considerations. Thus, the explained semantics of views and their implications for interconnection only hold where its underlying component is referenced.

Ideally, the non-empty facets of all types in a program can be determined, leaving no unclassified types or objects. This is in general not achievable in practice, as evident from the deliberations on the classification of immutable types and objects. Classification of types and objects predetermines subtypes and instances thereof, due to the invariance of classification. Depending on existing or foreseen subtypes, one might refrain from classifying a type or its instances as immutable, leaving the type's static or dynamic facet undetermined. Also, a developer might opt not to classify member types if these are private or only used in their enclosing class. Unclassified types and objects yield a less pristine model of a program, but should be considered nevertheless for the component-oriented view of interconnection.

In the optimal case an obtained reference to an object that is not immutable is either transitory or assigned to a field, connecting the object. This clear distinction requires careful programming and usually does not hold in practice unless with the presented concepts in mind, as shown in Section 4.2. Often, it cannot be decided for every reference to an object whether it is retained (i.e. the object is connected) or not (i.e. the reference is transitory). Such *unclear* references can obscure interconnection, but have to be considered.

In summary, interconnection is based on a component-oriented view on a program, where the components as nodes integrate subcomponents and are linked through referrals to class components and references to dependencies. Also, in certain cases object components that are views can be discounted. These measures yield less nodes in the component network of a program and a representation of programs on a higher level.

4. Components in Practice

This chapter illuminates how components and related concepts can be employed in practice. We begin by showing how code annotations can serve to enhance Java programs with the presented concepts without modifying the language itself or requiring additional code processing or compilation steps. The annotations can be used to declare the classification of types or fields connecting subcomponents, for instance. Component concepts manifested by annotations are summarised and exemplified in a concise example program implementing a compiler.

The second part explains our approach to static program analysis for determination of components and their interconnection. We first describe a tool that implements a heuristic to identify components in a Java program. Afterwards, we introduce a code scanner that determines interconnection of a program, providing a visual result of the interconnection analysis to the user. This scanner utilises the aforementioned annotations. These annotations are also the foundation of dynamic checks introduced in the final section of this chapter. The annotations allow programs to be monitored at runtime and requirements of concepts such as encapsulation or connectivity to be enforced. Dynamic checks are a safeguard for developers using the presented concepts, supporting them in respecting the requirements associated with these concepts.

4.1. Employing component concepts with annotations

To employ the presented concepts in Java programs and render them visually in the code, we use Java *annotations*. These annotations allow for additional semantics in Java code. Using annotations we can represent our concepts in a structured way without extending the Java language. ArchJava [1], a similar approach to enhancing Java with components, introduces new syntactical elements to represent devised concepts in the code.[1] This adaption of the syntax of Java is a major difference to our approach, which relies on the Java language as it is.

We begin our explanations on representing component concepts in the code with an overview on annotations in Java.

[1] This extension of the Java syntax renders ArchJava incompatible with common Java compilers and requires a special compiler for ArchJava programs.

4.1.1. Annotations in Java

An annotation is a special code element attached to parts of the code such as types, fields, or methods. Annotations have an *annotation type* that is declared just as other Java types. Annotation type names start with an @-symbol and are of the form `@<annotation-name>`. Annotation types are associated with and can be restricted to certain program elements, so-called *targets*. Targets of an annotation type can be types, methods, fields, local variables, parameters, and (although not used here) constructors and packages. Annotations of a certain type must only be used with the targets allowed for the type. More than one annotation can occur on any such target, e.g. on a method, but not more than one per annotation type. The following declaration of a class `MyClass` is annotated with an exemplary annotation `@Java`:

```
1 @Java
2 class MyClass {
3   ...
4 }
```

An annotation can have an *element*[2] that has a certain type. We only consider Enum types for elements, i.e. special class types with predetermined instances, so-called *enum constants* (see JLS 8.9.1). Elements can be assigned with a *value* of the element type, i.e. an enum constant in our case. The element value of an annotation can be queried at runtime. An annotation with an element has the form `@<annotation-name>(<value>)` and is illustrated below, where an annotation type `@Code` with an element type that enumerates programming languages is used:

```
1 @Code(JAVA)
2 class MyClass {
3   ...
4 }
```

Static program analysis and dynamic checks as presented in the later sections use the structured information embodied in annotations, either by scanning the code or by checking for annotations at runtime. This runtime retention is a particular advantage of annotations over plain textual comments and even Javadoc, and is crucial for dynamic checks that operate at runtime.

Annotations also help developers to convey their perception of types. In particular, annotations allow coping with subjective criteria inherent to some concepts: The developer of a type can declare whether a type or object exhibits state. He can also annotate an encapsulated field of an object or class component, declaring that the objects assigned to this field manifest the state of the component (which makes the assigned objects subcomponents). By using the annotation types discussed in the following, a developer can

[2] In fact annotations can have several *element-value-pairs*, but these are not used here.

state that he deems these subjective criteria fulfilled.

Explicit declaration of the presented concepts using annotations has another benefit: Annotations are not only part of the code, i.e. available at runtime, but they also become apparent through Javadoc. Annotations applied to types, methods, and method or constructor parameters are obvious through the documentation of these program elements in Javadoc. Therefore, users do not need access to the code in order to understand the developer's intention declared by annotations, but only adequate documentation in form of Javadoc.

4.1.2. Facet annotations

Components are based on the two facets of a type, namely the static and dynamic facet. As types can be a target of annotations, we use one annotation to declare each type facet. It is permitted to use more than one annotation (of different type) on a single code element, thus annotations for each facet can be applied together to classify a type:

The annotation types @StaticFacet and @DynamicFacet target types and provide the following information:

- @StaticFacet describes the static facet of the annotated type
- @DynamicFacet describes the dynamic facet of the annotated type

Both annotation types have an element that state what the respective facet defines, or that it is empty, as shown below:

The element value of @StaticFacet and @DynamicFacet is one of the following:

- COMPONENT, meaning the facet defines a component
- IMMUTABLE, meaning the facet defines an immutable
- NONE, meaning the facet is empty[3]

While classification of types and objects as components or immutables is based on the subjective criterion of state, determining empty facets is straightforward. Empty facets can be identified programmatically, as evident from the Definitions 2 and 5 of facets in Section 2.2. However, we allow explicit declaration of empty facets with the above annotations, which renders this property of types more explicit.

A small example shows how facet annotations can be applied:

[3] In the following, a type with a non-empty static or dynamic facet is also said to *have* a static or dynamic facet, respectively.

```
1 @StaticFacet(NONE)
2 @DynamicFacet(COMPONENT)
3 class Comp {
4   int[] register;
5 }
```

The non-static field register of class Comp is mutable. The dynamic facet of this class defines a component, which corresponds to the element value of the applied annotation @DynamicFacet. The static facet is empty, as reflected by the annotation @StaticFacet.

4.1.3. @Transitory for parameters and variables

The previous chapter explained that short-term references in a program are of marginal interest in the static view of interconnection. We devised requirements for references to be considered transitory, meaning these references are not retained by an object or class. Here, we describe how an annotation @Transitory on parameters and local variables can be used to render transitory references explicitly in the code. References through parameters or assigned to variables can be declared as transitory with such an annotation. The annotation also indicates these references to dynamic checks, which can monitor references declared transitory and verify that these are not retained. Transitory references are allowed to be passed to methods that preserve this property (requirement T5 of Definition 23). Thus, annotating parameters with @Transitory informs users that the reference received through such a parameter is transitory in the invoked method or constructor. Then, it is safe to pass a transitory reference via such a parameter.

The annotation type @Transitory targets parameters and local variables and identifies

- parameters through which a method or constructor receives transitory references
- local variables assigned with a reference that is transitory

@Transitory annotations applied to method or constructor parameters can be seen as an extension of the contract that a method or constructor signature symbolises. This contract describes what a user of the method has to provide (the parameters), what he can expect as result (the return value for methods with return type different from void), and exceptional situations that can occur during execution of the method (thrown exceptions and errors). Parameters annotated with @Transitory add more semantics to this contract by stating that references passed through such parameters are not retained. This extension of method signatures with @Transitory is also apparent through Javadoc.

Parameters of methods and constructors annotated with @Transitory parallel the concept of *borrowed* parameters as introduced by Boyland [3]. Objects referenced by a borrowed parameter must not be assigned to fields, returned from a method (or procedure in terms of [3]), or passed via non-borrowed parameters. Boyland also suggests code annotations to declare parameters as borrowed. The term “borrowed” corresponds to the

semantics of transitory references: A method only borrows an object passed via a parameter annotated with @Transitory from the caller of the method, as it does not retain the reference to the object. In turn, the method may borrow the object to others by only passing it via parameters marked @Transitory.

Both targets of @Transitory annotations are a means for the developer to state his intended use of references. Acquired references must be assigned to a local variable annotated with @Transitory to declare them to be transitory. Transitory references that are not assigned cannot be annotated in that way: It is not possible to associate annotations with bare expressions by which a reference is obtained. Expressions are not among the possible targets of annotation types. In order to mark unassigned references transitory and to enable dynamic checks for them, they must be assigned to a local variable annotated with @Transitory.

The following example of three methods shows how the annotation type @Transitory can be applied in practice:

```
1  class T {
2    void m(@Transitory Pair p) {
3      @Transitory P n = new P(0, 1);
4      merge(p, n);
5      p.add(timesTwo(n.flip()));
6    }
7
8    static void merge(@Transitory Pair p1, @Transitory Pair p2) {
9      p1.add(p2);
10     p2.add(p1);
11   }
12
13   static Pair timesTwo(Pair p) {
14     p.add(p);
15     return p;
16   }
17 }
```

In the method m a reference to a Pair object is received through the parameter p. This reference is declared transitory by using @Transitory on p, and indeed the reference of this parameter is transitory in m: It is only passed via one of the parameters marked @Transitory of the method merge (as per requirement T5), and used to invoke the method add on. This method is anonymous, as evident from its implementation in the example on page 61: add does not create aliases to the Pair instance it is invoked on. Hence, this invocation conforms to requirement T4. The references received through parameters in the method merge are also transitory, as again only the anonymous method add is invoked on them. Moreover, any reference passed to add is transitory in this method. Although the respective annotation @Transitory is not applied to the parameter of add, we can

deduce this property of references from the code of add.

The method m also features a declaration of the local variable n annotated with @Transitory. The reference acquired through a new expression is assigned to n and thus declared transitory. This claim is true, as n is passed via a transitory parameter of merge and used to invoke the method flip on, which is also anonymous (see example on page 61).

The reference acquired through the expression n.flip() in line 5 cannot be directly annotated with @Transitory, as explained above. Declaring this reference transitory would require to assign it to a local variable before passing it to the method timesTwo. However, the reference acquired through the expression n.flip() is not transitory: It is passed to the method timesTwo via a parameter through which this method receives a non-transitory reference: timesTwo adds the passed pair to itself and returns it. Returning the pair creates an alias to the reference of the pair and conflicts with requirement T3 of transitory references.

Restrictions on using @Transitory

In order to identify references acquired in expressions as transitory, local variables annotated with @Transitory can be used, as stated above. Recall that static program analysis as conducted here is based on single expressions. This analysis model requires that assignment of acquired references occurs in the initialisation of such a local variable. Only then static analysis can associate the transitory property with the reference. Hence, the following is no legal use of @Transitory:

```
1 Pair p = new P(0, 1);
2 @Transitory Pair t = p;
```

The local variable t is initialised with the reference assigned to another local variable, although the reference is acquired in the previous line. @Transitory can be applied in the following way:

```
1 @Transitory Pair p = new P(0, 1);
2 Pair t = p;
```

Now p is declared transitory and initialised with a newly created Pair object. Dynamic checks can monitor the reference to this object to verify that the reference is not retained, which is claimed by using @Transitory.

If a reference in a method, constructor, or initializer is declared to be transitory through @Transitory, we conceive this property to hold for other references to the same object. To illustrate this idea, consider a method one as follows:

```
1 Pair one(@Transitory Pair t, Pair p) {
2   return p;
```

```
3 }
```

The method one has two Pair parameters t and p, where t is annotated with @Transitory. The reference received through t is not retained by this method, yet it creates an alias to p by using it in a return statement. Therefore, the reference through p is not transitory. Consider now an invocation of this method using the same reference for both parameters:

```
1 Pair n = new P(0, 1);
2 n = one(n, n);
```

Passing the reference assigned to n as the second parameter means it will be returned by this method. However, as the reference is also passed as the first parameter of one, it must be transitory in this method as per the parameter annotation @Transitory. As the reference is returned from the method, it is not transitory. We conceive the transitory property to prevail. Thus, a developer applying @Transitory must exercise caution when a reference might occur more than once in a method, constructor, or initializer.

4.1.4. Declaring connectivity with @Connected

In the previous chapter we conceived references assigned to fields as a counterpart of references assigned to local variables. Arrays or object components assigned to a field of an object or class component are connected to that component. We also identified different modes of connectivity that restrict assignments to a field. Here, we define an annotation that enables explicit declaration of connectivity. This annotation is recognised by dynamic checks as introduced in Section 4.4.

- The annotation type @Connected targets fields and declares their connectivity.
- The element value of @Connected is one of FLEXIBLE, SINGLE, PERMANENT, and CONSTANT, and declares the connectivity of the annotated field to be flexible, single, permanent, or constant, respectively.

As flexible connectivity imposes no assignment restrictions on a field, the annotation @Connected(FLEXIBLE) has no practical effect, i.e. it is disregarded by dynamic checks. Nevertheless, as the other element values, this annotation serves to express the developer's intention regarding assignments to a field.

To illustrate annotations of type @Connected, we modify the example of connections and connectivity from page 52, declaring connectivity with @Connected instead of comments:

```
1 class C {
2   @Connected(CONSTANT)
3   static C instance = new C();
4
```

```
  @Connected(PERMANENT)
  D child = new D(this);

  @Connected(FLEXIBLE)
  C friend;

  ...

  static class D {
    @Connected(FLEXIBLE)
    static D last;

    @Connected(SINGLE)
    private C owner;

    ...
  }
}
```

4.1.5. Distinguishing subcomponents from dependencies

We introduced subcomponents as arrays or object components connected to encapsulated fields of object or class components. Besides the requirement of encapsulation, subcomponents are based on a subjective criterion: Subcomponents manifest the state of the component they are connected to. The annotation `@Subcomponent` can be applied to fields and declares this criterion to be fulfilled. A such marked field is then subject to dynamic checks, which monitor references assigned to the field and ensure the encapsulation requirements of subcomponents.

The annotation type `@Subcomponent` targets fields and identifies fields connecting subcomponents.

Fields annotated with `@Subcomponent` must have an array type or one whose dynamic facet defines a component, as per requirement F1 of Definition 26. Moreover, the field must be declared `private` as per F2. F1 and F2 can be checked statically. The requirements F3 and E1 to E7 can be verified at runtime by dynamic checks.

We exemplify usage of `@Subcomponent` by revisiting the `Box` example (page 55):

```
class S {
  ...
}

class Box {
  @Subcomponent
  private S sub;
```

```
8
9   Box last;
10
11  ...
12 }
```

Recall that S instances serve to manifest the state of a Box object. As the connecting field sub is encapsulated, we can annotate this field with @Subcomponent, declaring that it connects subcomponents. Applying this annotation to sub also indicates this field to dynamic checks, which monitor any object assigned to sub for leaking from its Box object.

We do not conceive a similar annotation for dependencies. Encapsulation is not required for fields that connect dependencies of an object or class component. Thus, there are no dynamic checks for dependencies. Instead, dependencies can be inferred automatically as fields that connect arrays or object components and that are not annotated with @Subcomponent. In the example above, last as a field of the type Box, whose instances are object components, is a dependency, as the annotation @Subcomponent was not applied.

4.1.6. @GetView for methods yielding views

A view is defined as an object providing a different perspective on an object component. While we do not conceive means to manifest the notion of perspective in the code, we define an annotation that identifies methods yielding views. As per requirement V1 of Definition 27, a view must be obtained from its underlying component by invocation of a method. Such a method can be annotated using the following annotation type:

The annotation type @GetView targets methods and identifies methods that return view objects.

Annotating methods with @GetView allows dynamic checks to verify V2 for objects returned by such methods, as views are required to be tied to the underlying component. Thus, a view is either an instance of an inner class of the object component's type, or has a constant connection to the object component, or is the object component itself.

Consider again the example of the classes Dice and Sum (page 63), where instances of the latter represent views on Dice objects. Such a view is returned by the method getSum, on which @GetView can be applied:

```
1 class Dice extends P {
2   ...
3
4   @GetView
5   Sum getSum() {
6     return new Sum(this);
7   }
```

```
}

class Sum {
  ...
}
```

4.1.7. Annotations and type hierarchy

As per Section 3.1.3 classification of types and objects must be invariant with respect to subtyping. This invariance must also be respected by facet annotations. Therefore, the classification as component or immutable declared by element values of @StaticFacet and @DynamicFacet annotations must not contradict the value of such an annotation on a supertype. E.g. the following must not occur:

```
@DynamicFacet(IMMUTABLE)
class I { ... }

@DynamicFacet(COMPONENT)
class C extends I { ... }
```

It is not a contradiction if a type is annotated to be a component or an immutable using a facet annotation, and a supertype has an annotation @StaticFacet(NONE) or @DynamicFacet(NONE), stating the respective facet is empty. Thus, the following is legal:

```
@DynamicFacet(NONE)
class N { ... }

@DynamicFacet(COMPONENT)
class C extends N { ... }
```

Although Java provides an inheritance mechanism for annotations that target types,[4] we do not make use of this mechanism. Instead, static program analysis and dynamic checks take facet annotations of supertypes into account. Static program analysis also enforces the invariance of classification, and thus reports if facet annotations conflict with this property.

Another aspect of subtyping is overriding of methods as defined below:[5]

Definition 29. A non-static method n of a type T *overrides* an accessible non-static method m of a supertype if

- n and m have the same signature (i.e. equal names and parameter types)

[4] if a type is checked for an annotation of a certain type at runtime, this check will include all supertypes
[5] see JLS 8.4.8.1, 8.4.8.3

- for any exception m declares to throw, n declares to throw an exception of the same type or of a supertype
- n and m both return no result (void), both have the same primitive return type, or the return type of n is a subtype of the return type of m
- n is at least as accessible as m, i.e. the accessibility of n does not occur before that of m in the following list: private, package private, protected, public

Overriding of m by n imposes no restrictions on annotations on m or n or their parameters, meaning these can be annotated independently. However, we conceive that the annotation types @Transitory and @GetView should be repeated by overriding methods, as shown below:

```
1  class Dice extends P {
2    ...
3
4    @GetView
5    Sum getSum() {
6      return new Sum(this);
7    }
8
9  }
10
11 class Dice2 extends Dice {
12   ...
13
14   @GetView
15   Sum getSum() {
16     ...
17     return new Sum(this);
18   }
19 }
```

The method getSum in Dice2 overrides the method getSum inherited from Dice (both methods have equal names, no parameters, no throws clause, equal return type, and both are package private) and repeats the @GetView annotation. Thus, for the overriding method in Dice2 similar requirements as for the overridden method in Dice hold: The view object returned from the overriding method must be tied to its underlying object component (the Dice2 object).

We also require annotations on parameters to be repeated when overriding methods, as in the following example:

```
1  class T {
2    Pair one(@Transitory Pair t, Pair p) {
3      return p;
4    }
```

```
5 }
6
7 class T2 extends T {
8   Pair one(@Transitory Pair t, Pair p) {
9     return new P(t.low(), p.high());
10  }
11 }
```

The annotation @Transitory is repeated on the first parameter t of the method one in T2, which overrides one of T. The overriding method one preserves the transitory property of the reference received through t. This preservation is important when one invokes the method one on an object of type T. As per the substitutability of types, this object can actually be an instance of T2 at runtime. Only if @Transitory annotations on parameters are repeated and the transitory property of passed references is preserved by method overriding, one can safely pass a transitory reference via t. Then, any method of a subtype of T overriding one preserves the transitory property of the parameter t.

The reference received through the parameter p in the method one of T2 is also transitory, as only the anonymous method high is invoked. However, we do not conceive to introduce annotations during overriding: @Transitory is not used for the parameter p in the overridden method in T. Therefore, it is not used for p in T2 either. Similarly, @GetView cannot be applied to overriding methods if it is not present on the overridden method.

Connectivity of a field of a type T annotated with @Connected must be respected by subtypes. As there is no such concept of field overriding in Java, subtypes cannot change the connectivity of a field by means of a @Connected annotation. Dynamic checks monitor assignments to fields also when the field is not declared by a type, but inherited. Private fields are not inherited. Thus, as fields that connect subcomponents are required to be private, subtyping is irrelevant for the annotation type @Subcomponent.

4.2. The Compiler example

In the following we present an example program[6] that implements a simple compiler architecture. This compiler translates code of a fictitious programming language called FPL into an executable form. Instructions of the language FPL are separated by a "." symbol. The syntax of this language is explained by examples in the following. After an overview on the program functionality we highlight how the presented concepts were applied in the code of individual classes of the program. In the next section, the compiler program is used to visualise interconnection of a program.

A Compiler instance takes code files represented by String objects as input. The compiler passes these code files to a Scanner instance, which scans the code and passes each

[6] adapted from an exemplary compiler architecture sketched in [1]

instruction modelled by a Token object back to the compiler. The compiler collects these tokens in a list, which he passes to a CodeGen object. Instances of the class CodeGen model code generators that transform the code as a sequence of tokens into an executable form, a CompileUnit object. After the last token of the code is processed, the code generator returns the generated compile unit to the compiler. The compiler stores generated compile units in a field, where they can be looked up when referred to by other compile units.

The first token of the code is interpreted as the name of the compile unit. The name can be followed by a parameter list enclosed in parentheses (parameters correspond to arguments[7] of a Java program). A token can refer to other compile units (similar to referral to types in Java) by using the name of the compile unit and specifying its parameters. When a token refers to another compile unit, this compile unit is linked to the compile unit being generated.

For the sake of simplicity the listed code does not contain sanity checks of programs (e.g. for cyclic referral among compile units or referring to unknown compile units). Also, we omit all import declarations, which results in an offset of line numbers. By preserving the actual line numbers, we can relate the results of the tools presented in the following sections more easily to the code.

4.2.1. The main class Compiler

The class Compiler controls the interaction in the program and is the entry point for compiling programs. The code of the class Compiler is given below.

The class Compiler

```
17 @StaticFacet(COMPONENT)
18 @DynamicFacet(IMMUTABLE)
19 public class Compiler {
20   @Connected(CONSTANT)
21   private final CodeGen generator;
22
23   @Connected(FLEXIBLE)
24   private Scanner scanner;
25
26   @Connected(SINGLE)
27   private static List<CompileUnit> compiled;
28
29   public Compiler() {
30     generator = new CodeGen();
31   }
32
33   public static List<CompileUnit> getCompiled() {
34     return compiled;
```

[7] String objects passed via parameters to the main method

```
35   }
36
37   public void parse(String input) {
38     if (compiled == null) {
39       compiled = new LinkedList<>();
40     }
41     scanner = new Scanner(input);
42     @Transitory List<Token> list = new LinkedList<>();
43     Token token = scanner.nextToken();
44     while (token != null) {
45       list.add(token);
46       token = scanner.nextToken();
47     }
48     CompileUnit unit = generator.compile(list);
49     compiled.add(unit);
50   }
51 }
```

The constructor of Compiler instantiates a new CodeGen object, which is assigned to the field generator (line 21) of the compiler object being initialised. As this field is declared final, its connectivity is constant, which corresponds to its @Connected annotation. The main functionality is implemented by the method parse (line 37), which accepts code as a String parameter. The static field compiled (line 27) is assigned with a newly created LinkedList on the first invocation of parse. As compiled is never again assigned, its @Connected annotation correctly declares single connectivity for this field. By using the parametrised type List<CompileUnit> for compiled, we restrict stored elements to objects of type CompileUnit.

In line 41 a Scanner instance is created and assigned to the field scanner. As this field is null upon instantiation of the Compiler object, we derive flexible connectivity. In line 42 another LinkedList object is created to collect the Token objects returned from the scanner in the following. This list is assigned to the local variable list. The reference to the list is not retained, but only passed to the compile method of generator via a parameter that is annotated with @Transitory (see the code of CodeGen, page 84). Thus, the reference is transitory and the annotation @Transitory on list is appropriate. Line 49 adds the compile unit returned from the code generator to the list assigned to compiled. As this field is static, all Compiler instances can commonly access any compile unit generated so far.

The static facet of Compiler only contains the method getCompiled (line 33), which allows retrieval of the compile units generated up to the point of calling this method. The following small code example illustrates how a compiler can be used. We provide an exemplary main method of Compiler, where two trivial FPL programs are compiled. The first program prints the parameter it is passed. The second program combines its two parameters and uses the first program to print the result.

```
1  public static void main(String[] args) {
2    String code1 = "Printer(a).print a.";
3    String code2 = "AddNumbers(a,b).c=a+b.Printer(c).";
4    Compiler compiler = new Compiler();
5    compiler.parse(code1);
6    CompileUnit unit = getCompiled().get(0);
7    ...
8    compiler.parse(code2);
9    unit = getCompiled().get(1);
10   ...
11 }
```

Both code files modelled by code1 and code2 are valid FPL code. The first program is split into the tokens "Printer(a)" and "print a", where the first token defines the name of the program (Printer) and its parameter. The second program is split into the tokens "AddNumbers(a,b)", "c=a+b", and "Printer(c)", which defines the name AddNumbers. The last token of AddNumbers refers to the Printer program. The CompileUnit objects generated from the code files are retrieved in lines 6 and 9.

From the above code example we can derive that the static method getCompiled implies state of the class Compiler. As every new compile unit is added to the list that can be accessed via the method getCompiled, the state of the class Compiler is obvious through this method. We will show that getCompiled implies state by applying the code extension scheme introduced in Section 3.1.2 to this method. As a result of this scheme, the code is extended with reload(Compiler) expressions and changes as follows:

```
1  public static void main(String[] args) {
2    String code1 = "Printer(a).print a.";
3    String code2 = "AddNumbers(a,b).c=a+b.Printer(c).";
4    Compiler compiler = new Compiler();
5    compiler.parse(code1);
6    CompileUnit unit = getCompiled().get(0);
7    reload(Compiler);
8    ...
9    compiler.parse(code2);
10   unit = getCompiled().get(1);
11   reload(Compiler);
12   ...
13 }
```

Reloading the Compiler class in lines 7 and 11 also reinitialises its static field compiled, whose assigned list can be accessed via getCompiled. Thus, compile units stored in the list assigned to compiled are lost after reloading. Calling get with parameter 1 in line 10 accesses the second list element. However, the list returned by getCompiled only contains a single compile unit after reloading Compiler, namely that compiled from code2. Due to the illegal index 1, this access results in an exception that did not occur before code

extension, and changes the outcome of the program. The code extension scheme indicates that getCompiled implies state. Therefore, we consider Compiler a class component, which is reflected by the annotation @StaticFacet on the class.

The dynamic facet of the class Compiler includes no fields (all fields of that class are private). The only method parse does not imply state as per the code extension scheme for objects: parse can always be invoked on a new instance of Compiler, as the generated compile units are not stored by instances, but in a static field of that class. Thus, Compiler instances are immutable objects, which is declared by the annotation @DynamicFacet on the class. By using this annotation, we also demand the same property from subtypes as per the invariance of classification. That is, instances of any type that extends Compiler must also be immutable objects.

4.2.2. Scanners produce tokens

Compiling requires scanning of the code and splitting it into single instructions, which are modelled by Token objects. This preprocessing is the task of instances of the class Scanner.

The class Scanner

```
8  @StaticFacet(NONE)
9  @DynamicFacet(COMPONENT)
10 public class Scanner {
11   private final String input;
12   private int offset = 0;
13
14   public Scanner(String input) {
15     this.input = input;
16   }
17
18   public Token nextToken() {
19     int next = input.indexOf('.', offset);
20     if (next > -1) {
21       String token = input.substring(offset, next);
22       offset = next + 1;
23       return new Token(token);
24     }
25     return null;
26   }
27 }
```

The class Scanner declares no static fields or methods. Hence, its static facet is empty. Its dynamic facet contains only the method nextToken. This method returns the next instruction from the code that was passed to the scanner instance through the constructor. Retrieving another token increases the counter in the field offset. After the last token has been retrieved, nextToken will return null henceforth. The behaviour of this method

indicates that this method implies state: Invoking nextToken on a new Scanner instance (which is enforced when applying the code extension scheme for objects) changes the outcome of a program, as this method then always returns the first instruction. Therefore, the annotation @DynamicFacet states that Scanner instances are object components.

The class Token

```
9  @StaticFacet(NONE)
10 @DynamicFacet(IMMUTABLE)
11 public final class Token {
12   public final String value;
13
14   public Token(String value) {
15     this.value = value;
16   }
17
18   public boolean calls(@Transitory CompileUnit unit) {
19     return unit.name.equals(getName());
20   }
21
22   public String getName() {
23     if (!Character.isUpperCase(value.charAt(0))) {
24       return null;
25     }
26     int index = value.indexOf('(');
27     String name = value;
28     if (index > -1) {
29       name = name.substring(0, index);
30     }
31     return name;
32   }
33 }
```

Token objects are immutable, as the annotation @DynamicFacet indicates. The dynamic facet of Token contains a single final field value of type String. String objects are considered immutable, as we have already shown. Therefore, value is not mutable and does not imply state. The methods calls (with a single parameter that is correctly annotated with @Transitory) and getName in the dynamic facet of Token do not imply state, as they provide read access on the token's field value. The static facet is empty, as Token declares no static fields or methods.

4.2.3. Code generators create compile units

A code generator (class CodeGen) inspects the tokens of the code and resolves compile units referred to by tokens. In the end, the tokens and referred compile units are combined in a CompileUnit.

The class CodeGen

```
12 @StaticFacet(NONE)
13 @DynamicFacet(IMMUTABLE)
14 public class CodeGen {
15   public CompileUnit compile(@Transitory List<Token> tokenlist) {
16     Token first = tokenlist.remove(0); // first token = name
17     CompileUnit unit = new CompileUnit(first.getName());
18     for (Token token : tokenlist) {
19       for (CompileUnit c : Compiler.getCompiled()) {
20         if (token.calls(c)) {
21           unit.dependsOn.add(c);
22           break;
23         }
24       }
25       unit.add(token);
26     }
27     return unit;
28   }
29 }
```

Instances of CodeGen accept code as a list of tokens passed to the only method compile of the dynamic facet of CodeGen. In compile a new CompileUnit object unit is instantiated, passing the first token of the code as the name of the new compile unit. The first token is removed from the list in line 16. This manipulation of the list of tokens does not conflict with the annotation @Transitory on the parameter tokenlist through which the reference to the list is received. Transitory references must not be retained, but allow manipulation of the referenced object. In the following, every token of the code is checked for referral to a compile unit that was compiled previously. For that purpose CodeGen refers statically to the class Compiler by invoking the method getCompiled to retrieve the compile units generated already. If one of the tokens is a call to another compile unit, this compile unit is added to the list assigned to the field dependsOn of unit. The static facet of CodeGen is empty. The method compile is non-static and does not imply state: One can translate any program with a new instance of CodeGen. Hence, CodeGen objects are immutable.

The class CompileUnit

```
17 @StaticFacet(NONE)
18 @DynamicFacet(COMPONENT)
19 public class CompileUnit {
20   public final String name;
21
22   @Connected(CONSTANT)
23   protected final List<CompileUnit> dependsOn = new LinkedList<>();
24
25   @Subcomponent
26   @Connected(CONSTANT)
27   private final List<Token> code;
28
```

```
29  public CompileUnit(String name) {
30    this.name = name;
31    code = new LinkedList<>();
32  }
33
34  public void add(Token token) {
35    String ref = token.getName();
36    if (ref != null) {
37      for (CompileUnit unit : dependsOn) {
38        if (token.calls(unit)) {
39          break;
40        }
41      }
42    }
43    code.add(token);
44  }
45
46  @GetView
47  public Iterator<Token> iterator() {
48    return new DeepIterator(code.iterator());
49  }
50
51  class DeepIterator implements Iterator<Token> {
52    private Iterator<Token> called;
53    private Iterator<Token> it;
54    private Token next;
55
56    public DeepIterator(Iterator<Token> it) {
57      this.it = it;
58    }
59
60    public boolean hasNext() {
61      if (next != null) {
62        return true;
63      }
64
65      if (called != null) {
66        if (called.hasNext()) {
67          next = called.next();
68          return true;
69        }
70        called = null;
71      }
72
73      while (next == null && it.hasNext()) {
74        next = it.next();
75        for (CompileUnit dependency : dependsOn) {
76          if (next.calls(dependency)) {
77            called = dependency.iterator();
```

```
78              if (called.hasNext()) {
79                next = called.next();
80                return true;
81              }
82              called = null;
83              next = null;
84              break;
85            }
86          }
87        }
88        return next != null;
89      }
90
91      public Token next() {
92        if (next == null) {
93          hasNext();
94        }
95        Token n = next;
96        next = null;
97        return n;
98      }
99
100     public void remove() {}
101   }
102 }
```

A CompileUnit instance models code in executable form. Code as a sequence of Token objects is retained by the final field code (line 27) of type List<Token>. As this list is instantiated and assigned to code in the constructor of CompileUnit, the connectivity of code is constant, which corresponds to the @Connected annotation of that field. The same assignment restriction hold for the field dependsOn declared in line 23, which is of type List<CompileUnit>. The list assigned to dependsOn stores references to all other compile units referred to by a compile unit. As the tokens of its code constitute a compile unit, we can say that the field code manifests the state of a CompileUnit instance. Moreover, this field is private and satisfies all other encapsulation criteria as per Section 3.2.6. Therefore, we opt to declare code as a field that connects subcomponents through the annotation @Subcomponent.

The static facet of CompileUnit is empty due to the absence of static fields and methods. The dynamic facet contains the protected field dependsOn. This field is mutable and implies state, as the dynamic facet of its type List<CompileUnit> defines a component, as already argued. Other compile units that an instance of CompileUnit depends on are stored in the list assigned to dependsOn. Moreover, the method add (line 34) is part of the dynamic facet of CompileUnit and implies state: Adding a further token to a CompileUnit instance manipulates the state of the instance. This manipulation would become ineffective by reinitialisation as per the code extension scheme: Reinstantiating a compile

unit recreates its token list code; thus, tokens added to a CompileUnit instance (e.g. by the code generator that creates the compile unit) become lost with reinitialisation.

The state of a CompileUnit instance becomes obvious through the field dependsOn and through the method iterator in line 47, which provides a java.util.Iterator object for sequential access on the tokens of a CompileUnit instance. Thus, such an instance exhibits state, which is implied by the field and the method add of the dynamic facet of CompileUnit. Consequently, we consider objects of that class as object components, which is reflected by the annotation of type @DynamicFacet.

The method iterator of the class CompileUnit is annotated with @GetView. This indicates that any Iterator object returned from this method provides a view on a compile unit. The requirements on views as per Section 3.2.7 are fulfilled, as the view must be obtained through the method iterator (V1). Also, the view object is an instance of DeepIterator (line 51), an inner class of CompileUnit, which satisfies requirement V2. The name DeepIterator indicates the kind of perspective the view yields. A DeepIterator instance allows sequential access over all tokens of its underlying compile unit. Moreover, tokens that refer to other compile units are resolved: Instead of returning the referring token, all tokens of the referred compile unit are returned by the method next of DeepIterator (line 91).[8] Resolving of tokens creates a flat perspective on the hierarchy of depending compile units. Thus, one can consider a view represented by a DeepIterator instance as an extended perspective on a compile unit.

4.3. Static program analysis

In this section we demonstrate possibilities to distinguish components from immutable types and objects programmatically. Our concept of components is based on the subjective criterion of state. If the state of a type or object is based on mutable fields, this can be detected programmatically. Methods implying state cannot be determined automatically, as we only devised a theoretical test that indicates state through methods. However, this section presents a heuristic that identifies methods implying state with high probability. This heuristic is implemented by a tool that also tests fields for state, assisting in the classification of types.

The second part of this section introduces a code analysis tool that visualises the interconnection of a program. This analysis also takes the classification of common Java packages (discussed in Chapter 5) into account. The tool provides support for types reused from libraries, where one might not have access to the code of types. With such libraries our concepts cannot be applied using annotations. Instead, we show how to classify types and objects imported from libraries in a different way.

[8] If tokens of referred compile units refer to further compile units, they are not resolved (for reasons of simplicity).

4.3.1. Type facet analysis

Classification of types is supported by type facet analysis. We implemented this analysis with a Java program named TypeFacetsAnalyser. This tool works on a package basis. By specifying a package name, users of the TypeFacetsAnalyser determine the types to be analysed, namely all top-level types in the selected package (including those of subpackages, if desired). The analysis focusses on class and interfaces types, excluding deprecated types, Enum types, annotation types, and java.lang.Throwable and its subtypes. The TypeFacetsAnalyser determines every field and method that belongs to the static or dynamic facet of a type. If no fields or methods qualify as per the Definitions 2 and 5, the program yields that the analysed facet is empty.

The TypeFacetsAnalyser checks every field in the static or dynamic facet of a type to determine if the field is mutable. Checking if a field is mutable and thus implies state is straightforward: Definition 6 identifies three kinds of fields that are mutable, namely non-final fields, fields of an array type, and fields of a type whose dynamic facet does not define an immutable. If the TypeFacetsAnalyser can identify a field that implies state, this field determines the facet, i.e. the facet defines a component.

If no field of a facet implies state, the TypeFacetsAnalyser checks the methods of the facet. Methods can also imply state based on the subjective criterion of state. As this criterion cannot be tested programmatically, we devised a heuristic to identify methods implying state. Inspection of types in commonly used Java packages (e.g. java.lang and java.io) revealed that many methods that imply state suggest an effect on an object or type by their name. Method names should express the purpose of a method and contain a meaningful verb, as the *Java Code Conventions* suggest [15, ch. 9]. We identified a set of recurring verbs prefixing names of many methods that imply state. The TypeFacetsAnalyser considers any method that starts with such a verb as a method that implies state. In the next chapter, we show that this heuristic yields very good results.

We identified 16 prefixes as indicators for methods implying state. Many of them, like add, are common for components that are data structures, e.g. List objects. The prefixes are:

set	put	add	push	append	insert	next	roll
remove	clear	flush	pop	drop	delete	reset	init

The TypeFacetsAnalyser also respects the facet annotations @StaticFacet and @DynamicFacet on types. By using these annotation types, developers can classify types manually. If one of these annotations is found on a type or one of its supertypes (as per the invariance of classification), the annotation's element value (one of NONE, COMPONENT, and IMMUTABLE) will be taken into account for type facet analysis. We exemplified the implications of facet annotations on supertypes in Section 4.1.7, which also showed that facet annotations of types and their supertypes can conflict. The TypeFacetsAnalyser detects and reports these conflicts in facet annotations.

Results of type facet analysis

For every analysed type facet the TypeFacetsAnalyser yields one of the following assessments:

- NONE if the facet is empty
- IMMUTABLE if the analysed type or a supertype has a facet annotation with element value IMMUTABLE
- COMPONENT if the facet defines a component
  - due to a field implying state, or
  - because of a facet annotation with element value COMPONENT on the analysed type or a supertype
- POSSIBLE COMPONENT if the heuristic considers a method of the facet to imply state, which indicates that the facet defines a component
- EXISTS otherwise

If the result of the TypeFacetsAnalyser for a facet is EXISTS, the facet is not empty. However, no facet annotation on the type or a supertype determines the facet, and the program could not identify any fields or methods (using the heuristic) that imply state. Such a facet can still define a component if there is a method implying state that does not fit into the name prefix scheme of the heuristic. On the other hand, the facet might also define an immutable if no method implies state. We refrained from using IMMUTABLE as an assessment of the heuristic. Due to the invariance of classification, a facet defining an immutable must take every subtype into account. The TypeFacetsAnalyser disregards subtypes of analysed types, however, for reasons of simplicity. Moreover, our heuristic discounts methods that do not start with a prefix from a fix set of prefixes. Therefore, our heuristic is a conservative strategy, and facets with the indecisive assessment EXISTS might define components or immutables. Consequently, facets with the result EXISTS or POSSIBLE COMPONENT must be checked manually, as the type facet analysis cannot determine these facets.

The results of type facet analysis for the top-level types of a package (and possibly subpackages) are aggregated by the TypeFacetsAnalyser into classification statistics. We exemplify these statistics by analysing the compiler example from the previous section. First, we analyse the types from the compiler example without facet annotations. Then, the TypeFacetsAnalyser yields the following output (consider the classes of the compiler example reside in the package maxbe.cinamon.example):

```
5 types analysed in maxbe.cinamon.example (excl. subpackages, excl. subtypes of java.
    lang.Throwable)
```

```
class maxbe.cinamon.example.CodeGen: Static facet NONE, dynamic facet EXISTS
class maxbe.cinamon.example.CompileUnit: Static facet NONE, dynamic facet
    POSSIBLE_COMPONENT
class maxbe.cinamon.example.Compiler: Static facet EXISTS, dynamic facet EXISTS
class maxbe.cinamon.example.Scanner: Static facet NONE, dynamic facet POSSIBLE_COMPONENT
class maxbe.cinamon.example.Token: Static facet NONE, dynamic facet EXISTS
Of 5 fully analysed types
  1 (20 %) have a static facet
    of which 0 (0 %) are found to be COMPONENTs
    and 0 (0 %) are found to be IMMUTABLEs
  5 (100 %) have a dynamic facet
    of which 0 (0 %) are found to be COMPONENTs
    and 0 (0 %) are found to be IMMUTABLEs
  and 1 (20 %) have both facets
    of which 0 (0 %) are determined equally (both IMMUTABLE or COMPONENT)
    and 0 (0 %) are determined differently (one IMMUTABLE, other COMPONENT)
```

The type facet analysis finds the static facets of all classes of the compiler example except Compiler as empty. The static facet of Compiler only contains a single method parse that implies state. However, the name of this method does not start with one of the prefixes listed above. Hence, the TypeFacetsAnalyser cannot determine this facet, yielding the result EXISTS. The same holds for the dynamic facets of the classes Compiler, CodeGen, and Token. The facets of the latter two define immutables, as argued above. For CompileUnit, the heuristic suggests the method add as implying state: The name of this method matches one of the prefixes that indicate state. Similarly, the heuristic considers the method nextToken of the class Scanner to imply state, due to its prefix next. Thus, the TypeFacetsAnalyser assesses both CompileUnit and Scanner as POSSIBLE COMPONENT. The dynamic facets of CompileUnit and Scanner indeed define components, as we explained in the previous section.

The statistics show the distributions of classification of non-empty facets. As we did not take facet annotations into account yet, no immutable objects or types could be identified. Also, as none of the facets analysed contains mutable fields, no components could be identified.

Taking facet annotations of the compiler example into account, type facet analysis yields more complete results. Now every non-empty facet of the example classes can be determined, and defines either a component or an immutable:

```
5 types analysed in maxbe.cinamon.example (excl. subpackages, excl. subtypes of java.
    lang.Throwable)
class maxbe.cinamon.example.CodeGen: Static facet NONE, dynamic facet IMMUTABLE
class maxbe.cinamon.example.CompileUnit: Static facet NONE, dynamic facet COMPONENT
class maxbe.cinamon.example.Compiler: Static facet COMPONENT, dynamic facet IMMUTABLE
class maxbe.cinamon.example.Scanner: Static facet NONE, dynamic facet COMPONENT
class maxbe.cinamon.example.Token: Static facet NONE, dynamic facet IMMUTABLE
Of 5 fully analysed types
  1 (20 %) have a static facet
```

```
  of which 1 (100 %) are found to be COMPONENTs
  and 0 (0 %) are found to be IMMUTABLEs
5 (100 %) have a dynamic facet
  of which 2 (40 %) are found to be COMPONENTs
  and 3 (60 %) are found to be IMMUTABLEs
and 1 (20 %) have both facets
  of which 0 (0 %) are determined equally (both IMMUTABLE or COMPONENT)
  and 1 (100 %) are determined differently (one IMMUTABLE, other COMPONENT)
```

The statistics of the type facet analysis are more meaningful when the manual classification represented by facet annotations is considered. However, as the compiler example is a very small program, the results are not representative. In the following chapter we show statistics of larger packages, allowing to search for tendencies with respect to classification of types as components or immutables.

4.3.2. Interconnection analysis

We devised a type-wise static program analysis inspecting the code of Java programs in order to determine the interconnection. This analysis considers the annotations introduced in Section 4.1 through which the presented component concepts can be manifested in the code. The interconnection analysis is implemented in form of a plug-in for the *Eclipse*[9] Integrated Development Environment (IDE). Integration into an IDE fosters regular application of interconnection analysis along the development process of programs employing the introduced component concepts. Ideally, the plug-in helps to devise a component-oriented program design already at an early stage.

Our implementation of interconnection analysis inspects the code bodies of the top-level types of a Java program. The interconnection analysis yields a list of all links from a type or its instances to other types and their instances. These links are distinguished by their duration, i.e. the analysis lists transitory references and those assigned to fields separately. As discussed already, not all references can be clearly identified as transitory or assigned to a field. Hence, the interconnection analysis has to cope with unclear references, which are enumerated in a category of their own. References to object components or arrays through fields are further subdivided into subcomponents and dependencies, which are listed independently. Furthermore, methods are scanned for annotations of type @GetView, meaning a method yields a view on a component.

The analysis is based on code bodies, i.e. declared methods, (field) initializers, constructors, and code bodies of enclosing classes and declared member type. As such, inherited fields, methods, or member types are not considered, and their links are not taken into account. As a type or object can use inherited members just as declared members, links in supertypes also contribute to interconnection. To reflect this contribution, the analysis result lists all supertypes of analysed types.[10]

[9] see www.eclipse.org

[10] but not java.lang.Object as the implicit supertype of types that do not declare supertypes

Interconnection of the compiler program

We revisit the compiler example in order to visualise the results of our interconnection analysis. The screenshot in Figure 4.1 shows how our plug-in visualises the results of the interconnection analysis. The categorised links for both static and dynamic code bodies of analysed types are listed separately. Appending "(static)" to a type name distinguishes static from dynamic code bodies.[11] Although transitory references and immutable types and objects can be disregarded in interconnection, the plug-in lists them for a more complete overview on links. Immutable types and objects can optionally be hidden from the result.

Each top-level element of the result corresponds to the static or dynamic code bodies of a type. The compiler program consists of 5 (top-level) types, which yields 10 top-level elements in the analysis result of the interconnection of this program. Each element is followed by an assessment of the static or dynamic facet of a type. The assessment of a facet is based on facet annotations, but also takes the classification of common Java packages into account (see Section 5.1). For instance, the first element lists links from dynamic code bodies of the class CodeGen. Recall that the dynamic facet of CodeGen defines an immutable, which is declared by an annotation on that class (see example on page 84). This element only contains a category for transitory references: CodeGen declares no supertype, and the other link categories only apply to components.

Any entry in one of the categories underneath the top-level elements represents references to an object of a type T or referrals to T. Each consists of several parts: First, T's simple name, and its package. Then, an assessment of T's static facet for referrals, or of T's dynamic facet for references. Finally, the last part lists line numbers where a referral occurs or a reference is obtained in the code. As an example, consider the first entry in the category "Transitory references" of the element CodeGen. This entry describes transitory references from dynamic code bodies of CodeGen to List objects from the package java.util. As per the third part, the dynamic facet of List defines a component. The remainder states that CodeGen obtains transitory references to List objects in two lines: Through a method parameter marked @Transitory in line 15 and by an unassigned reference to a List object in line 19.

Caveats on interconnection analysis

Our approach to interconnection analysis scans code on an expression basis. As such, references are distinguished with respect to duration directly when they are acquired. An acquired reference that is directly[12] assigned to a local variable annotated with @Transi-

[11] which is also facilitated by a superscript "s" preceding the type name

[12] if the expression through which the reference is obtained is cast before assigning it, this is also considered direct

Figure 4.1.: Interconnection of the compiler program

tory is considered transitory. An example of such a reference can be found in line 42 of the class Compiler (see page 79), where a reference is acquired by instantiating the class LinkedList. This reference is the only transitory reference identified by the interconnection analysis for Compiler, as the results show. Similarly, an obtained reference that is directly assigned to a field can be identified as subcomponent or dependency, depending on the field. This case is exemplified in line 31 of CompileUnit (see page 84), where a new LinkedList instance is assigned to a field. Also, acquired references assigned to a field in its initializer are taken into account, as can be seen in line 23 of the class CompileUnit. The field dependsOn is directly assigned with a reference to a new LinkedList object, and the analysis result lists this reference as a dependency of CompileUnit instances.

If a reference cannot be identified as transitory or associated with a field, it will be listed as unclear reference in the result. For instance, the reference to another compile unit in line 37 of CompileUnit is unclear, as the reference is assigned to the local variable unit, which has no @Transitory annotation. In many cases, unclear references are received through parameters. While the annotation type @Transitory can be used to declare references received through parameters as transitory, there is no similar means to indicate that such a parameter is intended to be assigned to a field. A solution to this shortcoming would be to defer categorisation of references received through parameters. For instance, one could check the first expressions of a method or constructor body to find out if the object referenced by a parameter is assigned to a field. Then, one could consider the object as connected, and the reference could be listed as subcomponent or dependency (depending on the field). However, this procedure exceeds the limits of our expression-based analysis model that categorises references directly.

Assessment of facets of analysed types is mainly based on facet annotations as introduced in Section 4.1. Also, we classified types of common Java packages, and the results of this classification are utilised by the plug-in realising the interconnection analysis. For other Java packages, and for third-party code in general, a developer must conduct the classification on his own. However, annotating types as a means to introduce components into Java programs requires access to the code. The code might not be available when using types from third-party libraries. Instead, the plug-in allows to classify types through a textual classification file, in which each line must be in the form <key>=<value>. For instance, the line java.util.List=C states that the plug-in should consider List objects as object components. When adding the line java.lang.String@static=I to the classification file, the plug-in treats the class String as an immutable type. Entries in the classification file override the classification declared by facet annotations or the classification of common Java packages integrated into the plug-in.

Connectivity of fields is not considered in our implementation of interconnection analysis. Instead, fields are monitored by dynamic checks in order to verify that assignments to fields adhere to the declared connectivity. Dynamic checks also verify encapsulation of fields that connect subcomponents. Two requirements of encapsulation as per Definition 26 are verified during interconnection analysis: A field annotated with @Subcomponent

must be private (F2) and only allow assignment of arrays or object components (F1).

4.4. Dynamic compliance checking

To support developers employing components and related constructs, the previous section presented tools that facilitate static program analysis. The criteria of our concepts impose strict requirements on fields or references to objects which a developer can easily violate even with care during programming. However, some of these criteria are difficult to test statically. We conceive dynamic checks operating at runtime of a program as a safeguard for the developer and a mechanism to ensure proper use of components and related concepts. Dynamic checks monitor certain fields and references at runtime and thus have to be connected to the program under test. The checks should not be part of the program code in order to separate the monitoring logic from the actual program logic. Defining the checks independently of the code also allows reuse in other programs utilising our concepts.

We implement dynamic checks using *aspect-oriented programming* (cf. Kiczales et al. [9]). As the name of this programming paradigm indicates, it is based on *aspects*. Aspects allow extraction of commonly used code parts such as logging or audit logic and combine this logic in a single piece of code. Aspects can be related to different events in the execution of a program (e.g. invocation of methods or occurrence of exceptions) and perform defined actions when such an event occurs. We use AspectJ [12] as an aspect-oriented language based on Java to realise dynamic checks. The following overview on aspect-oriented programming refers to AspectJ and uses terms and mechanism of that language in order to explain aspects in more details.

4.4.1. Overview on AspectJ

An aspect in AspectJ is a collection of *advise* definitions, which states the actions to be performed for an event (also called *advising* of events). An action in AspectJ can be expressed with Java code in the advise *body*, similar to the body of a method. For dynamic checks, actions are used to report violation of encapsulation requirements of subcomponents, for instance. AspectJ allows arbitrary actions to be performed for an event. For instance, dynamic checks could throw an exception for any violation of the criteria we test. It is even possible to stop the tested program in case of a violation. Also, AspectJ can prevent events from occurring.[13] For instance, our dynamic checks could catch field assignments violating declared connectivity, forestall the assignment, and proceed with the program. As this manipulation of the program behaviour can have unpredictable consequences, we opt not do use this feature. Instead, violations are reported by printing a message on the console, which is therefore the only type of action we consider.

[13] using so-called *around-advise*

Events of a program are called *join points* in AspectJ, which are categorised in *pointcuts*. We consider field assignments, method and constructor invocations, and object and class initialisations as join points. The pointcuts for these kinds of events can be denoted with the following syntax in AspectJ:[14]

```
1 set(<Annotations> * *)
2 call(* *(..))
3 call(new(..))
4 initialization(new(..))
5 staticinitialization(*)
```

Line 1 represents any assignment to fields annotated with <Annotations> in a program. The first asterisk symbol (*) in a set pointcut indicates that an assigned field can have any type. The second asterisk represents an arbitrary field name. For instance, the pointcut set(@Subcomponent * *) subsumes every assignment to a field that is declared to connect subcomponents. It is not possible in AspectJ to specify the element value of annotations directly in a set pointcut. Instead, the element value can be checked in the body of advise. We abstract from this restriction here and define the element value directly with the annotation in the pointcut.

In line 2, a pointcut for method invocations is given, where ".." stands for arbitrary many or no parameters. The first asterisk allows any return type of methods captured by the pointcut, the second stands for an arbitrary method name. Both field and return types can be restricted to certain types by using a *type pattern* instead of a generic asterisk. We disregard this possibility for the sake of simplicity. However, we assume that the field and return types in the following are of a non-primitive type.[15] One can also specify the names of fields and methods that set and call pointcuts encompass, which is not required here.

Line 3 shows a pointcut that represents any constructor invocation invoked through a new expression,[16] also with a generic parameter list. Line 4 and 5 symbolise any object initialisation (when a constructor is executed) and class initialisation (occurring when a class is loaded) event, respectively.

Advise definitions

An advise definition specifies pointcuts and the action to be performed for every event captured by the pointcuts. The definition also states if the action is to be performed before or after the event, signalled by the keywords before and after, respectively. In order to apply advise at proper times, the execution of a program for which aspects have been defined is monitored. Before-advise is applied when an event is about to occur, e.g. a

[14] We only present simple join points and do not make use of the full syntax

[15] which would require to use the type pattern java.lang.Object+ (where "+" includes all subtypes of the preceding type) in pointcuts, as we did in our implementation of dynamic checks

[16] Alternate or super constructor invocations are no join points in AspectJ

method is about to be invoked. After-advise is applied when the event is over, e.g. after returning from a method.

Advise definitions can further restrict the join points captured by its pointcuts. Then, actions are only performed for certain events. In particular, advise definitions may constrain the *context* of join points. The context is defined by the Java type from which the event that a join point represents originates, and by the type that is the target of the event. The context can be further constrained with Java code in the body of advise. The body can determine the class or object where an event originates (e.g. the object a method is invoked from), and the type or object that is the target of a join point (e.g. an interface where a field is accessed on). For instance, a pointcut for method calls can be restricted to originate from one object, and to target a different object. To achieve this requirement, the advise body using this pointcut compares the origin and the target objects. Only if a method invocation does not occur on an object itself, this invocation is reported. In programs utilising our concepts it is also possible to require the origin or target to be an object or class component. The dynamic checks we implemented facilitate this restriction in advise bodies by inspecting the origin and target types (and their supertypes) for facet annotations (@StaticFacet and @DynamicFacet).

We use a simplified advise syntax based on AspectJ. Advise definitions start either with before or after and only specify a single pointcut. Context constraints implemented in advise bodies are represented by a targeting clause constraining the target of join points, and a from clause constraining the origin of join points. The action of the advise body is denoted by a report clause, where one can specify further conditions for an event to be reported. The following summarises the advise syntax:

```
before|after:
  <pointcut>
(targeting:
  <target restriction>)
(from:
  <origin restriction>)
report:
  <conditions>
```

The targeting and from clause are optional. Thus, a simple advise definition that reports every method invocation looks as follows:

```
before:
  call(* *(..))
report:
  true
```

The keyword true indicates that the report should occur for every event, i.e. method invocation, and before defines that the report is emitted prior to the invocation. We can as well restrict the advise definition to method invocations on object components:

```
before:
  call(* *(..))
targeting:
  object component p
report:
  true
```

When verifying the encapsulation criteria of subcomponents, we require the notion of external access as per Definitions 15 and 16. External access or invocation can be determined by comparing the target and origin of join points for field assignments and method invocations in advise body. By doing so, one can determine if the event captured by the pointcut, e.g. a method invocation, occurs externally. Here, we simply denote this requirement with the keyword external in the targeting clause. Thus, assignments of externally accessed fields of object components are advised as follows:

```
before:
  set(* *)
targeting external:
  object component p
report:
  true
```

AspectJ allows access to the *arguments* of a join point. Arguments can be the parameters of a method or constructor invocation or the value being assigned to a field. The arguments of a join point can be accessed in the advise body by invoking the method getArgs() on the special reference thisJoinPoint. This method returns a java.lang.Object array. For join points of method or constructor invocations, this array contains the values or objects passed via the parameters of the method or constructor. Variables of the array can also be null, if null is passed as a parameter. For field assignments, the array returned by getArgs() contains only a single element, namely the value or object that is assigned to a field (or null). Thus, we can restrict an advise definition to assignments of null to fields of object components:

```
before:
  set(* *)
targeting:
  object component p
report:
  thisJoinPoint.getArgs()[0] == null
```

The introduced advise syntax is sufficient to illustrate the dynamic checks for connectivity and encapsulation. The syntax allows to explain the idea and important parts of advise definitions that are dense and difficult to read in the actual AspectJ syntax.

4.4.2. Checking connectivity of fields

We distinguished fields by their connectivity, which helps to differentiate degrees of coupling complexity in a program. Different kinds of connectivity restrict assignments to fields in temporal and quantitative ways. We introduced the annotation type @Connected as a means for the developer to declare connectivity of fields in the code. To test this declaration, we present an aspect that monitors fields with a @Connected annotation for assignments. This aspect contains advise definitions for fields of single, permanent, and constant connectivity.[17] Assignments that violate the declared connectivity of fields are reported, just as initialisations of fields with null if the declared connectivity prohibits this value.

Assignments to fields with single connectivity can be advised as follows:

```
before:
  set(@Connected(SINGLE) * *)
report:
  [if the assigned field is not null]
```

The advise definition above causes assignments to fields with the annotation @Connected(SINGLE) to be reported if there is already an object assigned to the field. This event is reported before it occurs, which is indicated by the keyword before. The criterion of this advise definition is actually stricter than required for single connectivity: It reports assigning null to a field if an object is already assigned to the field, meaning it is not null. Single connectivity only disallows assigning objects to a field that is not null. However, if assigning null was allowed, the advise could afterwards not decide if an object was already assigned to a field or not. Keeping track of previous assignments to a field would require further measures. To keep the dynamic checks simple, we realised the aspect for single connectivity in the above way.

Advise for fields with permanent connectivity requires us to check the argument of assignment join points:

```
before:
  set(@Connected(PERMANENT) * *)
report:
  thisJoinPoint.getArgs()[0] == null
```

As a field with permanent connectivity must never be null, the only argument of a set join point is tested to determine if null is assigned to the field. Assigning null violates the declared connectivity and is reported.

Finally, advise to verify constant connectivity can be defined as follows:

[17] Recall that there are no restrictions to test for fields with flexible connectivity

```
before:
  set(@Connected(CONSTANT) * *)
report:
  [if the assigned field is not null] || thisJoinPoint.getArgs()[0] == null
```

This advise definition is a combination of the definitions for single and permanent connectivity: An assignment of a field declared to have constant connectivity will be reported if the field is not null, or if the assigned value is null. Thus, only the initial assignment of an object to such a field can pass this check.

The advise definitions for permanent and constant connectivity monitor assignments; however, they are not sufficient. Both kinds of connectivity disallow the field to be null after the object or class the field belongs to is initialised. Field initialisation is not taken into account yet. Therefore, we define further advise employing the pointcuts initialization(new(..)) and staticinitialization(*):

```
after:
  initialization(new(..))
targeting:
  object o
report:
  [for any field of o with permanent or constant connectivity that is null]
```

```
after:
  staticinitialization(*)
targeting:
  class C
report:
  [for any field of C with permanent or constant connectivity that is null]
```

The first advise definition applies after an object has been initialised, i.e. its constructor has completed execution. Any field of this object with a permanent or constant connectivity declaration that is null violates the connectivity requirements, which will be reported. Similarly, the second advise definition requires that after a class is loaded, any field of the class with permanent or constant connectivity is not null. Otherwise, this connectivity violation is reported too.

Example

The code in Example 4.1 shows how dynamic checks monitor assignments in a program based on the aspect defined above. The portrayed program consists of the single class Connect that declares three non-static fields, one for each of the connectivity kinds single, permanent, and constant. It also contains two static fields declaring permanent and

Example 4.1: The class Connect

```
8  public class Connect {
9    @Connected(SINGLE)
10   Object single;
11
12   @Connected(PERMANENT)
13   Object permanent;                  // Violates connectivity (initialised as null)
14
15   @Connected(CONSTANT)
16   Object constant_;                  // Violates connectivity (initialised as null)
17
18   @Connected(PERMANENT)
19   static Object permanentStatic;     // Violates connectivity (initialised with null)
20
21   @Connected(CONSTANT)
22   static Object constantStatic;      // Violates connectivity (initialised with null)
23
24   public static void main(String[] args) {
25     Connect o = new Connect();
26
27     o.single = new Object();         // OK, initialises field
28     o.single = new Object();         // Violates connectivity (already initialised)
29     o.single = null;                 // Violates connectivity
30
31     o.permanent = new Object();      // OK, initialises field (initialised as null)
32     o.permanent = new Object();      // OK, reassigning possible
33     o.permanent = null;              // Violates connectivity
34
35     o.constant_ = new Object();      // OK, initialises field (initialised as null)
36     o.constant_ = new Object();      // Violates connectivity
37     o.constant_ = null;              // Violates connectivity
38   }
39 }
```

constant connectivity. Violations of connectivity are indicated in comments after each field declaration or assignment.

When executing this program, its main method is invoked. The dynamic checks report violations by printing a message on the console. These messages are of the form <Type>:<Line> (<JoinPoint>): <Message> and indicate the code position of connectivity violations. The dynamic checks emit the following messages for the program Connect:[18]

[18] AspectJ yields incorrect line numbers for join points of initialization and staticinitialization pointcuts

```
Connect.java:0 (staticinitialization(Connect.<clinit>)): Field with PERMANENT
    connectivity initialised with null
Connect.java:0 (staticinitialization(Connect.<clinit>)): Field with CONSTANT
    connectivity initialised with null
Connect.java:8 (initialization(Connect())): Field with PERMANENT connectivity
    initialised with null
Connect.java:8 (initialization(Connect())): Field with CONSTANT connectivity initialised
     with null
Connect.java:28 (set(Connect.single)): Reassigning field with SINGLE connectivity
Connect.java:29 (set(Connect.single)): Reassigning field with SINGLE connectivity
Connect.java:33 (set(Connect.permanent)): Assigning null to field with PERMANENT
    connectivity
Connect.java:36 (set(Connect.constant_)): Reassigning field with CONSTANT connectivity
Connect.java:37 (set(Connect.constant_)): Reassigning field with CONSTANT connectivity
Connect.java:37 (set(Connect.constant_)): Assigning null to field with CONSTANT
    connectivity
```

At the start of execution of this program, the class Connect is loaded. We defined advise that checks if static fields adhere to their declared connectivity after a class is loaded and initialised. The two fields permanentStatic and constantStatic are initialised with null, which is disallowed for their declared connectivity permanent and constant, respectively. In the main method, an instance of Connect is created, which is when two further violations are reported: The fields permanent and constant_ are initialised with null and therefore violate the connectivity declared with @Connected. In the following, the individual fields of the Connect instance are assigned with objects or null. First, the illegal assignments of the fields with single and permanent connectivity are reported (lines 28, 29, and 33). Finally, the reassignment of the field permanent with permanent connectivity is reported for lines 36 and 37. Line 37 provokes two reports, as it not only reassigns a field with permanent connectivity, but reassigns it with null.[19]

4.4.3. Verifying encapsulation of subcomponents

We require a set of criteria to be fulfilled for fields that connect subcomponents. As subcomponents manifest the state of a component and are crucial for a component, we require proper encapsulation of fields that connect subcomponents. Such a field must be private, for instance, which is tested statically. Further requirements prevent leaking of subcomponents from their components. Here, we only present advise for three of these requirements. Leaking of subcomponents through external field assignments, external method invocations, and constructor invocations are reported by advise defined in the following.

As per requirement E2 of Definition 26, assigning a subcomponent to an externally accessed field is forbidden. The dynamic check verifying this requirement is implemented by the following advise definition:

[19] This strict handling of permanent connectivity is not obvious from the advise definitions here, but is how we implemented the dynamic checks.

```
before:
  set(* *)
from:
  component p
targeting external:
  object o or class C
report:
  [if p has a subcomponent s and s == thisJoinPoint.getArgs()[0]]
```

We use a from clause to specify the origin of the field assignment denoted by the set pointcut: The assignment must occur in an object or class component p. As the targeting clause contains the keyword external, only assignments to externally accessed fields are advised. The report clause states that a subcomponent of p must not be the argument of such a field assignment. Otherwise, the subcomponent leaks by assigning it to an external field, which is reported.

Encapsulation also means that a subcomponent must not be used as a parameter of a constructor or external method invocation. These criteria are described in E6 and E7 of Definition 26. We can check these requirements by using advise for constructor and method invocations defined as follows:

```
before:
  call(new(..))
from:
  component p
report:
  [if p has a subcomponent s and s is contained in thisJoinPoint.getArgs()]
```

```
before:
  call(* *(..))
from:
  component p
targeting external:
  object o or class C
report:
  [if p has a subcomponent s and s is contained in thisJoinPoint.getArgs()]
```

Both advise definitions restrict the origin of join points to object or class components. Any constructor invocation (which is external per se, hence does not require a targeting clause in the first advise definition) or external method invocation occurring in a component must not pass a subcomponent via parameters of the invoked constructor or method.

Example 4.2: The class Capsule

```
7  @DynamicFacet(COMPONENT)
8  public class Capsule {
9    @Subcomponent
10   private Capsule sub;
11
12   public Capsule() {}
13
14   public Capsule(Capsule other) {}
15
16   void pass() {
17     sub = new Capsule();       // Assign new capsule as subcomponent
18     take(sub);                 // OK, passed to method invoked on this
19     Outside.leaked = sub;      // Leaking through field assignment (external)
20     Outside.take(sub);         // Leaking through parameter of method (external)
21     new Capsule(sub);          // Leaking through constructor parameter
22   }
23
24   private void take(Capsule leak) {}
25
26   public static void main(String[] args) {
27     Capsule leaky = new Capsule();
28     leaky.pass();
29   }
30 }
31
32 class Outside {
33   static Object leaked;
34
35   static void take(Object leak) {}
36 }
```

Example

Application of encapsulation advise as defined above is exemplified by the program in Example 4.2. The class Capsule has an annotation of type @DynamicFacet that declares instances of this class to be object components. These instances have a field sub that is declared to connect subcomponents, also of type Capsule. The main method instantiates a new Capsule object and invokes the method pass on it. The method pass violates encapsulation requirements of the field sub. The following displays the reports that dynamic checks emit when running the program Capsule:

```
Capsule.java:19 (set(Outside.leaked)): Leaked subcomponent of field sub
Capsule.java:20 (call(Outside.take(..))): Leaked subcomponent of field sub
Capsule.java:21 (call(Capsule(..))): Leaked subcomponent of field sub
```

The method pass instantiates a new Capsule object in line 17 and assigns it to sub, which turns the object into a subcomponent. Passing this subcomponent to the method take is legal, as this method invocation does not occur external, but on the current object. When the field leaked of class Outside is assigned with the subcomponent, this is an external access and reported as a violation of encapsulation of the field sub. Similarly, passing the subcomponent to the method take of Outside is a violation, as the method is externally invoked on another class. In line 21, a constructor is invoked with the subcomponent as parameter, also violating the encapsulation criteria.

4.4.4. Remarks on aspects

Besides the presented advise definitions for subcomponents, we implemented further checks for the requirements of encapsulation. However, AspectJ has limits that prevent coverage of all encapsulation criteria. For instance, there is no pointcut in AspectJ capturing access to arrays, meaning one cannot monitor storing of subcomponents in arrays. Still, AspectJ enables realisation of dynamic compliance checking in a modular way without obtruding the program code. Moreover, aspects implemented in AspectJ can be packed and deployed in libraries similar to Java code, allowing reuse independent of the advised programs.

Applying aspects to a Java program requires a special compiler that adds an additional step to the compilation of programs. In this step, a program is *woven* with aspects. Weaving a program with aspects can have drastic performance impact. E.g. our aspects checking connectivity and encapsulation of fields advise events such as method invocations that occur frequently during execution of a program. Thus, these aspects are meant to be used during the development and testing phase of a program. If aspects are woven into a productively used program, they require careful selection of join points and programming of actions in order to constrain the overhead of monitoring logic.

5. Evaluation

In this chapter we assess the applicability of our devised concepts in actual code. In the first part we evaluate our notion of facets and components with the Java library. We have classified types of common Java packages and identified class components, object components, and immutable types and objects. This chapter presents statistics of this classification along with examples and observations of our analysis.

In the second part we describe how our concepts can be applied to a real-life project. We classified types of Sat4J, a program for solving satisfiability problems, and identified component concepts in the code. We present the result of this analysis and discuss observations made during the application of our concepts.

5.1. Components in the Java library

The Java library was the starting point of our attempts to define components in Java. Inspection of the code of types in commonly used packages of the Java library suggested that both types and objects are suitable to manifest the notion of components. This finding motivated the definition of facets, which components are based on. In the first part of this section we discuss the results of type facet analysis for types in this library. Afterwards, we present the results of the classification of types of common Java packages. These results are summarised in a statistic allowing for the extrapolation of the results to the complete Java library.

5.1.1. Facet analysis of the Java library

The Java library is part of the Java Development Kit and used in many programs. It contains over 1600 top-level class and interface types in 58 packages. To determine the static and dynamic facet of types in the Java library, we use our tool for type facet analysis, the TypeFacetsAnalyser. As our analysis disregards deprecated types, annotation and Enum types, and java.lang.Throwable including its subtypes (i.e. types modelling exceptions or errors), the type facet analysis covers 1380 class and interface types of the Java library.

Every package of the Java library is a subpackage of the package java. Hence, the analysis is based on this package and spans all subpackages. Recall that the TypeFacetsAnalyser can only determine immutable types and objects if facet annotations are applied to the code. As our classification of common Java packages discussed in the next section was not reflected in the code with annotations, the type facet analysis cannot identify any immutable types or objects in the Java library. The following lists the results of the

TypeFacetsAnalyser for this library. We omit the assessment of the facets of individual types.

```
1380 types analysed in java (incl. subpackages, excl. subtypes of java.lang.Throwable)
...
Of 1380 fully analysed types
  709 (51 %) have a static facet
    of which 53 (7 %) are found to be COMPONENTs
    and 0 (0 %) are found to be IMMUTABLEs
  1291 (94 %) have a dynamic facet
    of which 328 (25 %) are found to be COMPONENTs
    and 0 (0 %) are found to be IMMUTABLEs
  and 633 (46 %) have both facets
    of which 0 (0 %) are determined equally (both IMMUTABLE or COMPONENT)
    and 0 (0 %) are determined differently (one IMMUTABLE, other COMPONENT)
```

The type facet analysis reveals that 51 % of the analysed types have a static facet, 94 % have a dynamic facet, and 46 % have both a static and a dynamic facet. This means that about half of all types can be used statically without creating any instances. 5 % (the difference between all types with a static facet and those with both facets) of the analysed types can only be used statically. The majority of types has a dynamic facet and can be used in form of objects. This result corresponds to the fact that Java is an object-oriented language, in which objects are the principle constructs to represent program logic.

The analysis also determines a number of components in the Java library. Methods implying state, which is the criterion to segregate components from immutables, can only be suggested by a heuristic. Thus, the assessment COMPONENT in the result statistics is only based on mutable fields in the static or dynamic facet of a type. Only 7 % of the analysed types with a static facet are class components due to mutable fields. If type facet analysis also considers those facets that the heuristic indicates to define components, this percentage increases to 18 %. As the heuristic yields good results (as we show in the next section), we can expect that about a tenth of the analysed types of the Java library ($36\ \% \times 51\ \% \approx 9\ \%$) can be classified as class components.

25 % of the non-empty dynamic facets contain mutable fields, thus define components. Taking into account those facets that possibly define components as per the heuristic, this value changes to 60 %. Thus, we can expect that the instances of about half of all analysed types ($60\ \% \times 94\ \% \approx 56\ \%$) are object components. The expected fractions of object and class components can only be verified with a manual classification of the Java library. However, the results of analysing common Java packages presented in the following section support these estimations.

5.1.2. Classification of common Java types

We applied our concept of components to the three packages java.lang, java.io, and java.util of the Java library by classifying the top-level types of these packages. Types of these

	Distribution of assessments		
Assessment	Static facet	Dynamic facet	Both facets
Empty	124 (54 %)	28 (12 %)	4 (2 %)
Non-empty	104 (46 %)	200 (88 %)	80 (35 %)
Component	35 (15 %)	146 (64 %)	9 (4 %)
Immutable	67 (29 %)	46 (20 %)	14 (6 %)

Table 5.1.: Results of classifying 228 types from the Java library

packages are frequently used in many programs. Moreover, every type in java.lang can be referred to by its simple name in every compilation unit of any program (see JLS 7.3). Our analysis excluded annotation, Enum, and deprecated types. We also classified a few types of other packages of the Java library. In total, our analysis spanned 228 class and interface types.

We did not determine the facets of every type. Recall that the invariance of classification requires that the classification of a type holds for every subtype. Hence, we refrained from determining facets of types with many subtypes if the facet defines an immutable. As java.lang.Object is the root of the Java type hierarchy and the supertype of every other type, it remains unclassified in our classification. Also, we did not determine the dynamic or static facet of java.lang.Enum, as this class is the superclass of every Enum type in Java. The detailed results of this analysis are given in the Appendix, where all analysed facets and their assessment are listed. Here, we aggregate our findings into a statistic, which is shown in Table 5.1.

The results of our classification correspond to the type facet analysis of the complete Java library. 46 % of the static facets and 88 % of the dynamic facets of the 228 analysed types are non-empty (in comparison to 51 % non-empty static facets and 94 % non-empty dynamic facets of all types in the Java library). 35 % of the classified types have both facets (compared to 46 % of all types).

15 % of the static facets of the classified types define components, as do 64 % of the dynamic facets. These percentages correlate with the results of the type facet analysis with heuristic on the complete Java library, where 9 % of the static facets and 56 % of the dynamic facets were assessed to define components. This correspondence is an indication for the applicability of the heuristic implemented by the TypeFacetsAnalyser. The next section shows that the heuristic can yield results close to those of manual classification of types.

Observations

During our analysis of types in the Java library we have made observations that we enlist in the following.

The so-called *wrapper* classes in java.lang represent values of a primitive type in form an object. The wrapper classes are Boolean, Byte, Character, Double, Float, Integer, Long, and Short. Instances of these classes model a primitive value that cannot be changed. There are no mutable fields or methods implying state in the dynamic facet of these classes. Hence, these facets define immutables, and instances of wrapper classes are immutable objects. This classification resembles the nature of primitive values, as these are also unchangeable.

The dynamic facet of interface types cannot contain any fields, as fields of an interface are always static. Interfaces can declare non-static methods, however, which are always public and therefore part of the dynamic facet. If a method in the dynamic facet of an interface implies state, the facet defines a component. Due to the invariance of classification, objects of a subtype of such an interface are object components. Even a single method of an interface can cause such a classification if the method implies state. Examples for such interface methods are the method close declared by java.io.Closeable and the method append in java.lang.Appendable. Invoking close on the data source that an object of type Closeable represents closes the source, allowing it to release its resources (usually in the form of other objects). In general, a data source cannot be used any longer after closing it. The method append of Appendable allows adding further characters to the sequence of characters that the Appendable object represents. Appending characters is a manipulation of the state of the sequence, thus the method append implies state. We classified both the interfaces Closeable and Appendable as object components.

During our inspection of the types in the Java library we detected methods through which a view on the instances of a type can be obtained. These views are objects of type java.util.Iterator, for instance. An example is the List interface in java.util, as we explained already. List also defines the method subList. This method returns another List object through which only a part of the underlying object component, i.e. the original list, can be accessed. The view obtained through subList is an example for a restricted perspective on object components. An extended perspective is given by a view on an instance of java.io.ObjectInputStream. This class allows the reconstruction of an object after it was *serialised*. Serialisation is the process of transforming an object into a form that allows for transmission or persisting, e.g. writing the object into a file. The reconstruction requires reading of each field of the serialised object in a fix order. Alternatively, the non-static method readFields of ObjectInputStream yields a view for random access on the serialised fields. By using the view obtained through this method, one can retrieve the objects and values of the fields of the serialised object in an arbitrary order. A similar kind of view exists for ObjectOutputStream, which allows for serialisation of objects. This class is the counterpart of ObjectInputStream.

Some of the analysed types were found to have no facet, i.e. both the static and dynamic facet are empty. We only observed this case for interfaces without fields or methods. Interfaces that do not declare any fields or methods are often called *marker* interfaces. These interface types only *mark* instances of subtypes to have a special property. For instance, objects can be marked to be serialisable or cloneable by implementing the marker interfaces java.io.Serializable and java.lang.Cloneable, respectively.

As types of the Java library are used in many programs, the discussed classification results can serve as a starting point to use component concepts in other programs. The Eclipse plug-in for interconnection analysis uses our classification of common Java types. This plug-in can be used for a component-oriented view on programs without much classification effort in advance. Using our classification results, developers can directly identify a set of components and immutable types and objects in their own program, if the program utilises types from the packages we analysed.

5.2. Analysis of Sat4J

Libraries, such as the Java library, are usually a collection of types useful for other programs, but often do not form a closed program. In addition to analysing the Java library, we opted to apply our concepts to a complete program from a real-world project. We searched for a medium-sized open-source program that can be used on its own instead of serving as a library. We chose Sat4J, a Java tool for solving and optimising satisfiability problems. This tool takes text files containing a set of satisfiability problems as input, applies a solver to it, and emits the result. We restricted our analysis to the Core module of Sat4J.[1] The Core module can be used independently of other modules of Sat4J, and we consider it a program on its own. In the following, we refer to the Sat4J Core module simply as "Sat4J".

We inspected and classified all top-level types of Sat4J. For a set of types that we classified as components, we analysed the links to other types and objects and distinguished these links with our concepts introduced in Section 3.2. The first part of this section illustrates the results of our classification of the types of Sat4J. Afterwards, we discuss observations made during the analysis of this program.

5.2.1. Classification of Sat4J

We classified top-level class and interface types of Sat4J, excluding test classes and deprecated types. Our analysis covered 191 types in total. The results of the classification were reflected in the code by applying facet annotations on the types. The TypeFacetsAnalyser, our tool for facet analysis, uses these annotations to calculate classification statistics. The output of this tool (omitting the results for single types) is provided below.

[1] in particular of the Sat4J Release 2.3.3, see http://www.sat4j.org/maven233/org.sat4j.core/index.html

```
191 types analysed in org.sat4j (incl. subpackages, excl. subtypes of java.lang.
    Throwable)
...
Of 191 fully analysed types
  37 (19 %) have a static facet
    of which 2 (5 %) are found to be COMPONENTs
    and 31 (84 %) are found to be IMMUTABLEs
  185 (97 %) have a dynamic facet
    of which 158 (85 %) are found to be COMPONENTs
    and 17 (9 %) are found to be IMMUTABLEs
  and 31 (16 %) have both facets
    of which 4 (13 %) are determined equally (both IMMUTABLE or COMPONENT)
    and 21 (68 %) are determined differently (one IMMUTABLE, other COMPONENT)
```

As per our classification, 19 % of all types in Sat4J have a static facet, and 97 % have a dynamic facet. This distribution resembles that of types of the Java library, where we also found less types with a static facet than types with a dynamic facet. 16 % of the checked types have both facets. 3 % (19 % − 16 %) only have a static facet, which also corresponds to our analysis of types in the Java library: Most types with a static facet also have a dynamic facet.

We found 5 % of the types with a static facet to be class components, and 84 % to be immutable types. The instances of 85 % of analysed types with a non-empty dynamic facet are object components, and 9 % are immutable objects. Of the 31 types with both facets, 13 % were determined equally (i.e. both facets either define an immutable or a component). The facets of about two thirds (68 %) of types were determined differently: One facet of these types defines an immutable, the other facet defines a component. Thus, the majority of types diverge with respect to classification. These percentages should not be overrated, due to the small amount of types with both facets. However, this divergence (13 % versus 68 %) indicates that subdividing types into facets and defining the terms component and immutable for both facets is reasonable. The distribution of immutables and components does not match the distribution in the Java library well. A reason for this difference might be the limited analysis set size. Also, the different nature of programs and libraries might account for the disparity in distribution.

Before classifying the types of Sat4J, we performed a type facet analysis on this program using the TypeFacetsAnalyser. The results below include facets assessed to define components by the heuristic.

```
191 types analysed in org.sat4j (incl. subpackages, excl. subtypes of java.lang.
    Throwable)
...
Of 191 fully analysed types
  37 (19 %) have a static facet
    of which 0 (0 %) are found to be COMPONENTs
    and 0 (0 %) are found to be IMMUTABLEs
  185 (97 %) have a dynamic facet
```

```
   of which 153 (83 %) are found to be COMPONENTs
   and 0 (0 %) are found to be IMMUTABLEs
and 31 (16 %) have both facets
   of which 0 (0 %) are determined equally (both IMMUTABLE or COMPONENT)
   and 0 (0 %) are determined differently (one IMMUTABLE, other COMPONENT)
```

No class components were identified by using the heuristic, whereas we identified two class components during the manual analysis. 153 of the dynamic facets define components when enabling the heuristic, which is little less than the actual 158 that result from our classification. The results show that the heuristic can assist in identifying object and class components. As the type facet analysis cannot identify immutable types or objects without declaring them with facet annotations, we cannot compare the results with our manual classification.

Observations

The following enumerates observations from our analysis of types of Sat4J and describes the application of our concepts and problems we encountered.

Private fields in the inspected types of Sat4J often fulfil many of the encapsulation criteria we devised for subcomponents (see Definition 26). Frequently, they do not satisfy requirement E1, meaning assigned objects are not obtained newly (e.g. org.sat4j.tools.CheckMUSSolutionListener). Consequently, these fields cannot be considered to connect subcomponents, even if assigned objects manifest the state of a component. Nevertheless, we could apply the annotation @Subcomponent in several types where all encapsulation criteria are fulfilled.

The classes of Sat4J regularly use types of the Java library, e.g. java.lang.System. This class provides a method arraycopy for copying arrays. This method is *native*, which means its implementation is not given in Java code. Hence, we cannot tell from the code of System if the references passed to arraycopy are transitory in this method. Due to this uncertainty we could not apply the annotation @Transitory in many places of the code involving the method arraycopy (e.g. org.sat4j.minisat.constraints.card.MinWatchCard).

Many Sat4J types override the method toString from Object. By overriding, instances of these types provide a more meaningful String representation of an object than the overridden method of Object. Most of these overriding methods instantiate the class StringBuffer (e.g. org.sat4j.core.VecInt). A StringBuffer allows concatenation of String objects or literals. After building the String representation, the method toString is invoked on the StringBuffer instance to obtain the final String. This String object is then returned from the overriding toString method. Only anonymous methods of the StringBuffer instance are invoked and the reference to this instance is not retained or returned. Thus, we could use @Transitory on local variables assigned with a StringBuffer object in many toString methods.

The inspected types often declare set-methods that allow assigning of a (usually private) field. These set-methods rarely check if a passed reference is null and simply assign the reference to the field. Similarly, constructor parameters are usually not tested for a reference to null before assigning the reference to a field (e.g. org.sat4j.minisat.constraints.card.AtLeast). In these cases, we could not derive the field's connectivity as permanent or constant, as null must not be assigned to fields with these kinds of connectivity.

In summary, we could apply our concepts frequently in the selected types. However, we refrained from adapting the code to apply the concepts in more places. Such adaption would have required only little rewriting with some types. Nevertheless, the analysis showed that the ideas of our devised concepts can be found in real-world code, even if they are not considered at the start of development.

6. Conclusion

This thesis presented our approach to extending an object-oriented language with the notion of components. We analysed the Java language and showed how components can be identified as the major elements of object-oriented programs in this language. We consider stateful types and objects as components, and abstract from immutable, i.e. stateless, types and objects. We presented an annotation-based approach for seamless integration of component concepts in the code. To assist developers in the discovery and application of these concepts in their own programs, we developed and introduced analysis tools, also in form of an Eclipse plug-in. Our monitoring tools help check programs for compliance to the requirements of concepts related to components.

The definition of components is based on the two facets of a type, which determine how a type and its instances can be used. Facets allow us to define state as a differentiating property of types and objects. State can become obvious and be manipulated through fields and methods of a type or object. Also, we explained type classification as the process to segregate components from immutables.

The interconnection of a program is a component-oriented view on the program, modelling the interaction of components. This program model is based on the analysis of links between components and their distinction by duration. We explained how transitory references can be disregarded, whereas references through fields constitute coupling of components in the component-oriented model of a program. This coupling can be further qualified with the notion of connectivity of fields, through which coupling complexity can be assessed. Distinguishing subcomponents from dependencies is the basis for aggregation of components for a higher-level view of the program. The concept of views aids in abstracting from objects that merely yield a different perspective on components.

We showed how our devised concepts can be represented in the program code using annotations. Annotations are the basis for the tools and the dynamic checks we devised that help employ our concepts and verify programs for adherence to the requirements of these concepts.

By applying components and the related constructs to the Java library and a real-world program, we assessed the applicability and distribution of our concepts. We found that defining the term component for both types and objects is sound. Our inspection of a set of types from the Java library revealed that most types with both facets diverge with respect to classification. It is rare in our analysis results that a type and its instances are both immutable or both components. Thus, it proved to be reasonable to subdivide

types into facets as the basis for components.

We determined that half of the types of the Java library can be used statically, and most of the types can be used in form of instances. Using a heuristic for identification of components, we estimated a tenth of the types in the Java library to be class components, and the instances of half of the types to be object components. The distribution of object and class components indicates that state is a reasonable criterion to distinguish components from other types and objects. This criterion is neither too selective, nor does it yield too many types and objects as components. Our definition of components allows developers to distinguish types and objects on a semantic level and yields a clearer view on the elements of a program. We integrated the results of classifying parts of the Java library into our tool for interconnection analysis, through which developers can directly benefit from the presented concepts in their own programs. As most programs make regular use of types from the Java library, developers can identify components in their programs already before classifying their own types.

The notion of components in Java is foremostly connected to programs governed by component frameworks. We showed that components can be identified in many Java programs, including small or medium-size programs. The concepts introduced here can serve as a foundation to devise further methodologies and techniques for components in object-oriented programs. For instance, one can adapt the variety of coupling measures for object-oriented programs (cf. Dietrich et al. [4], for example) and devise a coupling measure based on components. Such a coupling measure can be based on qualified references among components and take connectivity into account. References through fields can be weighted differently than transitory references.

We evaluated our concepts on a finished program, but refrained from adapting the code. Adaption would have allowed us to employ the component concepts to a greater extent. However, we expect that consideration of these concepts already at the start of development yields a more pervasive application of component constructs. Ultimately, the concepts we introduced can be used to devise a component-oriented design principle for programs, where not objects are the prime elements, but components.

Acknowledgements

I would like to thank Prof. Arnd Poetzsch-Heffter for supervising this thesis, and Ilham Kurnia for his very helpful reviews and suggestions. I also thank Malte Brunnlieb for allowing me to use his Eclipse plug-in and extend it with component concepts.

Bibliography

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197. ACM, 2002.

[2] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *ECOOP'97—Object-Oriented Programming*, pages 32–59. Springer, 1997.

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

[4] Jens Dietrich, Vyacheslav Yakovlev, Catherine McCartin, Graham Jenson, and Manfred Duchrow. Cluster analysis of java dependency graphs. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 91–94. ACM, 2008.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[6] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java(R) Language Specification, Java SE 7 Edition, February 2013. URL `http://docs.oracle.com/javase/specs/jls/se7/html/index.html`. Last access: June 16, 2013.

[7] Jacob Jenkov. Dynamic class loading and reloading in java. URL `http://tutorials.jenkov.com/java-reflection/dynamic-class-loading-reloading.html`. Last access: June 16, 2013.

[8] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, Andy Clement, Dave Syer, Oliver Gierke, Rossen Stoyanchev, and Phillip Webb. Spring framework reference documentation, 3.2.2.release, 2013. URL `http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-introduction`. Last access: June 16, 2013.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*. Springer, 1997.

[10] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP'98—Object-Oriented Programming*, pages 158–185. Springer, 1998.

[11] OSGi Alliance. About the OSGi Service Platform. Technical Whitepaper, Revision 4.1, 2007. URL `http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf`. Last access: May 25, 2013.

[12] Palo Alto Research Center, Inc. The AspectJ(TM) Programming Guide, 2003. URL `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`. Last access: June 12, 2013.

[13] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[14] Paul Rogers. Encapsulation is not information hiding, 2001. URL `http://www.javaworld.com/javaworld/jw-05-2001/jw-0518-encapsulation.html`. Last access: May 21, 2013.

[15] Inc. Sun Microsystems. Code Conventions for the Java (TM) Programming Language, April 1999. URL `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`. Last access: June 16, 2013.

[16] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

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

A. Appendix

The following table lists the results of our analysis of 228 types from the Java library.

		Assessment	
Type name	Type kind	Static facet	Dynamic facet
java.io.Bits	Class	Immutable	Empty
java.io.BufferedInputStream	Class	Empty	Component
java.io.BufferedOutputStream	Class	Empty	Component
java.io.BufferedReader	Class	Empty	Component
java.io.BufferedWriter	Class	Empty	Component
java.io.ByteArrayInputStream	Class	Empty	Component
java.io.ByteArrayOutputStream	Class	Empty	Component
java.io.CharArrayReader	Class	Empty	Component
java.io.CharArrayWriter	Class	Empty	Component
java.io.Closeable	Interface	Empty	Component
java.io.Console	Class	Empty	Component
java.io.DataInput	Interface	Empty	Component
java.io.DataInputStream	Class	Immutable	Component
java.io.DataOutput	Interface	Empty	Component
java.io.DataOutputStream	Class	Immutable	Component
java.io.DeleteOnExitHook	Class	Component	Empty
java.io.ExpiringCache	Class	Empty	Component
java.io.Externalizable	Interface	Empty	Component
java.io.File	Class	Immutable	Component
java.io.FileDescriptor	Class	Immutable	Component
java.io.FileFilter	Interface	Empty	Immutable
java.io.FileInputStream	Class	Empty	Component
java.io.FileOutputStream	Class	Empty	Component
java.io.FilePermission	Class	Empty	Immutable
java.io.FilePermissionCollection	Class	Empty	Component
java.io.FileReader	Class	Empty	Component
java.io.FileSystem	Abstract class	Immutable	Component
java.io.FileWriter	Class	Empty	Component
java.io.FilenameFilter	Interface	Empty	Immutable
java.io.FilterInputStream	Class	Empty	Component
java.io.FilterOutputStream	Class	Empty	Component
java.io.FilterReader	Abstract class	Empty	Component
java.io.FilterWriter	Abstract class	Empty	Component
java.io.Flushable	Interface	Empty	Component
java.io.InputStream	Abstract class	Empty	Component
java.io.InputStreamReader	Class	Empty	Component
java.io.LineNumberReader	Class	Empty	Component
java.io.ObjectInput	Interface	Empty	Component
java.io.ObjectInputStream	Class	Immutable	Component
java.io.ObjectInputValidation	Interface	Empty	Immutable
java.io.ObjectOutput	Interface	Empty	Component
java.io.ObjectOutputStream	Class	Immutable	Component
java.io.ObjectStreamClass	Class	Immutable	Component
java.io.ObjectStreamConstants	Interface	Undetermined	Empty
java.io.ObjectStreamField	Class	Empty	Component
java.io.OutputStream	Abstract class	Empty	Component
java.io.OutputStreamWriter	Class	Empty	Component

		Assessment	
Type name	Type kind	Static facet	Dynamic facet
java.io.PipedInputStream	Class	Immutable	Component
java.io.PipedOutputStream	Class	Empty	Component
java.io.PipedReader	Class	Empty	Component
java.io.PipedWriter	Class	Empty	Component
java.io.PrintStream	Class	Empty	Component
java.io.PrintWriter	Class	Empty	Component
java.io.PushbackInputStream	Class	Empty	Component
java.io.PushbackReader	Class	Empty	Component
java.io.RandomAccessFile	Class	Empty	Component
java.io.Reader	Abstract class	Empty	Component
java.io.SequenceInputStream	Class	Empty	Component
java.io.SerialCallbackContext	Class	Empty	Component
java.io.Serializable	Interface	Empty	Empty
java.io.SerializablePermission	Class	Empty	Immutable
java.io.StreamTokenizer	Class	Immutable	Component
java.io.StringReader	Class	Empty	Component
java.io.StringWriter	Class	Empty	Component
java.io.Win32FileSystem	Class	Immutable	Immutable
java.io.WinNTFileSystem	Class	Immutable	Immutable
java.io.Writer	Abstract class	Empty	Component
java.lang.AbstractStringBuilder	Abstract class	Empty	Component
java.lang.Appendable	Interface	Empty	Component
java.lang.ApplicationShutdownHooks	Class	Component	Empty
java.lang.AssertionStatusDirectives	Class	Empty	Component
java.lang.AutoCloseable	Interface	Empty	Component
java.lang.Boolean	Class	Component	Immutable
java.lang.Byte	Class	Component	Immutable
java.lang.CharSequence	Interface	Empty	Undetermined
java.lang.Character	Class	Component	Immutable
java.lang.CharacterData	Abstract class	Immutable	Immutable
java.lang.CharacterData00	Class	Component	Immutable
java.lang.CharacterData01	Class	Component	Immutable
java.lang.CharacterData02	Class	Component	Immutable
java.lang.CharacterData0E	Class	Component	Immutable
java.lang.CharacterDataLatin1	Class	Component	Immutable
java.lang.CharacterDataPrivateUse	Class	Immutable	Immutable
java.lang.CharacterDataUndefined	Class	Immutable	Immutable
java.lang.CharacterName	Class	Immutable	Empty
java.lang.Class	Class	Immutable	Component
java.lang.ClassLoader	Abstract class	Component	Component
java.lang.ClassLoaderHelper	Class	Immutable	Empty
java.lang.ClassValue	Abstract class	Component	Component
java.lang.Cloneable	Interface	Empty	Empty
java.lang.Comparable	Interface	Empty	Undetermined
java.lang.Compiler	Class	Component	Empty
java.lang.ConditionalSpecialCasing	Class	Component	Empty
java.lang.Double	Class	Component	Immutable
java.lang.Enum	Abstract class	Undetermined	Undetermined
java.lang.Float	Class	Component	Immutable
java.lang.InheritableThreadLocal	Class	Immutable	Component
java.lang.Integer	Class	Component	Immutable
java.lang.Iterable	Interface	Empty	Undetermined
java.lang.Long	Class	Component	Immutable
java.lang.Math	Class	Immutable	Empty
java.lang.Number	Abstract class	Empty	Immutable
java.lang.Object	Class	Empty	Undetermined
java.lang.Package	Class	Immutable	Immutable
java.lang.Process	Abstract class	Empty	Component
java.lang.ProcessBuilder	Class	Immutable	Component
java.lang.ProcessEnvironment	Class	Immutable	Component
java.lang.ProcessImpl	Class	Immutable	Component
java.lang.Readable	Interface	Empty	Component

		Assessment	
Type name	Type kind	Static facet	Dynamic facet
java.lang.Runnable	Interface	Empty	Undetermined
java.lang.Runtime	Class	Immutable	Immutable
java.lang.RuntimePermission	Class	Empty	Immutable
java.lang.SecurityManager	Class	Empty	Immutable
java.lang.Short	Class	Component	Immutable
java.lang.Shutdown	Class	Component	Empty
java.lang.StackTraceElement	Class	Empty	Immutable
java.lang.StrictMath	Class	Immutable	Empty
java.lang.String	Class	Immutable	Immutable
java.lang.StringBuffer	Class	Immutable	Component
java.lang.StringBuilder	Class	Immutable	Component
java.lang.StringCoding	Class	Immutable	Empty
java.lang.System	Class	Component	Empty
java.lang.SystemClassLoaderAction	Class	Empty	Immutable
java.lang.Terminator	Class	Component	Empty
java.lang.Thread	Class	Component	Component
java.lang.ThreadGroup	Class	Empty	Component
java.lang.ThreadLocal	Class	Immutable	Component
java.lang.Void	Class	Component	Empty
java.net.URI	Class	Immutable	Immutable
java.net.URL	Class	Component	Component
java.security.BasicPermission	Abstract class	Empty	Immutable
java.security.Permission	Abstract class	Empty	Immutable
java.security.PermissionCollection	Abstract class	Empty	Component
java.util.AbstractCollection	Abstract class	Empty	Component
java.util.AbstractList	Abstract class	Empty	Component
java.util.AbstractMap	Abstract class	Empty	Component
java.util.AbstractQueue	Abstract class	Empty	Component
java.util.AbstractSequentialList	Abstract class	Empty	Component
java.util.AbstractSet	Abstract class	Empty	Component
java.util.ArrayDeque	Class	Empty	Component
java.util.ArrayList	Class	Immutable	Component
java.util.Arrays	Class	Immutable	Empty
java.util.BitSet	Class	Immutable	Component
java.util.Calendar	Abstract class	Immutable	Component
java.util.Collection	Interface	Empty	Component
java.util.Collections	Class	Immutable	Empty
java.util.ComparableTimSort	Class	Immutable	Empty
java.util.Comparator	Interface	Empty	Immutable
java.util.Currency	Class	Component	Immutable
java.util.Date	Class	Immutable	Component
java.util.Deque	Interface	Empty	Component
java.util.Dictionary	Abstract class	Empty	Component
java.util.DualPivotQuicksort	Class	Immutable	Empty
java.util.EnumMap	Class	Empty	Component
java.util.EnumSet	Abstract class	Immutable	Component
java.util.Enumeration	Interface	Empty	Component
java.util.EventListener	Interface	Empty	Empty
java.util.EventListenerProxy	Abstract class	Empty	Immutable
java.util.EventObject	Class	Empty	Component
java.util.Formattable	Interface	Empty	Undetermined
java.util.FormattableFlags	Class	Immutable	Empty
java.util.Formatter	Class	Immutable	Component
java.util.GregorianCalendar	Class	Component	Component
java.util.HashMap	Class	Immutable	Component
java.util.HashSet	Class	Immutable	Component
java.util.Hashtable	Class	Immutable	Component
java.util.IdentityHashMap	Class	Empty	Component
java.util.Iterator	Interface	Empty	Component
java.util.JapaneseImperialCalendar	Class	Component	Component
java.util.JumboEnumSet	Class	Immutable	Component
java.util.LinkedHashMap	Class	Immutable	Component

		Assessment	
Type name	Type kind	Static facet	Dynamic facet
java.util.LinkedHashSet	Class	Empty	Component
java.util.LinkedList	Class	Empty	Component
java.util.List	Interface	Empty	Component
java.util.ListIterator	Interface	Empty	Component
java.util.ListResourceBundle	Abstract class	Component	Immutable
java.util.Locale	Class	Component	Immutable
java.util.LocaleISOData	Class	Immutable	Empty
java.util.Map	Interface	Empty	Component
java.util.NavigableMap	Interface	Empty	Component
java.util.NavigableSet	Interface	Empty	Component
java.util.Objects	Class	Immutable	Empty
java.util.Observable	Class	Empty	Component
java.util.Observer	Interface	Empty	Undetermined
java.util.PriorityQueue	Class	Empty	Component
java.util.Properties	Class	Immutable	Component
java.util.PropertyPermission	Class	Immutable	Immutable
java.util.PropertyPermissionCollection	Class	Empty	Component
java.util.PropertyResourceBundle	Class	Component	Immutable
java.util.Queue	Interface	Empty	Component
java.util.Random	Class	Immutable	Component
java.util.RandomAccess	Interface	Empty	Empty
java.util.RandomAccessSubList	Class	Empty	Component
java.util.RegularEnumSet	Class	Immutable	Component
java.util.ResourceBundle	Abstract class	Component	Immutable
java.util.Scanner	Class	Empty	Component
java.util.ServiceLoader	Class	Immutable	Component
java.util.Set	Interface	Empty	Component
java.util.SimpleTimeZone	Class	Component	Component
java.util.SortedMap	Interface	Empty	Component
java.util.SortedSet	Interface	Empty	Component
java.util.Stack	Class	Empty	Component
java.util.StringTokenizer	Class	Empty	Component
java.util.SubList	Class	Empty	Component
java.util.TaskQueue	Class	Empty	Component
java.util.TimSort	Class	Immutable	Empty
java.util.TimeZone	Abstract class	Component	Component
java.util.Timer	Class	Empty	Component
java.util.TimerTask	Abstract class	Immutable	Component
java.util.TimerThread	Class	Component	Component
java.util.TreeMap	Class	Immutable	Component
java.util.TreeSet	Class	Empty	Component
java.util.UUID	Class	Immutable	Immutable
java.util.Vector	Class	Empty	Component
java.util.WeakHashMap	Class	Immutable	Component
java.util.XMLUtils	Class	Immutable	Empty
java.util.concurrent.atomic.AtomicBoolean	Class	Empty	Component
java.util.concurrent.atomic.AtomicInteger	Class	Empty	Component
java.util.concurrent.atomic.AtomicIntegerArray	Class	Empty	Component
java.util.concurrent.atomic.AtomicIntegerFieldUpdater	Abstract class	Immutable	Immutable
java.util.concurrent.atomic.AtomicLong	Class	Empty	Component
java.util.concurrent.atomic.AtomicLongArray	Class	Empty	Component
java.util.concurrent.atomic.AtomicLongFieldUpdater	Abstract class	Immutable	Immutable
java.util.concurrent.atomic.AtomicMarkableReference	Class	Immutable	Component
java.util.concurrent.atomic.AtomicReference	Class	Empty	Component
java.util.concurrent.atomic.AtomicReferenceArray	Class	Empty	Component
java.util.concurrent.atomic.AtomicReferenceFieldUpdater	Abstract class	Immutable	Immutable
java.util.concurrent.atomic.AtomicStampedReference	Class	Immutable	Component