Minor Thesis

Evaluation of Reuseware based on the development of a web-based community-platform

submitted by

Rene Gastmeier
born 28.03.1982 in Dresden

Technische Universität Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: Dipl.-Medieninf. Jendrik Johannes
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted September 1, 2010
# Contents

1 Introduction 8

2 Foundations 10
   2.1 Object-Orientation 10
   2.2 Aspect-Orientation 11
   2.3 Component-Orientation 11
      2.3.1 Component 11
      2.3.2 Composition System 12
      2.3.3 Invasive Software Composition 12
   2.4 Feature 13
   2.5 Ontology 14

3 Technology Stack 15
   3.1 Java Technology 15
      3.1.1 Java Platform Enterprise Edition 16
   3.2 Spring Framework 17
   3.3 Hibernate 18
   3.4 Wicket 21
   3.5 AspectJ 23
   3.6 Reuseware 24

4 The Project: Ontology Components 28
   4.1 Component Framework 28
      4.1.1 Entity Package 30
      4.1.2 Term Package 30
      4.1.3 Association Package 30
      4.1.4 Concept Package 31
      4.1.5 Ontology Package 32
      4.1.6 Workflow Package 32
      4.1.7 Collection Package 33
   4.2 Realization 34
      4.2.1 Software Process 34
      4.2.2 Architecture 35
5 Reuseware Incorporation and Evaluation

5.1 Non-polymorphic Inheritance for Hibernate

5.1.1 The Native Solution

5.1.2 The Reuseware Solution

5.1.3 Comparison and Evaluation

5.2 Exchangeable Class References for Hibernate

5.2.1 The Native Solution

5.2.2 The Reuseware Solution

5.2.3 Comparison and Evaluation

5.3 Using Hibernate Entity as Component Definitions

5.3.1 Native Solution

5.3.2 Reuseware Solution

5.3.3 Comparison and Evaluation

5.4 Aspects for Java, Hibernate, and Wicket-HTML

5.4.1 Native Solution

5.4.2 Reuseware Solution

5.4.3 AspectJ Solution

5.4.4 Comparison and Evaluation

5.5 Advanced Aspects for Java

5.5.1 Native Solution

5.5.2 Reuseware Solution

5.5.3 AspectJ Solution

5.5.4 Comparison and Evaluation

5.6 Exchangeable Class References for Spring

5.6.1 Native Solution

5.6.2 Reuseware Solution

5.6.3 Comparison and Evaluation

5.7 Overall Comparison and Evaluation

6 Summary and Perspective
## List of Figures

2.1 Basic feature diagram ........................................... 13
3.1 Multi-tiered architecture of an enterprise application .......... 17
3.2 A tuple .......................................................... 19
3.3 Composition program example ................................... 27
4.1 Package Structure ............................................... 29
4.2 Core Entity ...................................................... 30
4.3 Core Term ........................................................ 30
4.4 Feature tree of the term component ................................ 31
4.5 Core Association ............................................... 31
4.6 Core Concept .................................................... 31
4.7 Core Ontology ................................................... 32
4.8 Core Workflow .................................................. 33
4.9 Collection edit panel ............................................ 33
4.10 Domain and application engineering ............................. 34
4.11 Layers and their technologies ................................... 35
4.12 Classes in the business and persistence layer ................. 36
5.1 Modularization concepts ........................................ 40
5.2 Concept mapping files ........................................... 41
5.3 Ontology mapping files ......................................... 41
5.4 Non-polymorphically tables ..................................... 43
5.5 Hibernate mapping file Ecore model ............................ 46
5.6 Inheritance mapping ............................................ 48
5.7 Mappings belonging to clazz component ....................... 54
5.8Clazz component associating PropertyRestriction components .... 56
5.9 Composition program with usage of a Hibernate component .... 60
5.10 Term component package structure ............................. 61
5.11 Abbreviation feature ........................................... 61
5.12 Language feature ............................................... 61
5.13 Event communication .......................................... 66
5.14 Assigning the feature extension to Wickets term edit panel .... 74
5.15 Workflow composition diagram .................................................. 82
5.16 A fragment with many ports ...................................................... 83
5.17 Composition program for a Spring configuration file ..................... 87
5.18 Reuseware composition technique evaluation ................................... 88
5.19 Reuseware Eclipse tools evaluation .............................................. 88
5.20 Evaluation of the native Hibernate solutions .................................. 89
5.21 Evaluation of the native Wicket solution ...................................... 90
5.22 Evaluation of the native Java solutions ....................................... 90
5.23 Evaluation of the AspectJ solutions ....................................... 91
5.24 Evaluation of the Spring solution .............................................. 92
<table>
<thead>
<tr>
<th>Listings</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 XML driven configuration of the Spring IoC container</td>
<td>18</td>
</tr>
<tr>
<td>3.2 An example Java class that is to be persisted</td>
<td>19</td>
</tr>
<tr>
<td>3.3 An example for a mapping file</td>
<td>20</td>
</tr>
<tr>
<td>3.4 Hibernate entity example</td>
<td>20</td>
</tr>
<tr>
<td>3.5 An example deposit class</td>
<td>21</td>
</tr>
<tr>
<td>3.6 Hibernate component example</td>
<td>21</td>
</tr>
<tr>
<td>3.7 An example Wicket web page Java class</td>
<td>22</td>
</tr>
<tr>
<td>3.8 An example Wicket web page HTML file</td>
<td>22</td>
</tr>
<tr>
<td>3.9 A logging aspect defining a pointcut and two advices</td>
<td>23</td>
</tr>
<tr>
<td>3.10 A logging aspect defining a pointcut and two advices</td>
<td>24</td>
</tr>
<tr>
<td>3.11 An example composition system definition</td>
<td>25</td>
</tr>
<tr>
<td>3.12 An example of a reuse extension</td>
<td>26</td>
</tr>
<tr>
<td>3.13 An example class defining some attributes</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Java generics for expressing component dependencies</td>
<td>36</td>
</tr>
<tr>
<td>4.2 An example with evolved classes</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Ordering of interfaces</td>
<td>37</td>
</tr>
<tr>
<td>4.4 Wrong DAO component chosen</td>
<td>38</td>
</tr>
<tr>
<td>4.5 Component deployment in Spring IOC container</td>
<td>39</td>
</tr>
<tr>
<td>5.1 Concept.hbm.xml</td>
<td>41</td>
</tr>
<tr>
<td>5.2 Instance.hbm.xml</td>
<td>41</td>
</tr>
<tr>
<td>5.3 ConceptProperties.xml</td>
<td>43</td>
</tr>
<tr>
<td>5.4 Concept.hbm.xml with XML entity usage</td>
<td>43</td>
</tr>
<tr>
<td>5.5 OntologyConceptProperties.xml</td>
<td>44</td>
</tr>
<tr>
<td>5.6 OntologyConcept.hbm.xml with XML entity usage</td>
<td>44</td>
</tr>
<tr>
<td>5.7 Inheritance composition system</td>
<td>45</td>
</tr>
<tr>
<td>5.8 Inheritance reuse extension for Hibernate</td>
<td>46</td>
</tr>
<tr>
<td>5.9 Mapping file skeleton for an ordered ontology component</td>
<td>48</td>
</tr>
<tr>
<td>5.10 Association.hbm.xml</td>
<td>49</td>
</tr>
<tr>
<td>5.11 Mapping file for concept component</td>
<td>50</td>
</tr>
<tr>
<td>5.12 Mapping file for association component with the use of the any element</td>
<td>51</td>
</tr>
<tr>
<td>5.13 Reference composition system for exchangeable class references</td>
<td>52</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.14</td>
<td>Reference reuse extension for Hibernate</td>
</tr>
<tr>
<td>5.15</td>
<td>Association mapping with adopted class references</td>
</tr>
<tr>
<td>5.16</td>
<td>AssociationAssociatedAsComponent.hbm.xml</td>
</tr>
<tr>
<td>5.17</td>
<td>AssociationAssociatedAsComponent.hbm.xml with a DTD entity</td>
</tr>
<tr>
<td>5.18</td>
<td>Adopted reference reuse extension for the use of Hibernate components</td>
</tr>
<tr>
<td>5.19</td>
<td>Inheritance composition system with added Hibernate component support</td>
</tr>
<tr>
<td>5.20</td>
<td>Inheritance reuse extension with added Hibernate component support</td>
</tr>
<tr>
<td>5.21</td>
<td>Feature implementing class abbreviation</td>
</tr>
<tr>
<td>5.22</td>
<td>Wicket edit panel class for abbreviation feature</td>
</tr>
<tr>
<td>5.23</td>
<td>HTML file for abbreviation feature</td>
</tr>
<tr>
<td>5.24</td>
<td>HTML file for the core that is a term component</td>
</tr>
<tr>
<td>5.25</td>
<td>Interface for adding features to a core</td>
</tr>
<tr>
<td>5.26</td>
<td>Interface for adding features to a core</td>
</tr>
<tr>
<td>5.27</td>
<td>Advisor interface</td>
</tr>
<tr>
<td>5.28</td>
<td>Interface for adding advisors to a core</td>
</tr>
<tr>
<td>5.29</td>
<td>An advisor for adding feature Wicket components into the Wicket component tree of a core</td>
</tr>
<tr>
<td>5.30</td>
<td>Feature edit panel gets assigned an advisor</td>
</tr>
<tr>
<td>5.31</td>
<td>Core edit panel for the term component gets assigned feature edit panels</td>
</tr>
<tr>
<td>5.32</td>
<td>SolidCoreTermEdit.html</td>
</tr>
<tr>
<td>5.33</td>
<td>Wicket HTML inheritance</td>
</tr>
<tr>
<td>5.34</td>
<td>Inheritance reuse extension for Java</td>
</tr>
<tr>
<td>5.35</td>
<td>Composition system definition for aspects</td>
</tr>
<tr>
<td>5.36</td>
<td>Reuse extension of aspect composition system for Java</td>
</tr>
<tr>
<td>5.37</td>
<td>Inter-type declarations for adding feature properties</td>
</tr>
<tr>
<td>5.38</td>
<td>Use of the abbreviation aspect in component order phase</td>
</tr>
<tr>
<td>5.39</td>
<td>Advice to add feature Wicket components</td>
</tr>
<tr>
<td>5.40</td>
<td>Attributes of a workflowed entity</td>
</tr>
<tr>
<td>5.41</td>
<td>Life-cycle methods of CoreEntityManager</td>
</tr>
<tr>
<td>5.42</td>
<td>Advisor for an workflowed entity</td>
</tr>
<tr>
<td>5.43</td>
<td>Advice code for including the management of a workflowed entity</td>
</tr>
<tr>
<td>5.44</td>
<td>Aspect reuse extension with added variable referencing support</td>
</tr>
<tr>
<td>5.45</td>
<td>An ordered term manager for an instance component</td>
</tr>
<tr>
<td>5.46</td>
<td>Aspect to add feature Wicket components</td>
</tr>
<tr>
<td>5.47</td>
<td>Typical deployment of business components</td>
</tr>
<tr>
<td>5.48</td>
<td>Reuse extension for exchangable references for Spring</td>
</tr>
</tbody>
</table>
1 Introduction

The dream of software engineering is to compose software from a set of components. You need to create a web-store with users that can login and put goods into a shopping cart? Then go to a component market, look for the components that fit your needs and put them together by simple mouse clicking in a component assembly application. It is like going in a computer store buying a mainboard, a processor, a graphic card, and later putting these components together resulting in a ready to use computer system. Programming itself could be as simple as that.

This dream is not new. It was already stated in 1968 by McIlroy at a conference in Garmisch, Germany [McI69]:

Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality, and time-space performance.

Today, about 40 years later, components yet are not state of the art. But what can already be done to still use components or at least to come close to what a component-based approach would look like? This work investigates an approach using Reuseware. It is a utility with which composition systems can be defined. It is defined what kinds of components exist and in which ways they are combinable. Once defined, composition system can be put on top of any language representable with an Ecore model.

The question is, can Reuseware satisfy the needs posed by a component-oriented development approach? Are the needed modularization requirements realizable? If they are realizable, is it worth using them? These questions are answered in the context of a project whose aim is the development of ontology components that can be used in a web application. It uses Java Enterprise Edition along with Spring, Hibernate, and Wicket. These technologies represent a contemporary development environments encountered in professional projects whose aim it is to produce a reliable product.

To be able to make a better proposition regarding the use of Reuseware in a production environment it is also important to investigate solutions realized with only the native concepts the used technologies offer. In the end, both of the solutions are compared to each other. Solutions for the modularization concepts inheritance, exchangeable references, and aspects will be examined. For aspects, a solution achieved with AspectJ will be made and compared
Section 2 gives attention to object-orientation, aspect-orientation, components, features, and ontologies. Section 3 gives an overview over the used technologies Java, Hibernate, Wicket, Spring, AspectJ, and Reuseware while explaining their basic usage. Section 4 introduces the project and its development process. Finally, Section 5 presents solutions for the modularization requirements while comparing and evaluating them. The last Section 6 gives a summary and future prospects.
2 Foundations

This chapter gives an overview on object-orientation and its extension aspect-orientation. Both of them will be used to realize components. The second part introduces components and composition systems as an environment for components. Invasive software composition is an incarnation of a composition technique which is the foundation for Reuseware. Furthermore, features are introduced which serve as language independent requirement analysis technique for components. Finally, the projects aim ontologies are explained.

2.1 Object-Orientation

Object orientation is centered around objects which are runtime entities representing some concern. A concern can be a bank account or a customer owning this bank account. Every object has a class as its construction plan and is therefore also called an instance of a particular class. A class is technically spoken a collection of attributes (data) and methods (functions). The set of function heads of a class is called its interface. Arbitrary many instances can be generated from a class and every one of them will have its own instance specific data (every customer has a different name) while using the methods defined by the class to work with this data. Finding the right classes and therefore the right abstractions is a difficult task that needs a lot of experience.

Beneath objects the second key feature of object orientation is inheritance. This concept provides means for expressing different levels of abstraction. Something can be more general or more specific than something else. A bank account is more specific than an account and the latter is more general than the former. In object orientation this concept is realized through three different types of inheritance which in popular systems work together to provide a consistent experience. The first one is interface inheritance (subtyping) which enriches the interface of a subclass with that of an assigned superclass. Implementation inheritance (code inheritance, subclassing) is an extension to the former by also inheriting the source code plus attributes to the subclass. Also a subclass is allowed to override the implementation of methods already provided by the superclass to alter or extend behavior. This kind of inheritance is a powerful technique which can yield troubles if not used correctly. The problem that can appear is called the fragile base class problem [MS98]. The last type of inheritance is the substitutability (polymorphism) of superclasses through its subclasses. This becomes
possible due to the fact that a subclass can do everything a superclass can do plus potentially more. Due to the fact that a subclass can overwrite method implementations the right implementation has to be chosen at runtime. This is done through a technique called late binding.

For a complete overview on object orientation and its features refer to [AC96].

Object orientation is an evolutionary advancement of procedural languages. Anyway there are scenarios that can not be addressed satisfactorily with it.

2.2 Aspect-Orientation

Aspect-orientation is an extension of object-orientation by means that it enriches it with the concept of aspect. An Aspect is meant to grasp crosscutting concerns in a program code. A crosscutting concern is a concern about which code is scattered over the code base crossing the boundaries of a class. Logging is a very good example for such a concern. It can not be grasped through a class, it appears potentially everywhere on the beginning and ending of methods of arbitrary classes leading to scattered code. Aspect-oriented programming is ought to grasp the scattered code and abstract it in a new kind of module called an aspect. For a more complete overview and rationale behind it refer to [KLL+02].

2.3 Component-Orientation

2.3.1 Component

A component is for composition, as its name states (nomen est omen). It should be combinable with others of the same kind to create something more complete up to a complete software system. This is the main and in the end the only defining property. In [KE00] it is argued that the concept component is a natural in contrast to an artificial concept. This imposes that it can not be defined by a set of properties. An object is an example for an artificial concept. It can be defined by three properties: identity, state, and behavior. A table is an example for a natural concept. You cannot find properties that every table has. A table has an arbitrary number of legs, it has an arbitrarily looking shape, it is made from arbitrary material.

The same holds for components. The way a component looks like depends on the environment. A component can be realized with procedures, object orientation, aspect orientation, or any other modularity concept. It is only important that components are combinable in every way they are needed. If a complete software system can be built by combining components then it is called component-based software engineering. A software engineer doing so is called a
component assembler.

2.3.2 Composition System

A composition system is a system that allows for composing components. Therefore it is the foundation of component-based software engineering. It consists of a component model, a composition technique, and a composition language. The component model prescribes what a component looks like and which qualities it has. How can it be substituted, how can it be parameterized, what kind of standard components exist? The composition technique prescribes by which means components are combined internally. Can components be adapted and glued, can extensions be integrated, are aspects supported, is the binding time hidden, is there a metamodel? The composition language prescribes by which a component assembler is able to express composition. Can consistency constraints be defined, is the software process supported, is meta-composition possible?

2.3.3 Invasive Software Composition

Invasive software composition (ISC) defines a composition system for composing source-code entities called fragment components or fragments for short. The term fragment stems from the programming language BETA [MNMP93]. This programming language is developed in the Scandinavian school of object orientation where the first object oriented language Simula was developed. BETA is divided into an object language and a modularization language. The latter one is it that ISC was inspired from. In BETA a fragment is defined as a sentential form which is a sentence in a formal language that encloses nonterminal symbols. Those nonterminal symbols are given a name and called slots which lead to points of variation.

Similarly a fragment in ISC is defined as a code fragment which contains hooks. A hook also has a name and is there to be replaced, therefore providing a point of variation too. It is the same concept as slot in BETA. There are two types of hooks, implicit and declared ones. The former ones are positions or program elements contained in every programming language by definition, whereas the latter ones have to be explicitly declared by the component designer. A fragment box is a set of fragments. An invasive composer uses these hooks to combine fragments by transformation. They are built from basic composition operators like bind which replaces a hook with a value, or extend which adds a value to a list hook. Due to the reason that components get adapted through transformation of hooks ISC is a graybox composition approach. An explanation on the invasive software composition can be found in [Afm03].

COMPOST is a demonstrator realization of ISC for Java. Reuseware takes it a step further by supporting ISC for arbitrary languages. It is explained in section 3.
2.4 Feature

A feature is a reusable and configurable requirement. A component can have a set of features which describes the variants it can occur in. Feature models prescribe those sets and additionally add constraints on which feature combinations are allowed and which are not. Out of this configuration knowledge, concrete component configurations that have a set of certain features can be chosen.

A feature model most importantly consists of a feature diagram. It uses a tree-like graph which nodes are features and which arcs can be decorated to allow further semantics. The root node represents the core concept for which features are to be defined. Figure 2.1 depicts a basic feature diagram with only the parts needed to understand diagrams used in this work.

![Basic feature diagram](image)

Figure 2.1: Basic feature diagram

The concept node has three direct features $f_1$, $f_2$ and $f_3$. $f_1$ is a mandatory feature indicated by the filled circle whereas the others are optional features indicated by the hollow circle. $f_4$ is an indirect feature or a subfeature of $f_1$. For an exhausting overview on feature models and diagrams refer to [KE00].

Features itself can be realized through components. In the end, feature components have to be composed with core components to incarnate certain component variants.
2.5 Ontology

An ontology is a representation of knowledge. It consists of different entities. The entity *class* is for defining an abstract term which subsumes a set of *instances*. For example a *person* is a class whereas *Paul* is an instance of that class. A class can define *properties* which instances can use to refer to other instances. For example *friend* is a property which *Paul* can use to refer to *John* to denote him as his friend.
3 Technology Stack

This section introduces the technologies used in the project. These are Java, Spring, Hibernate, and Wicket. Furthermore, AspectJ as an incarnation of aspect-orientation is explained and of course Reuseware.

3.1 Java™ Technology

Java™ Technology consists of a programming language and a platform. The programming language called Java is a general-purpose, concurrent, class-based, object-oriented language. Its name is a registered trademark of Oracle Systems. The platform is the environment the applications written in the Java programming language run in.

Java is a strongly typed language and relatively high-level. It hides machine details such as storage management which is done by a garbage collector. This prevents from deallocation problems found in lower-level languages like C++. It’s a production language that is supposed to produce real life applications in contrast to a scientific language that is supposed to test new features. Nonetheless does it evolve in a well defined community process [javb]. Edition number 5.0 was released in 2004 and brought a number of new features such as generics, annotations, asserts, enum types, foreach loops and more. For a complete overview on the language and it’s features refer to [GJSB05].

A Java platform is supposed to fulfill the needs of particular class of applications. There are three different platforms available:

- Java Platform, Standard Edition (Java SE)
- Java Platform, Enterprise Edition (Java EE)
- Java Platform, Micro Edition (Java ME)

Java SE is the standard platform for standard software. Java EE is an environment tailored for large-scale, multi-tiered, scalable, reliable, and secure network applications. Java ME is tailored for the needs of applications running on small devices like cell phones. Each platform provides a Java Virtual Machine (JVM) and an application programming interface (API). A virtual machine is a program that is responsible for running applications written in a programming language that can compile to the particular virtual machine. The API is a
collection of classes that provide solutions for common tasks such as parsing XML files or the display of a graphical user interface (GUI). Both Java EE and Java ME build upon Java SE whereas Java ME uses only a subset of Java SE.

Since virtual machines can be developed for arbitrary hardware and software platform, programs written in Java programming language can run on arbitrary systems and are thus platform-independent. They are compiled to an intermediate representation called byte code which consists of instructions of the JVM. HotSpot is the primary JVM developed by Sun Microsystems. It features different optimization such as just-in-time (JIT) compilation which is responsible for translating JVM byte code to platform specific native machine code that is faster. This compilation is done during loadtime and therefore can slow down the start of the program. To keep this effect small it is only done for hot spots and mostly for server applications which are started once and keep running for a long time.

Java technology is available as open source. Every platform has its own open source community:

- Java SE: OpenJDK (Open Java Development Kit)
- Java EE: Glassfish
- Java ME: Mobile and Embedded

Most parts are available under the GNU General Public License (GPL) version 2.0, like the compiler javac or the virtual machine HotSpot. The GPL ensures that every program is for free use and that its sources are available for study and derivation. Derived programs that get published also have to be licensed under the GPL which is known as the copyleft principle. That is the reason why the API libraries are available under the GPL v2 with the classpath exception, which means that code using these libraries do not necessarily have to fall under the GPL license as well. This enables closed source code which uses the open source libraries. More about it can be read in [java].

### 3.1.1 Java Platform Enterprise Edition

Java Platform Enterprise Edition is the specification facilitating the development of so called enterprise applications. In this kind of applications, typically a client makes a request to a server which sends back a response by involving data from a database, see Figure 3.1.

The aim of the Java EE platform is to present solutions for the server tier, which is also called the middle tier, which is why such kind of platforms are also called middleware. Among the solutions is an API specification and a Java EE server providing the services. It most importantly consists of a web container and an application container. The former is for delivering web pages and the latter for delivering business functionality. The latter is the
reason why this kind of server is also called an application server. There are different vendors implementing the Java EE specification like IBM’s Websphere, Oracle Application Server, Bea Weblogic or Sun’s own open source Glassfish. The biggest contender to the Java EE platform is Microsoft’s .NET platform.

To build an enterprise application, developers can choose from different component models. There are models for the client tier, web server tier, and application server tier:

- Client tier: Application components, JavaBeans, and applets
- Web server tier: Servlets and Java Server Pages (JSP)
- Application server tier: Enterprise JavaBeans (EJB)

Relevant with respect to this work are Enterprise JavaBeans. They are not to be confused with JavaBeans, which in fact belong to the Java SE platform and are entirely different. EJBs exist in different flavors (stateless sessions, stateful sessions, entity, and message-driven beans), each fulfilling another kind of task. The important thing is that they get deployed to the Java EE server’s application container. When deployed, the container is able to provide the middleware services like transactions, security or persistency. The configuration of these are expressed declaratively in a deployment descriptor file which uses the XML.

For a complete overview of Java EE platform refer to [jav07].

3.2 Spring Framework

Spring is a framework with the aim to enable Java EE development without EJBs. EJBs are argued to be an over-complex solution for most of the enterprise applications encountered
in reality. They revolve around distributed applications which brings a lot of overhead to
the development process and the application itself. Therefore those applications are hard to
maintain and even show bad performance most of the time. For a complete overview on this
discussion refer to [Joh04].

The Spring framework presents a lightweight alternative to EJBs. Firstly, there is a container
which permits the deployment of plain old Java objects (POJO) instead of EJBs. Therefore,
POJOs become components which are called beans in Spring. Secondly, Spring provides
inversion of control (IoC) by means of dependency injection (DI). IoC means that objects are
put in an environment which has control over these objects, not vice versa. Frameworks are a
popular incarnation of this principle. In the case of DI this means that dependencies objects
may have among each other can be defined in a different place than the program itself.

One implementation of the Spring IoC container permits its configuration via XML-files which
is the place DI is happening. Listing 3.1 shows such an XML-file:

```
    <bean id="beanA" class="my.library.BeanAImpl"/>
    <bean id="beanB" class="my.library.BeanBImpl">
        <property name="beanADependency" ref="beanA"/>
    </bean>
```

Listing 3.1: XML driven configuration of the Spring IoC container

In the example, a bean named beanA and a bean named beanB are deployed. beanB
has a dependency to a bean with the interface my.library.BeanA and is dependency injected
with beanA which is an implementation of that interface. As this example tries to show,
programming against interfaces becomes more rewarding, because implementations can be
switched easily.

For a complete overview on the features and their usage refer to [spra].

3.3 Hibernate

In an enterprise application there are a lot of entities like bank accounts or ontology entities
that have to be persisted. Persistent data is data that outlives its program. The most popular
technology used to solve this task is relational database technology. They are based on a
relational model consisting of relations and a set of operations defined over these relations. A
relation is a set of tuples of the same kind. A tuple is a set of attributes each consisting of an
attribute name, an attribute type, and an attribute value. Each tuple most likely represents
data belonging to a particular concern e.g. a bank account. Figure 3.2 depicts a tuple.

A set of operations called the relational algebra offers the ability to produce new relations
out of existing ones as a result of some query. A query can be formulated in SQL (Structured
Query Language) [sql]. For an exhausting overview on relational databases refer to [DD00]. Whereas the relational database world is based on a relational model, programming is most likely done in an object oriented way. Obviously these two paradigms differ. A mapping from the world of objects to the world of relations has to be done. The problems encountered in creating such a mapping are subsumed by the “object relational impedance mismatch problem”. One problem is that a class can have an arbitrary structure and can also be part of a class inheritance hierarchy, whereas in the relational model there are no comparable concepts. An overview and classification of these problems can be found here [IBNW09].

An object relational mapper (ORM) is a framework which strives to solve the mentioned impedance problem. Hibernate is one of them and very popular too. It supports the object oriented idiom including association, inheritance, polymorphism, composition, and collections. It also supports querying for objects with its own query language HQL (Hibernate Query Language) which is optimized for an object oriented context, although it also supports plain SQL.

For Hibernate to know which attributes of a class are to be persisted, a mapping has to be created. This can either be done in the code with Java 5 annotations or external in XML files. The author’s preferred method is XML mapping files because of the code remaining untouched. Listing 3.2 shows a simple class that is to be persisted.

```java
1 class BankAccount {
2     private Long id;
3     private Float deposit;
4     private String currency;
5
6     public String getId() {...};
7     public void setId(Long id) {...};
8
9     // getter and setter for every attribute
10 }
```

Listing 3.2: An example Java class that is to be persisted

Hibernate is a non-intrusive solution which means that it does not require following many

![Figure 3.2: A tuple](image-url)
specific rules. But there are some and one of them is the id attribute that has to be included in every class that should be retrievable from the database in its own right. In Hibernate those are called entities. The data type of the id is not restricted to Long, virtually anything can be an identifier type. The very simple class of example 3.2 is mapped like in Listing 3.3.

```xml
<hibernate-mapping>
  <class name="BankAccount">
    <id name="id" type="long">
      <generator class="native"/>
    </id>
    <property name="deposit"/>
    <property name="currency"/>
  </class>
</hibernate-mapping>
```

Listing 3.3: An example for a mapping file

In a real application a customer would most likely be represented by its own class Customer. In Hibernate this can be modeled as an Hibernate entity with its own identity and table. The class needs to have an id attribute and its own mapping file. The mapping of BankAccount can then refer to it by a foreign key association. Listing 3.4 shows how such kind of relationship is mapped.

```xml
<hibernate-mapping>
  <class name="BankAccount">
    <id name="id" type="long">
      <generator class="native"/>
    </id>
    <many-to-one name="name" class="Name"/>
  </class>
</hibernate-mapping>
```

Listing 3.4: Hibernate entity example

There is also the possibility of one-to-many and many-to-many relationships which could be represented by any type of collection Java offers. For an explanation on those refer to [KBA+].

Furthermore Hibernate supports the fine grained object model. Assuming the property deposit should be represented by its own class for more detail. Listing 3.5 shows an example of
such a class.

```java
class Deposit {
    private Float amount ;
    private String currency ;

    public String getAmount ( ) {...} ;
    public void setAmount( Float id) {...} ;
    // getter and setter for every attribute
}
```

Listing 3.5: An example deposit class

This class is tightly coupled to the `BankAccount` class and shares its life cycle. This means if a `BankAccount` instance is created also a `Deposit` instance will be created. There is no need to have a separate entity which is persisted in its own table. Such kind of relationship is called composition in UML. To model those with Hibernate if offers Hibernate components which in contrast to Hibernate entities are not persisted in their own tables but in the same table as the component holding entity. Listing 3.6 shows the definition of a Hibernate component.

```xml
<hibernate-mapping>
  <class name="BankAccount">
    ...
    <component name="deposit" class="Deposit">
      <property name="amount"/>
      <property name="currency"/>
    </component>
  </class>
</hibernate-mapping>
```

Listing 3.6: Hibernate component example

The shown mapping elements `property`, `many-to-one`, and `component` are all there to map properties (attributes) of a class. Therefore those elements will be called `property mappings` throughout this work.

For a detailed overview on Hibernate and its features refer to [KBA+].

### 3.4 Wicket

Wicket is a web application framework. The first of two main features is component-orientation. This means that web sites can be composed through the use of GUI components just like in desktop GUI frameworks like Swing. A web page or a component itself consists of a tree of components. Components get embedded in HTML through the use of special
Wicket tags representing a tree structure that is defined in a Java class. In the end a web page or a component consists of a Java class and a HTML file. Listings 3.7 and 3.8 show a very simple example of such a pair.

```java
public class Example extends WebPage {
    private int counter = 0;

    public Example() {
        Link link = new Link("link") {
            @Override
            public void onClick() {
                counter++
            }
        }
        Label label = new Label("label", "A\text{link}\_label");
        link.add(label);
    }
}
```

Listing 3.7: An example Wicket web page Java class

```html
<html>
<body>
    <a wicket:id="link">\text{link}\_label</a>
</body>
</html>
```

Listing 3.8: An example Wicket web page HTML file

The Java class represents a web page by extending the Wicket class `WebPage`. In the constructor there are two components added, one of type `Link` and one of type `Label` which represents the link text. The first parameter of both of them is the component id which is needed to identify the corresponding embedded component in the HTML file. To do so, HTML elements can have a `wicket:id` attribute which holds the same component identifier as the component in the Java part. Wicket parses the HTML file and replaces those elements that refer to Wicket components with content coming from the Java class realizing the Wicket component. That way a separation of presentation and business logic is achieved.

The second main feature of Wicket is the automatic storage of state that is needed when interacting with the user in more complex ways. In the example above a variable `counter` is incremented each time the link is clicked. This variable is a private attribute of the web page class `Example`. Without an additional mechanism this attribute would be zero every time the example web page is requested because HTTP is a stateless protocol where every request is handled independently. Wicket serializes therefore stores the web page every time
the same user requests it. The next time the user calls the same page again the stored one is deserialized and sent. That way the variable counter really increases every time the link is clicked.

In a Java environment there are objects that have state. In HTTP communication this state has to be added by some solution. Therefore Wicket is said to solve the impedance mismatch between the stateless HTTP protocol and object-oriented Java programming.

### 3.5 AspectJ

AspectJ is a realization of aspect-oriented programming for Java. It introduces some concepts that define the language of it. An aspect itself is a module encapsulating a crosscutting concern. It consists of advices, pointcuts, and inter-type declarations. Furthermore it can contain ordinary attributes and methods, it even can inherit from an ordinary Java class. AspectJ includes a compiler, a debugger, a documentation generator, a program structure browser, and integration with Eclipse, Ant, and other tools.

A pointcut is a set of join points which can be for example the beginning or the end of a method. A join point and in the end a pointcut too is a similar concept as a hook in invasive software composition. An advice is a piece of code that gets introduced before, after, or around pointcuts. This introduction happens on Java byte code and is called weaving. It means that the code is inserted in the other code by program transformation. The original code is extended by advice code. This is done by the AspectJ compiler. Listing 3.9 shows an aspect defining logging code to be woven into some class.

```java
public aspect LoggingAspect {

    pointcut someMethodPointcut () {
        call(void someClass.someMethod(int));
    }

    before() : someMethodPointcut () {
        System.out.println("entering method someMethod");
    }

    after() : someMethodPointcut () {
        System.out.println("leaving method someMethod");
    }
}
```

Listing 3.9: A logging aspect defining a pointcut and two advices

This simple example defines a pointcut someMehtodPointcut which consists of a join point that picks out every call to a method of the signature someClass.someMethod(int). In every
part of the signature there can be used the wildcards * to express that there can be arbitrary characters.

Inter-type declarations declare attributes and methods which are defined in the aspect but belong to the target class. That way classes can be extended with new behavior from the outside. Listing 3.10 gives an example.

```
public aspect SomeAspect {
    private int someClass.someAttribute = 0;

    public void someClass.setSomeAttribute(int someAttribute) {
        this.someAttribute = someAttribute;
    }

    public int someClass.getSomeAttribute() {
        return this.someAttribute;
    }
}
```

Listing 3.10: A logging aspect defining a pointcut and two advices

This aspect defines an attribute someAttribute on a class someClass with its accessor methods. The attribute is of scope private which means that even if there are other attributes with the same name on the target class or other aspects there will be no conflict because it is only visible for the declaring aspect itself. Differently, methods which are duplicates will lead to compilation errors within the AspectJ compiler.

For a complete overview on AspectJ refer to their website [asp].

### 3.6 Reuseware

Reuseware aims at bringing ISC to arbitrary languages that can be described by a grammar. Amongst the targeted group are domain specific languages (DSLs). They often only support the most important constructs to fulfill their purpose but do not cover concepts for modularization. The reason for this is that adding modularization concepts is a complex task in its own right. A light-weight dedicated composition system (LWDCS), Reuseware that is, can make up for this drawback. It is light-weight because the target language will not be aware of it and it is dedicated because it addresses concerns for a single language only.

Reuseware presents a formalism that describes how ISC fragment components can be defined for arbitrary languages that are described either by a grammar or by an Ecore metamodel. Those languages get extended with the concepts needed to enable composition and then
are called *reuse language* while the original language is called the *core language*. The reuse language introduces fragments and variation points that can reside inside fragments. In terms of an EBNF grammar a variation point is just like in BETA a named nonterminal whereas a fragment is a nonterminal potentially different from the starting nonterminal. There are three kinds of variation point pairs:

- prototype/hook:
- value prototype/value hook:
- anchor/slot:

Variation points can be replaced by fragments of the same kind of nonterminal as the variation point. This happens through composition operators. Reuseware supports four types:

- bind: replaces a slot with a fragment and therefore removing the variation point
- extend: add a fragment to a hook while not removing the variation point
- prepend: add a fragment before a hook
- append: add a fragment after a hook

Reuseware implements its concepts in the Reuseware composition framework which has a frontend that is available as plugin for the Eclipse platform [The].

To demonstrate the definition of a composition system, a small example will be used. The goal is to transfer attributes from one Java class to another. The composition system in Listing 3.11 defines two fragment roles. One is the provider of attributes called *Attribute_Provider* and the other one is the receiver of attributes.

```plaintext
compositionsystem example.attributes {  
fragment role Attribute_Provider {  
dynamic port Prototype;  
}  
fragment role Attribute_Receiver {  
static port Hook;  
}  
contributing association copy {  
    Attribute_Provider.Prototype ⟷ Attribute_Receiver.Hook  
}  
}
```

Listing 3.11: An example composition system definition
Both fragments define one port. A port is a collection of variation points. Port Prototype will refer to all attributes a class possess. It is a dynamic port because there can be arbitrary many attributes. For every one a port is created. Port Hook is a static one because there is exactly one location in Java classes to add attributes. Furthermore an association is defined that determines which ports can be assigned to which others. In this case, the port Prototype of fragment role Attribute_Provider can be associated to port Hook of fragment role Attribute_Receiver. This means that attributes are copied from an Attribute_Provider to an Attribute_Receiver.

The next step is to write a reuse extension to assign variation points to the defined ports. In this case a prototype/hook combination is chosen. Listing 3.12 shows it.

```java
componentmodel example.attributes java
implements example.attributes
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {

  fragment role Attribute_Provider {
    port Prototype {
      java::members::Field is prototype {
        port expr = $self.name$
      }
    }
  }

  fragment role Attribute_Receiver {
    port Hook {
      java::classifiers::Class.members is hook {
      }
    }
  }
}
```

Listing 3.12: An example of a reuse extension

When defining a reuse extension an Epackage providing the Ecore model for the desired language has to be given. In this case a model for Java is needed. The Epackage’s identifier refers to JaMoPP (Java model parser and printer) [jam]. It provides the needed Ecore model. The model consists of different Eclasses from which CompilationUnit denotes a Java file as a whole. Therefore it is used as rootclass. Port Prototype uses Eclass Field as prototype which is an attribute of a Java class. For the reason of this port being a dynamic one there can be arbitrary many Prototype ports. To have a distinct name for every one a point expression which denotes a name for every port is given. It is an OCL expression. self denotes the Eclass Field’s instance while attribute name is the name of the recent field.
Later on a composition program can be used to combine fragments. Suppose a class `ClassA` like in Listing 3.13 and an empty class `ClassB`.

```java
public class ClassA {
    private String aString;
    private int anInt;
}
```

Listing 3.13: An example class defining some attributes

The composition program in Figure 3.3 copies both of the attributes `ClassA` defines to `ClassB`.

![Figure 3.3: Composition program example](image)

As can be seen, fragment `ClassA.java` has three ports, two of them belonging to `Attribute_Provider` fragment role providing the two attributes `ClassA` defines. Both of them are connected to port `Hook` of fragment role `Attribute_Receiver` of fragment `ClassB.java`. Both fragments play both roles which means that fragment `ClassA.java` can also receive attributes and fragment `ClassB.java` can also provide attributes. Fragment `ClassB.java` is filled with a gray color which means that it is a target which in turn means that a new composed fragment for this fragment is created. In this case it means that a new Java file containing class `ClassB` is created which also contains the two attributes defined by class `ClassA`. By default, composed fragments are put into a subdirectory `composed` of the directory of the target fragment.
4 The Project: Ontology Components

The project serving as the case study is a framework for ontologies. The aim is to represent the entities of an ontology in a component-oriented fashion. There will be different features assignable to a component. Both of them, components and features, will reside in a component framework. Later on, specific component configurations can be ordered from that framework.

A component in this work is through the use of Spring technically an ordinary Java class. This means that the component framework will consist of a set of Java classes implementing the different concerns. Conceptually those classes are components because they are to be composable. Therefore the terms class and component will be used interchangeably depending on the context which can have either a technical or conceptual nature. Sometimes the classes will be called component implementing classes to build a mental bridge between these two natures.

The resulting ontology components are to be used in a web environment. There, a large community should be able to manage an ontology by adding, deleting, and editing ontology entities. These activities should take part in a community process defined through a workflow. To provide such functionality for the web tier, HTML or some kind of HTML generating technology is needed to present forms to the user. Once created, ontology entities have to be persisted in a database. Additionally, transaction management is needed to resolve conflicts emerging from the concurrency of the system. Summa summarum the system should fulfill the requirements of a typical enterprise application.

4.1 Component Framework

The components residing in the component framework are realized in Java by POJOs which is possible through the use of Spring. There are seven different packages that are important for this work:

- Entity Package: Provides abstract entities that have an identifier and basic management functionality for them (create, edit, delete).
- Concept Package: Based on the entity package it provides components that can have arbitrary many names. This is needed for some of the ontology entities.
- Term Package: Provides terms for naming things.
- Association Package: Provides associations to associate components while adding metadata like creator or time of creation.
- Ontology Package: Using the concept components for some of its entities, it provides the entities an ontology consists of.
- Workflow Package: Provides means to define workflows.
- Collection Package: Provides Wicket panels for displaying and editing arbitrary collections.

The ontology package provides representations of ontology entities. Some of them, like classes and instances, have to have names. These are based on the concept package which provides components that can have names. The names in turn are provided by the term package, each name represented by a term component. The term is associated to a named entity through an association component coming from the association package. The entity package provides a framework for components that occur in a large amount and therefore have to be managed (be created, edited, and deleted). It is the base for the concept, association, and term component. A collection of terms can be displayed and edited by Wicket panels defined in the collection package. The life cycle of a term managed by a community can be defined with the help of the workflow package. Figure 4.1 depicts the package structure and the relations between them.

![Figure 4.1: Package Structure](image)

The challenge here is the three different involved languages Java, Hibernate Mapping Markup and Wicket extended HTML. It is desirable to use the same modularity concepts on every tier. This work investigates how this can be achieved by using native means supported by
the used languages and with the help of Reuseware while comparing both approaches.

4.1.1 Entity Package

The entity package is the base for every component that should be represented as a Hibernate entity. This means that it is represented by its own table in a database which leads to entities that are retrievable by their own. To achieve this, entities have an identifier attribute which acts as primary key in the database and which is the prerequisite to act as an Hibernate entity. This is the only attribute that is defined, see Figure 4.2.

![Figure 4.2: Core Entity](image)

4.1.2 Term Package

A term is a word that gives a name to something. This package provides components that represent terms. The core version of this package holds a component that simply has a string representing the term, see Figure 4.3.

![Figure 4.3: Core Term](image)

Two features are to be realized within the scope of this work. The first one is to additionally associate arbitrary many languages with a term and the second one is to associate arbitrary many abbreviations with it. Figure 4.4 depicts the term component’s features in a feature diagram.

4.1.3 Association Package

The association package provides components to enrich an association with meta data like time of creation or even more functionality. The core package provides an empty association...
that only knows its associator and associated object, see Figure 4.5.

**Figure 4.5: Core Association**

### 4.1.4 Concept Package

In philosophy a concept (lat. concipere = conceive) is a mental representation of something that we humans found to exist. This can be something that we can see or something more abstract, for example the entities of an ontology. In this package components are provided to gather the commonalities of concepts. The core version of it only provides a plain component that is associating an arbitrary number of term components through association components, see Figure 4.6.

**Figure 4.6: Core Concept**

In this version the component is a named object. It can be used as base for more sophisticated components.
4.1.5 Ontology Package

The ontology package provides the entities an ontology consists of. The interesting ones for this work are ontology, class, property, and instance. Those need to have a name and therefore inherit from the concepts package Concept component, see Figure 4.7.

![Core Ontology Diagram](image)

Figure 4.7: Core Ontology

4.1.6 Workflow Package

The workflow package provides means to define a workflow for arbitrary items. A workflow consists of arbitrary many states which are transitioned when certain conditions are met. These conditions are defined in connection with activities such as a vote. A condition then could be: more than 1000 votes and 51 percent voted yes. If it is met a state transition occurs and an action can be executed that might do something with the item being workflowed. Figure 4.8 depicts the workflow’s elements and their containment relationship.

Furthermore, an item to be managed by a workflow needs to keep some state. It needs to know which state it is in and the results of the state’s activities. This task is taken care of by the component WorkflowedEntity. It will be introduced more appropriately in Section 5.5.
4.1.7 Collection Package

The collection package contains Wicket panels for displaying and editing arbitrary Java collections which are the ones that implement the interface `java.util.Collection`. It provides means to edit, add, and delete arbitrary Java objects. Figure 4.9 shows what it looks like using the example of a collection of `term` components.

In this case the panel provides one HTML input element for entering a term string for every `term` component. Since it is possible to edit collections of arbitrary Java objects, there has to be a possibility to display the HTML form elements needed. This is done through the use of another edit panel which provides everything needed to edit a specific object. Thus the collection edit panel receives a factory that is capable of creating the needed edit panel which then can be displayed for every object that resides in the collection. Furthermore the collection edit panel needs to create new objects when clicking the `add` button. Also this is done through with the help of a factory.
4.2 Realization

4.2.1 Software Process

The described components can occur in different variations tailored for a specific context (which happens through the assignment of features). This requires some kind of framework which consists of reusable assets that can be used to create specific components. It needs to describe a family of components. Therefore it will be called *component framework* throughout this work.

*Domain engineering* [KE00] is a process that concentrates on providing reusable solutions for families of systems. A second process called *application engineering* reuses the component families by choosing specific component variants to build a concrete system. This process is called the *ordering* of components from the component framework. Figure 4.10 depicts the process.

![Figure 4.10: Domain and application engineering](image)

The first part in domain engineering is domain analysis. Part of it is to describe the concepts belonging to a domain in, say, UML and prescribe their features in feature diagrams. Domain design and domain implementation realize the gathered requirements. A software architecture is chosen which in this case is a layered architecture which is best practice for web applications, see next Section 4.2.2. The result of the domain engineering process is the component framework.

In application engineering a set of features is chosen to represent the requirements identified in the first step the requirements analysis. If requirements arise that are not addressable...
by the component framework there are two possibilities. The first one is to adapt to ordered components and the second one is to add the new requirements by the means of new features to the component framework. The result of the application engineering process is the specific product.

4.2.2 Architecture

An architecture is, according to [SG96], the following:

Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among these components. Such a system may in turn be used as a (composite) element in a larger system design.

The typical feature of enterprise applications is their multi-tiered architecture. The middle or server tier also consists of tiers or layers. A layer has the characteristic that it can only communicate with the layer underneath it. That way a stack of layers come into existence, each of which only communicates with the layer underneath it.

When using EJB technology, a two layered architecture results, see Figure 3.1. Since the project is using Spring, persistency becomes an additional layer. The resulting layers and their used technologies can be seen in Figure 4.11.

The typical classes to be developed in business layer is a business class and its service class also called a manager class. The business class can be something like a bank account or an ontology entity while the manager class will provide services for working with the business objects. The most basic ones are database related CRUD (copy read update delete) operations for simple
retrieval or deletion of a business object. The access of the database is the responsibility of the business layer. The class that will be developed here is a so call DAO (data access object) class. Figure 4.12 depicts these classes and their relationships using the example of the entity package. It is the base for any business object or phrased more abstractly for any object that appears in a large amount and has to be persisted.

![Diagram of classes in the business and persistence layer](image)

Figure 4.12: Classes in the business and persistence layer

The colors used are the same as in the previous Figure 4.11, which helps to identify the layer the class resides in. Every class is programmed against an interface to allow for variations of the same interface. For example the EntityDao interface can be realized with Hibernate usage or with Java collections for in memory “persistency”. This pattern is the same for term, association, and ontology package.

### 4.2.3 Component dependencies

By using the Spring container, components are ordinary Java classes. As depicted in the previous section, functionality is spread over different classes to allow for good modularization. For example a manager class uses a DAO class for data access. Both classes can evolve independently. If these are to be components, a component vendor could develop a new DAO interface with new methods that help improving the performance. A different vendor could then rely on the new interface and develop a new manager using it. This means a manager component depends on a certain incarnation of a DAO component.

To express this dependency, Java generics are used. Listing 4.1 shows its usage using the example of EntityManager and EntityDao interfaces.

```java
1 interface EntityDao {  
2      ...  
3   }
4
5 interface EntityManager < EntityDaoType extends EntityDao  
6      {  
7          ...  
8      }
9```

36
The generic type parameter `EntityDaoType` extends the interface type `EntityDao`. This means that the used DAO can be of type `EntityDao` or any subtype. Furthermore this means evolution of components will happen inside an inheritance hierarchy. An evolved manager/DAO pair is shown in Listing 4.2.

```java
class EvolvedEntityDao extends EntityDao {
    ...
}

interface EvolvedEntityManager
    <
    EntityDaoType extends EvolvedEntityDao
    >
    extends EntityManager<
    EntityDaoType
    > {
    ...
}
```

Listing 4.2: An example with evolved classes

The clear disadvantage of this technique is that a list of generic type parameter has to be maintained. If a type parameters changes in a superclass, every subclass has to be altered.

### 4.2.4 Ordering Components from the Component Framework

The activity of choosing specific components out of the component framework is called ordering of components. The component framework has a set of possible component variants. The ordering process is realized through subclassing of the component implementing classes. Thereby, all of their generic type parameters are bound so that the resulting class will be parameter free. Listing 4.3 shows the ordering of the interfaces of an `EntityManager/EntityDao` pair.

```java
interface OrderedEntityDao extends EntityDao { \empty }

interface OrderedEntityManager
    extends EntityManager
    <
    EntityDaoType extends OrderedEntityDao
    > { \empty }
```

Listing 4.3: Ordering of interfaces
By binding every generic type parameter, the configuration as a whole will be verified. Using the previous example, the question is: Is the chosen EntityDao compatible with the chosen EntityManager? If a more general DAO was chosen the Java compiler would raise an error. More general, that means this DAO is one of the superclasses of the demanded DAO. The resulting error can be interpreted as that the DAO component was not compatible with the EntityManager component. Listing 4.4 shows an example with a wrong DAO chosen.

```java
1 interface OrderedEntityDao extends MoreGeneralDao { \empty }
2
3 interface OrderedEntityManager
4   extends EntityManager
5   <
6     EntityDaoType extends OrderedEntityDao <- ERROR!
7   > { \empty }
```

Listing 4.4: Wrong DAO component chosen

That way the component framework user is warned if an illegal component combination was chosen. The instantiation can happen for the implementation classes only or additionally for the interfaces if needed. Furthermore, the subclassed component’s implementing classes are the place where features are assigned. This can happen through the help of some natively with only the means the used languages support or with the help of Reuseware. For the Java side there is also the possibility to use AspectJ. All three approaches are investigated and compared in this work.

A disadvantage of this approach is that subclasses are technically new components which potentially can have more functionality. This is problematic when using adapter classes for example. An adapter class [GHJV95] provides a certain interface which the adapted class is needed to provide. If such a class is used in the context of an ordered component, which will have its classes generic parameters bound to subclasses and therefore to interfaces that potentially have more methods, the adapter class will not work because it only supports the interface of the superclass. The adapter class had to be rewritten to support the new subclassed interface. The conceptual problem is that an ordered component is a component that binds depending component variants and features. It does not offer a new interface. At least it does not offer a new core interface. Features may provide new methods and attributes but the core interface remains as it is. The mentioned adapter would be an adapter for its core interface, not for its features. It is not possible to express this kind of semantics with the Java language directly. A possible solution would be the use of proxies. The problem with them is that they show significantly worse performance when accessing an object. Addressing this issue will be part of this work and could be addressed in further investigations.
4.2.5 Deployment in Spring IOC Container

The deployment in Spring IOC container is trivial. The instantiated components are deployed as beans (which are instances of the component implementing classes). Their depending components are set via dependency injection (DI). This is done through an XML configuration file. Listing 4.5 shows an example.

```xml
<bean id="entityDao"
    class="components.entity.impl.InstantiatedEntityDaoImpl" />
<bean id="entityManager"
    class="components.entity.impl.InstantiatedEntityManagerImpl">
    <property name="entityDao" ref="entityDao" />
</bean>
```

Listing 4.5: Component deployment in Spring IOC container

This work investigates how the deployment of beans into the IOC container can be enhanced with Reuseware concepts.

4.3 The Task

The task is to realize the specified components. This ought to happen in a component-oriented fashion so that in the end the there are components that have to be composed. Thus two distinct task are identifiable. The first one is the realization of components in the component framework and the second one is the composition of them.

The implementation of the components happens on the three tiers web, business, and persistence. Every tier has its specific implementation language. A challenge therefore is to realize the same abstractions on every tier. Languages like Java already are very sophisticated in contrast to Hibernate mapping files which mostly concentrate on supporting good concepts to define mappings but not on concept supporting good modularity. Since it is desirable to use the same kind of abstractions on every tier the missing modularity concepts have to be integrated into the language.

The composition of the components of the component framework is the second task. There will be needs for modularity that will exceed the ones faced when creating the component framework. For example Hibernate mappings provide means for mapping classes as stand alone entity in the database or just as a bunch of columns. A component composer might want to choose between these two possibilities to potentially tweak the performance of the system.

For both of these tasks Reuseware seems to be a good choice for adding the missing modularity concepts. This work evaluates if it is capable to present solution for the given challenges.
5 Reuseware Incorporation and Evaluation

There are three modularization concepts realized in this work, see Figure 5.1.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Spring</th>
<th>Hibernate</th>
<th>Wicket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>✔</td>
<td>?</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Class Reference</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Aspects</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inter-type decl.</td>
<td>✗</td>
<td>?</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Advice</td>
<td>✗</td>
<td>n.n.</td>
<td>n.n.</td>
<td>n.n.</td>
</tr>
</tbody>
</table>

✔ available ✗ not available ⚫ realized in this work n.n. not needed

? not investigated if it is needed

Figure 5.1: Modularization concepts

Inheritance is a common mechanism and it is not only supported by Java but also by Hibernate mappings files and Wicket extended HTML which are tightly coupled to the object-oriented paradigm. Nevertheless it is investigated if Reuseware can do better. Inheritance support for Spring was not investigated and it is not sure if it is needed at all. Class reference means exchangeable class references. In Java for example these can be introduced through the use of generics. In every other used technology there are no means to modularize class references. Again Reuseware will be used to add the missing concept. Aspect support consists of two parts. The first one is inter-type declarations. Technically it is very similar to multi-inheritance which means that a class can have multiple superclasses. Consequently, ordinary single-inheritance and multi-inheritance and inter-type declarations are treated as the same concept throughout this work. Again it was not investigated if the concept of inter-type declarations make sense in the Spring tier. The second one, advices, is only needed for Java. Advices add executable statements to target statements to extend their functionality. This is not needed for the other languages because they only focus on the data side.
5.1 Non-polymorphic Inheritance for Hibernate

In the component framework, the ontology package is based on the concept package by means of inheritance (see Figure 4.7). For persistency both packages define Hibernate mapping files. Figures 5.2 and 5.3 depicting the package structures and the locations of the mapping files.

Figure 5.2: Concept mapping files
Figure 5.3: Ontology mapping files

*Concept.hbm.xml* maps the two properties its respective class has, see Listing 5.1. Subclass *OntologyConcept.hbm.xml* does not add anything since its class does not define any new properties. The mapping files for the ontology entities do, see Listing 5.2 for an example of *Instance.hbm.xml*.

```
<hibernate-mapping>
  <class name="Concept">
    <id name="id" column="ID" type="long">
      <generator class="native"/>
    </id>
    <set name="associatedTerms">
      <key column="concept_id"/>
      <one-to-many class="Term"/>
    </set>
  </class>
</hibernate-mapping>
```

Listing 5.1: Concept.hbm.xml

```
<hibernate-mapping>
  <class name="Instance">
    ...
  </class>
</hibernate-mapping>
```

Listing 5.2: Instance.hbm.xml
The task at hand is to reflect the inheritance relationship as it is in Java in the Hibernate mappings. This means inherit the contents of `Concept.hbm.xml` to `OntologyConcept.hbm.xml` and from there to the different ontology entity mappings. Hibernate offers three strategies to solve this problem resulting in three different table structures all with its advantages and disadvantages:

1. Table per concrete class: Use a designated table for each concrete subclass.
2. Table per class hierarchy: Use a table for a whole class hierarchy.
3. Table per subclass: Use a table for each subclass which only holds the added attributes.

The last two strategies support polymorphism, the first one does not. Since the concept package could be the base for a lot of different use cases, its subclasses most likely are independent of each other. At least the ontology package should be stand alone in its own right. Therefore, `table per concrete class` (1) is the strategy of choice because polymorphism is not needed. This strategy uses a designated table for every class where every mapped attribute is represented by a column including the inherited ones from the superclasses. Figure 5.4 depicts three out of four resulting tables Hibernate will use to persist the objects of the ontology package. `OntologyConcept` does not have a table on its own because it is an abstract superclass.

The attributes `id` and `termAssociations` are defined by the superclass and thus are duplicated in every table.

In the component ordering process, subclasses of the component implementing classes of the component framework are made. If those are entity components they need to have a mapping file so that they can be persisted. Such a mapping file once again can be filled with inheritance mechanisms.

Listing 5.2: Instance.hbm.xml

```
<many-to-one name="ontology"
  column="ONTOMETRY_ID"
  class="Ontology"
  not-null="true"/>

<many-to-one name="clazz"
  column="CLAZZ_ID"
  class="Clazz"
  not-null="true"/>
</class>
</hibernate-mapping>
```
5.1.1 The Native Solution

The simplest solution to realize table per concrete class strategy is to write a mapping file for every concrete subclass. Since this means duplication of XML code, the Hibernate handbook [KBA+] suggests in Section 9.1.6. the use of XML entities. These are variables used to define shortcuts to some text, see the XML reference documentation [BPSM+]. An entity is declared like this ⟨ENTITY allproperties SYSTEM "allproperties.xml"⟩ in the DTD header and referenced like this &allproperties; in the body. The contents of the file allproperties.xml will be inserted at the position of the &allproperties; statement while replacing it.

This means to avoid XML code duplication properties that should be inherited have to be defined in another file and have to be included where needed. Listing 5.3 shows such an external property file for the Concept class, while Listing 5.4 depicts how it gets included into the actual mapping file.

```
1 <id name="id" column="ID" type="long">
2   <generator class="native"/>
3 </id>
4
5 <set name="associatedTerms">
6   <key column="concept_id"/>
7   <one-to-many class="Term"/>
8 </set>
```

Listing 5.3: ConceptProperties.xml

```
1 <!DOCTYPE hibernate-mapping PUBLIC
2   "-//Hibernate/Hibernate-Mapping_DTD_3.0//EN"
3   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" [ 
4   ENTITY conceptProperties SYSTEM "ConceptProperties.xml">
5 ]>
6
7 <hibernate-mapping>
8   <class name="Concept">
9     &conceptProperties;
10   </class>
```

Listing 5.4: ConceptProperties.xml
Listing 5.4: Concept.hbm.xml with XML entity usage

This example shows how to define a property file and a mapping file for the base class Concept. The task at hand is to define mapping files for every class with respect to the inheritance tree (see Figure 4.7). The first subclass of Concept is OntologyConcept. Again a property file and a mapping file is created. Because OntologyConcept inherits from Concept, the properties of the superclass also have to be included. This means two property files have to be declared as XML entities in the DTD header and referenced in the body. While the declaration can only be done in one way (in the header), there are two possibilities for referencing. The first one is to put both references in OntologyConcept.hbm.xml, the second one is to only put the reference for the OntologyConcept properties in the former file while putting the reference for the Concept properties in the property file of OntologyConcept. The latter way is shown in listings 5.5 and 5.6.

Listing 5.5: OntologyConceptProperties.xml

Listing 5.6: OntologyConcept.hbm.xml with XML entity usage

This way has the advantage of avoiding the reference for the Concept properties for every subclass of OntologyConcept because it will get included when referencing the OntologyConcept properties. The DTD entity declaration, in contrast, can not be avoided. For every superclass of a class, an entity declaration has to be made.

When ordering an ontology component, a mapping file for it is needed too. This can be done by creating a new mapping file which only contains an empty class element. The missing property mapping elements inside the class element can be included via another
use of the DTD entity. The important thing is to not forget the DTD entity definitions for
every superclass the ordered component’s implementing class possesses. In the end, such a
mapping file looks similar to the one shown in Listing 5.6. The only difference is the name of
the mapped class, which in this case is the name of the ordered component’s implementing
class.

The native solution works but it has drawbacks. If the inheritance tree is too deep, mainte-
nance becomes tedious.

5.1.2 The Reuseware Solution

To solve the task of inheriting Hibernate mapping files in Reuseware, a composition system
and its reuse extension has to be defined. In the latter case an Epackage has to be specified
which relates this extension to a specific language. Since there is no support for Hibernate
mapping files from scratch, it has to be added. This means at first an Ecore model has to be
created. For that purpose, EMF offers an XSD (XML schema definition) import function.

Hibernate mapping file definition is in DTD (document type definition) format. To use
the EMF importer, the DTD file has to be converted to a XSD file. To solve this task, a
combination of the small tools DTDinst [Ltda] and Trang [Ltdb] produce useful results. The
first one converts the DTD file to a RELAX NG file. The second one takes this newly created
file and converts it to the final XSD format. The quality of this process is that it preserves
the original DTD structure.

Now that there is an XSD file available for Hibernate mapping files, an Ecore model can be
generated through the help of EMF. To do so, a new EMF project has to be created and in
the next step the project wizards asks for a model importer where XSD schema has to be
chosen. An Ecore model and a generator model is created. Those are the base to create the
model code and the edit code which are used to parse and manipulate files corresponding to
this model. A complete guidance for doing so can be found in [emf]. After creating the code,
it has to be exported as Eclipse plugin and copied into the Eclipse plugin folder.

Now that an Ecore model was created and made available via an Eclipse plugin, it can be
used when defining reuse extensions. Its epackage id called nsURI can be found inside the
Ecore model file. It also can be changed there. For rootclass, if complete mapping files are
supposed to be valid components (fragments), DocumentRoot has to be used.

Now that the prerequisites are met, the composition system can be defined. The task at
hand is to transfer the property mapping elements of a class element of one mapping file to
the class element of another mapping file. This means that there are two roles involved. One
mapping file is the provider of property mapping elements and the other one is their receiver.
Listing 5.7 shows the resulting composition system definition.
The next step is to define the corresponding reuse extension for Hibernate mapping files. To know which Eclases from the Ecore model to reference, a look into the corresponding Ecore model file can be taken. Eclipse is able to open those files and show its contents in a tree. Figure 5.5 shows such a tree, Listing the elements a `ClassType` can possess.

```
compositionsyste m components.systemdefinition.inheritance {

fragment role Inheritance.Provider {
    static port Content;
}

fragment role Inheritance.Receiver {
    static port Class_Hook;
}

contributing association inherit {
    Inheritance.Provider.Content --> Inheritance.Receiver.Class_Hook
}
```

Listing 5.7: Inheritance composition system

The items above the ellipsis (three dots) are represented in XML through their own XML elements which in this example are types of property mappings whereas the items underneath are XML attributes of the class element. Every one of these property mapping elements should be prototype and hook at the same time to enable arbitrary inheritance relationships between mapping files. Listing 5.8 shows the resulting reuse extension.

Figure 5.5: Hibernate mapping file Ecore model
componentmodel components.systemdefinition.inheritancehibernate
implements components.systemdefinition.inheritance
epackages <http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd>
rootclass DocumentRoot {

    fragment role Inheritance_Provider {
        port Content {
            ClassType.id is prototype {}
            ClassType.compositeId is prototype {}

            ClassType.property is prototype {}
            ClassType.manyToOne is prototype {}
            ClassType.oneToOne is prototype {}
            ClassType.component is prototype {}
            ClassType.dynamicComponent is prototype {}
            ClassType.properties is prototype {}
            ...
        }
    }

    fragment role Inheritance_Receiver {
        port Class_Hook {
            ClassType.id is hook {}
            ClassType.compositeId is hook {}

            ClassType.property is hook {}
            ClassType.manyToOne is hook {}
            ClassType.oneToOne is hook {}
            ClassType.component is hook {}
            ClassType.dynamicComponent is hook {}
            ClassType.properties is hook {}
            ...
        }
    }
}

Listing 5.8: Inheritance reuse extension for Hibernate

To allow for a complete inheritance every possible property mapping element has to be declared as prototype and hook.

Now that a composition system and its reuse extension has been created, it is possible to map the inheritance hierarchy to Hibernate mapping files by the means of table per concrete class strategy. This is done in a composition program, as shown in Figure 5.6.
Figure 5.6: Inheritance mapping

When ordering an ontology component, a mapping file for it is needed too. Just like in the native solution, a skeleton mapping file containing only an empty class element has to be created, as shown in Listing 5.9.

```
<hibernate-mapping>
  <class name="InstantiatedOntologyComponent">
  </class>
</hibernate-mapping>
```

Listing 5.9: Mapping file skeleton for an ordered ontology component

A simple composition program then transfers the property mapping elements from the mapping file of the component framework to the mapping file for the ordered component.

5.1.3 Comparison and Evaluation

The native solution has a serious drawback regarding maintainability. If the inheritance hierarchy changes in terms of classes added or removed, the mapping files of every subclass have to be adjusted. In every mapping file, DTD entity definitions have to be made for every superclass a class may possess. If a new class is added or removed, these definitions have to be added or removed as well.

Furthermore, the separation of property mappings and class mapping seems artificial. The mapping file always contains a nearly empty class mapping element which only contains a
DTD entity reference. It seems more like a fast but dirty solution for a modularization problem. It is an example for a modularization problem not being addressed well enough.

The Reuseware solution in contrast seems to be natural and well fitting. The inheritance hierarchy can even be reproduced graphically in the composition programs. Therefore Reuseware is suited to model inheritance relationships in DSLs.

A point of criticism is the technical limitation of composed and original files. Composed files are put into a folder structure that is equal to the one the original file resides in and there it is put into a subfolder named composed. The fragment explorer which Reuseware provides for Eclipse displays folders with original files and composed files next to each other. This way both of them are easily accessible without navigating into another folder structure. Though an inconvenience regarding Hibernate arises when deploying mapping files. Hibernate can be given a folder containing mapping files which it parses to provide persistency support for the classes defined in the mapping files. If a folder contains mapping files that will not be composed and files that will, this folder can not be used for Hibernate. Therefore the mapping files have to be specified individually. Nonetheless this is not an issue within this project since every used mapping file will be a composed one due to the component ordering process.

5.2 Exchangeable Class References for Hibernate

The association component connects two components with each other. One component plays the role of the associator and one of the associated. For example, the concept component uses the association component to associate arbitrary many term components. The class diagram in Figure 4.6 depicts this relationship.

An association component is persisted in Hibernate as an entity which means that it is stored in its own table. To achieve this, the component implementing class has to be mapped as an Hibernate entity using the class mapping element. Listing 5.10 shows what the resulting mapping looks like.

```xml
<class name="Association">
  <many-to-one name="associator"
               class="Associator"/>
  <many-to-one name="associated"
               class="Associated"/>
</class>
```
This is the mapping file residing in the component framework. There are two property mappings of the type many-to-one which reference the respective associator and associated classes. These can be arbitrary ones since an association component is a very general one. The problem is that Hibernate needs to know a certain class so that it knows from which table to look it up.

When ordering an association component, a mapping file for Hibernate persistency will be needed.

The association component will be used in a broad spectrum of use cases. Once again, the different uses of the association components will most likely have nothing in common with each other. One could be used to associate a bank account to a customer and the other one to associate a name to an ontology entity. Therefore, it is desired to store different usages of association components in different tables.

Furthermore the associator component itself needs to know about the association class. Therefore, the associators mapping file will have a property that refers to one or a collection of associations. Listing 5.11 shows the mapping file of the concept component for an example of a component associating arbitrary many association components.

```
<class name="Concept">
  <set name="associatedTerms">
    <key/>
    <one-to-many class="Association"/>
  </set>
</class>
```

Listing 5.11: Mapping file for concept component

The one-to-many elements class attribute references the association component implementing class. Since every kind of association will have its own table and therefore its own implementation class, this reference will be a different one for every ordered component of this type.

### 5.2.1 The Native Solution

When ordering the association component from the component framework a subclass of the component implementing class will be made. Furthermore for persistency with Hibernate, a
mapping file for the ordered component will be needed. The component framework offers the needed file. The task at hand is to create a dedicated mapping file for the ordered component.

The problem to take a look at first is the one of both many-to-one associations referring to arbitrary classes which are not known beforehand. For Hibernate to map an ordinary many-to-one association, it needs to know the concrete class beforehand. Nevertheless Hibernate offers the any mapping element. Its purpose is to associate classes mapped in arbitrary tables. The use of it is demonstrated in Listing 5.12.

```xml
<class name="Association">

  <any
    name="associator"
    id-type="idtype_of_associator_class">
    <meta-value value="ASSOCIATOR" class="associator_class"/>
    <column name="value_column_name"/>
    <column name="id_column_name"/>
  </any>

  <any
    name="associated"
    id-type="idtype_of_associated_class">
    <meta-value value="ASSOCIATED" class="associated_class"/>
    <column name="value_column_name"/>
    <column name="id_column_name"/>
  </any>

</class>
```

Listing 5.12: Mapping file for association component with the use of the any element

The main problem by using the any mapping element is that the associator/associated class is still needed. The purpose in using this element was to avoid the specification of a concrete class. Thus ordinary many-to-one elements can be used, which in fact are the better choice anyway.

By using the ordinary many-to-one, we need to specify the concrete class associated. Thus there is no other choice than to make a copy of the mapping file coming from the component framework and adjust the class reference by hand. If the any tag allowed to avoid the specification of a concrete class, DTD entities could have been used. That way the property mapping elements could have been reused.

The second challenge is to adjust the associator mapping file for ordered associator components. For example, the concept component associates arbitrary many term components through the use of association components. The used mapping element is shown in 5.11.
The \textit{one-to-many} element has a class reference whose class is unknown beforehand to the component framework developer. In Hibernate there are no means of making such a class reference exchangeable. Thus the only solution is to make a copy of the mapping file and adjust the class reference by hand.

\subsection*{5.2.2 The Reuseware Solution}

For Reuseware to solve this task, at first a composition system and its reuse extension for Hibernate has to be defined. The former Section 5.1.2 describes how to add an Ecore model for Hibernate mapping files to enable Reuseware support for those.

When designing the composition system two fragment roles can be found. One fragment provides a class name the other one receives it. Listing 5.13 shows the resulting composition system definition.

```plaintext
compositionsystem components.systemdefinition.reference {

fragment role ClassReference_Provider {
  static port Class;
}

fragment role ClassReference_Receiver {
  dynamic port Class_Hook;
}

contributing association weave {
  ClassReference_Provider.Class --> ClassReference_Receiver.Class_Hook
}
}
```

Listing 5.13: Reference composition system for exchangeable class references

The fragment role \textit{ClassReference}_\textit{Receiver} contains a dynamic port which means that there can be arbitrary many of them. The association components mapping file (see Listing 5.10) shows a use case as that there are two class references that need to receive a class name.

The specific Hibernate reuse extension is shown in Listing 5.14.

```plaintext
componentmodel components.systemdefinition.reference.hibernate
implements components.systemdefinition..reference
epackages <http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd>
rootclass DocumentRoot {

fragment role ClassReference_Provider {
  port Class {
    ClassType is value prototype {
```
Listing 5.14: Reference reuse extension for Hibernate

The port `Class` of fragment role `ClassReference_Receiver` uses the `ClassType` as value prototype. The contained definition of a value expression (value expr) specifies which value will be used as the prototype, in this case, the value of the `ClassType` attribute name. The value hook on the other side complements to this. In this case, a different metaclass than `ClassType`, namely `OneToManyType` class attribute, is used. This is the reason to use a value prototype/value hook pair because different kinds of metaclasses serve as prototype respectively hook. Therefore, not complete Tags, but only names are composed. For the same reason a point expression (point expr) is defined in both ports which provide the same name. Normally Reuseware would identify the prototype/hooks pairs that should belong together by matching metaclasses. In this case different ones are used so Reuseware needs a point expression to know which prototype/hooks belong together.

For the reason that the component designer may want to decide which class reference should be replaceable and which one not, a naming convention is introduced. If the name of a referenced class ends with `_Hook` then it is supposed to be replaceable. The OCL expression which is followed by the `if` statement describes this. Furthermore a dynamic port needs a port expression (port expr) which gives a name to the port. This is needed because there are arbitrary many ports which would otherwise all have the same name. The OCL expression describing how this name should look like states that the `_Hook` part of the name is replaced with nothing thereby eliminating it. The resulting mapping file with adapted class references for the association component can be seen in Listing 5.15.
On component ordering, the available hooks are bound to the names of appropriate classes. In the following composition program example, every needed component for the ontology entity Clazz is shown. For every ordered component, a mapping file is needed. As described in the former Section 5.1.2 a skeleton mapping file containing only an empty class element is created for every ordered component. With the help of the inheritance composition system, the missing property mapping elements are inserted in the empty class element. In the next step, the newly created composition system is used to assign class references. Figure 5.7 shows this composition program.

Figure 5.7: Mappings belonging to clazz component

The ClazzTermAssociation.hbm.xml fragment is the skeleton mapping file for the association component associating a term component. It is a target fragment. It inherits every property mapping element made by fragment Association.hbm.xml which is the mapping file provided by the component framework. Also this is the place where the class references Associator_Hook and Associated_Hook are bound. They are bound to the class name specified in the skeleton mapping files Clazz.hbm.xml and ClazzTerm.hbm.xml. Also, both of them inherit from the respective mapping files coming from the component framework. Thereby, the fragment Clazz.hbm.xml2 additionally receives the name of the ordered association component’s
mapping file ClazzTermAssociation.hbm.xml.

5.2.3 Comparison and Evaluation

The native solution is to do everything by hand. Make a copy of the mapping file coming from the component framework and adjust its class name and references. The Reuseware solution in contrast demands to create a skeleton file which is filled by the inheritance composition system and whose references are bound by the newly created reference composition system.

The advantage of the Reuseware solution is that a component assembler ordering a mapping file does not need to know the inner workings of it. It is enough to compose fragments in a composition program. The rest is done by Reuseware. Nonetheless the user needs to know the basic pattern of filling a skeleton file with content coming from the component framework.

5.3 Using Hibernate Entity as Component Definitions

In the case of the association component, the component assembler might want to model the associated component as Hibernate component in contrast to a Hibernate entity (see Section 3.3 on what a Hibernate component is). This is done for reasons of performance. A Hibernate component is persisted in the same table as the Hibernate entity that defines this Hibernate component. When there is no independent access on the associated component needed but only in context of the association this is a possibility.

In Java there is no difference between a class mapped as Hibernate entity or Hibernate component. The needed property mapping elements are the same. Actually the definition of a Hibernate entity and a Hibernate component does not differ too much. There are two elements an entity definition (class mapping element) can possess that a component (component mapping element) can not:

- idbag
- properties

When those two elements can be avoided, it is desirable to have one mapping file with a class element that defines which properties are to be mapped and which then can be reused for Hibernate component definitions.

A mapping file for the association component which maps the associated component as Hibernate component is shown in Listing 5.16.

```
1 <class name="Association">
2 </class>
3     <many-to-one
```
An example of a component mapped as a Hibernate component is the `PropertyRestriction` component. It is used to restrict a property definition made by a `PropertyDefinition` component in the scope of a `Clazz` component. This means a `Clazz` component associates arbitrary many `PropertyRestrictions` through `Association` components. Figure 5.8 depicts this.

5.3.1 Native Solution

The native solution to this problem is using DTD entities as described in Section 5.1.1. That way it is possible to externalize arbitrary XML code and include it later on by using of DTD entities.

Since the native solution for non-polymorphic inheritance is solved through DTD entities, anyway most of the mapping files in the component framework are already split into two files. The ones that are not will be split right now so that every set of property mapping elements becomes usable for both entity and component definitions. Once done, the file defining the property mappings of a component can be inserted in the place of Hibernate component definitions. See Listing 5.17 for an example.

```
<!DOCTYPE hibernate-mapping PUBLIC
```
<hibernate-mapping>
  <class name="Association">
    <many-to-one
      name="associator"
      class="AssociatorClass"/>
    <component
      name="associated"
      class="AssociatedClass">
      &PropertyRestrictionProperties;
    </component>
  </class>
</hibernate-mapping>

Listing 5.17: AssociationAssociatedAsComponent.hbm.xml with a DTD entity

5.3.2 Reuseware Solution

To solve this requirement in Reuseware, both of the former defined composition systems can be reused. There are two tasks to solve:

- Replace the reference to the class
- Put the property mappings inside the component element

The first one can be solved by extending the referencehibernate.rex (Listing 5.14) reuse extension. The only thing that has to be done is to add a new type hook to the Class_Hook port of the fragment role ClassReference_Receiver. Listing 5.18 shows this new hook.

```java
fragment role ClassReference_Receiver {
  ...
  port Class_Hook {
    ComponentType.class is value hook if
    $self.class.oclAsType(String).endsWith(’_Hook’) {
      port expr = $self.class.oclAsType(String).replace(’_Hook’, ’’)$
      point expr = $’class’$
```
This definition is of the same style as the definition made in former section 5.2.2 except for
the used metaclass which in this case is ComponentType. Since the port Class_Hook is a
dynamic one every occurrence of a ComponentType which class attribute value ends with the
string _Hook will be represented as a port on its own.

The second task is to put the property mappings of an Hibernate entity definition into the
Hibernate component definition. In this case, the inheritance composition system (Listings
5.7 and 5.8) can be reused. The reuse extension of this system already describes how property
mappings from one entity definition are transferred to another entity definition. The task of
transferring property mappings to component definitions is almost the same except for the
two kinds of property mappings not supported.

The first thing needed is a new port in fragment role Inheritance_Receiver. Since as for now
there is only a static port for an entity definition. Component definitions can occur arbitrary
many times. Therefore a dynamic port is needed and its corresponding association definition.
Listing 5.19 shows both.

```java
compositionsysten components.systemdefinition.inheritance { 
  fragment role Inheritance_Provider { 
    static port Content; 
  } 
  fragment role Inheritance_Receiver { 
    static port Class_Hook; 
    dynamic port Prototype_Hook; 
  } 
  contributing association inherit { 
    Inheritance_Provider.Content --> Inheritance_Receiver.Class_Hook 
  } 
  contributing association transfer { 
    Inheritance_Provider.Content --> Inheritance_Receiver.Prototype_Hook 
  } 
}
```

Listing 5.19: Inheritance composition system with added Hibernate component support
The next thing to do is to adjust the reuse extension. The new defined port Prototpe_Hook needs to be specified with every supported property mapping as hook. Listing 5.20 shows the definition of the new port.

```plaintext
...  
fragment role Inheritance_Receiver { 
  port Prototype_Hook { 
    ComponentType.property is hook if 
      $self.class.oclAsType(String).endsWith('Hook')$ { 
        port expr = 
          $'ComponentHook'.concat(self.class.oclAsType(String)).replace('Hook', '')$ 
      } 
    ComponentType.manyToOne is hook if 
      $self.class.oclAsType(String).endsWith('Hook')$ { 
        port expr = 
          $'ComponentHook'.concat(self.class.oclAsType(String)).replace('Hook', '')$ 
      } 
    ComponentType.oneToOne is hook if 
      $self.class.oclAsType(String).endsWith('Hook')$ { 
        port expr = 
          $'ComponentHook'.concat(self.class.oclAsType(String)).replace('Hook', '')$ 
      } 
  } 
...  
```

Listing 5.20: Inheritance reuse extension with added Hibernate component support

Every hook definition contains an if expression to constrain the occurrence of a component definition as Prototype_Hook port. Only if the value of the class attribute of the component definition ends with _Hook then the contained property mapping is a hook. To affect the occurrence of the port this condition has to be set for every hook definition. If Reuseware does not find any hook definition that fulfills the condition, then the component definition will not be shown as a Prototype_Hook port in composition programs.

Furthermore there is a port expression (port expr) which gives the port a name. It also has to be stated for every hook definition, because it is not known beforehand which of the hooks actually exist. There can be only one property mapping as hook in a component definition and if this one would not possess the port expression Reuseware would not set a custom name for the port.

Figure 5.9 shows a part of the resulting composition diagram.
5.3.3 Comparison and Evaluation

The native solution uses the same technique as for inheritance (see Section 5.1.1). This is an advantage because two different modularization problems can be solved with the same technique. It saves the time to learn something new and modularization efforts do not have to be done redundant. Reuseware is able to offer the same advantages. It is possible to use the same composition system as for inheritance (see Section 5.1.2). This at least saves one more port on the fragment and instead of maintaining two composition systems it is enough to maintain one.

Another advantage is that even if the two not allowed Hibernate mapping elements idbag and properties are used, Reuseware abandons them when trying to insert them into a Hibernate component definition. The native solution has not a similar functionality.

5.4 Aspects for Java, Hibernate, and Wicket-HTML

The main goal of this work was to realize features. Features can be realized through components. In the end, the task is to compose core components which hold the essential functionality with feature components which add functionality. Thereby every class can act as a core or as a feature component. One time the term component is a core and its feature is the abbreviation component. Another time the abbreviation component is a core which will be extended through an precedence component, which in this case acts as a feature.

Features can be realized through the use of concepts of aspect-orientation. Inter-type declarations can add feature related attributes and methods while advices can introduce feature code into core code. Here, the term feature is used for the inter-type declaration concept.

Every file of the term component package belongs to one of the three concerns business,
Persistency, or web. Furthermore there are three different file types which are for Java source code files, Hibernate mapping files, and Wicket extended HTML files. Figure 5.10 depicts the package structure with its different file types and shows which part of it belongs to which layer.

Furthermore the term component has two optional features language and abbreviation which add language support respectively abbreviation support. These are optional which means that they can be selected in any combination or left out. Both of the features are represented by the same package structure and the same kind of files. Figures 5.11 and 5.12 shows the package structure.
CoreAbbreviation is the interface for one abbreviation object. CoreAbbreviations is one for a collection of abbreviation objects. Furthermore for both there is a Hibernate mapping file and a Wicket edit panel. The same is true for the language feature. Additional manager and dao classes are missing because of the use of Hibernate. It is sufficient to just adapt the Hibernate mapping file by including the new properties the term component implementing class will have.

The task of assigning features is to combine the core package with its feature packages, consisting of Java source code files, Hibernate mapping files, and Wicket extended HTML files.

Firstly Java classes consist of attributes and methods which have to be combined with core classes. For example class CoreAbbreviationsImpl defines an attribute abbreviations which is a java.util.Set holding arbitrary many CoreAbbreviationImpl instances. Of course it also provides the two standard accessor methods, see Listing 5.21.

```java
public class CoreAbbreviationsImpl implements CoreAbbreviations {

    private Set<CoreAbbreviation> abbreviations =
    new HashSet<CoreAbbreviation>();

    @Override
    public Set<CoreAbbreviation> getAbbreviations() {
        return this.abbreviations;
    }

    @Override
    public void setAbbreviations(Set<CoreAbbreviation> abbreviations) {
        this.abbreviations = abbreviations;
    }
}
```

Listing 5.21: Feature implementing class abbreviation

The whole content of this class has to be somehow combined with the core term class so that it gets assigned the feature.

Additionally there are situations where feature code has to be inserted into core code. For example in Wicket edit panels. There is an edit panel for the core SolidCoreTermEdit (see Figure 5.10) and there is one for both of the features SolidCoreAbbreviationsEdit and SolidCoreLocaleEdit (see figures 5.11 and 5.12). Since these classes are Wicket edit panels they consist of Wicket components arranged in a Wicket component tree. The task is to add the Wicket components of the features to the Wicket component tree of the core. This happens through a call of Wickets add method. Listing 5.22 shows this code by the example of the
The edit panel uses a collection edit panel which is created by a factory `CollectionEditFactory` assigned to the attribute `abbreviationsEditFactory`. The method `addAbbreviationComponents` creates an instance of the collection edit panel and adds it to the Wicket component tree. Also interesting is the use of Wicket’s `PropertyModel` instance which is responsible for accessing the needed data. In Wicket, a model is responsible for getting and setting the data a Wicket component works with. In the case of `PropertyModel` it happens through Java reflection. The second parameter with value "abbreviations" denotes the name of the property or attribute. It is used as base for generating the standard accessor method names `getAbbreviations` and `setAbbreviations`. The first parameter is the object on which the data is to be accessed. Here another model is given which means that at first the object is extracted from the given model before accessing its property. This way the abbreviations property can reside on any object which is what is needed. The next statement is for creating the Wicket panel for editing the collection of abbreviation objects. The creation of it is delegated to a factory. The last statements adds the collection edit panel to the Wicket component tree.

The second task is to combine the HTML files of the features and the core. An HTML file for a feature is depicted in Listing 5.23.
There is a `wicket:panel` element which holds the elements belonging to the Wicket edit panel class. The content of it has to be combined with the content of the core term HTML file which has a similar buildup, see Listing 5.24.

The elements of Wicket element `wicket:panel` of the feature HTML file have to be introduced into the same element of the core HTML file.

The last task is to combine the Hibernate mapping files of the features with those of the core. The only thing that has to be done is to copy the Hibernate property definitions. A solution for this was already described in Section 5.1. It can be reused without the need of any additions.

Feature assignment itself takes place in the component ordering process. When ordering a component its implementing class becomes the superclass of the ordered component implementing class. Every feature assignment will happen on it.
5.4.1 Native Solution

As explained earlier, there are two challenges to be solved for the Java side. Contribute to the core the attributes and methods the feature introduces and extend core methods with feature code where needed.

The first one is solved by giving the core a list of feature implementations which is consistent with inter-type declarations. Then the core knows which features it possesses and can easily access them. Listing 5.25 shows the interface.

```java
public interface Feature {
    // Feature role methods
    public String getFeatureName();
    public void setFeatureName(String featureName);

    // Core role methods
    public Object getFeature(String featureName);
    public void registerFeature(String featureName, Feature feature);
}
```

Listing 5.25: Interface for adding features to a core

Since a component can be a feature or a core getting a feature assigned, every component implementing class provides means for both cases. The core is able to add a feature through `registerFeature` and retrieve a certain feature by its name through `getFeature`. The feature gets assigned its name it has in the context of a core. Features are now represented through its own classes and combined with a core through object composition.

When ordering a component the needed features just have to be added. This is to be happen in the constructor. Listing 5.26 shows the ordering of a term component with added features `abbreviation` and `language`.

```java
public class InstanceTerm extends CoreTermImpl {
    public InstanceTerm() {
        this.registerFeature("abbreviations", new OrderedAbbreviations);
        this.registerFeature("languages", new OrderedLocales);
    }
}
```

Listing 5.26: Interface for adding features to a core

Since a feature can also be a core to which features can be added, both features added in the example are also subclasses of the original component implementing classes of the component.
framework. Both potentially have their own features assigned, for example one assigning a precedence to an abbreviation determining its importance.

To realize the extension of core methods an observer-like pattern [GHJV95] is used. The idea is that methods which should be extended expose their extension points by calling a special method which in turn calls feature code. To differentiate between extension points, the special method needs to be given some kind of identifier. This identifier is commonly called an event. Expressed with this term, extensible methods send events at extension points. An event is represented through a class and can encapsulate additional data which the event receiver might need. Now features can register to specific events to get called when the event happens. For this there is a dedicated event handling method which receives the event class as a parameter. The event class can encapsulate further data which might be needed by the receiver. Diagram 5.13 depicts the communication process.

Method `fireEvent(InitEvent)` is called somewhere in the core code. Thereupon the core looks up the registered features for event `InitEvent` through a call of `getListeningFeatures(InitEvent)`. Afterwards every registered feature’s dedicated event handling method is called which in both cases have the name `onInitEvent(InitEvent)`.

To be more variable those event handling methods should not be defined on the feature itself. There might be different possibilities to realize the same requirement or there might be functionality not really needed. For example there might be an abbreviation lookup mechanism which is able to look up standard abbreviations for a given term in some database. This could be added to the term component creation process by the means of looking up a list of possible abbreviations for a newly created term while adding this list to it. There might be
configurations where this feature is wanted and others where not. For this reason it is better to encapsulate those event handlers in distinct classes. The class doing so in this project is called Advisor. Its name refers to the term advice from aspect-oriented programming. An Advisor has several advice respectively several event handling methods. Listing 5.27 shows the interface.

```java
public interface Advisor
  <FeatureType, CoreType> {

  public void attachAdvisors(Feature core);

  public FeatureType getFeature();
  public void setFeature(FeatureType feature);

  public CoreType getCore();
  public void setCore(CoreType core);
}
```

Listing 5.27: Advisor interface

The advisor has access to the feature it belongs to and to the core it advices. Method attachAdvisors is a template method [GHJV95] which is used to attach every event handling method to an event of the core. Event handling methods are added to a class as many as needed. Later on they will get called through Java reflection mechanisms.

In the end a feature has a list containing every needed advisor. The little difference now is that not features register to certain events but their Advisor objects. Listing 5.28 shows the corresponding methods also belonging to interface Feature.

```java
public interface Feature {

  // Feature role methods
  public void addAdvisor(Advisor<Feature, ?> advisor);
  public void registerAtCore(Feature core);

  ...

  // Core role methods
  public void fireEvent(Event event, Point point);
  public void attachAdvisor(
    Class<? extends Event> event,
    Class<? extends Point> point,
    Advisor<?, ?> adviser,
    String methodName,
    int precedence
  );
```
Method \textit{addAdvisor} is for adding every needed advisor to a feature. When the feature is added to a core its \textit{registerAtCore} method is called to attach its advisors to the intended events. It goes through the features list of advisors and calls their \textit{attachAdvisors} method which in turn uses method \textit{attachAdvisor} to attach a single advisor. The event consists of two classes, \textit{Event} and \textit{Point}. The first one gives a name to the event and the latter one defines a specific point during that event, for example \textit{before} or \textit{after}. The parameter \textit{methodName} is the name of the event handling method as being defined on the advisor which will be called by the use of Java reflection. The last parameter \textit{precedence} defines some precedence which is used if there is more than one advisor registered for an event to determine an order.

An \textit{Advisor} class for adding the Wicket components of a feature edit panel into the component tree of a core edit panel may look like in Listing 5.29.

```
public class AddComponentsAdvisor

    < FeatureType extends WebMarkupContainer, 
    CoreType extends WebMarkupContainer

    extends AbstractAdvisorImpl<FeatureType, CoreType>

    implements Advisor<FeatureType, CoreType> {

    @Override
    public void attachListenersToCore(Feature core) {
        core.addListener(
            AddComponentsEvent.class,
            After.class,
            this,
            "afterAddComponents",
            1
        );
    }

    public void afterAddComponents(AddComponentsEvent event) {
        this.getCore().add(this.getFeature());
    }

```
Listing 5.29: An advisor for adding feature Wicket components into the Wicket component tree of a core

This *Advisor* class is general enough to be used with every Wicket panel which is achieved through class *WebMarkupContainer* from which for example *FormComponentPanel* inherits. In the component ordering process it can be used when adding the ability to edit the features *abbreviation* and *language* in the context of a term edit panel. Their respective edit panels *SolidCoreAbbreviationsEdit* and *SolidCoreLocalesEdit* both are features related to the core *SolidCoreTermEdit*. Both of them get the *AddComponentsAdvisor* assigned. See Listing 5.30 for an example.

```java
class InstanceTermAbbreviationsEdit extends SolidCoreAbbreviationsEdit {
    // Constructor here

    @Override
    public void addAdvisors () {
        this.addAdvisor(new AddComponentsAdvisor ());
    }
}
```

Listing 5.30: Feature edit panel gets assigned an advisor

Template method *addAdvisors* is overridden to add the needed advisor. Later on it is called when the feature is registered thereby attaching the advisors event handling methods to events of the core. The only thing left to do is to add both feature edit panels to the term edit panel. Again this is done in the constructor. See Listing 5.31 for how it is done.

```java
class InstanceTermEdit extends SolidCoreTermEdit {

    public InstanceTermEdit(String id , IModel model) {
        super(id , model);

        this.registerFeature("abbreviation",
                new InstanceTermAbbreviationsEdit("abbreviations",
                        FeatureModel(this , "abbreviations")));
        this.registerFeature("locales",
                new InstanceTermLocalesEdit("locales",
                        FeatureModel(this , "locales")));
    }
}
```

69
The Wicket model `FeatureModel` that is used as parameter for the feature edit panels is an adopted one. When needed, it looks up the needed feature object from the collection of the core.

The next step is to realize features for Wicket extended HTML files. The Wicket components defined in the features edit panel have to be referred in the HTML code. For reusing HTML files, Wicket provides an inheritance mechanism. It works the following way. Inside the `wicket:panel` element, a `wicket:child` element is defined. It denotes the position a subclass contributes HTML elements. Listing 5.32 shows how it looks like at the example of the HTML code for editing a term component.

```xml
<wicket:panel>
  <input wicket:id="term" type="text"/>
  <wicket:child/>
</wicket:panel>
```

Listing 5.32: SolidCoreTermEdit.html

An HTML file of a subclass then can extend the HTML file of its superclass by using `wicket:extend` elements instead of `wicket:panel` elements. This mechanism directly reflects the inheritance structure of the Wicket panels and thus its their Java classes. For the reason that features are added differently than with inheritance it is not usable for the problem at hand. Nevertheless it can be used to provide an HTML file for the ordered Wicket panel. That way the HTML elements defined in the core HTML file are transfered into the final HTML of the ordered component which then has to be enriched with the HTML elements for the features by hand. Listing 5.33 shows the resulting file for an ordered term component which uses both of the features.

```xml
<wicket:extend>
  <div wicket:id="abbreviations"/>
  <div wicket:id="locales"/>
</wicket:extend>
```

Listing 5.33: Wicket HTML inheritance

The native solution is intrusive. Events have to be inserted into methods and classes have to implement the `Feature` interface. Component ordering is extensive too. Methods of superclasses have to be overridden and in the case of Wicket panels a specialized class to support features (`FeatureModel`) has to be used. This is knowledge a component assembler needs to have to be able to order components.
5.4.2 Reuseware Solution

Again there are the two challenges to be solved for the Java tier. For combining attributes and methods of features with cores, the inheritance composition system can be reused. A reuse extension for Java has to be created. It will be able to add arbitrary many code to a class resulting in multi-inheritance which can be used for inter-type declarations. Listing 5.34 shows what it looks like.

```java
componentmodel components.systemdefinition.inheritancejava
implements components.systemdefinition.inheritance
epackages <http://www.emftext.org/java>
rootclass java::containers::CompilationUnit {

fragment role Inheritance_Provider {
    port Content {
        java::classifiers::Class::implements is prototype {}
        java::members::Member is prototype if
        $not self.oclIsTypeOf(java::members::Constructor)
        and eContainer().eContainer().
        oclIsTypeOf(java::containers::CompilationUnit)$ {}
    }
}

fragment role Inheritance_Receiver {
    port Class_Hook {
        java::containers::CompilationUnit.namespaces is value hook {
            point expr = '$package$
            list idx expr = $self.namespaces->size()$
        }
        java::classifiers::Class::implements is hook {}
        java::classifiers::Class::members is hook {}
    }
}
```

Listing 5.34: Inheritance reuse extension for Java

There are three things to be explained. The first one is the value hook for CompilationUnit.namespaces defined in port Class_Hook. It solves the problem that appears when the composed Java files are moved to the subdirectory for composed files which name is by default composed. In every Java compilation unit a namespace is defined that reflects the directory structure. If the composed file is moved to a subdirectory without adopting the namespace then it becomes invalid. The definition adds a value hook which can be set in the composition diagram for every Java fragment. It appears in the Eclipse property view and it needs to be assigned the name of the subdirectory. Reuseware then appends the given value to the end
of the namespace definition which is achieved through using the list idx expression.

The second one is that the interfaces a class may implement are copied. This makes sense since programming against an interface is used in the project. It means that every feature implementation will refer to an interface it implements. It is desired to let the core refer to the supported feature interfaces to know which features are assigned.

The second one is that constructors will not be copied. This is consistent with Java inheritance behavior. Code that is executed in the constructor of a feature has to be woven into the core constructor. For this reason a new composition system aspect is defined. The idea is to use every method of a class as advice code that can be woven before or after any method of another class. This means there are arbitrary many advice methods and target or core methods. Listing 5.35 shows what the composition system looks like.

```java
composition system components.systemdefinition.aspect {
    fragment role Core {
        dynamic port Method;
    }

    fragment role Aspect {
        dynamic port Advice;
    }

    contributing association weave {
        Aspect.Advice --> Core.Method
    }
}
```

Listing 5.35: Composition system definition for aspects

Both of the ports are dynamic because there are arbitrary many advice and core methods. The next step is to define a reuse extension for Java. Both ports need to expose every existent method of a class. Port Advice provides the code that can be inserted at the beginning or the end of the target port Method. This means the latter port has to offer two hooks, one for prepending and one for appending code. Listing 5.36 shows the resulting file.

```java
component model components.systemdefinition.aspectjava

implements components.systemdefinition.aspect

epackages <http://www.emftext.org/java>

rootclass java::containers::CompilationUnit {

    fragment role Core {
        port Method {
            java::statements::StatementListContainer.statements is hook {
```
Listing 5.36: Reuse extension of aspect composition system for Java

To denote the statements encapsulated by a method, the eClass `StatementListContainer` is used. It is the superclass for every method which means that every method is also a `StatementListContainer`. For the name of the port the eClass `NamedElement` is used. Every method is also a `NamedElement`. To realize the before and after semantics the mode expression is used. It actually determines which one of the four primitive composition operators is used.

Now, the abbreviation feature can be assigned to term components. One part in doing so is to add the features attributes to the core which is a term component. This can be done with the inheritance composition system like shown in Section 5.1.2. Since the newly defined aspect composition system is not needed a demonstration how it is done it is left out. The other part is to assign the feature extension to the Wicket edit panel of the term component. This task needs the use of the newly defined composition system. Figure 5.14 demonstrates how it is done.

There are two connections. One is from the inheritance composition system and is there to add the needed attributes and methods to the core edit panel. The other one is from the new aspect composition system and it prepends the contents of method `addAbbreviationComponents` to the constructor of the core edit panel. That way the needed Wicket edit components are added to the Wicket component tree the term edit panel.
When contributing feature code to core code, the component assembler needs to know which feature method provides code for which code method. So still technical knowledge of the inner workings of the component framework is needed.

The next step is to introduce feature support for Wicket extended HTML files. To do so a composition system with a reuse extension for HTML is needed. For this purpose, the inheritance composition system presented in Section 5.1.2 can be reused. A further reuse extension for a Wicket extended HTML language is needed. Of course there is no support for such a language from scratch thus at first an Ecore model has to be generated. How that is done is described in Section 5.1.2.

The problem in this case is that there is no valid DTD or XSD defining the format. There is a DTD which is basically a copy of XHTML-strict. The only addition is two HTML attributes that can appear on every HTML element, among them wicket:id. When converting this file to a XSD, the conversion process reports errors due to the attributes containing the namespace wicket which is undefined for the XSD. The addition of the two Wicket attributes with their undefined namespace are not valid. Thus there is not proper DTD or XSD that defined the Wicket extended HTML elements.

Nonetheless to be able to use Reuseware with Wicket extended HTML, a XSD has to be created. There are instructions in [xht] but they are not trivial. Several steps have to be taken which is why a creation of a XSD defining Wicket extended HTML was to time-consuming.

### 5.4.3 AspectJ Solution

Having found a native solution and one with Reuseware it may be worth to also investigate a solution achieved with AspectJ. Invasive software composition and thus its realization Reuseware claims that every new programming style can be modeled as different variants of invasive software composition techniques. So the question is, which realization of aspect-
orientation is working out better? Can Reuseware replace AspectJ?

Again the two tasks for the Java side are to combine feature attributes and methods and the
weaving of feature code into the core. The first one can be solved with inter-type declarations.
They have to be made on an interface or a class. Since the aspect represents a feature which
should be assignable to any core class, it should be parameterizable so that a target class can
be specified by binding the parameter. To do so AspectJ supports generic type parameters.
Listing 5.37 shows the aspect for the *abbreviation* feature.

```java
public abstract aspect CoreAbbreviationsAsp<Component> { 
    declare parents: Component implements CoreAbbreviations;
    private Set<CoreAbbreviation> CoreAbbreviations.abbreviations;
    public Set<CoreAbbreviation> CoreAbbreviations.getAbbreviations() { 
        return this.abbreviations;
    }
    public void CoreAbbreviations.setAbbreviations( 
        Set<CoreAbbreviation> abbreviations) { 
        this.abbreviations = abbreviations;
    }
}
```

Listing 5.37: Inter-type declarations for adding feature properties

The aspect is abstract because of its generic parameter. This is intended because the feature
should not be assigned to any class by default. Furthermore inter-type declarations can not
be made on generic parameters directly. An interface has to be declared that the generic
parameter will implement. The statement *declare parents* does do so. Afterwards a default
implementation for this interface can be given.

The assignment of the *abbreviation* feature to the term component happens when ordering
a component. To assign the feature a subclass of the abstract aspect has to be made while
binding the generic type parameter. Listing 5.38 shows the use of the *abbreviation* aspect.

```java
public aspect ClazzTermAbbrevationsAsp 
    extends CoreAbbreviationsAsp<ClazzTerm> {
}
```

Listing 5.38: Use of the abbreviation aspect in component order phase

The second task of weaving feature code into core classes again is demonstrated using the
example of feature Wicket components that need to be integrated into the Wicket component tree of a core Wicket panel using the example of abbreviation feature. Listing 5.39 shows the solution.

```java
public abstract aspect SolidCoreAbbreviationsEditAsp
  <EditComponent extends FormComponentPanel> {

  interface SolidCoreAbbreviationsEdit {
    public CollectionEditFactory getAbbreviationsEditFactory();
    public void setAbbreviationsEditFactory(
      CollectionEditFactory abbreviationsEditFactory);
  }

  declare parents: EditComponent
  implements SolidCoreAbbreviationsEdit;

  private CollectionEditFactory
    SolidCoreAbbreviationsEdit.abbreviationsEditFactory;

  // default implementation for accessor methods
  public Factory<CoreAbbreviation>
    SolidCoreAbbreviationsEdit.getAbbreviationsEditFactory() {
    return this.abbreviationsEditFactory;
  }

  // the advice
  after (EditComponent editComponent):
    this(editComponent) && execution(EditComponent.new(String, IModel)) {
      IModel model =
      new PropertyModel(editComponent.getModel(), "abbreviations");

      FormComponentPanel abbreviationsEdit =
        ((SolidCoreAbbreviationsEdit) editComponent).
      getAbbreviationsEditFactory().
      createInstance("abbreviationsEdit", model);

      editComponent.add(abbreviationsEdit);
    }

Listing 5.39: Advice to add feature Wicket components

The advice's pointcut denotes every constructor execution with parameters String and IModel. Furthermore denotes the calling class itself so that it can be used as a parameter for the ad-
vice. This way the advised edit panel becomes accessible. In fact this is the core edit panel of the *term* component. Now it becomes possible to add the feature’s Wicket components to the Wicket component tree of the core edit panel.

This solution is fully generic just like the previous one which means that it can be parameterized with any component implementing class just like shown in 5.38. Furthermore, if a stand-alone not aspect version of the edit panel is needed too, then an empty Wicket edit panel can be defined which gets its content assigned in the same way like an ordered Wicket edit panel does. This way code duplication is avoided.

### 5.4.4 Comparison and Evaluation

The native solution achieved with Java language concepts is of all three solutions the most extensive and intrusive one. Methods have to be enriched with events to identify extension points and classes have to implement the *Feature* interface. On component ordering process, methods of superclasses have to be overridden to appoint advisors and features. This leaves the component assembler with knowing technical details of the feature implementation. Furthermore, in the case of Wicket edit panels a specialized class *FeatureModel* has to be employed which meets the concerns of the feature implementation. On the positive side is that no additional technologies are needed and that code introduction can be defined in *Advisor* classes. Thus the component assembler does not need to know which methods need to be advised with which code.

The Reuseware solution is non-intrusive. The creation of a reuse extension for Java for the both composition systems might pose problems which stem from the bad documentation of the Eclasses of JaMoPP. Though it is possible to get a hold of the Ecore model file for JaMoPP, still it is not possible to find out which attributes an Eclass possesses. Both attributes *implements* and *members* of Eclass *java::classifiers::Class* were guessed. Once the reuse extensions are defined, the composition system can be used when ordering components. When contributing feature code to core code, the component assembler needs to know which feature code needs to be woven into which core code. Thus still a technical understanding of the inner workings of the component framework is needed.

The solution achieved with AspectJ is non-intrusive too. The definition of aspects is unproblematic and straight forward. The use of them is trivial, no technical details are known. The aspect itself knows which methods are to be advised.

Of all three solutions, the AspectJ solution is the best one. It is non-intrusive and no further knowledge is needed when using it. The Reuseware solution needs further knowledge but is non-intrusive and the native solution needs the most knowledge and is also intrusive.
5.5 Advanced Aspects for Java

Another application of the aspect composition system is the involvement of the workflow package. A term component that is created is to be managed by a workflow. For example if a term is not used for over a month it should be deleted. This means that there is some activity which does nothing more than to retain the time a term was created. A condition then would check if the time difference is greater than a month. Subsequently, an action would be responsible for deleting the term.

The workflowed term has to be associated with its recent state and the activity retaining the time of creation. Listing 5.40 shows a class providing the needed attributes.

```
public interface CoreWorkflowedEntity
    extends CoreEntity<Long> {

    public EntityType getEntity();
    public void setEntity(EntityType entity);

    public CoreState getState();
    public void setState(CoreState state);

    public Set<CoreActivity> getActivities();
    public void setActivities(Set<CoreActivity> activities);
}
```

Listing 5.40: Attributes of a workflowed entity

This class is not to be intended a feature that is going to be assigned to a term. It is a part of the workflow system which manages a collection of workflowed entities. As for the term itself, it gets the state token assigned to be able to quickly access this information. This is the first challenge to be solved. The second challenge is to connect the life-cycle of a CoreWorkflowedEntity instance to the one of a workflowed entity.

Both of the challenges are similar to those presented in 5.4. Nevertheless, there are differences. The first one is that the feature code that is getting introduced into the core will need to share variables defined in the core code. The second one is that the code is not introduced into the constructor but into methods defined on a superclass.

The CoreTermManager class is the manager component for a Term component. It is based on CoreEntityManager from package Entity (see Section 4.1.1). This component implementing class provides means for simple life-cycle management as creation and deletion. Listing 5.41 shows the signatures of the relevant methods.
public interface CoreEntityManager

    <
    IdType,
    EntityType extends CoreEntity<IdType>
    > {

        public IdType create(EntityType entity);
        public void delete(EntityType entity);

        ....
    }

Listing 5.41: Life-cycle methods of CoreEntityManager

As can be seen, both methods have a parameter entity which is also needed in the feature code which is responsible for creating a WorkflowedEntity instance which in turn needs to reference the workflowed entity. In this section it will be shown that the Reuseware solution has drawbacks.

The solutions for Hibernate mappings files and Wicket extended HTML files are not investigated again because they do not offer new insights.

### 5.5.1 Native Solution

The first thing that is needed is a class that acts as a feature to add the needed state token to the core class that is the Term component implementing class. This class is really simple, it only consists of one attribute representing the state token and of course it implements the Feature interface (see Listing 5.25 and reflst:featurejava). Therefore a listing of it is left out.

To connect the life-cycle of the two components, the create and delete method just have to send an event to designate the places feature code can be attached to. The event used has to have as an attribute the entity which is going to be created respective deleted. An advisor then can attach event handling methods for both of the events. Listing 5.42 shows the Advisor class.

```java
public class WorkflowEntityAdvisor
    extends AbstractAdvisorImpl<WorkflowEntity, CoreEntityManager>
    implements Advisor<WorkflowEntity, CoreEntityManager> {

    private Factory<CoreWorkflowedEntity> workflowedEntityFactory;
    private CoreWorkflowManager workflowManager;

    \\ setter and getter for the attributes
```
@Override
public void attachListenersToCore(Feature core) {
    core.addListener(
            CreateEntityEvent.class,
            After.class,
            this,
            "afterCreateEntity",
            1
    );
    // attach listener for DeleteEntityEvent
}

public void afterCreateEntity(CreateEntityEvent event) {
    workflowedTerm = workflowEntityFactory.createInstance();
    workflowedTerm.setEntity(event.getEntity());
    this.workflowManager.create(workflowedTerm);
}

public void afterDeleteEntity(DeleteEntityEvent event) {
    ...
}

Listing 5.42: Advisor for an workflowed entity

The entity that is being created is provided by the event. This solution works out as expected without any problems.

5.5.2 Reuseware Solution

Including the creation and deletion of a CoreWorkflowedEntity instance into the life-cycle of a Term instance reveals drawbacks of the Reuseware approach.

Again the feature implementation providing the state token is trivial and therefore is left out.

The second task is to introduce code to also create or delete a CoreWorkflowedEntity instance. It is captured in a dedicated class shown in Listing 5.43 which only shows the code for the case a term is created since the code for the other case is very similar.

class WorkflowEntityAsp {
    private Factory&lt;CoreWorkflowedEntity&gt; workflowEntityFactory;
    private CoreWorkflowedEntityManager workflowEntityManager;
    \\ setter and getter for the attributes
    private CoreEntity workflowEntity;
}
public void onCreate() {
    CoreWorkflowedEntity workflowedEntity = 
        workflowedEntityFactory.createInstance();
    workflowedEntity.setEntity(workflowEntity);
    this.workflowedEntityManager.create(workflowedEntity);
}

public void onDelete() {
    // work with attribute workflowedEntity too
}
}

Listing 5.43: Advice code for including the management of a workflowed entity

Method `onCreate` is to be woven into method `create` of class `CoreTermManager`. On the second line the entity attribute is set on the `CoreWorkflowedEntity` instance by assigning the attribute `workflowEntity`. This one is meant to be the same as the parameter `entity` from `CoreTermManager's create` method. It is the reference to the entity that is going to be workflowed. The task is to replace the `workflowEntity` attribute reference in `onCreate` by the `entity` variable reference of method `create`. To do so, a little enhancement of the Java reuse extension of the aspect composition system is sufficient, see Listing 5.44.

```java
...  
fragment role Core {
    port Method {
        ...
        java::parameters::Parameter is anchor {
            port expr = `$'var-anchor: 'concat(self.name)`$
            point expr = `$'variable '$`
        }
    }
}

fragment role Aspect {
    port Advice {
        ...
        java::variables::Variable is slot if $self.name.endsWith('_Slot')$ {
            port expr = `$'var-slot: 'concat(self.name).replace('_Slot', '')`$
            point expr = `$'variable '$`
        }
    }
}
```
Listing 5.44: Aspect reuse extension with added variable referencing support

To limit the number of variables that are being shown as slot, a naming convention is used. It demands that the variable name ends with _Slot. Thus variable workflowedEntity has to be renamed to workflowedEntity_Slot.

The next step is to order a TermManager for ontology component Instance by creating a subclass of component implementing class CoreTermManager. This will be the class Reuseware will work on. Listing 5.45 shows the ordered component’s implementing class.

```java
public class InstanceTermManager
    extends CoreTermManager<InstanceTerm, InstanceTermDao> {
```

Listing 5.45: An ordered term manager for an instance component

This class is empty, it only binds the generic type parameters of CoreTermManager. This is intended, see section 4.2.3 for an explanation. This class does not contain the two methods create and delete that are to be advised by Reuseware. Both of them are declared in the superclass (CoreEntityManager) and as long as they are not overridden, their definitions do not appear in a subclass again. The problem is that Reuseware needs their definition to be available on the class it works with to provide the ports defined by the composition system. A workaround for this is to override the methods that need to be advised in the ordered component’s implementing class. This is a drawback of the Reuseware approach.

The resulting composition diagram is shown in Figure 5.15.

Figure 5.15: Workflow composition diagram

As can be seen, the newly defined variable ports are connected to each other.

In more complex scenarios there might be a lot of methods that are to be advised. Assuming there are 20 methods each one possessing one parameter. Using the aspect composition
system there would be one port for each parameter and two ports for each method. This means there are 50 circles connected to the fragment shape in a composition diagram. Figure 5.16 shows a fragment of a class consisting of 19 methods with 15 parameters. It uses the aspect and the inheritance composition system.

![Composition Diagram](image)

Figure 5.16: A fragment with many ports

If challenges become more complex Reuseware reveals drawbacks. Methods have to be overridden by hand and fragments composition diagrams can have many ports.

### 5.5.3 AspectJ Solution

To solve the first challenge with AspectJ, an aspect introducing the state token on the term component through an inter-type declaration. For the reason that this aspect is trivial it is left out.

The second challenge is to connect the life-cycles of a term component and its corresponding workflowed entity managed by the workflow itself. Listing 5.46 shows how it is done.

```java
1 public aspect WorkflowManagerAsp
2
3   <ManagerComponent extends CoreEntityManager {
4     
5     interface WorkflowManagerFeature {
6       public Factory<CoreWorkflowedEntity> getWorkflowedEntityFactory ();
7       public CoreWorkflowManager getCoreWorkflowManager ();
8     }
9     // and setter for both attributes
10 }
```
As can be seen, AspectJ is able to provide means for the more sophisticated use cases. Therefore the solution does not need any workarounds.

Nevertheless, an observation was made during the work with AspectJ. There were different errors complicating the work. Classes did not compile correctly and showed errors that were no errors and the compiler itself reported errors in the compiling process resulting in no compilation at all. Those errors could be solved by Eclipse’s `Clean...` build option, by closing and opening the project, or by removing and adding the AspectJ capability to the project.

### 5.5.4 Comparison and Evaluation

The native solution provides means to solve the more sophisticated use cases. There is no need to introduce new concepts which would have make the solution more complex. Thus, there are no new insights. The same holds for AspectJ. The new technical requirements are met within it.

Differently Reuseware which needs to workaround its disadvantage that it only works with static source code. Methods inherited by superclasses are not defined on the subclass again. On runtime, methods are bound to objects through late binding. The appropriate method
code is looked up in a table. AspectJ has means to handle this. In Reuseware the only possibility is to redefine the needed methods in the subclass. Still, there is the theoretical possibility to have these methods be created by JaMoPP. Another drawback is that fragments can possess a lot of ports. A component assembler would need to have an API documentation and a composition recipe to know which feature code is to be woven into which core methods.

5.6 Exchangeable Class References for Spring

Deploying the ordered components in the Spring container is a tedious and error-prone work. A deployed component is called a bean in the context of Spring. A bean needs to have a name and a class it is instantiated from. Listing 5.47 shows a typical configuration of term components for naming ontologies.

```
1 <bean id="ontologyTermFactory"
2   class="components.term.OntologyTermFactory"/>
3 <bean id="ontologyTermDao"
4   class="components.term.OntologyTermDao"/>
5 <bean id="ontologyTermManager"
6   class="components.term.OntologyTermManager">
7   <property name="dao" ref="ontologyTermDao"/>
8 </bean>
```

Listing 5.47: Typical deployment of business components

To be able to name the remaining ontology entity components that are classes, instances, and properties, three additional term components as in Listing 5.47 are needed for each of them. Those are already twelve bean definitions. A complete configuration of a component system can be quite large. Therefore it is desired to have a template configuration which can be filled with concrete class references to obtain a ready to use Spring configuration file. Furthermore changes to a package structure affecting a set of references or systematic changes to single references will be more convenient and less error-prone.

5.6.1 Native Solution

A native solution can not be achieved. Spring does not have a modularization concept for exchangeable class references. Nevertheless there is an Eclipse plugin Spring IDE [sprb] which supports the creation and maintenance of Spring configuration files. They can be created textual and graphical. Furthermore there are wizards for creating configuration files. In addition, bean definitions are validated including their used classes and their reference among themselves (dependency injection). There is a view which displays all Spring-related files. Those files also can be searched through Eclipse’s search facility which the plugin
extends. Refactorings are supported like renaming and moving.

5.6.2 Reuseware Solution

To solve the task with Reuseware the reference composition system can be reused. A reuse extension for Spring configuration files is needed. Before it can be created, an Ecore model of Spring configuration files is needed. There is an XSD definition available which means that an Ecore model can be created with the help of EMF, see Section 5.1.2. After having added the Ecore model plugin to Eclipse it can be used in the needed reuse extension. Again, the needed metaclasses can be looked up in the corresponding Ecore model file. Listing 5.48 shows the resulting reuse extension.

```xml
<componentmodel components.systemdefinition.referencespring>
  <implements components.systemdefinition._reference>
    <epackages>
      <http://www.springframework.org/schema/beans>
      <http://www.emftext.org/java>
    </epackages>
    <rootclass>DocumentRoot{
      <fragment role>ClassReference_Receiver{
        <port>Class_Hook{
          <BeanType.class>is value hook{
            <port>expr = $self.name$
            <point>expr = "$'class'$
          }
        }
      }
      <fragment role>ClassReference_Provider{
        <port>Class{
          <java::classifiers::Class>is value prototype{
            <point>expr = "$'class'$
            <value>expr = $eContainer().oclAsType(
              java::containers::CompilationUnit).
              namespaces.concat(self.name)$
          }
        }
      }
    }
  }
</componentmodel>
```

Listing 5.48: Reuse extension for exchangable references for Spring

This reuse extension uses two Epackages, the one newly created for Spring configuration files and one for Java. The port `Class` of fragment `ClassReference_Provider` has a value prototype which provides the fully qualified name of a Java class. The other port `Class_Hook` receives
this name. It represents the *class* attribute of Spring configuration’s *bean* element. Through
the use of a value hook, Reuseware is able to connect both of the fragments although they
belong to different file types.

Figure 5.17 depicts the resulting composition diagram.

![Composition program for a Spring configuration file](image)

**Figure 5.17: Composition program for a Spring configuration file**

### 5.6.3 Comparison and Evaluation

Spring does not offer a real native solution for the problem but it offers an Eclipse plugin
*Spring IDE* which offers numerous functionality. It solves the hand at task and also offers
other features which might be needed. It is a convenient and working solution.

Reuseware solves the task at hand too. When renaming or moving a fragment, references to
it in the Spring configuration file should be updated too. The problem is, that at the time
of writing, Reuseware does not support the rename or move refactoring.

Assumed Reuseware did like it should. Still it had to be decided if Reuseware was chosen
over Spring IDE. The advantages of the latter are that it offers additional functionality apart
from solving the problem at hand. It has a community developing it further which means
that bugs are fixed, new features are added, and support can be obtained. Using Reuseware
these tasks have to be done by oneself and still there would be functionality that Reuseware
could not provide. Thus, in case there are already grown software tools for the work with
the files, Reuseware might not be able to compete.

### 5.7 Overall Comparison and Evaluation

The composition technique of Reuseware, invasive software composition, has different advan-
tages but also two disadvantages, see the comparison shown in Figure 5.18.
A major advantage of the composition technique is that enriched languages are not aware of it. Therefore every tool for that language can be used further on. Composition systems can be reused which reduces learning efforts by reusing the same concepts. The port concept enables to group different variation points which leads to reuse of defined ports which also reduces learning efforts.

The first disadvantage of only working with static code shows when using languages that implement some runtime logic like Java. Java inheritance mechanism allows for inheriting methods to subclasses where they do not need to be defined again except they should be overridden. At runtime, Java looks up methods from a comparison and associates them with a calling object (late binding). The problem which arises is that Reuseware only works with what is defined on the class at hand. Therefore not every method a class possesses can be used as a port.

The second disadvantage is that for using the composition technique, a dedicated software is needed which in this case is Reuseware. When deciding to use a tool in a production environment, it needs to be supported by an active company or community that can provide help, fixes bugs, and implements needed features. At the time of writing, Reuseware does not have such a support.

The comparison in Figure 5.19 evaluates the Reuseware Eclipse tools.

As explained, there is no community constantly developing Reuseware further. Also, an exhausting documentation is missing. Both factors make it difficult to learn Reuseware or to find help. Positive is the existence of code completion that helps to find out which statements are allowed.
Reuseware composition programs are graphical. This is an advantage because diagrams are easier to read than text and they can be used as documentation too. At the time of writing, composition programs can become unclear because of the presentation of fragment ports and their names (see Figure 5.16). A solution to this was to show the port names in a list inside the fragment. Positive is that arbitrary many compositions can be made in one composition diagram. Therefore one diagram can be sufficient for one package.

The fragment browser lists all of the fragments a project possesses and shows the provided ports to give an overview. Also, it becomes unclear the more files a project has. This is due to the lack of a tree view. The handling of the Reuseware tools lack different standard features as drag and drop, double click to open a package in the fragment browser, or a convenient way to mark a fragment as a target. When a fragment is updated, Reuseware also updates the corresponding composed fragment. This means that the composites are always up to date, which is positive. The downside is that at the time of writing this compositing takes very long due to Reuseware recomposing every target fragment. A solution for this problem is being worked on. On the other side there are no further refactoring mechanisms supported. Moving a file or renaming it breaks composition diagrams.

Another problem can be memory consumption in big projects. An Eclipse with installed Reuseware plugin consumes 450 megabytes up to 600 megabytes working memory for a project with 400 files. An Eclipse without it needs about 200 megabytes.

The described advantages and disadvantages hold for every use case of Reuseware. In the following, the solutions achieved with native means and with AspectJ are evaluated. These can be compared to Reuseware by using the previous comparison 5.19. Further arguments regarding a specific Reuseware solution are brought into consideration later on.

The native solutions achieved with Hibernate are evaluated in the comparison shown in Figure 5.20.

| Inheritance and Inter-type declarations | + no additional software needed | - two files per mapping needed |
| Class References | + multi-inheritance, inter-type declarations possible | - DTD entity definitions |
| | | - not possible, only by hand |

Figure 5.20: Evaluation of the native Hibernate solutions

Positive is that the Hibernate solution already provides the possibility to use multiple-inheritance which can be used for inter-type declarations. Negative is its technical realization. The use of DTD entities enforces to have two files per mapping. Furthermore DTD entity declarations for every superclass have to be made in the head of the mapping file. This complicates refactoring of inheritance relations because every mapping file of every subclass has to be adopted too. The Reuseware solution does not have these disadvantages and is there-
fore superior. Exchangeable class references are not possible with native Hibernate means. Thus the use of Reuseware is a good choice.

The comparison shown in Figure 5.21 shows the evaluation of the native solutions for Wicket extended HTML files.

Inheritance and Inter-type declarations + no additional software needed + usage is simple + inter-type declarations by hand result in better control of the appearance of HTML elements - only single-inheritance (no inter-type declarations possible)

Figure 5.21: Evaluation of the native Wicket solution

The usage of Wicket’s provided inheritance mechanism is very simple. Nevertheless it only provides single-inheritance. When ordering a component and assigning features to it, the needed Wicket components have to be inserted into the ordered component’s HTML file by hand. In this case this is not a drawback because it allows full control of the appearance of the resulting HTML code. In contrast, Reuseware does not allow to declare in which order a composition affecting one fragment has to be done. Even if it was possible, a component assembler (or web designer) might wish to add additional HTML elements, which would result in editing the composed file. This is not suggestive because Reuseware overwrites the file when one of the involved fragments is updated. The additional HTML elements would have to be inserted every time an update happens. It needed further investigations to clarify if Reuseware was an appropriate choice for combining HTML files, extended by Wicket elements or not.

The evaluation for the native Java solutions are shown in the comparison shown in Figure 5.22.

General + no additional software needed + robust language, literally bug-free + developing community + very good documentation and books - missing modularization concepts (advice - inter-type declarations)

Aspects + no additional software needed + uses robust Java language + no API knowledge of advised classes needed - intrusive solution (feature interface and events) - object schizophrenia - sophisticated usage - usage needs technical knowledge (FeatureModel)

Figure 5.22: Evaluation of the native Java solutions

The general advantage of using native Java is that it is is a proven technology which is literally bug-free. It has a company and a community offering support and developing Java
further. At the time of writing, Java does not support aspects natively. This means that this support has to be added by using Java native means. The positive side is that it is possible to add this support without any limitations. The drawback is that a framework for doing so adds overhead to the Java classes. They have to implement a certain interface Feature (see Listing reflst:featurejava) and methods have to identify extension points through events. This is an intrusive solution. Inter-type declarations are realized through object composition which results in object schizophrenia [Ste87]. When using the aspect system for assigning features, the component assembler needs technical knowledge regarding its usage. In the case of Wicket, edit panel classes that are added as a feature have to use a different model class FeatureModel (see Listing 5.31). This class has to be introduced to cope with the indirection caused by the object composition. A component assembler needs to have knowledge of this class, which is a drawback. The more complex the solution becomes, the more of such classes will be needed.

In contrast, using the Reuseware solution, there is no indirection through object composition, thus there are no adjusted solutions needed. Then again, weaving feature code into a core needs knowledge about the API of the involved classes. A component assembler has to know which feature code has to be introduced into which core code. This is a major drawback because a detailed composition instruction is needed.

The AspectJ solution is evaluated in the comparison shown in Figure 5.23.

| General                          | + non-intrusive solution | - additional software needed |
|                                 | + mature language concepts | - not bug free               |
|                                 | + developing community    | - memory consumption         |
|                                 | + good documentation and books |                     |

| Aspects (Inter-type declaration and Advice) | + multi-inheritance, inter-type declarations possible |
|                                           | + no technical knowledge needed (FeatureModel) |
|                                           | + no API knowledge of advised classes needed |
|                                           | + very easy usage (generic aspects) |

Figure 5.23: Evaluation of the AspectJ solutions

AspectJ is an aspect-oriented system with mature language concepts. It is able to provide non-intrusive solutions for the given problem of assigning features. Advanced requirements can be fulfilled too. The resulting solutions are easy to use. A component assembler does not need to know technical details regarding the solution nor does he need to know APIs of the involved classes. The latter one is expected when using Reuseware. Therefore it provides the best solutions of all three, that is native Java, Reuseware, and AspectJ.

Nevertheless it has a serious drawback. During the use of AspectJ in this work, it was found
that bugs appeared. Java classes showed errors that were no errors and the AspectJ compiler stopped its work due to bugs. Therefore AspectJ does not appear as the ideal choice at the time of writing. Furthermore, the AspectJ plugin consumes 550 megabytes working memory which is on par with Reuseware.

Comparison in Figure 5.24 shows the evaluation of the solution using Spring IDE.

<table>
<thead>
<tr>
<th>Class References</th>
<th>+ additional functionality</th>
<th>- an additional Eclipse plugin is needed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ supported by a community</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.24: Evaluation of the Spring solution

The solution using Spring IDE solves the task at hand. Furthermore it offers additional functionality which Reuseware can not offer. However, at the time of writing, Reuseware is not able to deliver the functionality expected. It does not offer the rename and move refactorings for its fragments. Even if it would, still Spring IDE seems to be the better choice. It offers community support.
6 Summary and Perspective

In one out of three cases Reuseware seemed to be an appropriate choice. It is not suitable for Java because of it only working with static source code and because of generating a too complex composition interface. It is not the better choice for Spring configuration files because there is the Eclipse plugin Spring IDE which has a rich and tailored functionality. For HTML files it can not be made an assertion without further investigations. Nevertheless, for Hibernate mapping files it appears to be the right choice. However, it has to be taken into consideration that Reuseware does not have a supportive community and lags some needed features regarding its handling.

In the end, a mixed approach of native Java, AspectJ, Reuseware, and Spring IDE could be used. Native Java realizes the aspect system for features, AspectJ could be used for simple tasks as weaving the generation of events into methods, Spring IDE for managing Spring configuration files, and Reuseware for combining Hibernate mapping files. Wicket extended HTML files would be composed by hand for full control.

A perspective is to improve Reuseware. The basic features like refactorings should be implemented. The composition diagrams and the fragment browser need to become clearer. Basic handling functionality like drag and drop needs to be implemented. Furthermore it needs a complete and explanatory documentation. A forum is needed for users that want to ask questions.

Another perspective for Reuseware is that its composition system framework CoMoGen will be used as part of specialized software tools which add modularization concepts to languages. Composition programs are to general to fulfill more sophisticated use cases as shown in this work. For example Spring IDE could use CoMoGen offer more modularization concepts.

Furthermore it can be investigated how Reuseware is able to address the requirements HTML composition has. The question is if there is a workflow which is supported by Reuseware and which gives a component assembler (or a web designer) the means to create every variation of a HTML file needed. An Eclipse plugin which uses CoMoGen could be created which offers specialized HTML support with Reuseware technology as its foundation.

Another implementation of the idea of using CoMoGen for specialized tools is to provide a tool for defining aspects. Such a tool needed to provide means to define pointcuts. This would make up for the problem when using composition diagrams. Furthermore the problem
with the non-existent methods from superclasses in subclasses could be solved. Because of AspectJ not being bug-free it could be a good alternative.
Bibliography


5Fschema\%5Fdeveloping.
Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, September 1, 2010