Applying the Universe type system to an industrial application

Case Study

Thomas Hächler

Master Project Report

Software Component Technology Group

Departement of Computer Science

Swiss Federal Institute of Technology Zurich

September 2004 - March 2005

Supervising Assistant: Dipl.-Ing. Werner M. Dietl

Supervising Professor: Prof. Dr. Peter Müller

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

http://sct.inf.ethz.ch
Software Component Technology Group

Abstract

The Software Component Technology Group has developed the Universe type system to control aliasing in object-oriented programming languages. The goal of this masters thesis is to get realistic experience in applying this type system to an industrial application.
The analyzed application is presented in chapter 1. In chapter 2 we give a short introduction to the Universe type system.
In addition to the application, some Java API has been annotated with universe types. Some examples and encountered problems are presented in chapter 3. To enable static fields and application wide resources, a global universe has been used as an extension of the Universe type system.
The results of this case study are structured in three parts: (1) In chapter 4 we present how the Universe type system has been applied to the application and what parts had to be refactored. (2) Problems that have been encountered and workarounds that have been used while annotating the application are documented in chapter 5. (3) In chapter 6 we present a few approaches to extend the Universe type system. The focus of these extensions is on usage in real world applications.

Acknowledgements

I would like to thank Werner Dietl. He has spent a lot of time to support me and to review this report extensively.
Furthermore I thank Prof. Peter Müller, who gave me the opportunity to accomplish my master thesis as a case study on an industrial application.
Special thanks to Lukas Eppler, leader of the software development team at Cinerent Open Air AG. He made it possible to realize this case study on the Yoshi software.
Without the personal support of Mr. Hans Dubach, administrative issues would have been much harder. I want to thank him for his commitment.
Last but not least, I would like to thank Janine Plüss and my family for encouragement and support.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Application Overview

This chapter is an introduction to the application analyzed in this case study. We start with a short historical background and point out a few business details (section 1.1). In section 1.2 the software components of the application are introduced. At the end of this chapter we select the components that will be analyzed in chapter chapter 4.

1.1 Background

For better understanding of this case study we give an overview over the business context the analyzed application is used in. We introduce the brand "starticket" and the basic ideas of its ticketing system. Most of the details in this section are based on the document "Projektbeschrieb: Entwicklung von Barcode basierten Eingangskontrollen" [EG02].

Print at home® tickets

In the year 2002 Cinerent Open Air AG started with a barcode-based ticketing system called "starticket" [Cin]. They have been one among the first companies selling tickets with a unique barcode over the Internet. By buying a Ticket, a customer receives a number encoded in a barcode. This number equals an authorization to enter a specific show. The customer may print the barcode, together with other ticket information, to paper. The developer of this system thought about security aspects, such that (1) tickets that are copied (illegally) are valid only once and (2) no barcode can be guessed. This allows clients to print their tickets at home. An organizers of an event saves expensive ticket paper and therefore a lot of money.

The Yoshi device

The software we are going to analyze reads the home-printed barcodes of these tickets and checks whether the ticket-holder is allowed to enter the show or not.
The devices this software is running on are designed and built by Cinerent Open Air AG itself. The (originally internal) name of these devices is "Yoshi". Probably because of its sweet design (see figure 1.1 for a picture of a Yoshi device), this name has established itself and therefore it now appears in official descriptions, software documentation and even class names of the software (e.g. `YoshiDataStructure`).
These electronic ticket control systems will be used at different kinds of events. Therefore the devices have two requirements to fulfill: (1) The physical requirements, e.g. they have to be fast, easy to use and provide acoustic and visual feedback to the users, and they have to be weather resistant. (2) The

Figure 1.1: "Yoshi": The device on which the analyzed application runs.

software part: They have to load data from a web-server. This provides a base for making the right decision when a ticket is read. Changes of the data on the server have to be updated within a few minutes. Locally produced data have to be propagated to other Yoshis, running at the same event, and later on be uploaded to the web-server for statistics. When the device is going to be shut down, all data have to be saved and restored on the next startup. Last but not least, a Yoshi has to be configured and managed through a user interface, which should be easy to handle and provide as much flexibility as possible.

1.2 Software Components

In this section the specific software components are introduced. The first one is the Data Structure (section 1.2.1), it is the core of the whole application. Other components, like the XML Download (section 1.2.2) or the Checkpoint (section 1.2.3) work on these data. For an overview over all components and their relations, have a look at figure 1.2.

> *For improved clearness component names are written with initial capital letters (e.g. Data Structure).*

1.2.1 Data Structure

The data structure of the Yoshi is basically a tree organized as a composite pattern [GHJV95] with an index over all components. In the next paragraph all classes involved in the data structure tree are introduced. The variety of the elements in the tree comes from the business meaning represented by these classes.
In figure 1.3 the runtime structure of an example data tree is drawn. (The syntax ': Type' means 'an object of type Type'.)
A single object of type `YoshiDataStructure` provides a facade of the data structure. It allows access to the data and provides an interface for the common operations on the data structure, like addition and

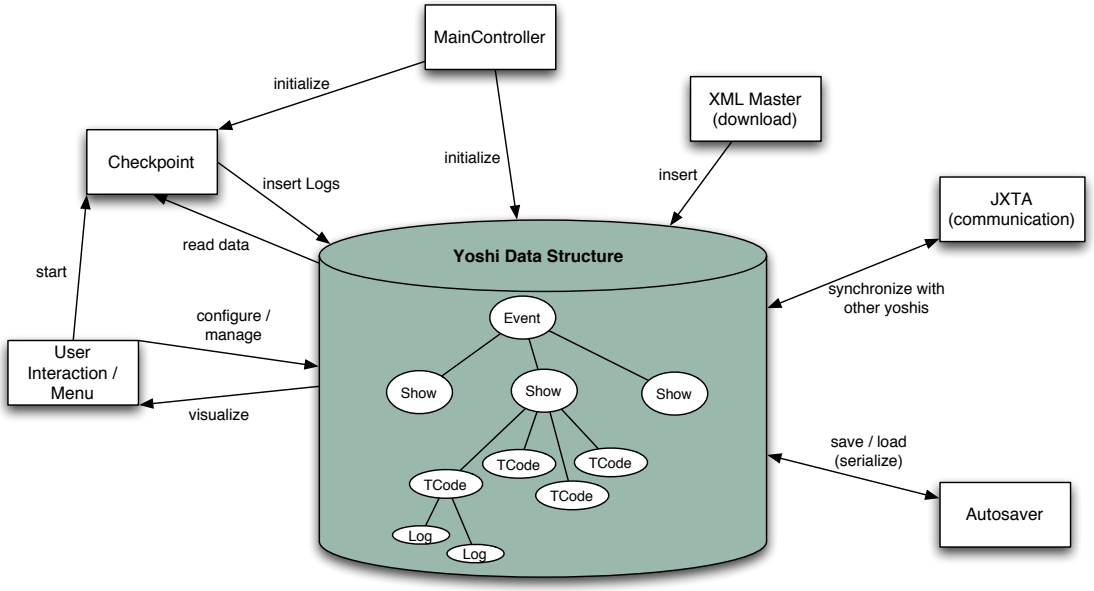

Figure 1.2: Yoshi Software Components and their relations.

retreival of data. Unfortunately, it does not fully encapsulate the data structure, since other components can perform operations on the entries directly.
The `YoshiDataStructure` object initializes the root of the data tree, which is of type `TicketingData`. The root of the tree contains all `Event` objects. An `Event` represents a happening like e.g. a festival and has several `Shows`. A `Show` takes place at a defined time and location. `Tickets` can be bought for `Shows`. A `Ticket` contains at most one valid `TCode` that defines what will be encoded in the barcode on the ticket. If a ticket is not yet sold or has been canceled, it does not contain a valid TCode. At runtime, the Checkpoint inserts `Logs` as children of a TCode. A log may have the meaning 'entered', 'left (exit)' or 'access denied'.
Additionally to the tree representation of the data, there is an index over all elements of the tree. It is stored in a static field `allInstances` of the class `TicketingData` and allows quick access to all data stored in the Data Structure. In figure 1.3 the index references are visualized as grey arrows.
All classes from the data tree extend the abstract class `TicketingComponent`.
They have a `TicketingUniqueKey`, which is composed by a unique ID per instance of a class (field `componentId`) and a static ID of the class (field `staticClassId`). Furthermore they contain a reference to their parent and are referenced from an index, which allows one to get quick access to all data stored in the Data Structure. The class diagram of the described classes (figure 1.4) visualizes the composite pattern.
The relations of the Data Structure to other components is visualized in Figure 1.2.

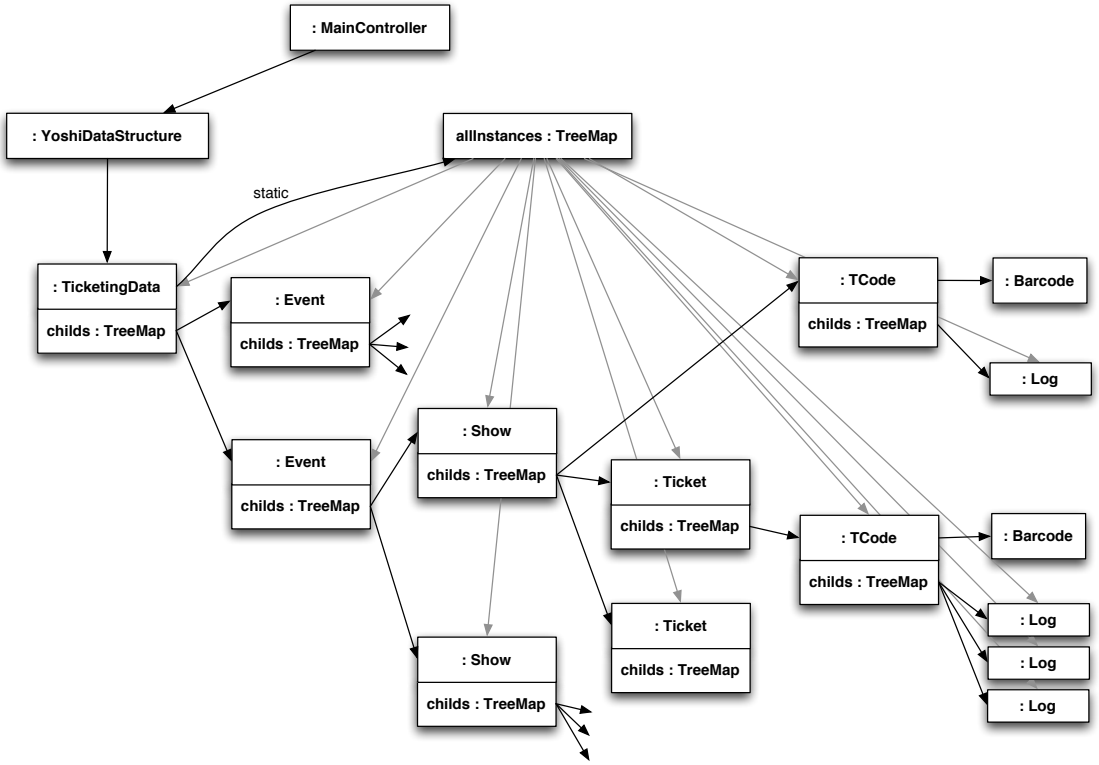

Figure 1.3: An example runtime object structure of the Data Structure component. `allInstances` is a static field in the class `TicketingData` and represents the index over all components in the tree; references of this index are drawn with grey arrows.

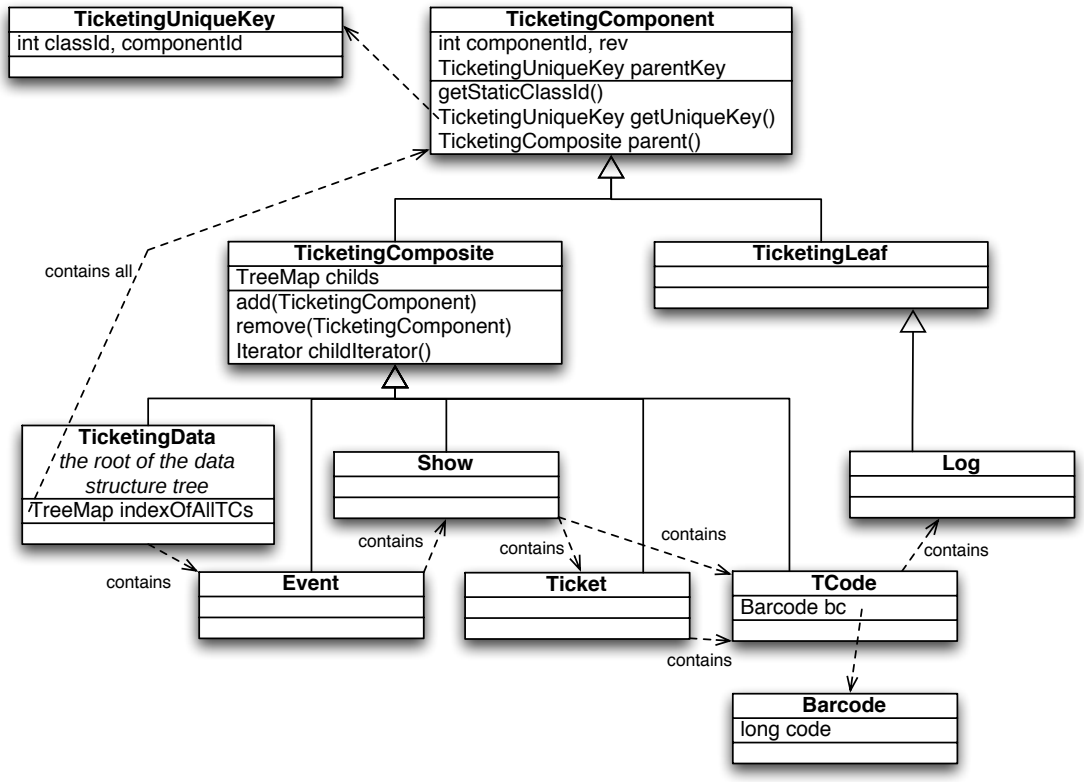

Figure 1.4: Class diagram of the data structure classes. These classes implement the composite pattern.

1.2.2 XML Download

Data import to the Data Structure is done by downloading an XML-file from an application server. The XML is parsed using the `org.xml.sax.*` parser. The application specific part of the parser instantiates `TicketingComponents`. These are merged into the existing tree in the data structure using the according methods in the class `YoshiDataStructure`. While merging, some criteria, like revision number of the `TicketingComponents`, are used to decide whether it is updated in the data structure tree or not.

A drawing of an example runtime object structure is presented later on in figure 4.6

1.2.3 Checkpoint

Concerning ticket checks, the Checkpoint component implements the main functionality. It waits for a barcode, read in by the according hardware, and produces two sorts of outputs: (1) The result of the check appears as output to the hardware-periphery, namely a message on the display and a light and sound effect. (2) The result has to be stored in the Data Structure (section 1.2.1), from where it will be propagated to other devices.

1.2.4 Communication

Several Yoshi-devices can be run at the same event. This enables greater throughput of people at the ticket control station. Additionally it gives more flexibility for the number of entries.
The Yoshis have to communicate with each other to ensure that they have the same information. For having more flexibility and to avoid a single point of failure, the programmer of the software decided to set up a peer-to-peer-network. As software library they used JXTA [Sunc], which promises to set up connections, so called pipes, through whatever the underlying physical connection is (Ethernet, WLAN, modem, GPRS).
At startup each peer sets up a broadcast pipe to talk to all Yoshis at once and an individual pipe to ensure other Yoshis can send messages to it. Normally all messages are sent over the broadcast pipe, except for commands that are to be executed only on one machine (e.g. a request to resend some messages of a remote peer).
As already mentioned, all peers contain all data. Therefore, in the case of a broken network connection, each device is still able to decide whether a ticket is valid or not. In this case, newly created data is only stored locally. When the network connection is restored the information will be synchronized again. This will be done by a resend-request of the missing messages.

1.2.5 Other components

The Main Controller is the first program started at runtime. It initializes other components like the Properties, the Data Structure (section 1.2.1) or the Checkpoint (section 1.2.3).
The Autosaver is used to serialize data to store on the disk and to recover them. Saving is done at shutdown of the software and loaded on startup. At runtime the autonomous Autosaver-thread stores data to disk every few minutes to avoid loss of data in case of a hardware machine stop, like a power outage.
There are some other components of the software like User Interface, XML Upload or Hardware Controlling. We will not analyze them in detail and therefore they are not presented in this application description.

1.3 Selection of components

Since the whole application has more than 50'000 lines of code[a], we want to annotate only the classes of a few components. Therefore we have to select these components and define the contained classes as our working set. For each class in the working set we are going to annotate the whole source code. The surrounding of the working set consists of direct dependencies of the working set, namely other classes, library- and framework-dependencies. For this surrounding we have to provide a stub, containing Universe annotations, but without implementation of the whole source code. Because we are working with the JML-tools [JML], we can write a *.refines-spec file for each class of the surrounding of the working set.
The Data Structure (section 1.2.1) is the most important part of the software and therefore I will focus on the package `com.cinerent.beans.*` where all the concerned classes are in.
Additionally, we are interested in components directly interacting with the data structure, namely we have chosen to analyze the XML Download (section 1.2.2). The sax-framework for XML is used in this component, but only its interface will have to be annotated.
Next, it would be interesting to analyze the Checkpoint (section 1.2.3), which is the part of the application which actively generates data. Investigations of this component will be done if there is enough time left.
The working set generates a big list of dependencies which can be reviewed in the table in appendix B.1.
All dependencies of the working set are mentioned in the list of necessary stubs (appendix B.3). Dependencies to the Java API are found in the list "Java API to annotate" (appendix B.4).

[a] including all empty lines, comments and some CVS logs that are included in the *.java source files

Chapter 2

The Universe type system

In object-oriented programming languages, having a reference to an object normally implies to have full read-write access to that object. This makes flexible data structures possible and supports architectures with complex control flow. But it makes it difficult to control all references to an object.
If there is more than one reference to an object, the object is called *aliased*. Aliasing is a direct consequence of object identity and appears in three origins: (1) An object can be *shared* among several objects, (2) an object passed to a data structure and stored there is called *captured* and (3) an *escaped* objects occurs when a reference to an object of a data structure, which is supposed to be internal, is passed outside [Mül04, lecture 6: Aliasing].

2.1 Concepts of the Universe type system

2.1.1 Encapsulation

The Universe type system is a technique to control aliasing [MPH01]. It introduces universes that hierarchically structure the object store of a runtime environment. Each object belongs to exactly one universe, namely the universe of its owner. Additionally, it defines its own universe. When an object is instantiated (keyword `new`) an additional keyword defines the ownership relation: (1) either the new objects belongs to the same owner and universe as the instantiator is in (`peer`) or (2) it is owned by the instantiator and belongs to its universe (`rep`). For the second case (`rep`) we can say that the new object belongs to the internal representation and therefore is encapsulated from the outside.
Alias control is enforced with the following rule:

Rule 2.1 *Unrestricted access from an instance method of an object* **u** *to another object* **v** *is only allowed if (1)* **u** *is the owner of* **v** *or (2)* **u** *and* **v** *are in the same universe.*

In contrast to conventional Java, where encapsulation is done on class level, universes provide encapsulation on object level.

2.1.2 Read-write and read-only references

Since rule 2.1 is too restrictive for many applications, the Universe type system introduces read-only references, which may refer to objects in any universe.
Access to read-only references is restricted. As the name implies, only operations are allowed that do not modify the internal state of the referenced object.

Since read-only references are less capable then read-write references, we can formulate the following rule:

Rule 2.2 *Read-write references can always be assigned to read-only references; but not vice versa.*

Since we want to forbid read-write access on any object through read-only references, we define `readonly` to be transitive:

Rule 2.3 *Read-only is transitive.*

Because read-write references are either `peer` (in the same universe) or `rep` (point into the own universe) we can formulate the following rule of thumb (we call this rule *rule of thumb* because introducing the global extension introduces exceptions to this rule):

Rule 2.4 (Rule of thumb) *References that cross universe boundaries are read-only.*

2.1.3 Annotations

In the source code, the universe relations have to be annotated. References that point to objects in the own universe, the so-called internal representation, are annotated with `rep`; references to objects in the same universe are `peer`; and read-only references are `readonly`.
E.g. a public field of a object belonging to the internal representation is declared as follows:

```
public rep Object o;
```

To instantiate an object, the universe in which the new object has to be created, has to be given. It is not allowed to use `readonly`, since a respective universe cannot be determined.
The following example declares an instantiation of an object in the same universe as `this` is:

```
new peer Object();
```

2.1.4 Type System

The properties of read-write and read-only references can be formulated as a type system: As *standard Java types*, we take the set of declared type identifiers of a given Java program. Each standard Java type combined with each universe annotation (`peer`, `rep` and `readonly`) yields in new universe types that are used.

Read-only as supertype of all universe types Corresponding to the rule 2.2 we can formulate:

Rule 2.5 *The read-only types* $\mathbf{R} = \mathit{readonly} \times \mathbf{V}$ *is the supertype of all corresponding read-write types* $\mathbf{U} = \{\ \mathit{peer}, \mathit{rep}\ \} \times \mathbf{V}, \quad \forall\, \mathbf{V} \in$ *standard Java types.*

E.g. `readonly Integer` is a supertype of the type `peer Integer`.
For each standard Java type `T` exists a subtype relation like drawn in figure 2.1.

No runtime overhead The Universe type system and the according rules that are introduced so far can all statically be checked. Therefore runtime errors can be avoided as well as any runtime overhead.

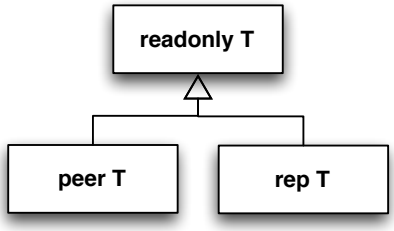

Figure 2.1: `peer T` and `rep T` as subtypes of `readonly T`. There is such a triple for each standard Java type `T`.

2.1.5 Downcasts

Due to the supertype relation of read-only types to the other universe types, we can provide downcasts from read-only references to read-write references respectively. If a read-only reference points into an universe U, only the owner of U and objects belonging to U can downcast the read-only reference into a read-write reference. Like downcasts of standard Java types, these universe downcasts need runtime checks and cannot be type checked at compile time.

2.1.6 Pure methods

Methods that do neither modify the objects internal state nor the one of any referenced object are called *pure methods*. Pure methods can be called on read-only references; they have to be annotated with the keyword `pure`.
A method that overrides a pure method has to be implemented as a pure method as well. We call this *inheritance of purity*.
Instance fields and method parameters are treated as read-only references in pure methods. Since casts from a read-only type to a read-write type are forbidden in `pure` methods, purity can be checked statically.

2.2 Notation and example

To demonstrate the universe notation of this report we present an example from [MPH01].

Example of a linked list and an iterator In listing 2.2, the class `Node` is used to compose a `LinkedList`. Additionally, the `LinkedList` provides an iterartor `Iter` to access all elements from the list consecutively.

2.2.1 Ownership diagram

Objects and references of a program at runtime can be visualized in an ownership diagram.
A square is used to represent an object, where the type of the object is written as `:Type`. Ovals represent universes. References are drawn as arrows: read-only references are arrows with dotted lines, while read-write references are arrows with continuous lines.
In figure 2.3, the ownership diagram of one possible run of the `LinkedList`-example is drawn, according to listing 2.2.

```
1  class Node {
2      peer Node prev, next;
3      readonly Object elem;
4  }
5  class LinkedList {
6      rep Node first, last;
7      void add(readonly Object o) { /*...*/ }
8      peer Iter getIter() { return new peer Iter(this); }
9  }
10 class Iter {
11     peer LinkedList list;
12     readonly Node position;
13     Iter(peer LinkedList l) {
14         list = l;
15         position = ((readonly List) l).first;
16     }
17     readonly Object next() {
18         readonly Object result = position.elem;
19         position = position.next;
20         return result;
21     }
22 }
```

Listing 2.2: Source code of a linked list with its node and iterator.

2.3 Introduction of a global universe

This case study is about a real world application. This is why we expect to meet static fields and program parts that are not assignable to a universe, as described in the previous sections. An approach to face these problems is presented in [Häc04]. It introduces a `global` universe outside of all previous universes (see figure 2.4).
Objects in the `global` universe have to be instantiated with the universe keyword `global`. Read-write references to objects in the global universe have to be annotated as `global` as well. The `global` universe type is a subtype of `readonly` like all other read-write universe types.
The `global` universe can be used for system resources (e.g. `System.out`), logging or singleton patterns [GHJV95].
As an example, in figure 2.4, each instance of the class `MyObject` has a `global` reference to a `Singleton` object **s**. A field of **s** points to another instance **globalO** of `MyObject`.

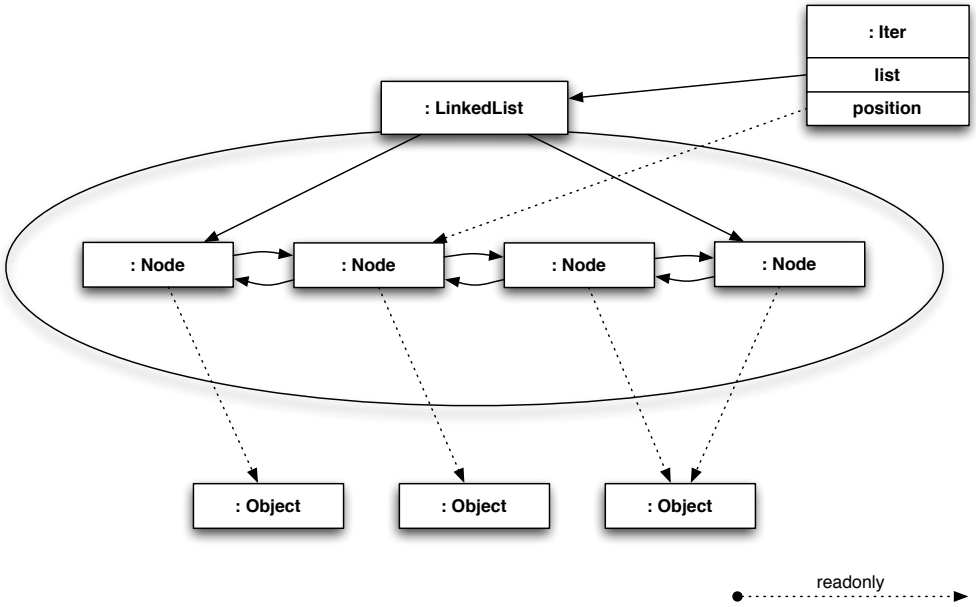

Figure 2.3: Ownership diagram for the `LinkedList`-example.

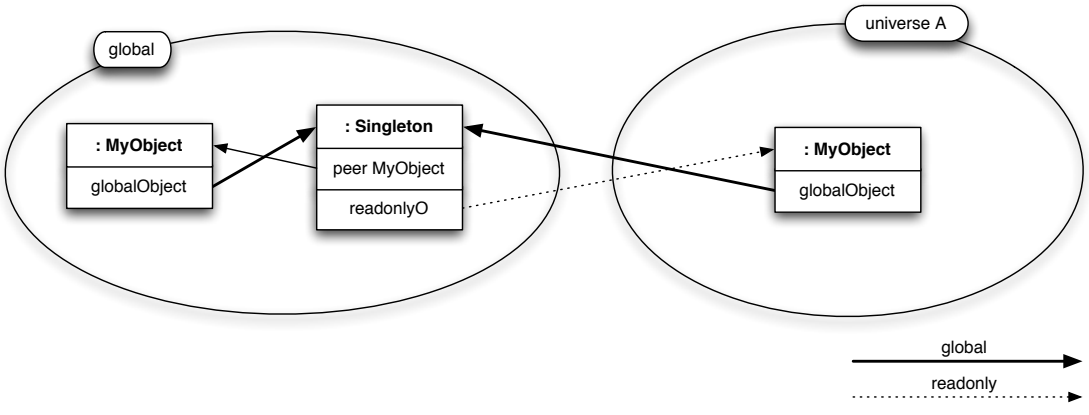

Figure 2.4: The `global` universe is outside of all previous universes. References to that universes and newly instantiated objects in that universe have to be declared as `global`.

2.4 Type combinator

The type combinator of the Universe type system describes, how universe types of two objects have to be treated in case of a method call or field access. (In the following enumeration *type* means universe type.)
(1) The actual type of the *return value* of a method call or field access `v.f` is calculated by combining the type of the target `v` with the formal type of the return value of `f`.
(2) The actual type of a *method parameter* must be a subtype of the combination of the type of the target with the formal parameter type.
The result type is defined by the rules 2.1, 2.3 and 2.5 of section 2.1.

	`peer`	`rep`	`readonly`	`global`
`peer`	`peer`	`readonly` (`rep` if called on `this`)	`readonly`	`global`
`rep`	`rep`	`readonly`	`readonly`	`global`
`readonly`	`readonly`	`readonly`	`readonly`	`readonly`
`global`	`global`	`readonly`	`readonly`	`global`

Table 2.1: Type combinator, including the `global` universe type

In table 2.1, the type combinator is presented including the `global`-extension. Some special cases are pointed out in the following:

Case `readonly` on at least one side Due to rule 2.3 the result type must be `readonly`.

Case `rep` on the right hand side If the first argument is `this`, the result is `rep`. In all other cases the result reference crosses the universe boundary of the first argument and therefore must be `readonly` (rule of thumb 2.4).

Case `peer` on the right hand side The result reference points into the same universe as the first argument (`peer` of the first argument).

Chapter 3

Annotation of Java API

We have annotated some Java API, listed in Table B.4. Additionally, we needed to annotate some classes from the log4j [Apa] and XML-SAX [SAX] libraries.
After a few technical notes, we will present some cases as examples, how the java API has been annotated (section 3.1). Some problems, that have been encountered will be presented and categorized in the next sections: (1) cloning in subsection 3.2.1, (2) iterators and `readonly`-collections in subsection 3.2.2 and (3) a need for a read-write parameter type subsection 3.2.3.

Some technial notes The SourceForge project JML-specs [JML] has been used and therefore only the signatures had to be annotated. The implemented or extended files can be inserted in the directory `specs` of the JML2 project. A script to generate a patch (`makeSpecsPatch.sh`) is included with the sources; the patch (`specsPatchXtom.tgz`) can be extracted in the root directory of the JML2 project, or even be added to the version control of the project.

3.1 Some examples of the annotated API

`java.lang.Object` In the class `Object`, we defined the compare method `equals(Object o)` to be pure and therefore, the parameter `o` is `readonly`.
Since objects of the class `String` are immutable, the method `toString()` returns a `readonly String` (see also section 6.1). This method could even be `pure`, but because of benevolent side effects, that often are implemented in overriding methods, we decided to leave `toString()` non-pure. (A possible benevolent side effect could be caching of the returned string.) We recommend that classes that override `toString()` without side effect make it `pure`.
The method `clone()` could not be annotated satisfyingly. See subsection 3.2.1.

`java.lang.Exception` Exceptions generally are easy to annotate, because they are always handled as `readonly` references [DM04] and in general have pure methods only.

`java.lang.System` To be accessible from all universes, we decided that system properties, system in-, out- and error-streams and other system resources are in the `global` universe.

Collections of `java.util` According to the well-known linked list example from [MPH01] collections operate on `readonly` keys and values. This implies a lot of casts everywhere in applications these collections are used. We recommend to state a comment on declarations of collections, if the owner

of all elements in the declared collection is invariant. As an example, a comment of a declaration of a field from the class `TicketingV2HandlerImpl`:

/ UTSCONVENTION: all objects in this Stack are rep. (they are parsed from the XML.) */*

The method `iterator()` cannot be `pure` since we want it to return a read-write `Iterator` (see subsection 3.2.2).

`java.util.Iterator` is treated in subsection 3.2.2.

`org.apache.log4j.Logger` Logging is done from everywhere in the code. So every object should be able to get a read-write reference to a logger. Therefore we decided, that all loggers are in the global universe. This is leading to the following signature of the loggers factory method:

```
public static global Logger getLogger(readonly Class c);
```

Nevertheless it is not possible to log from pure methods (because the log methods `debug(..)`, `info(..)`, etc. are non-pure).

3.2 Encountered Problems

3.2.1 Clone

Consider two objects **u** and **v** in two different universes $\mathbb{A}$ and $\mathbb{B}$.
Object **u** has a `readonly` reference `f` to **v**.
The question now is: Called from a method in **u**, in which universe will `f.clone()` be?
From a programmers point of view it probably would be nice to get the clone into 'his' universe $\mathbb{A}$, meaning the same universe the calling object is in, because this is a simple way to get a read-write reference to a copy of **v**.
But this cannot be achieved, because **u** is not allowed to instantiate objects in the universe $\mathbb{B}$.
Another possibility is to create the clone in the same universe as the origin is. This would end up in a signature of `clone()` like that:

```
peer Object clone();
```

In this case it is only possible to get a read-write reference to the clone if we already had a read-write reference to the origin.
To enable universe transfer copying, the clone concept, as implemented in Java is not feasible. In section 5.5 we present a workaround with a copy mechanism of a read-only object structure to grant full access to the copy of the given object.

3.2.2 Iterators

Iterators are used to run over a collection and perform some operation on each element. Intuitively we could say if we have a read-write reference to such a collection, we should be allowed to do any operation on it and if we have a read-only reference, we are only allowed to invoke pure methods on the objects of the collection.
The remaining question is, how to get an iterator, we are allowed to work on. Especially, if the reference to a given collection is read-only.
First problem is that the method `iterator()` is not pure. Therefore it is not possible to get an iterator on a read-only collection.

```
1  class Vector extends Collection {
2      peer Iterator iterator();
3      peer Enumeration elements();
4      synchronized pure readonly readonly Object[] toArray();
5  }
6
7  interface Iterator {
8      boolean hasNext();
9      readonly Object next();
10     void remove();
11 }
```

Listing 3.1: Parts of the interfaces of `java.util.Vector` and `java.util.Iterator`.

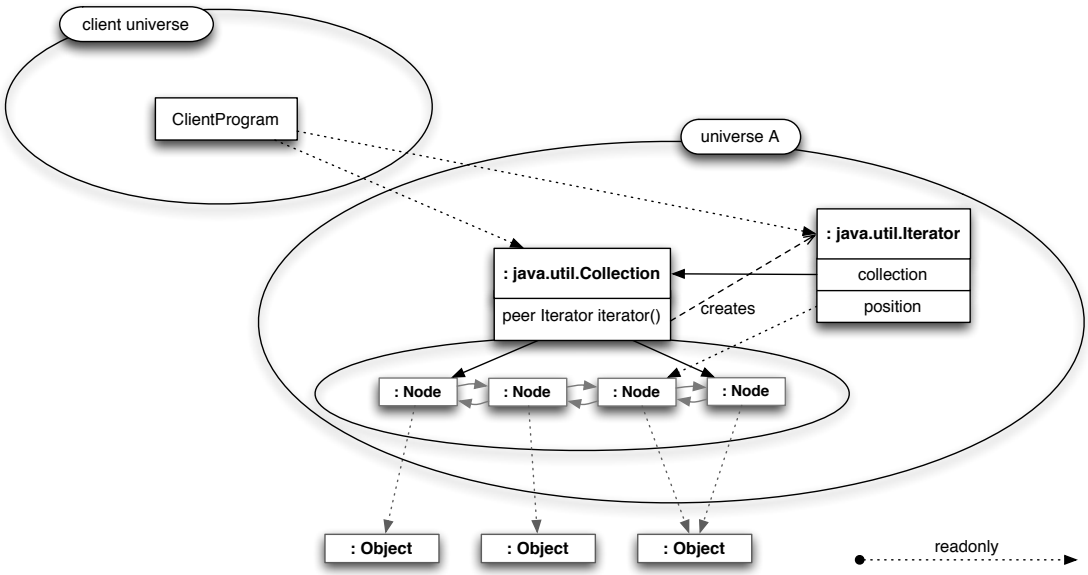

Figure 3.2: Collection and according iterator, like provided by the Java API.

If we found a way to call the method `iterator()`, regardless of its non-purity, the returned iterator would still be in the same universe as the originating collection. Therefore we still only had `readonly` access to the iterator; this situation is illustrated in figure 3.2.

A `readonly` iterator is useless, because the method `next()` has to be non-pure, since it has to modify the iterators internal state (e.g. store the actual position of iterating). Therefore a client needs a read-write reference to an iterator to operate on it (see subsection 3.2.2).

We can see, that the concept of iterators, how it is implemented in `java.util.*`, is not feasible in conjunction with the Universe type system. To work on read-only collections, we have to implement our own iterator classes. An according solution is presented in section 5.4.

There are similar complications with enumerations and all other collection classes.

3.2.3 Need for a writable parameter type

Some methods could have been annotated more generally, if we had an universe type, that is assignable for all read-write universe types (`rep`, `peer` and `global`). The universe type `readonly`, as supertype of all read-write universe types, does not fulfill this need, since it is not a read-write type.

In listing 3.3 two such methods are presented. The parameter `byte[] b` is used to write the read in bytes from the input stream. From the point of view of an `InputStream`, the parameter `byte[] b` could be either `rep`, `peer` or `global`. But since this class is part of a library, at programming time it is not yet known, which of the three types will be used.

```
1  package java.io;
2
3  /** JML's specification of InputStream.
4   * @author David Cok
5   *  (following Leaven's spec of OutputStream)
6   */
7  public abstract class InputStream {
8
9      /** XTOM: param should be writable. peer assumed.*/
10     public int read(peer byte[] b) throws IOException;
11
12     /** XTOM: param should be writable. peer assumed.*/
13     public int read(peer byte[] b, int i, int j) throws IOException;
14
15     // ...
16 }
```

Listing 3.3: Part of the file `InputStream.refines-java` as an example. The two methods `read(..)` could be annotated more generally with writable parameter `byte[] b`.

In the annotated API, we used the comment of the following form to mark such situations[a]:

/ param should be writable. peer assumed. */*

[a] To find these situations use the following grep command in the JML-specs directory: `grep -ri "param should be"` .

A first approach

One approach to such a more general type is to introduce a new abstract universe type *writable*. Like drawn in figure 3.4 writable is supertype of all read-write types and subtype of `readonly`.

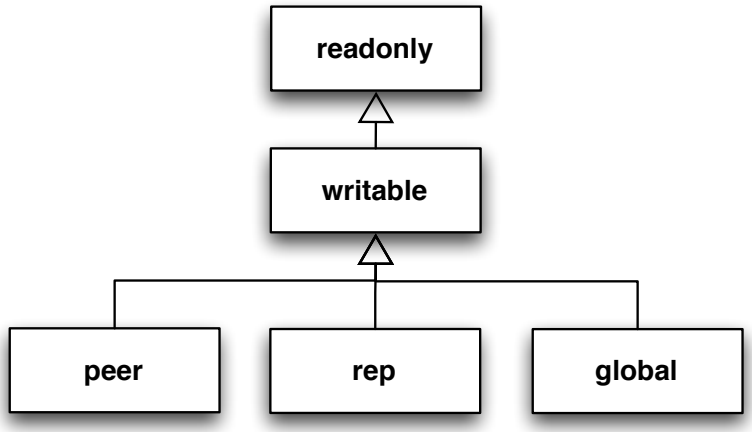

Figure 3.4: Writable as abstract universe type: direct subtype of `readonly`.

Problems are caused by static type checking of such an abstract universe type writable and its type combinator. Therefore we do not immediately propose writable to be introduced as new universe type, like presented in figure 3.4. More proposals facing this problem are presented in section 6.3.
Anyway, in the following we take the liberty of using *writable* to express situations related to the introduced problem in this section.

Chapter 4

Annotation of an application

In this chapter we first present approaches of annotation strategies (section 4.1). In section 4.2 and section 4.3 we describe, how we applied the Universe type system the selected software components. Section 4.4 is an annotation guide, where some indications for later application of the universe types system to existing code are presented.

4.1 Annotation Strategies

We developed two annotation strategies: (1) One of them looks at the whole application at once. It suggests an ownership diagram as first step, which has to be applied to the source code in some further steps. (2) The other strategy is an incremental approach. Changes are made at one place in the code and then dependent code is adapted.

4.1.1 One-step approach

This approach provides a strategy that allows one to assign universes to a whole application at once. It assumes a good knowledge of the application and its processes. Especially it is important to know which objects act on which other objects and what the meaning of the important references in the code is. References are important, if they are part of a data structure or if they connect several components of an application. (Unimportant references are those of type String or Number, and references that are only used within the same component, but are not essential for its internal structure). As a rule of thumb: important references are drawn in step A) of this strategy.
The strategy has four steps, but the first one is the most important one:

A) Draw the ownership diagram. Take the important objects of all components and make a picture, where for each object a unique owner is declared. By definition of the Universe type system, all objects with the same owner belong to the same universe. Draw the universe boundaries. The ownership diagram should clarify who is the owner of a whole component, if there are some.
Now try to map the ownership relations to the owning classes: State which fields and references (e.g. parameters of methods) have to be read-write and where `readonly`-references are sufficient.
(Ownership diagram is introduced in subsection 2.2.1.)

B) Declare universe types of the fields. Based on the ownership diagram, for each field of each class, the universe type can be determined. If the instance of the class is the owner of the object

referenced in the field, declare the field as `rep`. If the field is a reference to an object in the same universe, declare it as `peer`. References outside the same universe, even to the owner, have to be declared as `readonly`.

C) Annotate the methods. All types of the methods parameters and return values have to be annotated next. Make methods `pure` if they do not modify anything. (Consider the purity rules from the Universe type system, section 2.1.6; especially inheritance of purity.)
Additional methods might have to be provided, to perform operations across universe boundaries (e.g. operations on objects that are in the universe of a `peer` object **u** have to be called through a wrapping method on that `peer` object **u**; see workaround in section 5.1). Some additional methods might have to be introduced to allow operations to be executed on references that are `readonly` (see workaround in section 5.3).

D) Local variables and implementations. Finally the implementations of the methods and the according local variables have to be annotated. Auxiliary methods, introduced in step C, have to be called respectively. If the formal type of a return value of a called method is `readonly`, but you know it must be a read-write universe type, you have to use downcasts.
If all steps A) - C) have been executed carefully, step D) should be easy to implement.

Characteristics This approach is analytical. That is after having arranged the ownership diagram, universe annotations are defined for all classes. When all classes are annotated according to the ownership diagram, the application is fully annotated. The big disadvantage is that a lot of universe boundaries are introduced at once. So it easily could happen that the according restrictions imply a lot of workaround to implement. This may end up in a complex implementation session.

> *According to modern programming techniques, like extreme programming XP [Bec] or refactoring [Fow03], complex implementation sessions should be avoided.*

4.1.2 Incremental approach

While the one-step approach is analytical and plenty of work is to be done at once, we now present an incremental approach. Incremental in the sense of having an intact system, making a little step, and then reestablishing an intact system.
The first step seems simple, but may imply a bunch of side effects: Compile the application with the universe compiler and ensure there are no errors. Side effects that may complicate this step are depending on library classes, that provide `readonly`-references.
After having eliminated all compile errors, declare a universe annotation at one place in the code (e.g. make a field of a class `rep`). This change implies some other changes at the application. Implement these changes to reestablish an intact and compiling system. Then continue by choosing any restriction in the application you would like to introduce, make the change and reestablish the compiling system.

Characteristics Because small steps are taken and the system is kept intact, the incremental approach is easy to use for the programmer.
A big picture of the annotated system is missing (like the ownership diagram in the one-step approach). Therefore it might happen that a restriction is introduced which disables some other restriction that might have been useful. We made no further investigations about the likelihood of such a mischance.

4.1.3 Combination of the two approaches

If we combine the two approaches, we get a strategy, suitable for practice.

1) Preparation step. It seems useful to start with a compiling system. Therefore we first have to ensure that all library dependencies are fulfilled.

2) Draw the ownership diagram. An ownership diagram will help us to decide in the further steps. According to step A) from the one-step approach, draw a map with all important objects and components of the application. Define the owner for each object and draw the universe boundaries.

3) Annotate one unit. Choose a class (or a useful part of a class) and annotate it with the universe annotations, according to the ownership diagram, drawn in step 2). To simplify the process, you can apply steps B) to D) from the one-step approach to the chosen unit.

4) Clean all dependencies. According to the idea of the incremental approach, annotate all dependencies of the unit chosen in step 3), such that there are no compile errors left.

5) Iterate steps 3) and 4). Repeat steps 3) and 4) until you have annotated all classes. Orientate yourself at the ownership diagram drawn in step 2). Redraw 2) if there are better design ideas.

Characteristics The combination of the one-step approach and the incremental approach tries to get the advantages of both. Due to the drawn ownership diagram, we have an analytical approach, that declares an objective to achieve. The iteration steps of limited size prevent big implementation sessions. Further, they allow to estimate the upcoming effort.

4.2 Annotation of the Data Structure

In this section, we describe, how the Data Structure (section 1.2.1) has been annotated with the Universe type system.

4.2.1 Definition of the data universe

The Data Structure has two main parts (as described in 1.2.1): (1) The tree of all data components and (2) an index over all components stored in the tree for fast access.
The whole Data Structure has to be encapsulated; all changes should be done through the facade class `YoshiDataStructure`. Therefore the universe of the single instance of this class has been defined as the data universe (see figure 4.1). We introduced the following invariant for objects in the Data Structure:
Whatever is stored in the Data Structure is in the universe of the `YoshiDataStructure`.
To use one single universe for the whole Data Structure is not very restrictive, but two arguments made us use this approach: (1) There are two representations of the data, a tree and an index. Methods in the Data Structure component use both of them to access the data. As long as we can use the invariant a lot of downcasts to peer-references are justifiable. (2) If we tried to introduce a universe for every layer, a lot of syntax- and runtime-overhead had would be produced (as an example see the prototype implementation of the method `insertInChild(...)` in section 4.2.5).

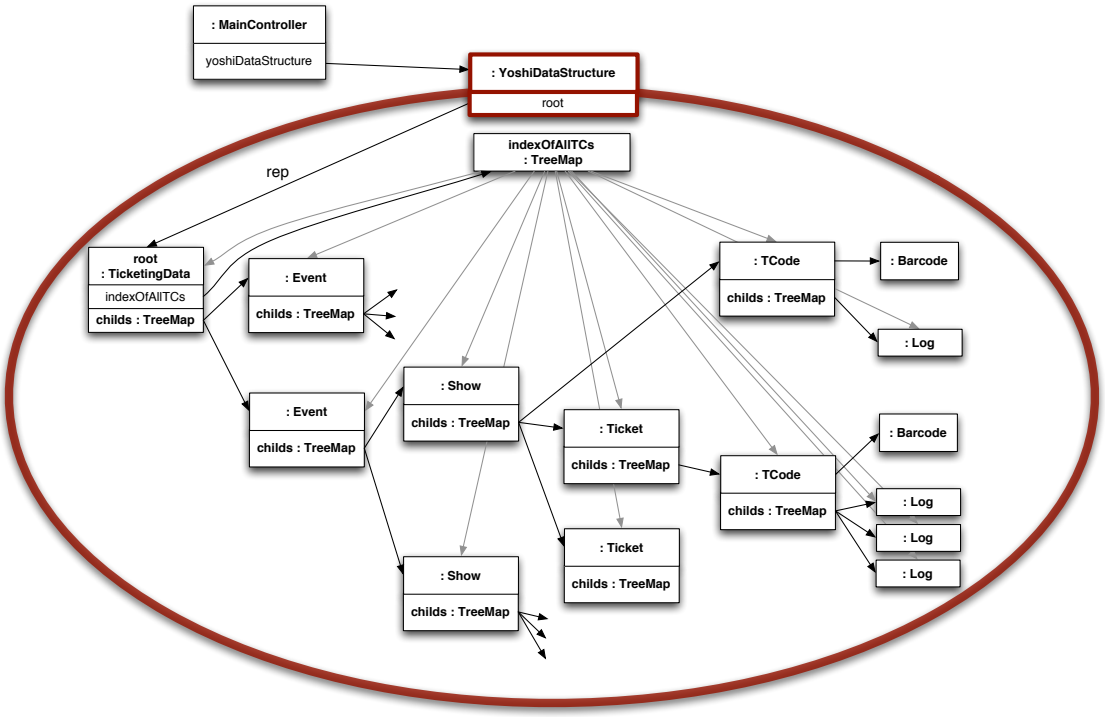

Figure 4.1: Writable as abstract universe type: direct subtype of `readonly`.

The introduction of the data universe gains the improvement that the usage of the class `YoshiDataStructure` as facade class is enforced, since no other write access to the data is allowed. Furthermore, the index in the class `TicketingData` had to be changed from a static field to an instance field (see next subsection).

4.2.2 Index as instance field

The index that links all components that are stored in the Data Structure has been stored in a static field of the class `TicketingData`, as you can see in Listing 4.2. To declare in what universe the index is to be initiated, the field has been changed to an instance field and instantiated in the data universe. The according methods have to be changed to instance methods and calls to these methods have to take place on the root object of the data tree, which is of type `TicketingData`.

```
1 /**
2  * Hold all valid instances of {@link TicketingComponent}.
3  */
4 private static SortedMap allInstances = Collections.synchronizedSortedMap(
      new TreeMap());
```

Listing 4.2: Static initializer part of the index in the Data Structure as it was before.

4.2.3 Root node as a `rep` field of `YoshiDataStructure`

The root of the data tree in the Data Structure has been implemented using the singleton pattern for the class `TicketingData`. To ensure the root object to be in the data universe, we moved its instantiation to a well-defined place in the initialization phase in the class `YoshiDataStructure`.

> *It seems to be a good approach to replace singleton patterns, by defining an explicit object to be the owner of the singleton object. The singleton instance is stored in an instance field of the owner. If there is no suitable structure in the program to put this field in (and ensure, the referenced object is the sole instance), one can still use the main program as a unique owner. (see section 5.2)*

4.2.4 Data structure in field of the main controller

```
1  public class YoshiDataStructure {
2
3      private static YoshiDataStructure instance = new YoshiDataStructure();
4
5      /**
6       * constructor.
7       */
8      private YoshiDataStructure() {
9          // ensure that a tree root node exists.
10         dataStructureRoot();
11     }
12
13     /**
14      * implementing a multithreaded singleton.
15      */
16     public static YoshiDataStructure getInstance() {
17         return instance;
18     }
19 }
```

Listing 4.3: Before applying the Universe type system to the software, the `YoshiDataStructure` has been implemented with the singleton pattern.

So far, the whole Data Structure component has been based on a singleton object of the class `YoshiDataStructure` (see listing 4.3). The single instance has been stored in a static field of the class `YoshiDataStructure`. This implies problems, since it is not defined, which universe this instance belongs to. Because we want full read-write control of the Data Structure, we decided to make the single instance of `YoshiDataStructure` a field of the `MainController`.

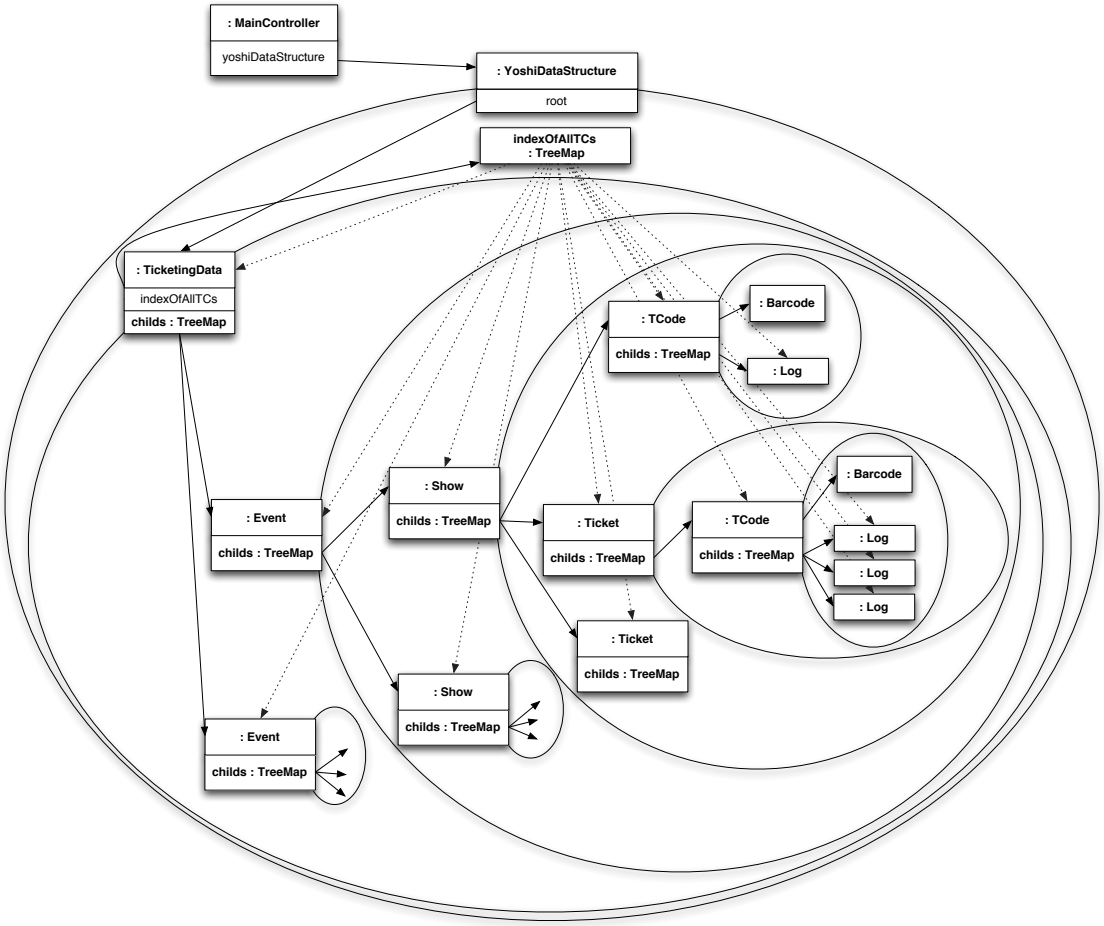

Figure 4.4: Writable as abstract universe type: direct subtype of `readonly`.

4.2.5 Deeply nested Data Structure

Instead of making one single data universe, like presented in subsection 4.2.1, a separate universe could be introduced for each node in the data tree. All children would be stored in the own universe respectively. See figure 4.4 for a ownership diagram. This approach implies a lot of restructuring and runtime overhead, since new, recursive methods are needed.

In listing 4.5 a method to insert a component in the structure is presented as example. On every level of the tree the path to the parent of the new component has to be searched (lines 2 to 6 in listing 4.5). This is implemented, by following the `parent()` reference of the new component (line 5). After the while-loop the child, in which the new component has to be inserted, is stored in `roParent`. Since `roParent` is a child of `this`, the cast to `rep` is allowed (in line 8). If the found component `repParent` is the direct parent of the new component, we can insert it directly; otherwise, we have to call `insertInChild()` recursively on `repParent`.

Such a method is needed for all operations on elements of the data tree, which are a lot of lines of code. Additionally, all the recursive searches are a lot of runtime overhead.

```
1  void insertInChild(readonly Component comp) throws Exception {
2      readonly Composite roParent = comp.parent();
3      while (!this.equals(roParent.parent())) {
4          if (roParent == null) throw new Exception("parent of " + comp + "
               not found.");
5          roParent = roParent.parent();
6      }
7      // cast roParent to rep.
8      rep Composite repParent = (rep Composite) roParent;
9      if (repParent.equals(comp.parent())) {
10         repParent.insert(comp);
11     } else {
12         repParent.insertInChild(comp);
13     }
14 }
15
16 /** replaces a possible old entry with same key */
17 void insert(readonly Component roComp) {
18     // Component implements Copyable
19     // children: field of type java.util.Map
20     this.children.put(roComp.getKey(), new rep Component(roComp));
21 }
```

Listing 4.5: In a nested universe structure: Method that inserts a component at the right place in a tree (according to the reference to its parent).

4.3 Annotation of the XML Download

In the last section (4.2), we basically used the one-step approach to annotate the Data Structure. To annotate the XML Download (section 1.2.2), we used the combination of the two annotation strategies (see subsection 4.1.3).
The runtime structure of the XML Download before any changes looked like figure 4.6.
The XML Download is used from several other components (e.g. a command that loads an XML immediately or a timer-job that loads the data). This other user-component is marked in figure 4.6 as "Initiator of the XML download". The objects, instantiated by parsing the XML, are marked with green color in the figure. Some of these objects are captured by the Data Structure; the inserted references are drawn with bold arrows.
In figure 4.7, the ownership diagram of the same XML download is drawn as in figure 4.6, but after restructuring of the XML Download component. We introduced three universes: (1) The outermost universe declares, that the whole download can be encapsulated, since it can be run in the universe of any initiator of an XML download. (2) The universe of the `XMLMaster`. In this universe are all objects located, that are needed by the org.xml.sax.* library. (3) The inner universe contains the parsed data, which are collected in a list `parsedObjects`. This result list is returned by a read-only reference to the initiator of the XML download, which can insert the new data into the Data Structure. In this figure, the Data Structure is drawn as one condensed universe; this emphasis that the data structure is encapsulated.
On the left hand side of the figure 4.7, we can see an instance of a `UserTalkback` is drawn in the global universe. The `UserTalkback` implements a callback mechanism, with the aid of which the user of the software can be informed about some events, e.g. that a XML download has failed or succeeded. Since the instance of such a `UserTalkback` can either be the display of the device or a wrapper to the logger, we decided to make it global.

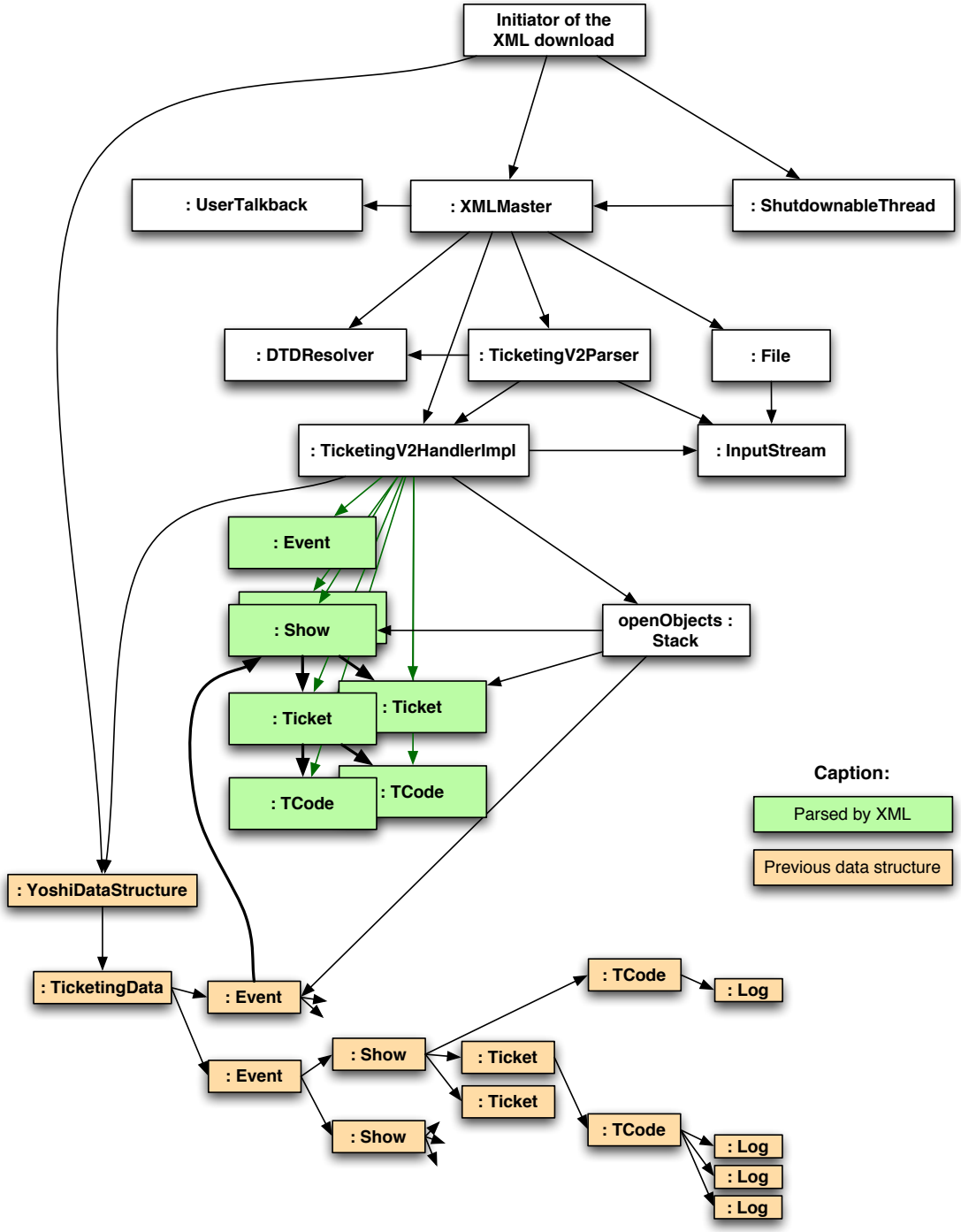

Figure 4.6: An example runtime structure of the XML Download before any universe annotations. It shows the object structure of a download of an XML, where a show is added to an existing event. Bold arrows mark references added to the Data Structure by parsing the XML.

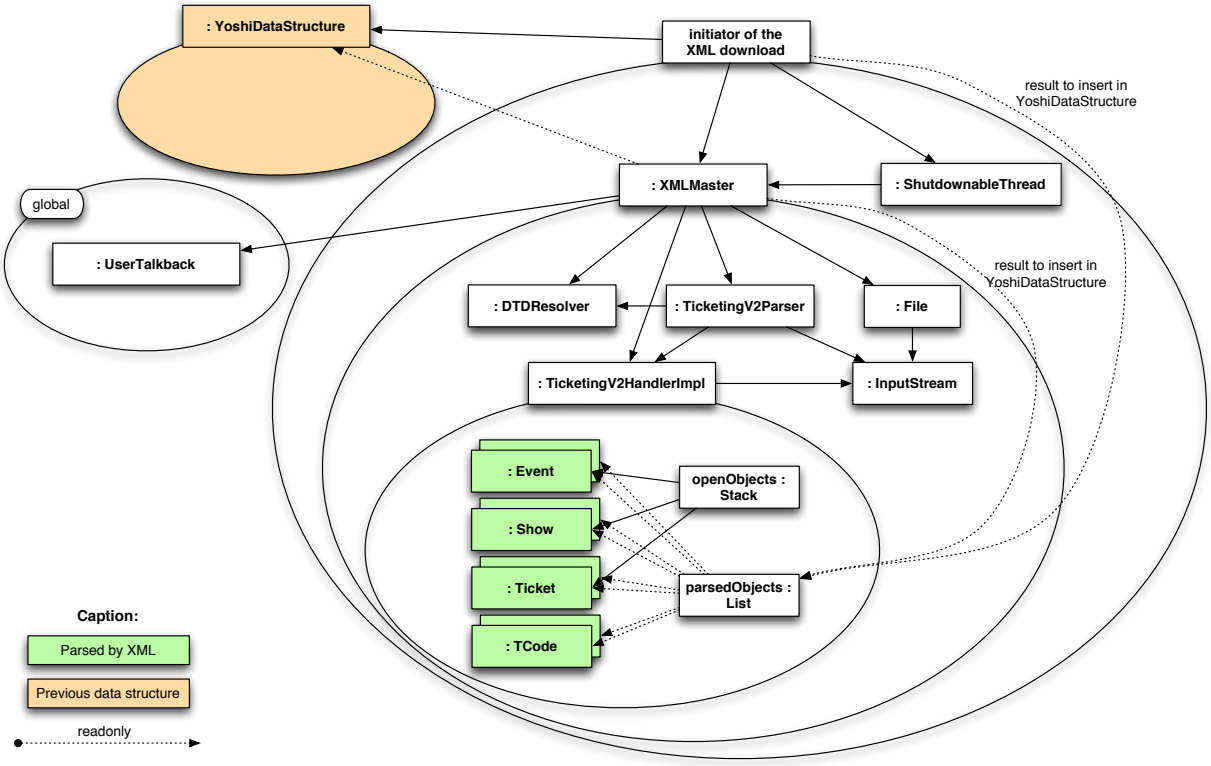

Figure 4.7: The ownership diagram of the annotated component XML Download.

4.3.1 Communicator thread

The `Communicator` is a thread that, among other things, schedules the download tasks. It uses a list of `TicketingComponents` that have to be downloaded; all entries of that list implement the interface `XMLDownloadable`. Other components can register `XMLDownloadables` that are desired to be loaded. To download one of these entries, the `Communicator` instantiates a `XMLMaster` and passes control over the `XMLDownloadable` to it.
Because other components should have the ability to add entries to the list, we decided to put the list into the global universe. The communicator, which is implemented as singleton pattern, provides a `global`reference to that list; therefore even if access to the communicator was read-only, another component can downcast this reference to `global` and insert some `XMLDownloadables`. An according ownership diagram can be reviewed in figure 4.8.

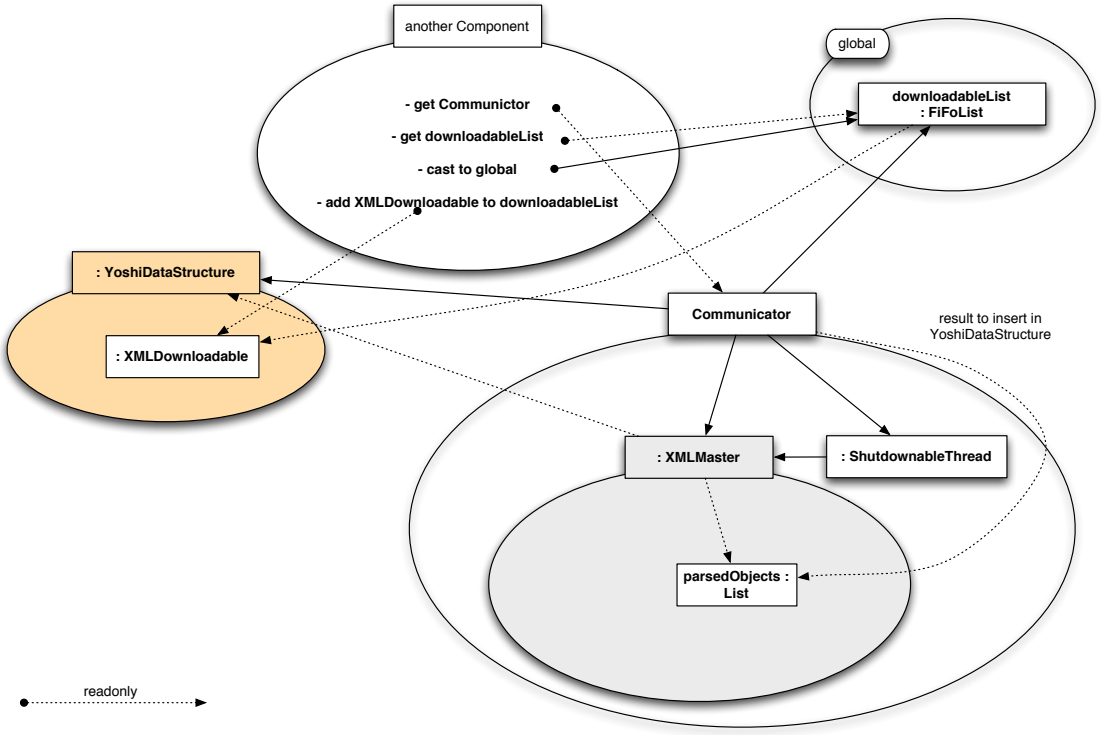

Figure 4.8: The `Communicator` thread is an initiator of the XML download. It maintains a `global` list of components that have to be downloaded.

4.4 Annotation guide

In this section we describe some guidelines, how to annotate an existing application with universe types. The predications are based on the experience we made by annotating parts of an industrial application, like described in the previous sections of this chapter.

4.4.1 Annotation strategy

In section 4.1 we described several approaches for annotation strategies. Regardless of which strategy is used, it seems to be important to have an ownership diagram. If the main part of the application is drawn on such a ownership diagram, this helps to keep track of the annotation process.

4.4.2 Superfluous Java access modifiers

Since the Universe type system provides proper alias control on object level (see subsection 2.1.1), we do not have to manually take care about aliasing anymore. Meaning, Java access modifiers to instance fields can be less restrictive.
Especially read-write access to `rep` fields can only be granted on `this`; all other objects can get a `readonly` access to the referenced objects at most. Therefore they do not have to be `private` or `protected` any more.

4.4.3 Flat versus nested data structures

Universes can be used to model data structures that represent the business logic of an application.
A disadvantage of building nested universe structures is, that for each non-pure method there has to be an additional facade method in the owner object of the universe. These additional methods may even be recursive (see subsection 4.2.5). This implies runtime overhead for the respective method calls.
If we can trust the objects in the data structure, it might be a better approach to introduce one single data universe (like we decided to do it in subsection 4.2.1).
Which one of these two approaches should be used, can be indicated by the following rule: The more complex methods of a class are, the more it should try to have its dependent objects in its own universe. In other words: If the data objects do not implement business dependent operations, it is sufficient to have one single data universe.

4.4.4 Top level universes

Read-write interaction between objects can only be performed, if the objects are in the same universe (or one object is the owner of the others). Therefore all components in a system that have to interact with each other have to be at the same universe level.
We recommend to give a particular attention to the top level universes, where the main components of the application communicate with each other. Interfaces and responsibilities should be determined clearly.

4.4.5 Singleton Pattern

There are two approaches to treat singletons: (1) avoid them, e.g. by refactoring or (2) make the single instance `global`.

The second approach is easy to implement and no refactoring is needed: The single instance is in the global universe and therefore can be accessed by every other object. Consequences, like described in [GHJV95], are all supported by this approach. Nevertheless, alias control is not possible.
Sometimes it is useful, that not all objects have access to a single instance of a class, but only the objects of a certain component. Conventionally, this is often implemented as a singleton pattern for simplicity. We recommend to make it an instance field of the universe provider instead. As examples, you can have a look at the root node of the Data Structure in subsection 4.2.3. (See section 5.2 as well.)

4.4.6 Global universe

Additionally to singletons, we recommend the following things to be in the global universe:

Properties are typically a singleton, which can be instantiated in the global universe.

Logging As already mentioned in 3.1, we decided to instantiate all loggers in the global universe, to be able to log from everywhere in the application.

4.4.7 Library object structures

In section 4.3 we used a library to parse an XML. As drawn in figure 4.7, we used two universes to deal with the according library object structures: (1) the universe of the `XMLMaster` in which all library objects are located and (2) the inner universe, where all generated, application-specific objects are stored.
A similar universe structure with two levels might be a good approach, if other library objects structures are used.

4.4.8 Result handling

Consider a component that builds a result object, or even object structure, in an inner universe. (E.g. like the parsed elements in figure 4.7.)
The result cannot be returned as read-write reference, since the result reference points in the inner universe. We recommend to return a read-only reference to the result objects. If desired, the client is still able to copy the whole result into its own universe, to get full read-write access to the result. Unfortunately, object identity is lost, since the objects are copied.

4.4.9 Annotated API

When an application is started to be annotated with universe types, there will be a few sticking points to pass: Since the used API is already annotated with universe types, there might be a lot of compile errors at the first compile time.
The reasons are (enumeration is not complete):

- Return types of library methods are `readonly` types, which cannot be assigned to the default `peer` type of references in the application.
  This case is especially for the collection framework of java.util.*. The situation will be enhanced, by support for Java generics [Suna].
- As soon as there are `readonly` references in a system, there will be errors for calls of non-pure methods on them.
- As soon as some methods are declared as `pure`, they will fail to compile, since almost all references in pure methods are treated read-only.

Chapter 5

Problems, Patterns and Workarounds

While applying the Universe type system to an existing application I had to use some workarounds to achieve that the universe constraints are fulfilled. These workarounds and accoring generalized solutions are presented in this chapter.

5.1 Additional methods to cross universe boundaries

Consider a situation like the following: An object **client** has to call a non-pure method `m()` on an object **inner**, which is in the universe of another object **outer**. Without universes such a situation is no problem iff **client** has a reference to **inner**; in other words, iff **inner** is aliased. Since the Universe type system introduces alias control, this is not possible anymore.

A suitable solution is that the owner **outer** provides a facade method `fm()`, that delegates the call to the according object in its universe. A standard way to implement this approach is to introduce a `readonly` parameter `p` in `fm()`, on which a reference to **inner** can be given. In `fm()` the parameter `p` can be casted to `rep` and then `m()` can be called on it.

5.2 Singleton Pattern

As already mentioned in subsection 4.2.3 and subsection 4.4.5 there are two ways of treating singletons: (1) The singleton object is created in the global universe or (2) a well-defined object is defined as the provider of the single instance of the desired class.

(1) has the advantage that all obligations of a Singleton Pattern, like described in [GHJV95] are fulfilled. The constructor can be made `private` and all access is done through a single point (method `getInstance()`). Nevertheless, there is no chance of any alias control. (2) allows to provide a singleton to be used from only a certain software component. The single instance is then stored in an instance field of the provider object. This is the same way, as singletons are simulated in Eiffel [AB04].

More flexibility can be achieved by combining the two approaches. A reference to the singleton object is stored twice: In a static read-only field of the singleton class and in an instance field of the provider object. It is even recommended to store it in a `rep` instance field of the provider object. In listing 5.1 such an alternative implementation is shown.

```
class Singleton {
    private static readonly Singleton instance = null;
    /** may return null */
    public static readonly Singleton getInstance() { return instance; }
    public static peer Singleton initialize() {
        if (instance != null) throw new RuntimeException("already␣
            initialized");
        return peer Singleton();
    }
    private Singleton() { instance = this; }
}
```

Listing 5.1: An alternative implementation of the Singleton Pattern: the initializer controls read-write references to the singleton object, while everyone is enabled to get `readonly` access.

5.3 Method needed twice

Sometimes a method is needed in a pure form as well as with a peer return value. As an example consider the method `parent()` in listing 5.2: It returns the parent node in a Tree-Data-Structure. The nodes in the tree all have an unique key object. Additionally, they store the unique key of the parent, to give the ability to restore the tree after having been serialized node by node. The original implementation of the method `parent()` (listing 5.2) checks whether a reference to the parent already exists, and if not, it sets the field `parent` to the according reference. This is a special case of caching (called lazy initialization).

Because the field `parent` is going to be set, this method cannot be declared as `pure` and because we know in this case the return value has to be in the same universe, we can return a `peer` reference. Because we need a read-write reference to call this method, we renamed it to `parentPeer()` (see listing 5.3).

To provide a `pure` method to access the parent node, I changed the default `parent()`-method to Listing 5.4. In this implementation writing to instance fields is avoided. Therefore we have no caching of the parent reference, but can provide the functionality with a `pure` method.

```
/**
 * The Parent Method returns the Parent TicketingComponent.
 * This is an essential Part of the Ticketing Data Structure.
 */
public TicketingComponent parent() {
    if (parent == null) {
        if (parentUniqueKey != null) {
            parent = YoshiDataStructure.getInstance().get(parentUniqueKey);
        }
    }
    return parent;
}
```

Listing 5.2: Method `parent()` before universe annotations.

```
1  public peer TicketingComponent parentPeer() {
2      if (parent == null) {
3          if (parentUniqueKey != null) {
4              readonly TicketingComponent p = YoshiDataStructure.getInstance()
                   .get(parentUniqueKey);
5              if (p instanceof peer TicketingComposite)
6                  parent = (peer TicketingComposite) p;
7              // else parent remains null.
8              // cannot return p, because is from another universe.
9          }
10     }
11     return parent;
12 }
```

Listing 5.3: Implementation of `parent()` as non-pure method with `peer` return value.

```
1  public pure readonly TicketingComponent parent() {
2      if (parent != null) return parent;
3      if (parentUniqueKey == null) return null;
4      else return YoshiDataStructure.getInstance().get(parentUniqueKey);
5  }
```

Listing 5.4: `pure` implementation of the method `parent()`.

5.4 Iterators

Iterators are used to visit a collection of objects. E.g. from another software module or to collect some information about a whole data collection. We often want to iterate over read-only collection, meaning over a collection, where we have only a read-only reference to.
As already stated in subsection 3.2.2, the first problem is that the method `iterator()` is not pure. And even if we could call `iterator()`, the returned iterator would be in the same universe than the collection is. But we do not have read-write access to that collection.
The first approach to face this problem is to not use the iterator defined by the the abstract method `java.util.Collection.iterator()`. Instead a generic iterator has to be used, which is able operate on a `readonly`-collection. To get read-write access to such an iterator, it has to be instantiated in the universe of the client (see section 5.4.1).
Another problem are methods that provide operations on the whole collection, using an iterator. For example search- or count-methods. From an operational point of view, one could think it should be easy to execute such a method in a pure environment, but because of the instantiation of the iterator within the `pure` context, it is not possible. I developed a workaround which is presented in section 5.4.2.

5.4.1 A generic iterator

To enable an iterator to operate on a `readonly` collection, we extended the `java.util.Iterator` to an `UTSIterator` (see Listing 5.5). This interface `UTSIterator` is basically a marker interface, telling the programmer, that the underlying `java.util.Collection` is `readonly`.
A first implementation of the interface `UTSIterator` is the class `SimpleIterator`, listed in Listing 5.7 (on page 38). In the constructor it takes the `readonly Collection`, gets an array of its elements and stores them in the instance field `readonly readonly Object[] array`. In another field `int`

```
1  import java.util.Iterator;
2  /**
3   * The basic idea is to provide an Iterator over a readonly collection.
4   * So an implementation of this interface should provide a constructor with
5   * a parameter of type <code>readonly {@link java.util.Collection}</code>.
6   */
7  public interface UTSIterator extends Iterator {
8
9      pure boolean hasNext();
10
11     readonly Object next();
12
13     /**
14      * An implementation of this method normally throws an {@link
            UnsupportedOperationException}
15      * because the underlying collection is <code>readonly</code>
16      * (and therfore cannot be modified, like {@link #remove()} would do it)
            .
17      */
18    void remove() throws UnsupportedOperationException;
19 }
```

Listing 5.5: The interface of an iterator over a `readonly` collection.

`position` the index of the element that will be returned when the method `next()` is called the next time is stored. See figure 5.6 for an example runtime object structure.
In this implementation I do not care about changes in the underlying collection. Another implementation `GenericIterator`[a] checks on every call of the method `next()` whether the index of the returned element and its precursor in the local array are still the same as in the original collection. If this constraint is violated, a `java.util.ConcurrentModificationException` is thrown.

5.4.2 Iterators in pure context

We encountered some methods provided by a data structure or one of its components, that allowed users from outside to perform some operation on the data;
e.g. recursive search functions (`TicketingComposite.containsRecursively()`) or functions that collect some data about the whole structure (`TicketingComposite.countRecursively()`).
There are two reasons, why they cannot be provided as `pure` methods. The main reason is that the method `next()` of the class `Iterator` cannot be called from a `pure` method; unfortunately not even if the iterator is instantiated locally. That is due to the method `next()` cannot be `pure` since it has to modify the internal state of the iterator. The other reason is, if there are some result- or callback-objects, handed to the method by reference-parameter, that obviously should be read-write . (e.g. an `OutputStream` to write results to or a `java.util.Map` to collect data in.)
If there are no side-effects to the data itself, the following refactoring pattern can be used to transform the method into a static one which therefore can be executed in the clients universe.

Refactoring Algorithms

1. Without refactoring tools of Eclipse

1. Make the method static.

[a] Listing A.1 in the appendix.

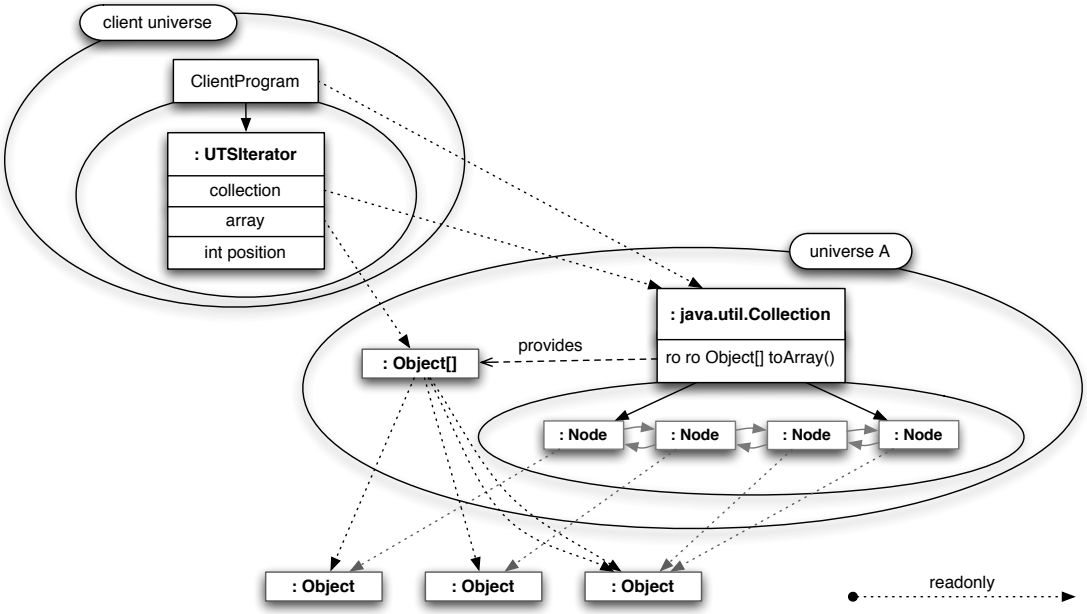

Figure 5.6: An example runtime object structure with a `UTSIterator`.

2. Add the old `this` as `readonly` parameter.
3. Change all accesses to `this` (including implicit `this`) to accesses to the introduced parameter.
4. Change callers according to new parameters.

2. With the aid of refactoring tools of Eclipse

1. Write a new static method (e.g. by copying and adapting the existing one and adding the `readonly` parameter).
2. Call the new static method from the original instance-method, use `this` as parameter.
3. Inline[b] the instance method, so the new static method is called directly. (replacement of the parameter `this` is done by the tool.)

This approach is a workaround that works for some specific cases. In section 6.2 we present an proposal that allows the usage of iterators in pure methods.

[b] To inline is a refactoring operation provided by the Eclipse refactoring tools [Fow03, Ecl].

```
1  import java.util.Collection;
2  /**
3   * This implementation of {@link org.multijava.uts.UTSIterator}
4   * stores all elements that will be returned in an array.
5   * It is pretty simple and does not care about changes on the
6   * underlying {@link java.util.Collection}. Therefore no
7   * {@link java.util.ConcurrentModificationException} will be thrown.
8   *
9   * @author Thomas Haechler
10  */
11 public class SimpleIterator implements UTSIterator {
12
13     protected readonly readonly Object[] array;
14
15     /** INV: position = the index of the element that will be returned when
            {@link #next()} is called the next time. **/
16     protected int position = 0;
17
18     /** Constructor.
19      * @param coll the underlying readonly Collection.
20      */
21     public SimpleIterator(readonly Collection coll) {
22         array = coll.toArray();
23     }
24
25     public pure boolean hasNext() {
26         return position < array.length;
27     }
28
29     /**
30      * @throws IndexOutOfBoundsException if there is no element left.
31      */
32     public synchronized readonly Object next() {
33         readonly Object retValue = array[position];
34         position ++;
35         return retValue;
36     }
37
38     /**
39      * not supported.
40      * @throws UnsupportedOperationException always. because we have
               readonly access to the underlying Collection.
41      */
42     public void remove() throws UnsupportedOperationException {
43         throw new UnsupportedOperationException("because " + this + " has
               readonly access to the underlying Collection.");
44     }
45 }
```

Listing 5.7: Simple implementation of `UTSIterator`.

5.5 Copy as a workaround for the universe-transfer-problem

In the Universe type system it is not possible to move an object from one universe to another. But in certain cases we want to enable such a transfer; e.g. a component generates an object in its universe and, after having called some modifying methods on it, wants to hand it over to another component.

Because a transfer is not allowed, a first idea has been to make a clone into the desired universe. But clone is a method executed in the context of the given object and therefore cannot instantiate its clone in another universe. Even if this would be allowed by any reason, the newly cloned object could not be handed over to the caller. Unless he has had a read-write reference to the primary object, but in this case its not a general universe transfer.

The sole possibility to get an object in a specific universe is to instantiate it in that universe. To have an equivalent object, we have to copy the fields from the originating object to the newly created one.

Approach A class, of which the instances have to be transferable from one universe to another, has to provide a copy-method. This method takes as parameter a `readonly`-reference to an object of the same type and copies all instance-fields according to the following rules:

- `readonly`-fields are copied by reference
- read-write references are copied recursively into the target universe. If there are such read-write fields, the types of these fields have to implement the `Copyable` interface as well.

These two rules describe a combination of shallow and deep copy; sometimes named as "sheep" copy.

Implementation An interface `Copyable` marks all classes that implement the sheep copy method. Listing 5.8 shows the source code of that interface.

```
1  /**
2   * It is recommended to implement a constructor of the following form:
3   * <code>MyClass(MyClass o) { copyFrom(o); }</code>
                                  */
4  public interface Copyable {
5          /**
6           * This method takes another Object of the same type
7           * and copies its internal state to this.
8           *
9           * implemenation of sheep-copy is recommended:
10          *    - new Objects for rep- and peer-references (sheep-copy as well)
                .
11          *    - copy the readonly-references and the values.
                                  */
12         void copyFrom(readonly Copyable o) throws ClassCastException;
13 }
```

Listing 5.8: The interface `Copyable` marks classes that implement the sheep-copy function.

The method `copyFrom(..)` is called on the new object, of which all fields are assigned according to the fields from the originating object, given in parameter `Copyable o`. A `ClassCastException` is thrown, if the object, given by parameter `o` is of another type than the new object itself.

It is recommended for classes that implement the `Copyable` interface to provide a constructor like an example can be reviewed in listing 5.9.

```
1 class MyClass implements Copyable {
2     /** recommended constructor */
3     MyClass(readonly MyClass o) { copyFrom(o); }
4     void copyFrom(readonly Copyable o) {
5         // nothing to do in the case of no instance fields.
6     }
7 }
```

Listing 5.9: A possible implementation of the `Copyable` interface; including the recommended constructor that takes an object of the same type.

5.6 "ambiguous" error message

We had some troubles with a sort of compile error of the same nature like the following one:

```
File ``AmbiguousExc.java'', line 4, character 63 error:  Call of method is ambiguous between (at
least) java.lang.Exception<init>( readonly java.lang.String ) and java.lang.Exception.<init>(
readonly java.lang.Throwable ).  [JLS 15.11]
```

When a method with at least one `readonly` parameter is called, this error message occurs if the actual parameter of the called method is (1) not `readonly` and (2) of a java-subtype of the formal parameter. E.g. see the call of the constructor of an exception in listing 5.10.

```
1 public class AmbiguousExc {
2     void m() throws Exception {
3         peer Object tc = new peer Integer(7);
4         throw new Exception(tc + " is peer Object / this is: " + this);
5     }
6 }
```

Listing 5.10: Compiling this class with the JML checker produces the "ambiguous" error message.

Workaround The error message can be avoided by casting the actual parameter to `readonly`. The example from above including this cast can be considered in listing 5.11.

```
1 public class AmbiguousExc {
2     void m() throws Exception {
3         peer Object tc = new peer Integer(7);
4         throw new Exception((readonly String) (tc + " is peer Object / this
              is: " + this));
5     }
6 }
```

Listing 5.11: The example with the "ambiguous" error mitigated by a cast (line 4).

Problem localization We have tried to put the above example (listing 5.10) to the MultiJava project as unit test. But unfortunately the unit test does not fail.
By further investigations, it has turned out that the MultiJava compiler accepts code like in listing 5.10, while the JML-tools checker fails (main class: `org.jmlspecs.checker.Main`).
We assume the ambiguity of parsing these code snippets is based on the two different direct supertypes of the actual parameter. For visualization see figure 5.12.

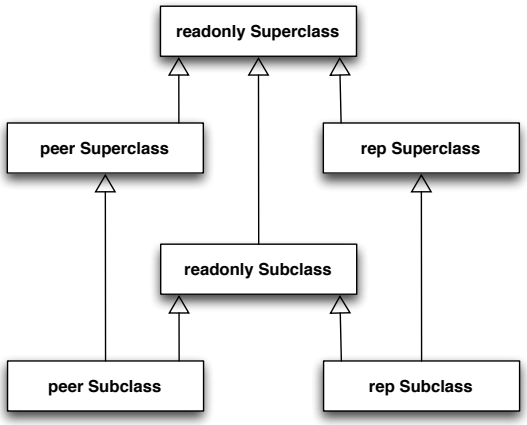

Figure 5.12: Read-write universe types have two different direct supertypes: (1) the according read-only type and (2) the corresponding read-write supertype in the class hierarchy of java.

As direct supertypes we consider instantiable types only; especially interfaces have been ignored.

> *No further investigations have been made. So this section is mainly a bug report, with an approach for the problem localization.*

Chapter 6

Ideas and Proposals

In this chapter we present a few ideas, how the Universe type system could be extended.
The first approach (section 6.1) presents, how some references could be typed implicitly as read-only. This is just to make the life of the programmer easier.
In section 6.2 we propose to introduce method-local universes, to gain more flexibility for pure methods.
Two ideas of a general read-write parameter type are presented in section 6.3.

6.1 Implizit `readonly`

In programs we often get `readonly` references. There are two ways to handle them: (1) we know in what universe the referenced object is, so we can make a downcast to the according universe or (2) a `readonly` reference is enough for the purpose we need it.
This proposal is about the second case: If a class has only `pure` and `readonly` features that are frequently used, references to objects of this class can be treated as `readonly` references by default.
Concerned classes are `String`, `Integer`, other numbers and all kinds of exceptions. Strings and numbers are immutable classes, while exceptions are propagated as `readonly` (according to [DM04]).

Approach The compiler treats all declarations of fields, parameters and local variables of immutable types and exceptions, that have no universe modifier declared, implicitly as `readonly`. Implicitly means that if there is another annotation, this one will be considered. But only iff there is no annotation for a reference, the compiler treats it as `readonly`.
Immutable types can be (1) a list of types, the compiler knows about or (2) all classes with the JML-annotation `immutable`. (1) has the disadvantage that only library types can be considered; references to user-defined classes cannot be declared to be implicitly treated as `readonly`. (2) is depending on JML [JML].

6.2 Local universes

6.2.1 Problem

In pure methods it is forbidden to invoke any non-pure methods, even on newly created objects. This is because the non-pure method could downcast a read-only reference and modify the object store.

6.2.2 Proposed solution

We introduce new method-local universes. Read-write access to these local universes is allowed during one method execution only.
A new type annotation `local` is used for that. `local` is a subtype of `readonly`, but not of anything else.
The annotation can only be used for local variables of a reference type.
Local references are read-write references; full access is allowed. We can even call non-pure methods on local references.
For arrays we allow:

```
local peer T[];
local readonly T[];
local local T[]
```

We can create new objects and arrays with a `local` annotation.
The only way a new local object could escape the method scope is if the object is returned through a read-only reference.
This new type will be most useful for pure methods, but it can also be used in non-pure methods to express that the objects created in the method should not escape it. It can be used in static methods and constructors.

6.2.3 Example

In listing 6.1 we present a class `C` that uses a local iterator (listing 6.2). Using the local iterator, it aggregates some information from a read-only Collection `c`.

6.2.4 Type combinator

	peer	rep	readonly	local	global
peer	peer	readonly (rep if called on this)	readonly	not possible	global
rep	rep	readonly	readonly	not possible	global
readonly	readonly	readonly	readonly	not possible	readonly
local	local	readonly (accord. to rep * rep)	readonly	not possible	global
global	global	readonly	readonly	not possible	global

Table 6.1: Type combinator, including the local universe type

```
1  class C {
2    readonly Collection c;
3
4    pure int someStatistic() {
5      // the constructor actual parameter must
6      // be local * readonly = readonly
7      local Iterator it = new local ROIterator( c );
8
9      int count;
10     count = 0;
11
12     while( it.hasNext() ) {
13       // the result type is
14       // local * readonly = readonly
15       readonly Object o = it.next();
16       count += o.aPure();              // aPure() is a pure method.
17     }
18
19     return count;
20   }
21 }
```

Listing 6.1: An example class `C` that uses a `local` iterator in a `pure` method to make an aggregation over a `readonly` collection.

```
1  class ROIterator implements UTSIterator {
2    ROIterator( readonly Collection c ) { .. }
3    pure boolean hasNext() { .. }
4    readonly Object next() { .. }
5  }
```

Listing 6.2: An iterator on a `readonly` collection as described in section 5.4.

In Table 6.1 we present the type combinator for the introduced local universe type. Only combinations with local on the left hand side are possible. Local can not be on the right hand side, because we only allow it for local variables.
So the new combinations are:

- `local` * `peer`, which should be local again.
- `local` * `rep`: according to `rep` * `rep` this returns `readonly`.
- `local` * `readonly`, which must be readonly again.
- `local` * `global`: according to `rep` * `global`, can be `global`.

Example of how a local reference can escape its method: In the listing 6.2, take as implementation of `ROIterator.next()`:

```
readonly Object next() { return this; }
```

Then in the method `someStatistics()` of class `C` (listing 6.1) we could have

```
// local * readonly => readonly
return it.next();
```

which returns a readonly Object, which is actually a reference to the newly created local object.

6.2.5 Runtime checks

Local objects are created in a new method-universe. Usually this Universe must only exist during the execution of a method, but local objects could escape through readonly references. For each method invocation we propose to use a different Universe.
We can call non-pure methods on local references. These non-pure methods could include downcasts of a readonly reference to peer or rep. But because the local objects are created in a separate Universe we can be sure that those downcasts would fail at runtime.
Statically we can not guarantee that a pure method with local variables has no side-effects. But at runtime all downcasts of non-local references will fail.
Unlike previous owners, the owner of local objects is not an object anymore. Since for each invocation of a method, another local universe is used, we recommend to use another owner for each invocation respectively.

6.2.6 Future work

Approach to enhance static checks We can restrict the method that are allowed to be called on local objects to *semipure* methods.
A method is semipure if it is `pure` or (1) it calls semipure methods only and (2) it contains no casts of parameters to read-write types.
Since these two properties can be checked statically, semipure methods can automatically determined by the compiler.

6.3 A general read-write paramater type

6.3.1 Motivation

Problem A As described in subsection 3.2.3, in some methods it is desired to have parameters of type read-write, but it is not important whether it is `peer`, `rep` or `global`. E.g. a method `void sayHello(PrintStream out)` takes an argument to write a "hello" message. If we want that this method can be called with either `global System.out` or our own `peer PrintStream p` as parameter, it is not possible to annotate the parameter with a universe type.

6.3.2 Approach with an abstract universe type

In subsection 3.2.3 we presented a first approach to face these problems. We introduced a new abstract universe type `writable` as direct subtype of `readonly` and supertype of all read-write universe types (`rep`, `peer` and `global`). `writable`, as the name suggests, is a read-write universe type as well. Like `readonly` types, `writable` types cannot be instantiated directly, therefore we call it abstract. The type hierarchy is drawn in figure 3.4.
In this section we adapt this approach a bit. But the main idea stays the same: Introduce a new read-write universe type, which is compatible with all existing read-write universe types.
The syntax of this approach is similar to Java generics [Suna]. If we want to allow a method to declare writable parameters, universe parameters have to be marked in front of a method declaration with the following notation: `<writable X>`, where `X` is the variable for the universe. The keyword `writable` reminds of the requirement of the parameter: the parameterized universe has to be read-write in both: the client and the method itself. At compile time the universe relations are assigned and type checked.

```
1  class C {
2      <writable U> void writeTo(U PrintStream p) {
3          p.writeln("hello universe!");
4      }
5  }
6
7  // client with peer reference c to instance of C:
8  peer C c = new peer C();
9  c.writeTo(System.out); // U is resolved to be global.
10 c.writeTo(new peer PrintStream(
11         new peer FileOutputStream("peerfile")
12 ));                    // U is resolved to be peer.
13 c.writeTo(new rep PrintStream(
14         new rep FileOutputStream("repfile")
15 ));                    // U is resolved to be readonly
16                        // => incompatible => compile time error.
```

Listing 6.3: A method which takes any read-write `PrintStream` as parameter, writes a message on it and returns.

Example to problem A In listing 6.3 we present a method that writes something on a given `PrintStream`, like described in Problem A. A client of this method tries to call it with parameters in different universes: `global`, `peer` and `rep`. The last call will fail, since **c** has no read-write permission in the rep universe of the client.
Let us have a closer look to the method `writeTo(..)`: Instead of defining the parameter `p` as `peer`, `rep` or `global` explicitly, the compiler can assign the universe, by resolving the parameters. If we wanted to provide this functionality with the upcoming Universe type system, we had to implement three versions of the method `writeTo(..)`.
As written in lines 9 and 12 of listing 6.3, the compiler is able to resolve the universe type of the parameter of the signature in line 2 correctly. In line 15 is stated, that the third call of `c.writeTo(..)` is typed wrong: the actual parameter is, due to the conventional type combinator, `peer` * `rep` $\Rightarrow$ `readonly`, which is not assignable to the formal parameter `writable`.

Type rule

The proposed type rule is:

Rule 6.1 *A formal parameter can be declared as `writable` iff the referenced object can be in any universe, where read-write access of the callee is allowed.*

The second part of this rule is checked at compile time by assigning a concrete universe to each `writable` keyword. To perform these checks the following type combinator can be used.

Type combinator

In Table 6.2 we present the type combinator belonging to the proposed introduction of `writable`.

We omit `writable` on the right hand side because it cannot be reasonably typed for all cases. Consider the case `rep` * `writable`: If we assume `writable` as the set of {`peer`, rep, global} then the resulting set should be {`rep` * `peer`, `rep` * `rep`, `rep` * `global`} = {`rep`, `readonly`, `global`}. And this resulting set is not equivalent to the assumed set for `writable`.
For the case `writable` * `writable`, the situation is getting even worse.

	peer	rep	readonly	global
peer	peer	readonly (rep if called on this)	readonly	global
rep	rep	readonly	readonly	global
readonly	readonly	readonly	readonly	readonly
writable	writable	readonly	readonly	global
global	global	readonly	readonly	global

Table 6.2: Type combinator, including the writable universe type

Return type If the return type of a method is `writable`, at compile time the actual universe is resolved and the call is treated using this universe information.

Conclusion

This approach introduces a writable universe type for methods only. It is useful as long as there is only one call to a method with writable parameters. But since it is not allowed to pass writable variables as actual parameters to other methods (right hand side of the type combinator), the area of application of this approach is very thin.

6.3.3 Approach with a template mechanism

Without introducing new concepts or keywords, we could simply introduce several methods, one for each read-write universe type respectively.
For the example, described in problem A (6.3.1), the solution would look like this:

```
void writeTo(peer PrintStream p) { .. }
void writeTo(rep PrintStream p) { .. }
void writeTo(global PrintStream p) { .. }
```

So for every read-write universe type, we have its own method declaration. Due to method overloading, the right method is chosen for execution.
But we do not want to declare each method once for each read-write universe type. To avoid duplicate code, we would like to declare the `writeTo(..)` method in a way like this:

```
void writeTo({peer, rep, global} PrintStream p) {
    p.writeln("hello universe!");
}
```

`{peer, rep, global}` means that there are three overloaded methods, each with one of the three universe types respectively.

The following problems have to be treated: (1) In the method body of the example above, we do not know what universe type p has. (2) `{peer, rep, global}` is ugly syntax without similarities to existing Java concepts. (3) The example above is not backward compatible with conventional Java, since it introduces several methods with the same signature, if universe types have been omitted.[a]

[a] E.g. a class has been compiled with MJ/JML and then the byte code is executed with a standard VM.

`writable` as a place holder

We introduce `writable` as a place holder for one universe type out of {`peer, rep, global`}. In each method, in which `writable` is used, it has to appear in the signature. The compiler replaces it for each read-write universe type and generates a separate method. If `writable` appears in the method body, it is replaced with the same read-write universe type as in the signature respectively.

```
1 /** @param m Map and all Entries in same universe (by convention). */
2 void collectInformation(writable Map m) {
3     // ensure there is a InfoObject for my category in m.
4     if (m.get(this.category()) == null) {
5         writable InfoObject io = new writable InfoObject();
6         m.put(this.category(), io);
7     }
8     m.get(this.category()).increment(this); // param of increment readonly
9 }
```

Listing 6.4: A map collects information about `this`, grouped by the category of `this`.

As an example in listing 6.4 we have a method that allows to collect information in a given `Map`. The information is stored in instances of a class `InfoObject` and grouped by a category which implements the `Comparable` interface. The compiler generates the three methods, like presented in listing 6.5.

```
1  void collectInformation(peer Map m) {
2      // ensure there is a InfoObject for my category in m.
3      if (m.get(this.category()) == null) {
4          peer InfoObject io = new peer InfoObject();
5          m.put(this.category(), io);
6      }
7      m.get(this.category()).increment(this); // param of increment ro
8  }
9  void collectInformation(rep Map m) {
10     // ensure there is a InfoObject for my category in m.
11     if (m.get(this.category()) == null) {
12         rep InfoObject io = new rep InfoObject();
13         m.put(this.category(), io);
14     }
15     m.get(this.category()).increment(this); // param of increment ro
16 }
17 void collectInformation(global Map m) {
18     // ensure there is a InfoObject for my category in m.
19     if (m.get(this.category()) == null) {
20         global InfoObject io = new global InfoObject();
21         m.put(this.category(), io);
22     }
23     m.get(this.category()).increment(this); // param of increment ro
24 }
```

Listing 6.5: The method of listing 6.4 is translated by the compiler to these three methods: one for each read-write universe type; in the body `writable` is replaced accordingly.

With the presented mechanism we have solved the two problems (1) and (2): Introducing the keyword `writable`, we have a nicer syntax and because of the replacement of `writable` by the compiler, we can even use the place holder `writable` in the whole method.
We are still not backward compatible (problem (3)) and we have introduced a new restriction: All `writable` of one method are interpreted as the same universe (problem (4)).

Parameterization of `writable`

To face problem (4), that only one `writable` type can be used per method, we can parameterize `writable`. The example in listing 6.4 now looks like presented in listing 6.6.

```
1 /** @param m Map and all Entries in same universe (by convention). */
2 <writable U> void collectInformation(U Map m) {
3     // ensure there is a InfoObject for my category in m.
4     if (m.get(this.category()) == null) {
5         U InfoObject io = new U InfoObject();
6         m.put(this.category(), io);
7     }
8     m.get(this.category()).increment(this); // param of increment readonly
9 }
```

Listing 6.6: Th example presented in listing 6.4 with the parameterized syntax.

Now we can have several writable parameter types, which can even be used in the method body as well. Following, a syntax example for two writable parameter types is presented:

```
<writable U, writable V> U Object method(U Object param1, V Object param2)
    { .. }
```

The compiler has to generated the according methods in several steps, for each `writable` declaration once. In the first step the following method declarations are generated:

```
<writable V> peer Object method(peer Object param1, V Object param2)
    { .. /* [ U / peer ] */ }
<writable V> rep Object method(rep Object param1, V Object param2)
    { .. /* [ U / rep ] */ }
<writable V> global Object method(global Object param1, V Object param2)
    { .. /* [ U / global ] */ }
```

The notation

/ [U / peer] */*

means that all occurrences of `U` in the method body are replaced by `peer`.
For the `<writable V>` the same replacement is done; so we end up with nine implicit method declarations (assumed there are three read-write universe types).

Backward compatibility

As already mentioned as problem (3), this approach is not backward compatible with conventional Java, so far. If a class that uses writable parameters is compiled with the MJ/JML compiler, and then the byte code is used with a java version that does not support universe byte code, there are ambiguous method declarations of the concerned methods.
To face this problem, we simulate overloading by introducing a dispatcher method, which calls a specific generated method according to the universe type. The following method renaming and generation steps are done by the compiler:

- All methods generated by the replacement (as described above), are renamed to the method name, concatenated with the universe type. (e.g. `methodPeer(..)`).
- In the originating method all `writable` parameters are changed to `readonly`. Its body is changed to a dispatcher method: An `instanceof` checks the actual type of the parameter; then the proper method (generated in the first step) is called with the parameter casted to the actual type.

As an example, the generated code of the problem A is presented in listing 6.7.

```
1  /* source code written by programmer */
2  <writable U> void writeTo(U PrintStream p) {
3      p.writeln("hello universe!");
4  }
5
6  /* code generated by the compiler */
7  void writeToPeer(peer PrintStream)    { .. /* [ U / peer] */ }
8  void writeToRep(rep PrintStream)      { .. /* [ U / rep] */ }
9  void writeToPeer(global PrintStream)  { .. /* [ U / global] */ }
10
11 /* the dispatch method (generated by the compiler) */
12 void writeTo(readonly PrintStream p) {
13     if (p instanceof peer PrintStream) {
14         writeToPeer((peer PrintStream) p); return; }
15     if (p instanceof rep PrintStream) {
16         writeToRep((rep PrintStream) p); return; }
17     if (p instanceof global PrintStream) {
18         writeToGlobal((global PrintStream) p); return; }
19     throw new ClassCastException("p is not writable");
20 }
```

Listing 6.7: The source code (lines 2 - 4) written by the programmer, is translated by the compiler into a dispatcher method (lines 12ff) and universe specific methods (lines 7 - 9).

The `ClassCastException` in line 19 is thrown only if the given parameter is not of a read-write type for the callee. This runtime exception occurs only with this last extension of backward compatibility. In the prior version, we checked read-write access of the callee with the standard universe type combinator and overloading of the method.
To prevent the `ClassCastException` in the dispatcher method, the compiler could remember for parameters of the dispatcher method, that they have been `writable` before and check for each call of that method, whether the callee has read-write access to the actual parameters. This check can be performed with the standard type combinator presented in section 2.4.

Conclusion

The approach with a template mechanism provides a suitable solution to the problem, like described in subsection 6.3.1. This approach reduces everything that is introduced to previously known concepts; and this reduction is performed by the compiler.

Chapter 7

Conclusion

We applied the Universe type system [MPH01] to an industrial application. Our main result is that it is possible to apply the Universe type system to an application of reasonable size.

We achieved this goal by annotating Java API classes (chapter 3) as well as the part of the application that we have researched (chapter 4). The Universe type system controls aliasing, meaning access to certain references is restricted. Due to these restrictions, we had to do some restructuring of the application and parts of the Java API.

7.1 Annotation of an application

An advantage of the restructuring process is that the structure of the application is improved: The number of objects that may modify another object is limited to the objects in the same universe (and its owner). This yields that a facade pattern can be enforced or whole object structures can be encapsulated. We get a cleaner distribution of responsibilities between several components.

Nevertheless, a disadvantage is, that more methods are needed: For each non-pure method that is allowed to be called from outside, there has to be a facade method in the owner object of this universe. These additional methods may even be recursive (see subsection 4.2.5). This implies runtime overhead for the respective method calls.

Ownership diagram To annotate one or several components with universe type, we recommend to draw an ownership diagram. This helps to keep track of the annotation process, independent of the annotation strategy that is used. (Some approaches to such annotation strategies are presented in section 4.1.)

Singleton Pattern In section 5.2 we describe two versions of how to treat singletons in existing applications: (1) with a global singleton object and (2) avoiding them. A combination of the two approaches yields a flexible solution and enables alias control.

Copying So far it is not possible in the Universe type system to transfer objects from one universe to another. In some situations it is suitable to copy these objects. The disadvantage is that object identity is lost. In section 5.5 we present an implementation of a copy mechanism.

Iterators Since the analyzed application makes versatile use of the java.util.* collection framework, we encountered two needs of iterators, that are not fulfilled by the existing framework. (1) Iteration over read-only collections and (2) the usage of iterators in pure methods. For both of these problems we present a workaround in subsection 5.4.1 and subsection 5.4.2. A generic iterator, like presented in subsection 5.4.1, combined with the proposal of a target universe type yields a reasonable solution for even the combination of (1) and (2).

Global universe Based on the experience of this case study, we can state that a global universe is needed to annotate a real world application. We used the concept of a global universe, like presented in [Häc04].

7.2 Annotation of Java API

We used the MultiJava/JML compiler [MJ, JML]. Using the JML `*.refines-spec` file format, we had to annotate the signatures for the desired Java API only.
Since the Universe type system introduces read-only references and restricts access to object in foreign universes, some Java API needs to be restructured.
In subsection 5.4.1 we introduce an `UTSIterator`, which is able to iterate over read-only collections. This is an example implementation of a Java API concept that has to be adapted for use together with the Universe type system.

7.3 Ideas and Proposals

In chapter 6 we present a few approaches to make life easier.
In section 6.2 we introduce a local universe type with a scope of only the actual method execution. In pure methods, this enables full access to objects, that are instantiated in the local universe. With this approach, among other things, we enable the usage of iterators in pure methods.
We present several approaches in section 6.3, to enable method parameters with read-write access for the callee. The approach with a template mechanism (subsection 6.3.3) seems to be the most suitable, since it uses conventional universe types and rules, and most of the work is done by the compiler. It is even possible to use method overloading, to avoid runtime overhead.
Last but not least, in subsection 6.3.3 we present an approach to instantiate objects inside any given universe.

Bibliography

[AB04] Karine Arnout and Eric Bezault. How to get a singleton in eiffel? *Journal of Object Technology*, 3(4):75–95, April 2004. Available from http://www.jot.fm/issues/issue_2004_04/article5. 33

[Apa] Apache Software Foundation. log4j. http://logging.apache.org/log4j/docs/. 15

[Bec] Kent Beck. Extreme programming XP. http://www.extremeprogramming.org/. 22

[Cin] Cinerent Open Air AG. starticket. https://www.starticket.ch/. 1

[DM04] W. Dietl and P. Müller. Exceptions in ownership type systems. In E. Poll, editor, *Formal Techniques for Java-like Programs*, pages 49–54, 2004. 15, 43

[Ecl] The Eclipse Foundation. The eclipse project. http://www.eclipse.org/, I used eclipse Version 3.0.1. 37, 76

[EG02] L. Eppler and T. Gresch. Projektbeschrieb: Entwicklung von barcode basierten eingangskontrollen. Firma Cinerent Open Air AG, CH-8702 Zollikon, Starticket https://www.starticket.ch, 2002. 1

[Fow03] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, September 2003. 22, 37, 76

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995. 2, 12, 32, 33

[Häc04] Thomas Hächler. Statische Felder im Universe Type System. 2004. 12, 54

[JML] The JML-specs project. http://www.jmlspecs.org/. 7, 15, 43, 54, 76

[Mas04] Vincent Massol. *JUnit in Action*. Manning, 2004. 76

[MJ] The MultiJava project. http://multijava.sourceforge.net/. 54, 76

[MPH01] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001. 9, 11, 15, 53

[Mül04] Peter Müller. Konzepte objektorientierter Programmierung, 2004. http://sct.ethz.ch/teaching/ws2004/koop/. 9

[SAX] SAX, Public Domain, SourceForge. Simple api for xml. http://www.saxproject.org/. 15

[Suna] Sun Microsystems, Inc. Generics in the java programming language (J2SE 1.5). http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html. 32, 46

[Sunb] Sun Microsystems, Inc. Java. http://java.sun.com/. 76

[Sunc] Sun Microsystems, Inc. Jxta. http://www.jxta.org/. 6

Appendix A

Some Details

A.1 Listing of the class GenericIterator

Listing A.1: Implementation of a generic iterator, working on a `readonly` collection.

```
1  /*
2   * $Id: GenericIterator.java,v 1.1 2005/01/18 11:40:04 xtom Exp $
3   *
4   * Copyright (C) 2004 Swiss Federal Institut of Technology Zurich
5   *
6   * This file is part of the MultiJava project.  More information is
7   * available from www.multijava.org.
8   *
9   * This program is free software; you can redistribute it and/or modify
10  * it under the terms of the GNU General Public License as published by
11  * the Free Software Foundation; either version 2 of the License, or
12  * (at your option) any later version.
13  *
14  * This program is distributed in the hope that it will be useful,
15  * but WITHOUT ANY WARRANTY; without even the implied warranty of
16  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
17  * GNU General Public License for more details.
18  *
19  * You should have received a copy of the GNU General Public License
20  * along with this program; if not, write to the Free Software
21  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
22  *
23  * Created on 20.12.2004
24  */
25 package org.multijava.uts;
26
27 import java.util.Collection;
28 import java.util.ConcurrentModificationException;
29
30 /**
31  * This implementation of {@link org.multijava.uts.UTSIterator}
32  * stores all elements that will be returned in an array.
33  * It check whether an element returned by {@link #next()} and the element
      before are
34  * still at the same position as in the underlying {@link java.util.
      Collection}.
35  * If these conditions are not fulfilled, {@link java.util.
      ConcurrentModificationException} will be thrown.
36  *
```

```
37   * @author Thomas Haechler
38   */
39  public class GenericIterator extends SimpleIterator implements UTSIterator {
40
41          /* CONSTANTS */
42
43          /** shall concurrent modification checks be done? */
44          static final boolean DO_CONCURRENT_MODIFICATION_CHECKS = true;
45
46          /* FIELDS */
47
48          /** hold the underlying Collection */
49          private /*@ \readonly @*/ Collection collection;
50
51          /** has a {@link java.util.ConcurrentModificationException} already
                been thrown? */
52          private boolean concModExpHasBeenThrown = false;
53
54          /* CONSTRUCTOR */
55
56          /**
57           * @param coll
58           */
59          public GenericIterator(/*@ \readonly @*/ Collection coll) {
60                  super(coll);
61                  collection  = coll;
62          }
63
64          /**
65           * @see java.util.Iterator#next()
66           */
67          /*@  private normal_behavior
68            @    assignable concModExpHasBeenThrown;
69            @*/
70          public synchronized /*@ \readonly @*/ Object next()
71          throws IndexOutOfBoundsException, ConcurrentModificationException {
72                  if (concModExpHasBeenThrown) concurrentModification();
73                  /*@ \readonly @*/ Object retValue = super.next();
74                  checkConcurrentModification(retValue);
75                  return retValue;
76          }
77
78          /**
79           * @param nextReturnValue
80           * @throws ConcurrentModificationException
81           */
82          /*@  private normal_behavior
83            @    assignable concModExpHasBeenThrown;
84        @*/
85          private void checkConcurrentModification(/*@ \readonly @*/ Object
                nextReturnValue)
86          throws ConcurrentModificationException {
87                  if (!DO_CONCURRENT_MODIFICATION_CHECKS) return;
88                  int pos = position - 1;
89                  /*@ \readonly \readonly @*/ Object[] actualCollection =
                        collection.toArray();
90                  if (actualCollection[pos] != nextReturnValue)
91                          concurrentModification();
92                  if ((pos > 0) && (actualCollection[pos-1] != array[pos-1]))
93                          concurrentModification();
94          }
95
96          /**
```

```
97              * @see java.util.Iterator#hasNext()
98              */
99             public synchronized /*@ pure @*/ boolean hasNext() {
100                    return !concModExpHasBeenThrown && super.hasNext();
101            }
102
103            /**
104             * @throws ConcurrentModificationException
105             */
106            /*@  private normal_behavior
107              @    assignable concModExpHasBeenThrown;
108              @*/
109            private void concurrentModification() throws
                   ConcurrentModificationException {
110                    concModExpHasBeenThrown = true;
111                    throw new ConcurrentModificationException("underlying␣
                           Collection␣has␣changed␣since␣instantiation␣of␣this␣("+
                           this+").");
112            }
113    }
114
115    /*
116     * CVS Log Entries for this file:
117     *
118     * $Log: GenericIterator.java,v $
119     * Revision 1.1  2005/01/18 11:40:04  xtom
120     * added source of GenericIterator to appendix.
121     *
122     * Revision 1.2  2004/12/22 12:31:50  xtom
123     * added javadoc.
124     *
125     */
```

Appendix B

Interim Reports

B.1 The selected subset of the application

...and its dependencies.
In the package com.cinerent.xml I chose the classes belonging to XML download only, but not XML writing or upload.

A summarization of the following table (B.1) can be found in table B.2.

Table B.1: Classes of the selected subset of the application and their dependencies.

class	depends on	remarks
com.cinerent.beans.	java.util.Set	
YoshiDataStructure	org.apache.log4j.Logger	
	com.cinerent.jxta.BagMessageSender	
	com.cinerent.jxta.YoshiGroup	

continued on next page

class	depends on	remarks
com.cinerent.beans.	java.lang.Object	java.lang.* is listed only once.
TicketingComponent	java.lang.Serializable	java.lang.* is listed only once.
	java.lang.Comparable	java.lang.* is listed only once.
	java.lang.NullPointerException	java.lang.* is listed only once.
	java.lang.String	java.lang.* is listed only once.
	java.lang.StringBuffer	java.lang.* is listed only once.
	java.io.Serializable	
	java.util.Collection	
	java.util.Iterator	
	java.util.LinkedList	
	java.util.Map	
	java.util.Set	
	java.util.TreeMap	
	net.jxta.id.ID	JXTA: stub only.
	org.apache.log4j.Logger	
	com.cinerent.jxta.IBagable	
	com.cinerent.util.SystemProperties	
	com.cinerent.xml.XmlException	
com.cinerent.beans.	java.lang.Integer	java.lang.* is listed only once.
TicketingComposite	java.lang.ClassNotFoundException	java.lang.* is listed only once.
	java.io.IOException	
	java.io.ObjectInputStream	
	java.io.ObjectOutputStream	
	java.io.PrintStream	
	java.util.Collection	
	java.util.Collections	
	java.util.Iterator	
	java.util.LinkedList	
	java.util.Map	
	java.util.SortedMap	
	java.util.TreeMap	
	com.cinerent.xml.XmlException	
com.cinerent.beans.	java.util.LinkedList	
TicketingLeaf	java.util.Map	

continued on next page

class	depends on	remarks
com.cinerent.beans. TicketingData	java.io.PrintStream java.util.Collections java.util.Iterator java.util.SortedMap java.util.TreeMap org.apache.log4j.Logger com.cinerent.controller.AutoSaver com.cinerent.xml.XMLDownloadData com.cinerent.xml.XMLDownloadable	
com.cinerent.beans. Event	java.io.IOException java.io.ObjectInputStream java.io.Serializable java.util.Calendar java.util.Date java.util.Iterator java.util.LinkedList org.apache.log4j.Logger com.cinerent.util.InstanceIsInvalidException com.cinerent.util.Saveable com.cinerent.util.StatisticData com.cinerent.util.YoshiProperties com.cinerent.xml.XMLDownloadData com.cinerent.xml.XMLDownloadable	
com.cinerent.beans. Show	java.io.IOException java.io.PrintStream java.text.FieldPosition java.text.SimpleDateFormat java.util.Calendar java.util.Date java.util.Iterator javax.xml.parsers.DocumentBuilderFactory javax.xml.parsers.ParserConfigurationException org.apache.log4j.Logger org.w3c.dom.Document org.w3c.dom.Element com.cinerent.command.CommandHandler com.cinerent.controller.MainController com.cinerent.jxta.BagMessageSender com.cinerent.util.YoshiProperties com.cinerent.xml.XMLDownloadData com.cinerent.xml.XMLDownloadable	

continued on next page

class	depends on	remarks
com.cinerent.beans.Ticket	java.util.Date java.util.Iterator java.util.Set org.apache.log4j.Logger com.cinerent.controller.MainController com.cinerent.jxta.BagMessageSender	
com.cinerent.beans.TCode	java.util.Calendar java.util.Collection java.util.Collections java.util.Date java.util.Iterator java.util.Set java.util.StringTokenizer java.util.Vector org.apache.log4j.Logger org.w3c.dom.Document org.w3c.dom.Element com.cinerent.controller.MainController com.cinerent.jxta.BagMessageSender com.cinerent.util.TimeComparator com.cinerent.xml.XmlException	
com.cinerent.beans.Log	java.text.SimpleDateFormat java.util.Collection java.util.Date java.util.Iterator java.util.LinkedList java.util.TreeMap org.apache.log4j.Logger com.cinerent.util.YoshiProperties	
com.cinerent.beans.PrintTag	java.util.Collection com.cinerent.controller.MainController	
com.cinerent.beans.Concession	java.util.Collection	
com.cinerent.beans.TicketingUniqueKey	java.io.Serializable java.lang.Comparable	
com.cinerent.beans.Barcode	java.io.Serializable java.util.Set com.cinerent.util.TreeMapOfSet	
com.cinerent.beans.BarcodeContainer	java.io.Serializable	

continued on next page

class	depends on	remarks
com.cinerent.beans. EventController	java.io.IOException java.io.ObjectInputStream java.io.Serializable java.util.Iterator java.util.LinkedList java.util.Set java.util.TreeSet org.apache.log4j.Logger com.cinerent.command.CommandHandler com.cinerent.command.ShowStateCommand com.cinerent.controller.AutoSaver com.cinerent.controller.MainController com.cinerent.controller.UserTalkback com.cinerent.display.PopUp com.cinerent.jxta.Communicator com.cinerent.util.InstanceIsInvalidException com.cinerent.util.MessageIsNullException com.cinerent.util.Saveable com.cinerent.util.YoshiProperties	
com.cinerent.beans. LogObserver		(no dependencies)
com.cinerent.beans. UniqueIdObserver		(no dependencies)
com.cinerent.beans. TicketingException	java.lang.Exception com.cinerent.YoshiException	java.lang.* is listed only once.
com.cinerent.beans. BarcodeNotValidException		(no dependencies)
com.cinerent.beans. AdminEventException		(no dependencies)

continued on next page

class	depends on	remarks
com.cinerent.xml. XMLMaster	java.io.BufferedReader java.io.IOException java.io.InputStreamReader java.net.MalformedURLException java.net.URL java.util.NoSuchElementException java.util.StringTokenizer javax.xml.parsers.ParserConfigurationException org.apache.log4j.Logger org.xml.sax.InputSource org.xml.sax.SAXException com.cinerent.YoshiException com.cinerent.beans.Event com.cinerent.beans.Show com.cinerent.beans.TicketingUniqueKey com.cinerent.beans.YoshiDataStructure com.cinerent.command.XMLStateCommand com.cinerent.controller.MainController com.cinerent.controller.UserTalkback com.cinerent.devices.FileGetter com.cinerent.jxta.Communicator com.cinerent.jxta.YoshiGroup com.cinerent.util.Shutdownable com.cinerent.util.SingletonException com.cinerent.util.TooEarlyException com.cinerent.util.YoshiProperties	
com.cinerent.xml. XMLMasterFile	java.io.BufferedInputStream java.io.FileInputStream java.io.FileNotFoundException java.io.InputStreamReader java.io.UnsupportedEncodingException org.apache.log4j.Logger com.cinerent.YoshiException com.cinerent.controller.UserTalkback com.cinerent.util.YoshiProperties	

continued on next page

class	depends on	remarks
com.cinerent.xml. XMLMasterURL	java.io.BufferedInputStream java.io.BufferedOutputStream java.io.File java.io.FileInputStream java.io.FileOutputStream java.io.IOException java.io.InputStreamReader java.net.HttpURLConnection java.net.MalformedURLException java.net.URL org.apache.log4j.Logger com.cinerent.YoshiException com.cinerent.controller.UserTalkback com.cinerent.util.TimeoutObserver com.cinerent.util.TimeoutSubject com.cinerent.util.TooEarlyException com.cinerent.util.YoshiProperties	
com.cinerent.xml. XMLDownloadable	java.io.Serializable	
com.cinerent.xml. XMLDownloadData	java.io.Serializable com.cinerent.util.YoshiProperties	
com.cinerent.xml. TicketingV2Handler	org.xml.sax.Attributes org.xml.sax.ContentHandler org.xml.sax.SAXException	

continued on next page

class	depends on	remarks
com.cinerent.xml.	java.text.ParseException	
TicketingV2HandlerImpl	java.text.SimpleDateFormat	
	java.util.Date	
	java.util.Iterator	
	java.util.Stack	
	java.util.TreeMap	
	org.apache.log4j.Logger	
	org.xml.sax.Attributes	
	org.xml.sax.SAXException	
	com.cinerent.beans.Barcode	
	com.cinerent.beans.BarcodeNotValidException	
	com.cinerent.beans.Concession	
	com.cinerent.beans.Event	
	com.cinerent.beans.Log	
	com.cinerent.beans.PrintTag	
	com.cinerent.beans.Show	
	com.cinerent.beans.TCode	
	com.cinerent.beans.Ticket	
	com.cinerent.beans.TicketingComponent	
	com.cinerent.beans.TicketingComposite	
	com.cinerent.beans.TicketingData	
	com.cinerent.beans.TicketingException	
	com.cinerent.beans.TicketingUniqueKey	
	com.cinerent.beans.YoshiDataStructure	
	com.cinerent.controller.MainController	
	com.cinerent.jxta.BagMessageSender	
	com.cinerent.jxta.YoshiGroup	
	com.cinerent.util.YoshiProperties	
com.cinerent.xml.	org.xml.sax.Attributes	TicketingV2Parser
TicketingV2Parser	org.xml.sax.ContentHandler	may probably be implemented
	org.xml.sax.EntityResolver	as stub.
	org.xml.sax.ErrorHandler	
	org.xml.sax.InputSource	
	org.xml.sax.Locator	
	org.xml.sax.SAXException	
	org.xml.sax.SAXParseException	
	org.xml.sax.XMLReader	

continued on next page

class	depends on	remarks
com.cinerent.xml.DTDResolver	java.io.BufferedReader java.io.IOException java.net.URL org.apache.log4j.Logger org.xml.sax.EntityResolver org.xml.sax.InputSource org.xml.sax.SAXException com.cinerent.devices.FileGetter	
com.cinerent.xml.XmlException	com.cinerent.YoshiException	
com.cinerent.controller.UserTalkback	com.cinerent.display.PopUpReceiver	
com.cinerent.util.YoshiProperties	java.io.File java.io.FileInputStream java.io.FileNotFoundException java.io.FileOutputStream java.io.IOException java.io.InputStream java.util.Enumeration java.util.Iterator java.util.Properties java.util.TreeMap org.apache.log4j.Logger	
com.cinerent.util.SystemProperties	java.io.FileInputStream java.io.FileNotFoundException java.io.IOException java.util.Properties org.apache.log4j.Logger	
com.cinerent.util.Saveable	java.io.Serializable	
com.cinerent.util.StatisticData		(no dependencies)
com.cinerent.util.TreeMapOfSet	java.util.Collection java.util.Collections java.util.Iterator java.util.Map java.util.Set java.util.SortedMap java.util.TreeMap java.util.TreeSet	
com.cinerent.util.Shutdownable	com.cinerent.YoshiException	

continued on next page

class	depends on	remarks
com.cinerent.util. TimeComparator	java.util.Comparator com.cinerent.beans.Log	
com.cinerent.util. TimeoutObserver	org.apache.log4j.Logger com.cinerent.YoshiException	
com.cinerent.util. TimeoutSubject		(no dependencies)
com.cinerent.util. SingletonException	com.cinerent.YoshiException	
com.cinerent.util. TooEarlyException	com.cinerent.YoshiException	
com.cinerent.util. InstanceIsInvalidException	com.cinerent.YoshiException	
com.cinerent.util. MessageIsNullException	com.cinerent.YoshiException	
com.cinerent. YoshiException	java.lang.Exception	

B.2 Summary of the subset of the application

Table B.2: Subset of the application; grouped by packages.

package	class to annotate	remarks
com.cinerent.beans	com.cinerent.beans.YoshiDataStructure	
	com.cinerent.beans.TicketingComponent	
	com.cinerent.beans.TicketingComposite	
	com.cinerent.beans.TicketingLeaf	
	com.cinerent.beans.TicketingData	
	com.cinerent.beans.Event	
	com.cinerent.beans.Show	
	com.cinerent.beans.Ticket	
	com.cinerent.beans.TCode	
	com.cinerent.beans.Log	
	com.cinerent.beans.PrintTag	
	com.cinerent.beans.Concession	
	com.cinerent.beans.TicketingUniqueKey	
	com.cinerent.beans.Barcode	
	com.cinerent.beans.BarcodeContainer	
	com.cinerent.beans.EventController	
	com.cinerent.beans.LogObserver	
	com.cinerent.beans.UniqueIdObserver	
	com.cinerent.beans.TicketingException	
	com.cinerent.beans.BarcodeNotValidException	
	com.cinerent.beans.AdminEventException	
com.cinerent.xml	com.cinerent.xml.XMLMaster	
	com.cinerent.xml.XMLMasterFile	
	com.cinerent.xml.XMLMasterURL	
	com.cinerent.xml.XMLDownloadable	
	com.cinerent.xml.XMLDownloadData	
	com.cinerent.xml.TicketingV2Handler	
	com.cinerent.xml.TicketingV2HandlerImpl	
	com.cinerent.xml.TicketingV2Parser	
	com.cinerent.xml.DTDResolver	
	com.cinerent.xml.XmlException	
com.cinerent.controller	com.cinerent.controller.UserTalkback	

continued on next page

package	class to annotate	remarks
com.cinerent.util	com.cinerent.util.YoshiProperties	
	com.cinerent.util.SystemProperties	
	com.cinerent.util.Saveable	
	com.cinerent.util.StatisticData	
	com.cinerent.util.TreeMapOfSet	
	com.cinerent.util.Shutdownable	
	com.cinerent.util.TimeComparator	
	com.cinerent.util.TimeoutObserver	
	com.cinerent.util.TimeoutSubject	
	com.cinerent.util.SingletonException	
	com.cinerent.util.TooEarlyException	
	com.cinerent.util.InstanceIsInvalidException	
	com.cinerent.util.MessageIsNullException	
com.cinerent	com.cinerent.YoshiException	

B.3 Necessary Stubs

Table B.3: Stubs to be implemented; grouped by packages.

package	stubs to implement	remarks
com.cinerent.controller	com.cinerent.controller.AutoSaver	
	com.cinerent.controller.MainController	
com.cinerent.command	com.cinerent.command.CommandHandler	
	com.cinerent.command.ShowStateCommand	
	com.cinerent.command.XMLStateCommand	
com.cinerent.jxta	com.cinerent.jxta.BagMessageSender	
	com.cinerent.jxta.IBagable	
	com.cinerent.jxta.Communicator	
	com.cinerent.jxta.YoshiGroup	
com.cinerent.display	com.cinerent.display.PopUp	
	com.cinerent.display.PopUpReceiver	
com.cinerent.devices	com.cinerent.devices.FileGetter	
java.net	java.net.URL	
	java.net.HttpURLConnection	
	java.net.MalformedURLException	
org.apache.log4j	org.apache.log4j.Logger	
net.jxta.id	net.jxta.id.ID	

continued on next page

package	stubs to implement	remarks
javax.xml.parsers	javax.xml.parsers.DocumentBuilderFactory	
	javax.xml.parsers.ParserConfigurationException	
org.xml.sax	org.xml.sax.InputSource	
	org.xml.sax.Attributes	
	org.xml.sax.XMLReader	
	org.xml.sax.ContentHandler	
	org.xml.sax.EntityResolver	
	org.xml.sax.ErrorHandler	
	org.xml.sax.InputSource	
	org.xml.sax.Locator	
	org.xml.sax.SAXException	
	org.xml.sax.SAXParseException	
org.w3c.dom	org.w3c.dom.Document	
	org.w3c.dom.Element	

B.4 Java API to annotate

Table B.4: Classes of the used Java API to be annotated; grouped by packages.

package	class to annotate	remarks
java.lang	java.lang.Object	
	java.lang.Comparable	
	java.lang.StringBuffer	
	java.lang.String	
	java.lang.Integer	
	java.lang.Exception	
	java.lang.ClassNotFoundException	
	java.lang.RuntimeException	
	java.lang.NullPointerException	
java.util	java.util.Comparator	
	java.util.Enumeration	
	java.util.Iterator	
	java.util.LinkedList	
	java.util.Set	
	java.util.Map	
	java.util.SortedMap	
	java.util.TreeMap	
	java.util.TreeSet	
	java.util.Collection	
	java.util.Collections	
	java.util.Stack	
	java.util.Calendar	
	java.util.Date	
	java.util.StringTokenizer	
	java.util.Vector	
	java.util.Properties	
	java.util.NoSuchElementException	

continued on next page

package	class to annotate	remarks
java.io	java.io.Serializable	
	java.io.IOException	
	java.io.ObjectInputStream	
	java.io.ObjectOutputStream	
	java.io.PrintStream	
	java.io.BufferedReader	
	java.io.InputStream	
	java.io.InputStreamReader	
	java.io.BufferedInputStream	
	java.io.BufferedOutputStream	
	java.io.FileNotFoundException	
	java.io.UnsupportedEncodingException	
	java.io.File	
	java.io.FileInputStream	
	java.io.FileOutputStream	
java.text	java.text.FieldPosition	
	java.text.SimpleDateFormat	
	java.text.ParseException	

B.5 MultiJava, JML-specs and eclipse

B.5.1 Motivation

I've been looking for an automated way to compile with the MultiJava compiler [MJ]. To provide source code compatibility with other java-compilers (like [Sunb]) I additionally used the JML-specs project [JML] which allowes to annotate the source code in `/*@...@*/`-style.
Eclipse [Ecl] is an integrated development environment providing features like source code browsing, refactoring [Fow03], JUnit integration [Mas04] and diverse plugins. The JUnit framework allows one to run a java program and do some assertions on success or failure.
Compiling a program with the MultiJava compiler is in fact invoking the according `compile()`-method as a java program. My approach is to run the MJ compiler as a JUnit Test and assert the return value of the `compile()`-method as `true`, meaning compiling with the MultiJava compiler was successful.

Problem of eclipse integration One cannot directly checkout MultiJava or JML-specs to an eclipse project, because compiling it is not possible without further complications. There are several steps to build these projects, organized with a Makefile. This is why I checked out the project into another directory, compiled there and then added the compiled project as an external library to the eclipse project.

B.5.2 Howto

Description how to setup MultiJava and JML-specs as JUnit-Test in Eclipse.

- Checkout the projects multijava and jmlspecs from the CVS location `pserver:anonymous@cvs.sourceforge.net:/cvsroot/...` to a separate cvs-location (i.e. ~/cvs/...).
- Build multijava using `make all`. (You need to set a few environment variables)
- Build jmlspecs using `make all`. (Don't forget to set `MJ_RELATIVE_ROOT`).
- Make a new project in eclipse (if you have not already done).
- Go to the Properties-Dialogue for the project (right-click on the project: properties). In the submenu 'Java Build Path / Libraries': You have to add two folders using 'Add Class Folder'. Here you have to create a new folder: give a name (i.e. 1. MJ / 2. JMLspecs) choose advanced mode and link to the according folder in the file system (i.e. 1. ~/cvs/MJ / 2. ~/cvs/JML2)
- In the same Properties-Dialogue you have to add all used *.jar-Archives for MultiJava and JML-specs. Add them as External JARs.

Now we are ready to write a JUnit-Test [Mas04]. In short, a JUnit-Test must have the following two properties: It extends and therefore is subclass of junit.framework.TestCase and has at least one method named `test...()` (i.e. `void testCompile()`).
As a possible implementation we can see in the method `testCompile()` in Listing B.1 that we have first to compose the arguments string containing the commandline option `-e` and then all classes we want to compile. At the end of the method we assert, the return value of the `org.jmlspecs.checker.Main.compile(...)`-method is `true`.
In contrast, we want the compiler to fail in the test method `testCompileWithoutSources()` of Listing B.1. Because we assert the compilation to fail, the execution of the test makes the compilert to print an error message(`error: No input files given`) while our test succeeds.

```
1  /*
2   * MjcTest.java
3   * Created by Thomas Haechler on 18.10.2004
4   */
5
6  import junit.framework.TestCase;
7  import org.jmlspecs.checker.Main;
8
9  /**
10  * This is a JUnit {@link junit.framework.TestCase} that with the method {
        @link #testCompile()}
11  * tries to compile the classes {@link LinkedList} and {@link Node} with the
        MultiJava compiler
12  * using directly {@link org.jmlspecs.checker.Main#compile(java.lang.String
        [])}.
13  */
14 public class MjcTest extends TestCase {
15     public void testCompile() {
16         String[] args = {"-e", // enable Universes in the compiler.
17                 "src/"+Node.class.getName()+".java",
18                 "src/"+LinkedList.class.getName()+".java" };
19         System.out.print("Arguments: ");
20         for (int i=0; i<args.length;i++) { System.out.print(args[i] + " ")
               ; }
21         System.out.println();
22         /* call the compiler */
23         boolean success = Main.compile(args);
24         assertTrue(success);
25     }
26     public void testCompileWithoutSources() {
27         String[] args = {"-e"}; // enable Universes in the compiler.
28         System.out.print("Arguments: ");
29         for (int i=0; i<args.length;i++) { System.out.print(args[i] + " ")
               ; }
30         System.out.println();
31         /* call the compiler */
32         boolean success = Main.compile(args);
33         assertFalse(success);
34     }
35 }
```

Listing B.1: MjcTest.java: A TestCase for the MJ compiler, using the JUnit Framework.

Appendix C

About this Masters Thesis

C.1 Mission Statement

The Software Component Technology Group has developed the Universe type system to control aliasing in object-oriented programming languages.
While a compiler for Java with the Universe-Extensions and some test programs have been implemented, only a few classes of standard Java API have been annotated with the extended type information and experience with large programs is still missing.
The mission of this masters thesis is to get realistic experience in applying the Universe type system to a real-world application.
Focussing on a delimited amount of classes (about 40 to 50) is preferred rather than typing the whole application. Therefore some stubs of other application classes have to be implemented and used by the observed component(s). Additionally, the used Java API classes will be identified and annotated with the Universe Types.
Based on the experience made while applying the Universe type system to the application the following topics are expected to be documented:

- Patterns that occured several times in the application.
- Problems occuring while applying the Universe type system to the application.
- If casts were necessary to be used, description of the situations and explanations why these casts work at runtime were needed.
- Relationship of upcoming problems with the Universe type system to the ownership model.
- Applicability of other type systems.

C.1.1 Possible Extensions

Type System Modifications Proposals how to modify the Universe type system such that encountered problems can be solved.

Invariants How can ownership-based invariants be introduced?

More APIs or components To advance the usability of the Universe type system for software developers, more APIs are needed to be annotated with the Universe Types. Some of those may be implemented and provided by library archives.
More components of the observed application may be annotated with Universe Types to get more experience with it.

C.2 Schedule

Date	*Description*	*Deliverables*
30.9.4		
1. Milestone	Project plan	Chart
	One page description of the topic	Mission
	Identification of separate subcomponents of the application	Picture about the architecture of the application
	Identification of the used Java API classes	
5.11.4		
2. Milestone	Annotation of the used Java API classes	tar-Archive of the used library classes
	Define working set boundary	Description of the working set boundary incl. list of classes, where i have to provide a stub
30.11.4		
3. Milestone	Annotation of Subcomponents and consequential Redesign	Status Report
	Introduction to the architecture of the application	Software Description
22.12.4		
4. Milestone	Annotation of Subcomponents and consequential Redesign	Annotated Classes
		Report of experience with programming with the Universe type system
		Collection of encountered problems
		First Results like Annotation Guide, Patterns or Problem Classifications
31.1.5		
5. Milestone	Analysis	Documentation of the analysis (Redesign, Patterns, Casts, Problems)
	Consider other type systems	Comparison
28.2.5		
6. Milestone	Extensions	Report of Extensions
16.3.5	Last beautifications	Final version

C.3 Slides of the presentation

Welcome

0

Applying the Universe type system to an industrial application

Case Study

Thomas Hächler

Master Project
September 2004 - March 2005

Supervising Assistant: Dipl.-Ing. Werner M. Dietl
Supervising Professor: Prof. Dr. Peter Müller

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

http://sct.inf.ethz.ch
Software Component Technology Group

Figure C.1: Slide 0: Welcome

Yoshi

1

Barcode based ticketing system.
Runs on device called "Yoshi".
Checks print at home® tickets bought on www.starticket.ch.

Figure C.2: Slide 1: Yoshi

Outline

2

- Introduction to the Universe type system
- Introduction to the application
- Real-world experiences
- Applying the Universe type system to Java API
- Proposals to face encountered problems
- Conclusion

Figure C.3: Slide 2: Outline

Universe type system

3

The Universe type system structures the object store.

- Ownership relations that define the universes.
- `rep`, `peer` and `readonly` references
- A type system guarantees the defined properties at compile time.
- Extension needed: `global` universe

Figure C.4: Slide 3: Universe type system

Universe type system with `global` extension

4

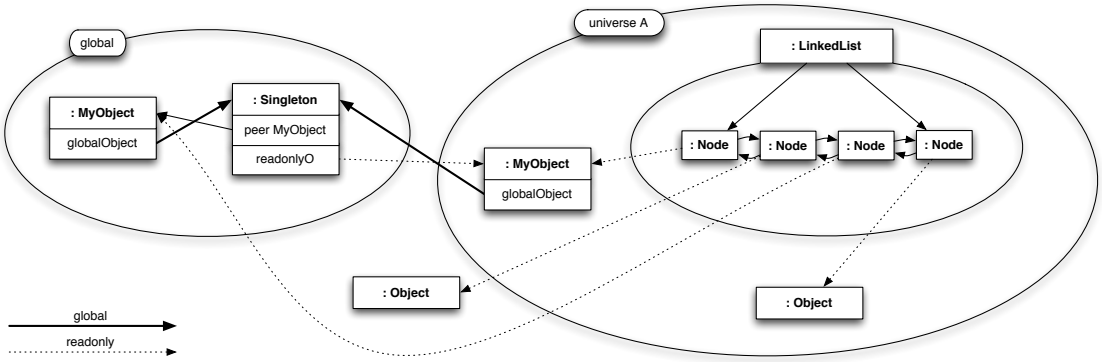

Figure C.5: Slide 4: Universe type system with `global` extension

Components of the application

5

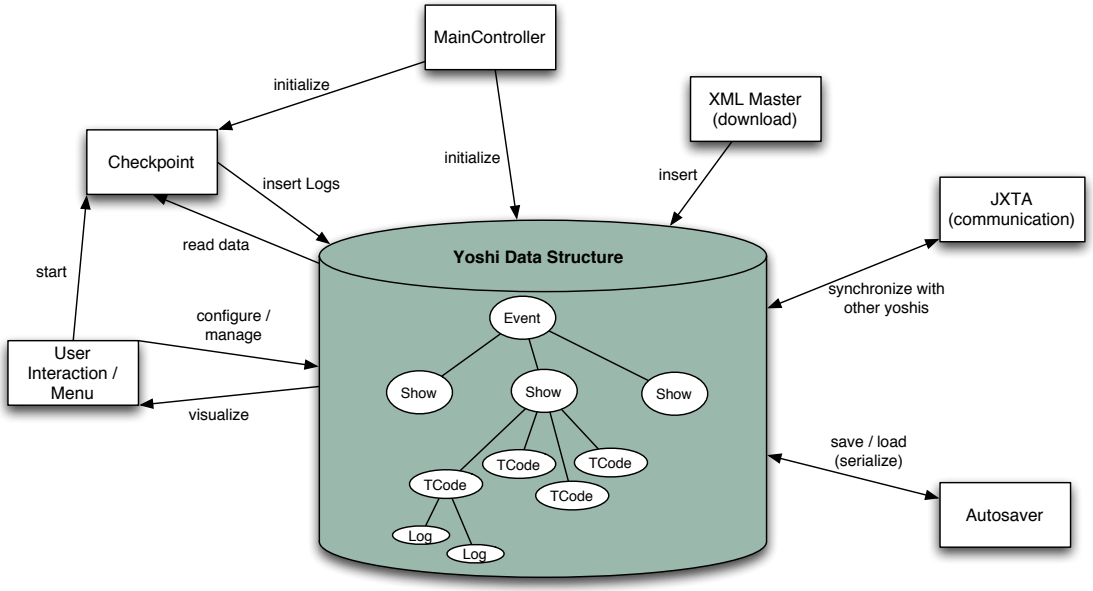

Figure C.6: Slide 5: Components of the application

Data Structure before Universe type system

6

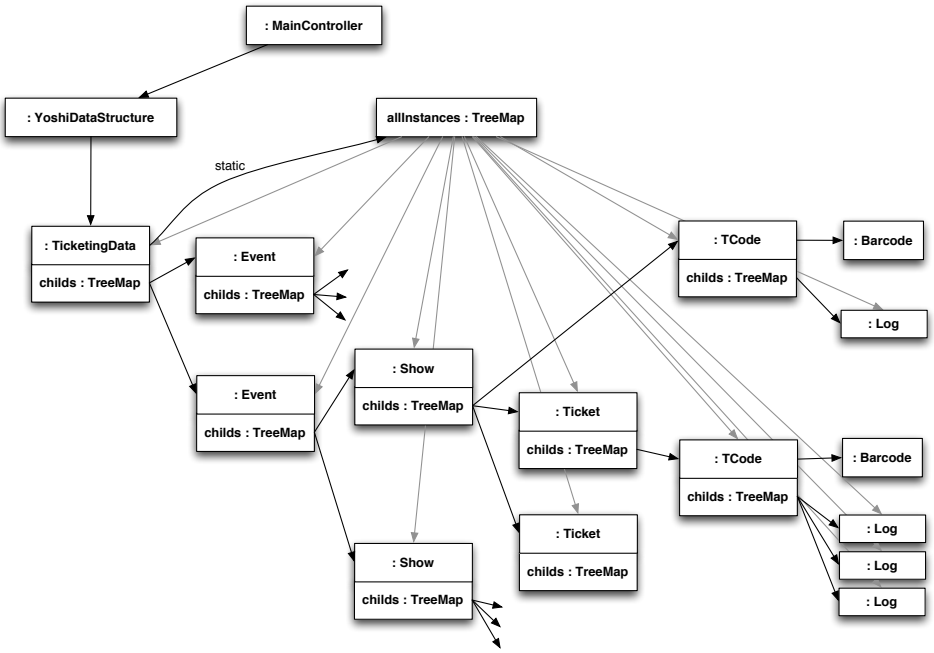

Figure C.7: Slide 6: Data Structure before Universe type system

Data Structure ownership diagram

7

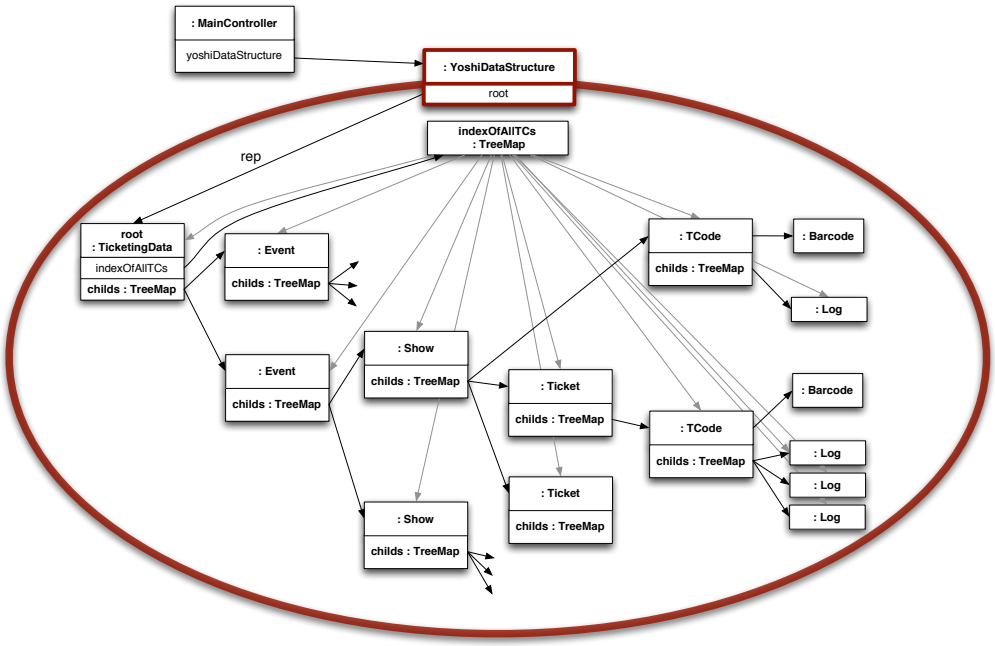

Figure C.8: Slide 7: Data Structure ownership diagram

Deeply nested Data Structure

8

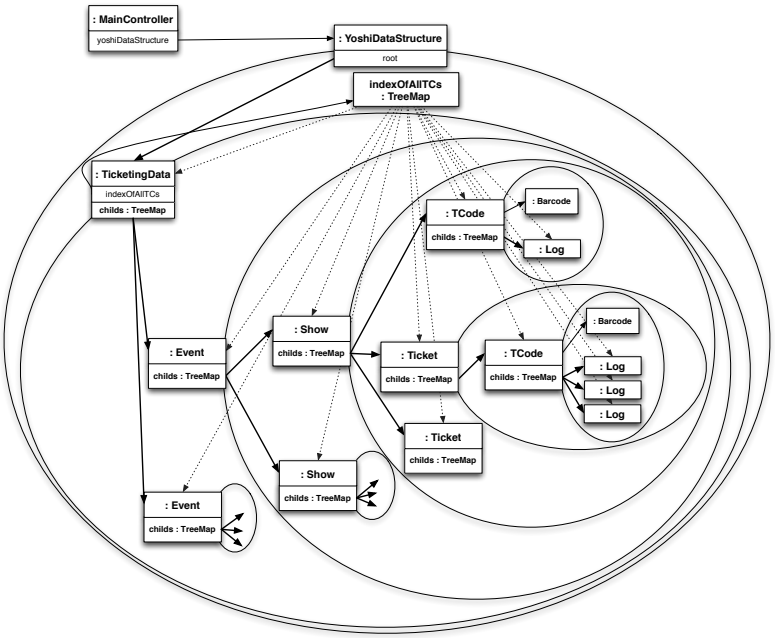

Figure C.9: Slide 8: Deeply nested Data Structure

XML Download before this case study

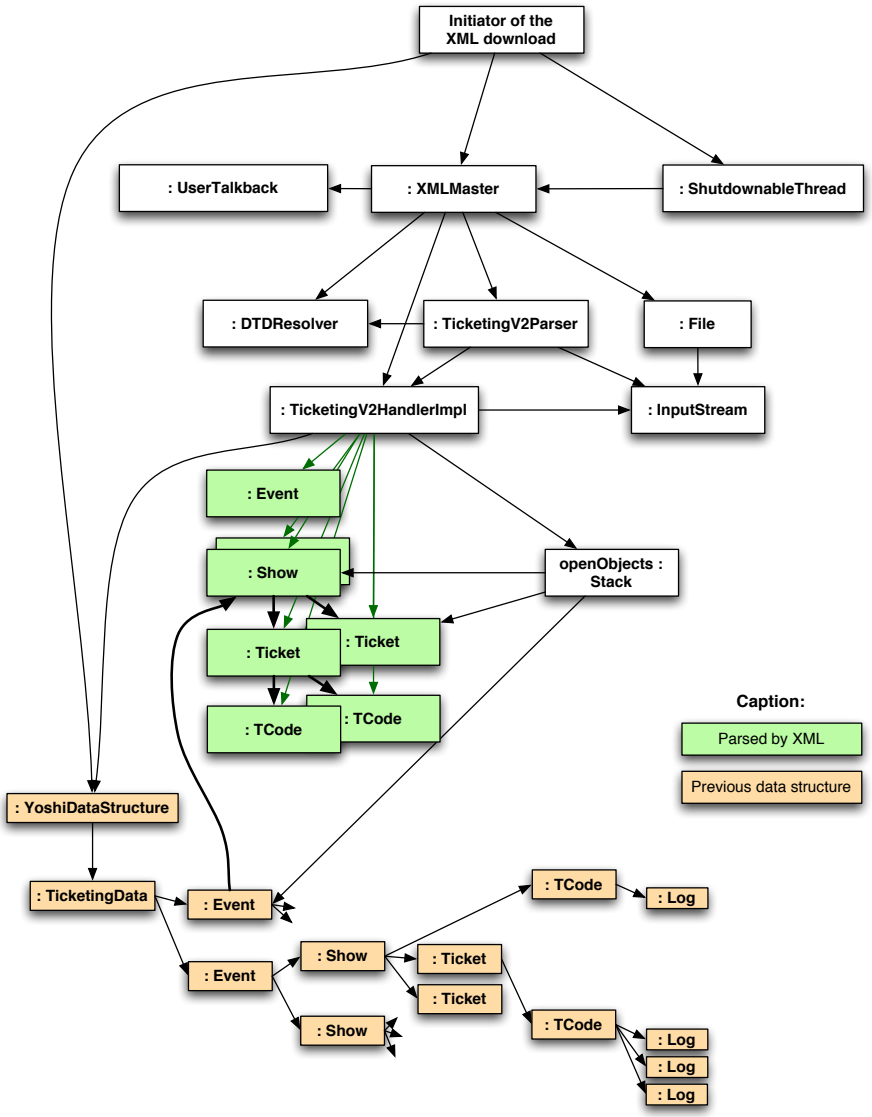

Figure C.10: Slide 9: XML Download before this case study

XML Download using the Universe type system

10

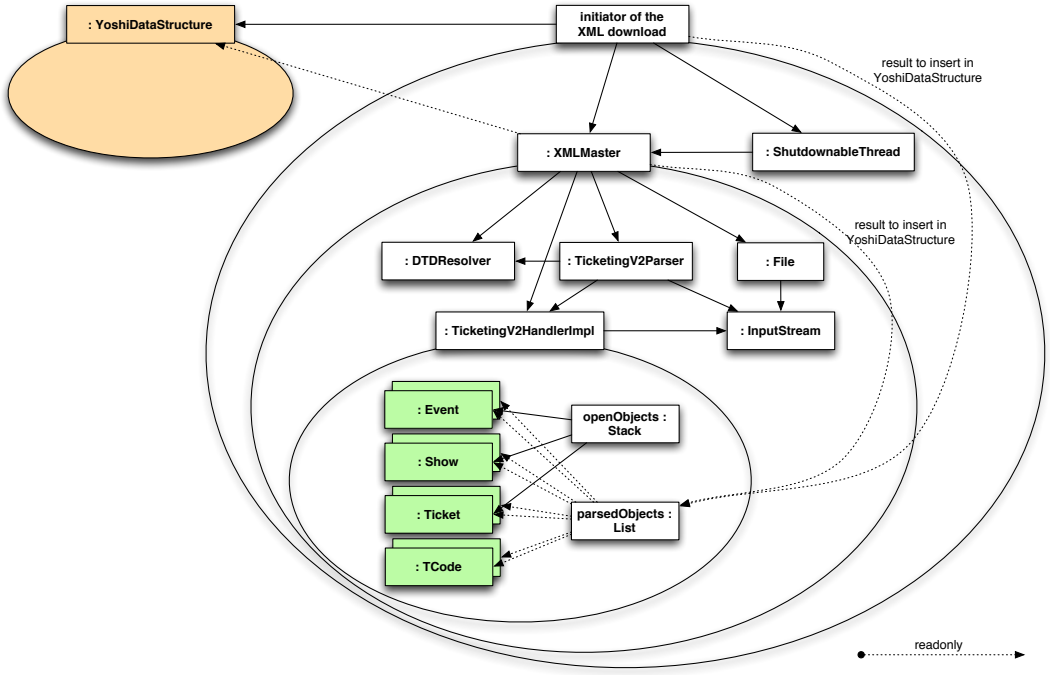

Figure C.11: Slide 10: XML Download using the Universe type system

Annotation of Java API

11

- In which universe should `System.out` be?
- Should it be possible to iterate over a `readonly` collection?
- How should the parameter `b` in `InputStream.read(byte[] b)` be annotated?

Figure C.12: Slide 11: Annotation of Java API

Annotation of Java API

12

- In which universe should `System.out` be?

```
public static global PrintStream out;
```

Figure C.13: Slide 12: Annotation of Java API

Problem with `java.util.Iterator`

13

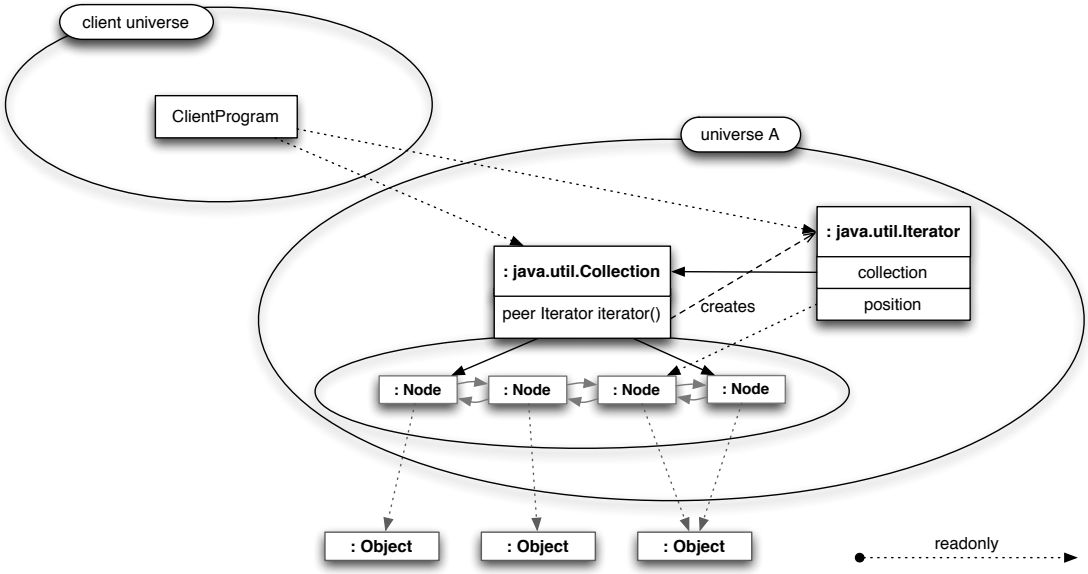

Figure C.14: Slide 13: Problem with `java.util.Iterator`

Iterator on read-only Collection

14

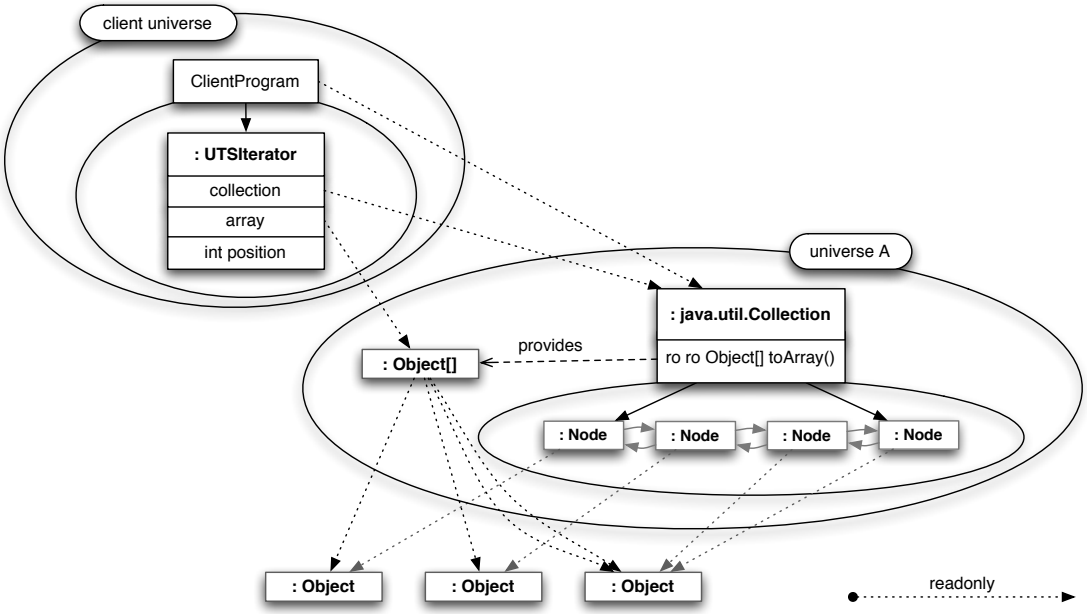

Figure C.15: Slide 14: Iterator on read-only Collection

Motivation of `writable` universe type

15

in `InputStream` (and other java.io.*):

```
InputStream.read(peer byte[] b);
InputStream.read(rep byte[] b);
InputStream.read(global byte[] b);
```

in `StringBuffer`:

```
void getChars(int srcBegin, int srcEnd, peer char[] dst, int dstBegin);
void getChars(int srcBegin, int srcEnd, rep char[] dst, int dstBegin);
void getChars(int srcBegin, int srcEnd, global char[] dst, int dstBegin);
```

in a composite pattern:

```
abstract void collectInformation(peer Map m);
abstract void collectInformation(rep Map m);
abstract void collectInformation(global Map m);
```

Figure C.16: Slide 15: Motivation of `writable` universe type

Example: method with a `writable` parameter

16

```
class C {
    <writable U> void writeTo(U PrintStream p) {
        p.writeln("hello universe!");
    }
}


// client with peer reference c to instance of C:
peer C c = new peer C();
c.writeTo(System.out);                  // U is resolved to be global.
c.writeTo(new peer PrintStream(..));    // U is resolved to be peer.
c.writeTo(new rep PrintStream(..));     // U is resolved to be readonly
                                        // => incompatible => compile time
                                            error.
```

Figure C.17: Slide 16: Example: method with a `writable` parameter

Idea of a `writable` universe type

17

- `writable` stands for { `peer`, `rep`, `global` }
- `writable` allowed for formal parameters, local variables and return values only
- Not allowed for fields – Restrictions for actual parameters
- Compiler checks whether the actual parameter is read-write for the callee

Figure C.18: Slide 17: Idea of a `writable` universe type

Other work

18

- Lot of work until application compiled with MJ/JML
- Bug reports for MJ/JML compiler
- `Copyable` – an interface to copy objects crossing universe boundaries
- Proposal for implicit `readonly` annotation
- Proposal for method-local universes
- Workarounds for programming patterns

Figure C.19: Slide 18: Other work

Conclusion

19

- Universe type system works
- Universe type system implies better structures
- Restructuring required – runtime overhead possible
- `global` universe needed in real world (logging, properties, singletons)
- Restructuring of API needed
- Some ideas to make life easier

Figure C.20: Slide 19: Conclusion