Master's Thesis

Verification of Software Architectures using Static Code Analysis for Java

Malte Brunnlieb
Technische Universität Kaiserslautern
AG Softwaretechnik
m_brunnl@cs.uni-kl.de

26. September 2012

Erstprüfer: Prof. Dr. Arnd Poetzsch-Heffter
Zweitprüfer: Dr. Gerhard Pews

Betreuer: Dipl.-Inf. Patrick Michel
M. Sc. Julian Meisel

Technische Universität Kaiserslautern
SOFTEX Arbeitsgruppe Softwaretechnik
Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

STATEMENT OF AUTHORSHIP

I confirm that the work presented in this research proposal/research report has been performed and interpreted solely by myself except where explicitly identified to the contrary. I confirm that this work is submitted in partial fulfillment for the degree of MSc in Computer Science and has not been submitted elsewhere in any other form for the fulfillment of any other degree or qualification.

Kaiserslautern, 26. September 2012

(Malte Brunnlieb)

Kurzdarstellung

Die vorliegende Masterarbeit beschäftigt sich mit einer Machbarkeitsstudie zur automatischen Verifikation von Architektur-Pattern gegen gegebene Implementierungen. Die Machbarkeitsstudie wird anhand der Register Factory® durchgeführt, welche von der Capgemini Holding GmbH entwickelt wurde und die Referenzarchitektur für administrative Software des Bundesverwaltungsamt darstellt. Die Machbarkeitsstudie verfolgt einen Pattern basierten Ansatz zur Strukturierung der Architektur. Für eine repräsentative Abdeckung der Architektur wird eine Menge verschiedenster Pattern zusammengestellt, die in einem zweiten Schritt in einer für diesen Zweck definierten Pattern Beschreibungssprache beschrieben werden. Da die entwickelte Beschreibungssprache maschinenlesbar ist, kann darauf folgend eine prototypische Umsetzung der Pattern Verification Engine (AVE) erfolgen. Mit diesem Architektur-Verifizierungstool lässt sich abschließend die definierte Menge an Pattern gegen die vorhanden Implementierungen des Kunden Capgemini prüfen.

Abstract

This Master's Thesis focuses on a feasibility analysis of an automatic verification tool for software architectures in the context of the Register Factory®. The Register Factory® is the reference architecture for administrational purposes developed by Capgemini Holding GmbH for the Federal Administration Office of Germany. The Feasibility analysis is done using a pattern based approach, such that the architecture can be converted into a more structural description than plain text. Afterwards, a representative set of patterns are brought into a self-developed machine readable pattern description language. Lastly, a prototype will be developed, which is able to verify the selected set of patterns against given implementations of the customer.

Table of Contents

Kurzdarstellung 5

Abstract 7

1 Introduction 11

2 Fundamentals 13

2.1 Register Factory® 13

2.1.1 Architectural Refinement 14

2.1.2 Technical Issues 15

2.2 Eclipse Framework 17

3 State of the Art 19

3.1 Architecture Compliance Checking 19

3.1.1 Techniques 19

3.1.2 Development Approaches 21

3.2 Architectural Patterns 22

4 Architectural Patterns 24

4.1 Abstraction Levels 24

4.2 Patterns for the Register Factory® 24

4.2.1 Program Structures 25

4.2.2 Implementation Facets 27

4.2.3 Component Structures 28

5 Formal Pattern Specification 29

5.1 Pattern Description Language 29

5.1.1 Syntax 29

5.1.2 Semantics 30

5.2 Pattern Specification 35

5.2.1 Pattern Preferences 35

5.2.2 Program Structures 36

5.2.3 Implementation Facets 37
5.2.4 Component Structures 37
6 Architecture Verification Engine (AVE) 39
6.1 Architecture 40
6.2 Verification Processing 41
6.2.1 Source Code Model Extraction 41
6.2.2 Spring Model Extraction 43
6.2.3 Realization of Predicate isExceptionFacade 43
6.3 Handling of Components 45
6.4 Data Management 45
6.5 Usage 46
7 Conclusions 49
7.1 Future Work 50
Bibliography 51
List of figures 53
Listings 55

1 Introduction

In today's Software Engineering quality assurance is one of the most important parts during the development of large software systems. Having well-established quality assurance processes will lead to lower costs generally. One essential aspect therefor is to assure a good maintainability of the software product, as the follow up costs will be reduced tremendously. This fact gets even more important, looking at the trend of distribution of work in the Software Engineering domain over time. Already in the 80's there were nearly as many developers as maintenance personal and furthermore the amount of maintenance tasks shows a fast growing curve. According to Capers Jones referenced by Deursen, Klint and Verhoef [1], there will be double as many maintenance personal as developers in the whole Software Engineering domain in 2020. This shows an alarming evolution in the Software Engineering domain, as the number of maintenance personal and programmers first equaled in the 90's. Therefore, the main focus on today's research in Software Engineering should be the effort reduction of maintenance tasks.

Essentially, good software maintainability is a result of a well-defined system architecture, its documentation, and its implementation. In addition the implementation has to be compliant to the documented architecture. Usually the last issue has been exposed to be the hardest one to achieve as software development is a dynamic process and it is not easy to keep the implementation and the architecture with its documentation mutually up to date. Consequently, processes have to be established which assure the compliance of an architecture and its implementation or at least check their compliance for controlling and monitoring issues. A possible effort and cost minimization for such maintenance tasks can be achieved by automatic architecture compliance checking.

Hence, this Master's Thesis discusses the question, whether it is feasible to verify different aspects of a given software architecture against its implementation automatically. For this purpose, a prototype of a compliance checking tool—called Architecture Verification Engine (AVE)—has been implemented in the specific context of the Register Factory®. The Register Factory® describes a highly standardized architecture for developing registers and business applications for the German administration institutions. Beside the implementation of the AVE, another aspect of this Master's Thesis is the development of a rudimentary pattern description language. In the context of the Register Factory® this description language is only used for enabling the specification of architectural patterns in a machine readable way. The AVE will be able to verify every pattern specified in this pattern description language on any given Java implementation. Furthermore, the AVE will be implemented as an Eclipse plug-in, as the Eclipse IDE is a well-established open source development environment for Java developers and it provides a very good framework for static code analysis.

Starting with all necessary fundamentals, the Register Factory® and its architecture will be described in Section 2.1, such that the terminology and the later deduced architectural patterns are understandable in the given context. In order to get a short overview of the frameworks and technologies used for the implementation of the AVE, the Eclipse and especially the JDT framework will be introduced in Section 2.2. Beside the fundamentals, there is a brief introduction into the current state of the art of pattern descriptions and verification techniques in Section 3. Furthermore, Section 3.1.2 provides an overview of different development approaches for architecture verification tools. Afterwards, Section 4 introduces the division of patterns into different abstraction levels. The identified architectural patterns for the Register Factory® are listed in Section 4.2 according to their abstraction levels. Providing the ability to describe these patterns in a machine readable way, Section 5.1 first defines the developed pattern description language. Following in Section 5.2, the previously specified patterns are implemented in the defined language. On top of the defined pattern description language, the implementation of the AVE is described in Section 6. Finally, all results of this Thesis are summarized and assessed in the Conclusions Section 7, followed by the future work in Section 7.1, which addresses open questions and open development tasks.

2 Fundamentals

2.1 Register Factory®

The Register Factory® is a reference architecture, which provides a well-documented standard for creating architectures for software in the context of governmental institutions for administrational purposes. It has been developed by the Capgemini Holding GmbH for the Federal Administration Office of Germany. The Capgemini Holding GmbH is one of the leading software development companies for governmental software and custom solutions in Germany. Besides, it is a globally acting management and consulting company for IT business.

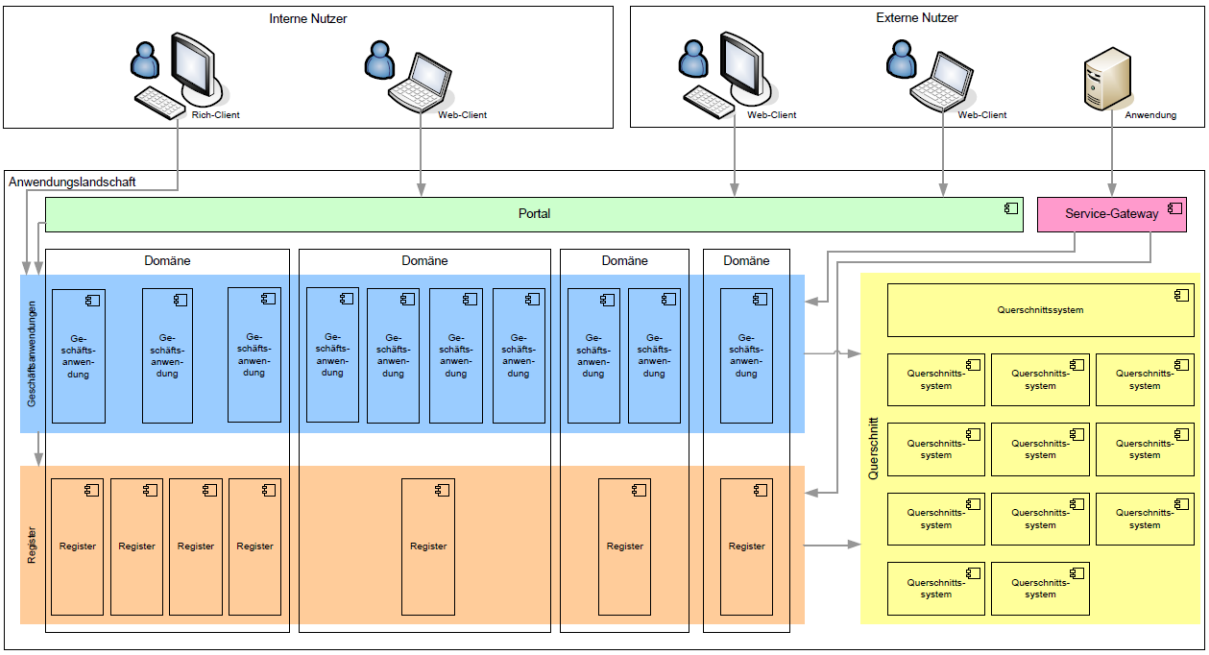

Figure 1: System Application Architecture [2]

The Register Factory® divides every system architecture, derived from the reference architecture, mainly into three components and two interface implementations, as shown in Figure 1. The three technical components are

1. Crosscutting Components (Ger.: “Querschnittssysteme”) are supporting services for logging, authentication and common used functionality.
2. Registers (Ger.: “Register”) have the focus on managing and providing data in a secure way for other components.
3. Business applications (Ger.: “Geschäftsanwendungen”) describe abstract functionality on data provided by the underlying registers. A business application can be seen as an

implementation of a frontend for a specific user role only focusing on some business use cases.

Beside the component definitions, the Register Factory® also prescribes the communication channels to be used. Thus, all communication across component borders has to be established via the Spring-Framework for Java, which leads to lower coupling and smaller independent implementation units. The Spring-Framework is a well-established open source framework mainly providing the ability of component decoupling by dependency injection.

The focus of this Thesis will be on registers, as these currently provide the highest degree of standardization given by well-defined architectural principles. The advantage of the architectural principles for registers is, that they are restrictive enough to enable verification on a very high level of detail.

2.1.1 Architectural Refinement

As already mentioned, the Register Factory® is a meta-architecture which helps to specify concrete architectures for a concrete implementation of a Register Factory® component. A short abstract description of the development process for business applications or registers will help to understand, how to come up with a verification engine for architectural patterns in this context. The development of the Register Factory® components is done in mainly two steps:

1. Creation and documentation of the concrete system architecture / design
2. Implementation of the concrete system design

As shown in Figure 2, each step during the development of the target component results in a refinement and enrichment of the architecture, the solution should be based on. Thus, every logical step can be seen as a new level of abstraction, whereas the highest abstraction level given by the Register Factory® is the development starting point.

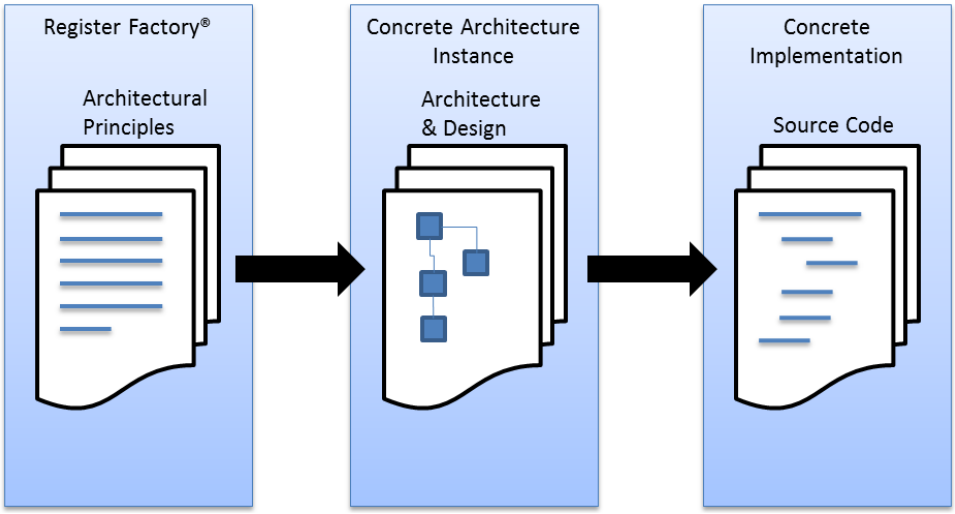

Figure 2: Steps of Refinement using the Register Factory®

On the highest level of abstraction, architecture descriptions are written in human readable text or sometimes also using very abstract UML diagrams. For the development of a concrete solution, concrete architecture and design documents have to be derived. Especially, the design documents define specific implementation strategies and technical dependencies for the intended solution. In addition, components are divided into smaller entities like types and their namespaces. Finally, it gets possible for developers to implement the solution by using the design documents.

However, from maintenance point of view a design document should not be used as input for deriving architectural patterns, as design documents are only valid for one specific implementation. Nevertheless, the architectural principles of the Register Factory® are abstracted from realization instances, such that it gets possible to derive generalized architectural patterns and use them for each implementation of the abstract common architecture. In order to close the gap between the high level abstraction of the architecture principles and the finally realized source code, the Register Factory® defines restrictive naming conventions.

2.1.2 Technical Issues

As one very effective architectural tool, naming conventions could be used for encoding architectural information in the source code without asking developers to understand the whole system architecture. Using naming conventions for types, packages, folders, and many other implementation entities increases the readability of code for each stakeholder. Even the introductory training for new employees will be as easy as for other people working on the code, e.g., maintenance personal.

Therefore, the overall architecture of the Register Factory® is encoded in the implementation by enabling restrictive naming conventions on different implementation entities. Besides, such

restrictive naming conventions are essential preconditions for an automatic verification process of architectural principles. The naming conventions of the Register Factory® are described in Listing 1.

Listing 1: Naming convention of the Register Factory®

```
de.bund.bva.<domain>.<business application/register>.<layer>.<subsystem/component>…
```

`<domain>`	"Domain name as stated in the technical architecture specification."
`<business application/ register>`	"Name of the business application or register as stated in the technical architecture specification."
`<layer>`	`common | gui | batch | service | core | persistence`
`<subsystem/component>`	`<for register> | <for business application> | <subsystem/component>`
`<for register>`	`meldung | auskunft | mitteilungen |protokollierung | fristenkontrolle | admin | datenpflege | analyse | basisdaten`
`<for business application>`	"Name of the business application respectively crosscutting component as stated in the technical architecture specification."

2.2 Eclipse Framework

The framework Eclipse is based on the well-known OSGi-platform, such that it can be adapted and extended by other plug-ins easily. It provides a very easy using interface for all resources managed by the IDE itself. Especially the Java Model of the Java Development Tools (JDT) provides an excellent entry point from Eclipse java projects down to compilation units, which are basically Java source files. A Java Model is only available for Java projects, which have to be a subtype of IJavaProject. A short introduction into the model has been given by Lars Vogel [3].

In order to step deeper into the Java source code than only listing method names of types, the Abstract Syntax Tree (AST) [4] has to be used. This syntax tree represents the whole Java source code as it is processed in the internal Eclipse Java compiler. Thus, it is well suitable for the static code analysis performed in this Thesis. In contrast to the Java Model, the AST is only accessible via the visitor-pattern. Dependent on what node element one wants to have a look at, the overloaded visit function has to be overridden from the super class ASTNode.

Unfortunately there is no complete and compact documentation of all available nodes of the AST. Instead, every node is documented isolated in its own Java documentation; see all subtypes of the root node org.eclipse.jdt.core.dom.ASTNode in the JDT API. In order to support all language constructs of the newest Java version 1.7, the ASTNode will be parsed using a JLS4 ASTParser. The JLS-Level describes the ASTParser compatibility level, which is staggered as follows:

- JLS2: Support since Java 1.4
- JLS3: Support since Java 1.5
- JLS4: Support since Java 1.7

In order to give a better understanding of the Java language grammar, there will be snippets of the documentation of some ASTNodes available later in this Thesis.

3 State of the Art

The following Section focuses on different approaches of today's research performing architecture compliance checking. As a well-structured architecture description is one precondition for automatic compliance checking, architectural patterns will be shortly introduced. They provide an appropriate medium for structuring plain text architectures in smaller well-defined entites. The introduction of pattern especially delimits architectural patterns from others and clarifies its contents.

3.1 Architecture Compliance Checking

Generally architecture compliance checking is based on architectural principles defined by the systems architecture documents. Hence, it can be simultaneously considered with pattern verification techniques.

There are different approaches and supporting tools for defining architectural patterns and checking them against an implementation. Most approaches focus on Architecture Description Languages (ADLs). Using such languages, architectures are usually specified on implementation level. They ensure compliance by construction like e.g. ArchJava [5]. For this purpose ArchJava uses its own type system. A very good overview and comparison of ADLs and the tool support till 2000 is given by Medvidovic and Taylor [6]. However, this Master's Thesis will not focus on specifying architectural constraints on implementation level.

More sophisticated approaches are dealing with generalized pattern descriptions and the intention to find violations of a given pattern in unknown source code. This opens the large area of pattern recognition, as the locations of the pattern occurrences are not known. Research in this area is often done on design patterns, as these are the best documented and generalized ones. There are several techniques for design pattern detection like model-driven [7], using static and dynamic code analysis [8] or even using template matching [9].

3.1.1 Techniques

For compliance checking of more abstract architecture descriptions there is the necessity for more generic approaches, discussing how to enable compliance checking of a concrete architecture against its implementation. As stated in [10] there are essentially three techniques:

1. Dependency-Structure Matrices (DSMs)
2. Source Code Query Languages (SCQLs)
3. Reflexion Models (RMs)

Dependency-Structure Matrix

Dependency-structure matrix based approaches are working on matrices encoding all inter-dependencies within a software system. Such a matrix has the dimensions N x N, where N is the overall number of observed entities. The definition of entities can be stretched from low level types to abstract components—defined by the software architecture. Setting a flag to the matrix element e_{ij} will denote that element i is dependent on element j. Now architectural compliance checking will be possible by generating a dependency structure matrix from source code as well as from the architectural specification. There is an easy way to compare both matrices and thus verify the implementation against its specification. However, this only refers to the verification of dependencies, as other structures cannot be easily encoded in a single matrix.

Source Code Query Language

In order to manage other structures than dependencies, source code query language based approaches use a database as their source code model. This enables arbitrary database requests on the implementation model as known from SQL. The technical limits of this approach are defined by the power of the used pattern specification language and the possible database request operations. Nevertheless, the information extracted from source code can also be a limit, when there is the need of checking e.g. compliance on an abstract component level.

Reflexion-Model

In contrast to both described approaches, reflexion-model based approaches do not consider the implementation and its entities as is. Such approaches consider two input values. On the one hand the implementation is taken as a source code model and on the other hand an abstract architecture description is taken into account for abstract definitions like components. To put both abstraction levels together, a manual mapping between source code elements and abstract architecture entities has to be done. For that purpose, e.g., the SAVE tool—developed by the Fraunhofer IESE—uses an explicit mapping of java packages to components [11]. After defining such a mapping, patterns described on a high abstraction level can also be verified against the underlying implementation.

The prototype developed in this Master's Thesis will follow the principles of a reflexion-model based approach. There will be a source model abstraction of the implementation and also an abstract mapping of Java packages to architectural components. In addition the mapping between architectural and implementation entities will be described by the restrictive naming convention of the Register Factory®, as introduced in Section 2.1.2.

3.1.2 Development Approaches

The development of a compliance checking tool essentially consists of five steps:

- Deriving the source code model
- Developing the description language for architectural patterns
- Developing the verification engine for the defined description language
- Defining architectural patterns for the target system
- Enabling usability by integration into development IDEs

There is no standardized strategy in which chronological order these steps have to be performed. Nevertheless, there are two recommended approaches how such a system can be implemented.

The most commonly performed approach in research is the bottom-up development approach. In this approach, a generalized description language will be defined first in order to describe architectural rules within it. Often there is a reasonable endeavor to reduce such language to a known logic or build the language upon a set of approved logics. Therefore, this approach benefits from the given power of the logics used. One example for this strategy is given by the dissertation of Sebastian Herold [12]. As the description language is the working base of this approach, it often results in a generalized, sometimes also language independent solution, like in [12].

Once a description language has been defined, the source code model has to be derived for automatic pattern verification. On top of that, the pattern verification engine can be implemented. Lastly the verification engine should be extended by a user interface. Actually, it does not matter whether the user interface only provides a text editor with syntax checking or implementing a mouse based user interface, but the user should be supported in any way specifying new patterns in the given description language. Nevertheless, this last step is mostly omitted as it is not a mandatory part of the performed research. In fact, it cannot be omitted if there is the purpose to evaluate the verification tool in real industry. At least it is necessary for introducing developers to the specified description language and therefore ending up in better acceptance rates.

In contrast to the bottom-up approach, the top-down approach does not define the description language at the beginning of the system implementation. This approach is more focused on the requirements of the target system and does not claim logical completeness. In a consequence a developed compliance checking tool in a top-down manner will usually not result in a generalized language. It can be easily reduced to well-established logics. Furthermore, significant for this development approach is a highly customized solution, which does not address universal applicability, e.g., programming language independence.

Usually the top-down development concurrently starts with choosing the target environment and defining architectural patterns for the target system in natural language. These tasks result in clearer requirements and a better understanding of the possibilities and restrictions, e.g., given by the target environment. The knowledge obtained in the identified requirements helps to derive an appropriate description language in order to express all previously identified patterns in a machine readable way. Finally, the source code model has to be extracted from the given source code and the verification engine has to be implemented. Afterwards the implementation is restricted to the customized description language and the necessarily extracted information of the source code model. Nevertheless, the implementation fulfills exactly the purpose of the described requirements.

The verification engine implemented in this Master's Thesis will follow the top-down or customizing development approach, as there is a strong endeavor to deploy the results in the processes of the customer represented by the Capgemini Holding GmbH.

3.2 Architectural Patterns

Patterns often describe occurring problems and how to solve them in a general sense. Thus, they are well-suited for structuring architectural principles at least in certain formalism. There are different well-known pattern libraries, for example, the patterns created by the Gang of Four (GoF) [13]. All patterns described by the GoF are design patterns, which solve common problems in OOP software implementations like the Observer or the Abstract Factory pattern. Nevertheless, all these patterns have nothing in common with architectural principles or patterns.

First of all architectural principles describe abstract rules for instance "a component should provide an external interface by always using an exception façade". For now this principle provides no notion of a solution resp. concrete implementation. However, architectural patterns provide a hint at least for a general solution. Therefore, architectural patterns are described by mainly two parts. First, there is the problem description, which is in this context described by an architectural principle. Second, the problem solution is provided as a combination of architectural design decisions and design patterns. Finally, in order to provide a machine readable solution for an automatic verification tool, the general pattern solution has to be refined to concrete implementation hints.

There are different approaches, which try to express solutions of pattern in a more or less formal way. On the one hand, there are some semi-formal techniques like describing the patterns in a restricted way using UML [14] [15]. On the other hand, there are also more formal techniques, which define their own pattern description language [16] [12]. Unfortunately, next to the approved logical and theoretical background of such languages, they are not intuitively usable by today's software engineers. This often is a side effect of a bottom up approach, as these pattern languages are defined

for a general purpose. Afterwards the concrete set of patterns has to be expressed in the defined language, which often is a non-trivial work.

The next chapter provides an overview of the pattern abstraction levels identified in the Register Factory®. Furthermore, a selection of extracted patterns will be described, which define the scope of this Master's Thesis.

4 Architectural Patterns

Architectural patterns can be defined in a wide spectrum of different levels of abstraction. Dependent on each level, it might be necessary to include more advanced verification techniques.

4.1 Abstraction Levels

In the knowledge domain of the Register Factory® patterns can be categorized in essentially three pattern abstraction levels, which have to be specifiable for a prototypical coverage of the system architecture. On the lowest level of abstraction, there are patterns for implementation facets which cover detailed implementation issues, e.g., object reference leaks or exception handling. Most commonly such patterns prevent possible implementation errors. Thus, enabling verification of patterns for implementation facets results in a generalized framework for fault detection.

On a higher level of abstraction, the source code is described by type dependencies, inheritance or other object-oriented implementation constructs. Furthermore—in scope of this Master's Thesis—dependencies enabled through the Spring-Framework have to be taken into account, as they are essentially for the Register Factory®. Patterns specified on this level of abstraction will be summarized in the following as patterns for program structures. In the context of a standardized architecture as the Register Factory®, patterns for program structures dominate the set of all available patterns, as they most commonly describe how to develop often occurring implementation entities.

Lastly the highest level of abstraction puts the notion of architectural components together with their related types, which are only specified on implementation level. In the following, these patterns will be called patterns for component structures. The verification of patterns for component structures is the most sophisticated part of this Master's Thesis. Only using the strict naming conventions of the Register Factory®—mentioned in Section 2.1.2—enables the verification of this kind of patterns. Nevertheless, patterns for component structures are very useful, as they can restrict communications across component borders.

4.2 Patterns for the Register Factory®

In the following, a representative set of all identified patterns will be presented for each pattern abstraction level. The patterns have been extracted from the Register Factory® architecture, which itself is only written in plain text. A pattern description will be divided into a problem part, solution part, and implementation part. The problem description states the general problem or task this pattern can be applied to. The solution part gives a general approach how to handle the problem in a common sense. This might include not only a general description, but also some detailed information

about the implementation, e.g., given by design patterns. At least the implementation part covers detailed information about how to apply the solution in the given context.

Values in angle brackets will represent a configurable value, which can be set for the current implementation or for each component itself. For detailed information see properties or variables of the pattern description language in Section 5.1.

4.2.1 Program Structures

4.2.1.1 Component-Façade

Problem: All communication across components has to be easily maintainable.

Solution: Each component should provide an interface for external usage. This interface should restrict the access to defined methods.

Implementation:

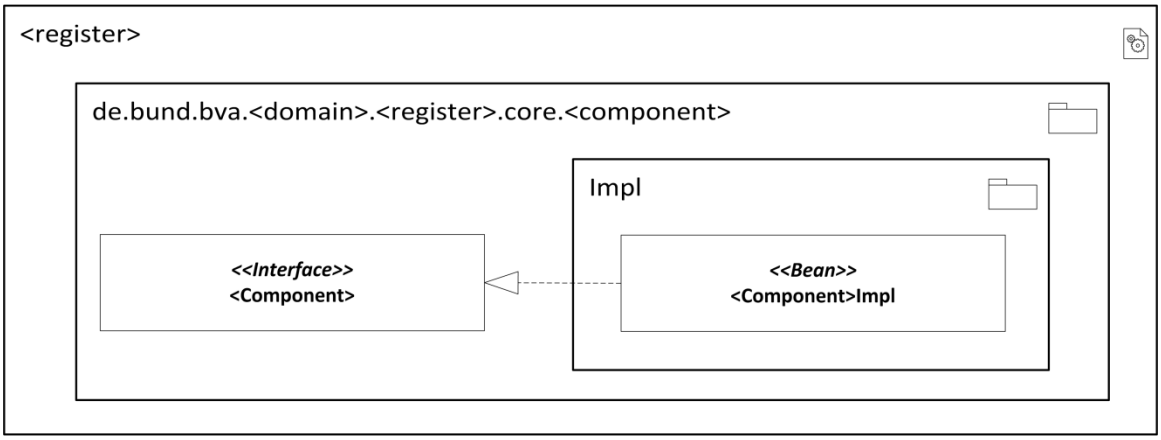

Figure 3: Patterns for program structures: Component-Façade

4.2.1.2 Exception-Façade for Component Interfaces

Problem: Accessing a component should always result in the defined behavior. It should be possible to access the component interface without potentially interrupting the system.

Solution: Each component interface should not throw any unhandled exceptions across the component border. This can be achieved by an exception façade implementation which has to handle all method forwards to the original component interface. For large systems it might be useful to establish a service layer for this behavior to gain better maintainability.

Implementation:

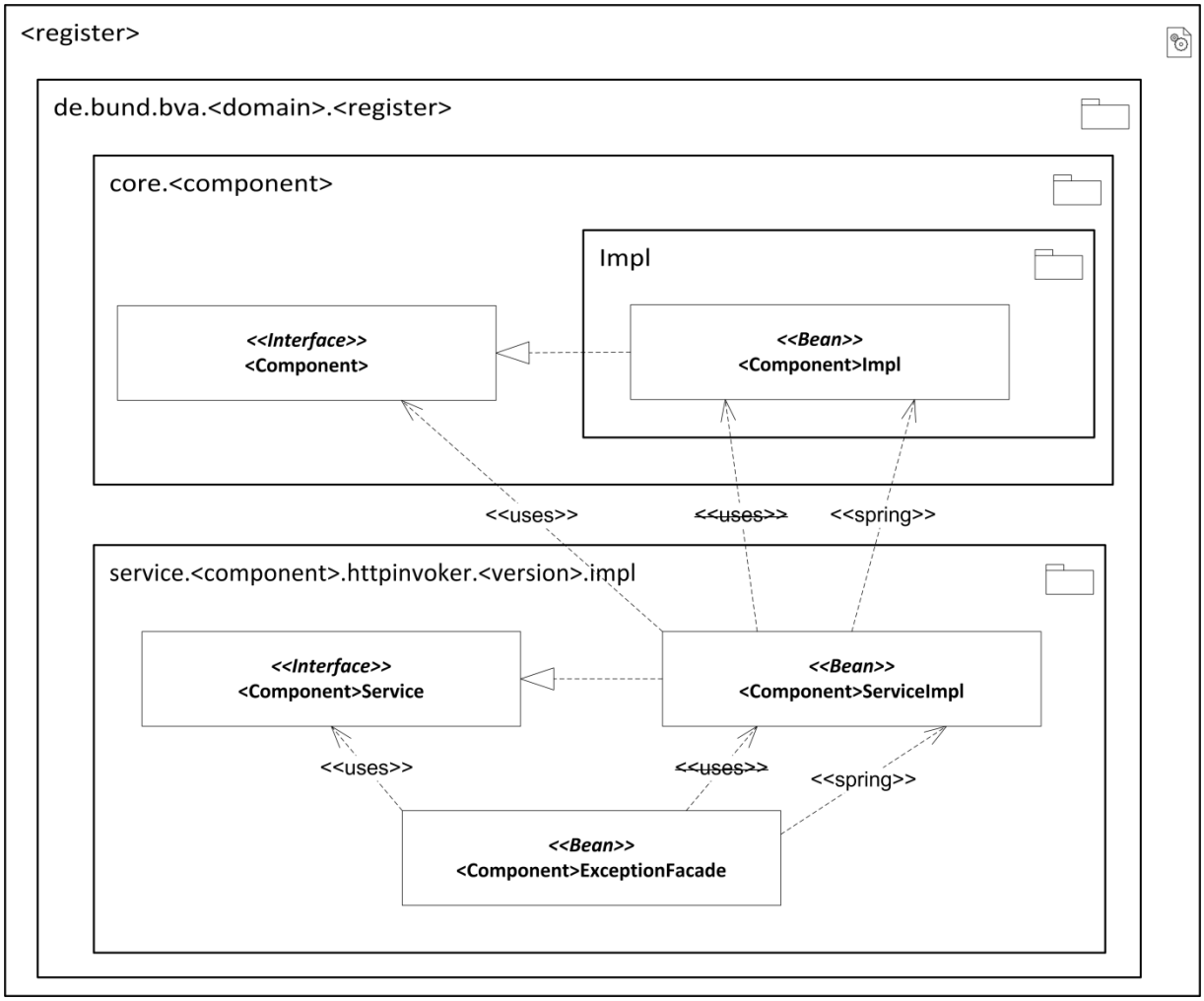

Figure 4: Patterns for program structures: Exception-Façade for Component Interfaces

4.2.1.3 HTTP-Interface for Components

Problem: Components can be distributed over different machines but they should be globally accessible for all services.

Solution: Each component which should be accessible for other components should provide an http interface.

Implementation:

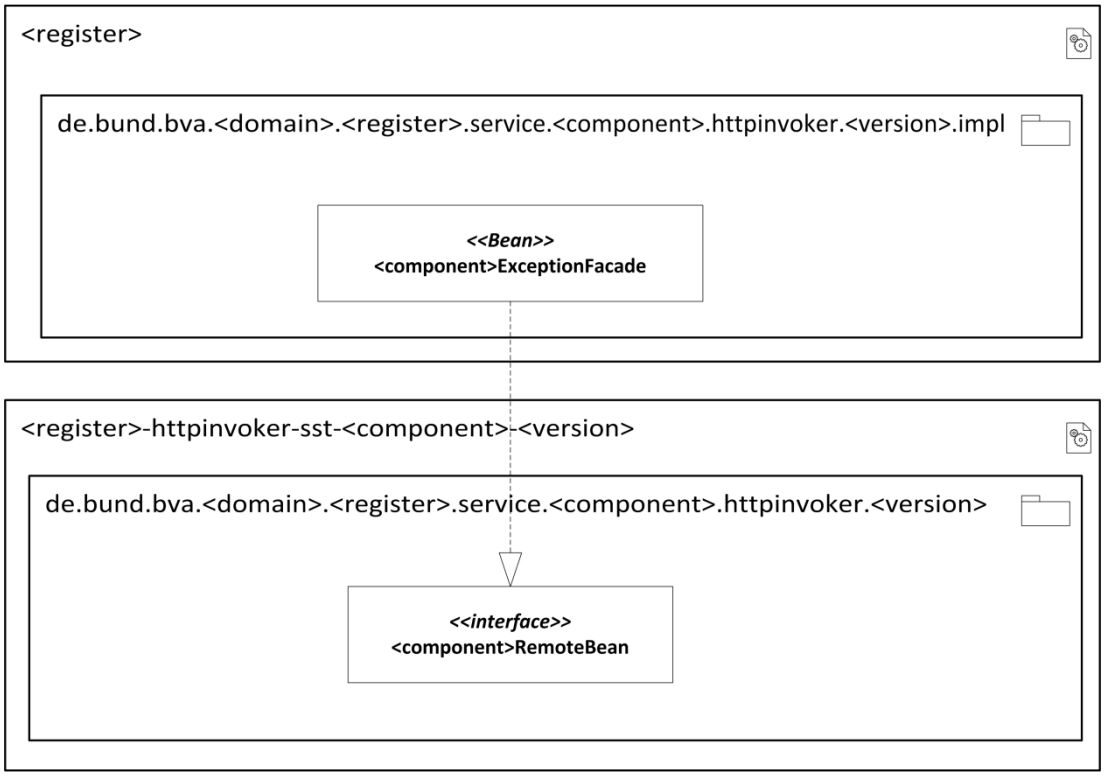

Figure 5: Patterns for program structures: HTTP-Interface for Components

4.2.2 Implementation Facets

4.2.2.1 Exception-Façade Security

Problem: An exception-façade should not throw any undefined exception over the component boundaries.

Solution: Every method call should be surrounded by a programming clause catching all exceptions that can occur and handle them by de-escalation or throwing another exception—defined by the boundary signature.

Implementation: Each method body in an exception-façade should be surrounded by a try-catch block catching at least the Java.lang.Throwable type. Only declared exceptions of the method signature can be thrown in a catch-block.

4.2.3 Component Structures

4.2.3.1 Access Restriction to Component-Façades

Problem: A component should be replaceable or maintainable easily without having an impact on other components

Solution: Each access to a component has to be forwarded via the component-façade, such that there are no dependencies of external types to the component internal types, despite to the component-façade itself.

Implementation: There should be no Java or spring dependency to any component internal class under the package de.bund.bva.<domain>.<register>.core.<component> despite the component-façade interface and implementation as stated in 4.2.1.1.

4.2.3.2 Data Sovereignty

Problem: All data have to be written in a controlled and consistent way. In addition the data structures should be maintainable easily.

Solution: Each component has to have data sovereignty over its own data structures. No other component should be allowed to access other components data.

Implementation: All component specific data structures are stored in the Java package de.bund.bva.<domain>.<register>.persistence.<component>. Only the correlated component to these data structures is allowed to have access on the objects directly.

5 Formal Pattern Specification

In order to be able to verify the identified patterns against real world implementations of the Register Factory®, a rudimentary pattern description language has been designed.

5.1 Pattern Description Language

5.1.1 Syntax

Listing 2 states the grammar for the pattern description language, designed to express the identified patterns mentioned in the last Section. All chars written in red are concrete language syntax elements. The **IDENT** token represents the identifier class of Java [17] as well as the **QUALIFIED_NAME** token, which represents the java definition of "qualified names". There are two grammar operators: single pipes which denote an exclusive "or" and squared brackets which imply an optional extension.

Listing 2: Grammar of the pattern description language

```
RULE = AND | OR | NOT | BRACKETS | PREDICATE | FORALL | EX
AND = RULE && RULE
OR = RULE || RULE
NOT = ! RULE
BRACKETS = ( RULE )

PREDICATE = isExceptionFacade(DATA [, DATA])
DATA = QUALIFIED_NAME | PROPERTY | VARIABLE
PROPERTY = <$IDENT>
VARIABLE = <IDENT>

FORALL = FORALL_SET | FORALL_COMPONENT
FORALL_SET = ForAll. [VARIABLE in SET] (RULE )
SET = {DATA, … , DATA}[(\SET) | (|SET)]
FORALL_COMPONENT = ForAll. [VARIABLE (in|not in) COMPONENT] (RULE )
COMPONENT = IDENT | (IDENT [(\SET) | (|SET)]) | (IDENT [(\COMPONENT) |
                (|COMPONENT)])

EX = EX_COMPILATIONUNIT | EX_PROJECT | EX_SUBTYPERELATION |
     EX_SPRINGDEPENDENCY | EX_SPRINGBEAN
EX_COMPILATIONUNIT = Ex. [(class | package)] (DATA [ inProject DATA])
EX_PROJECT = Ex. [project] (DATA)
EX_SUBTYPERELATION = Ex. [subtypeRelation] (DATA subtypeOf DATA)
EX_DEPENDENCY = Ex. [ (springDependency | javaDependency) ] (DATA to DATA)
EX_SPRINGBEAN = Ex. [springBean] ( (id=DATA && type=DATA | id=DATA | type=DATA) )
```

5.1.2 Semantics

In the following, the semantics and restrictions of each token of Listing 2 are described in detail. In order to understand all operator related issues, it should be mentioned anticipatory, that the developed verification engine of this Master's Thesis verifies each pattern on the scope of the current Eclipse workspace.

5.1.2.1 Root Node

RULE describes the top node of a pattern description. It represents a rule as part of an architectural pattern and always can be evaluated as a Boolean expression.

5.1.2.2 Simple Operators

AND | OR | NOT | BRACKETS

The tokens *AND*, *OR*, *NOT* are equal to the well-established logical operators. They can be combined in an arbitrary way, whereas the precedence of the operators are *NOT* > *AND* > *OR*. Besides, the brackets can surround a *RULE* and force the *RULE* to be evaluated before the formula around the brackets will be evaluated.

5.1.2.3 Complex Operators

PREDICATE

This token encloses all predicates resulting in a Boolean value and describing special verification logic. The only possible operator is currently the *isExceptionFacade*-operator.

The *isExceptionFacade*-operator has two parameters maximally. The first parameter states a type entity and the optional second one a method name. So the operator checks the given method of the given type for behaving as an exception façade. This means, that the given method cannot throw any unhandled exception when being executed. If the second parameter is omitted, all methods of the given type will be checked against this behavior. The result will be false whenever one or more methods do not behave as stated above, otherwise it results in true. How the evaluation is implemented in detail will be part of Section 6.2.3.

5.1.2.4 Universal quantifier

The ForAll-operator can be described as an encapsulated iterator. For all elements of a given set the rule will be evaluated. Therefor the specified variable holds the current element of the set, which then can be used within the rule of the ForAll-clause. The ForAll-operator results in a Boolean value, which will be true if all evaluations of the given *RULE* are valid. Otherwise it results in false. There are basically two different ways to work with the ForAll-operator.

FORALL_SET

The first possibility to use the ForAll-operator is given by the definition of iterators. The *VARIABLE* will be set to each value of the given *SET* once. In order to have the ability to exclude or include further variables, set operators have been established, which are described in Section 5.1.2.6.

FORALL_COMPONENT

Besides, there is the possibility to iterate over all types within a *COMPONENT*. Which types a *COMPONENT* consists of will be explained in detail in Section 5.1.2.6. For different analysis issues, it might be useful to include or exclude specific types or types of another *COMPONENT*. Therefore, it is possible to exclude or include a SET of types or all types of another *COMPONENT* via the *COMPONENT_ADAPTION* construct, see Section 5.1.2.6.

An automatically arising question of this syntax is the question of negation. In order to be able to get the set of all types, which are not within a given component, the keyword *in* should be simply replaced by *not in*. The universe, on which the set inverting is done, is described by the set of types implemented in the projects of the current workspace. Thus, types of the java language itself or types of external references will be excluded from further analysis. Besides, it is also possible to exclude namespaces from universe. Further information will be given by Section 6.

5.1.2.5 Existential quantifier

The Ex-operator is a wrapper operator for a lot of different existence verifications. In general the syntax of the existence operator can be described as *Ex. […] (…)*, whereby the contents of the squared brackets describe the type of entity, which should checked for existence. However, the contents in round brackets define the values, which identify the specific entity dependent on the type. The Ex-operator will also result in a Boolean value: true if the specified entity exists and false otherwise.

EX_COMPILATIONUNIT

The first existential operator *EX_COMPILATIONUNIT* verifies the existence of either a Java package *[package]* or a Java class *[class]*. The value of the existential clause has to be a package or class definition like *net.example.project* resp. *net.example.project.ClassName.Java*. Optionally, the related project can be specified, the entity has to be defined in. Then the existential clause has to be enriched by the keyword *inProject* after the class or package definition followed by the intended project name. If there is only a package or class definition given as parameter, the entity will be searched in the whole workspace. If there is an entity named as specified, *EX_COMPLIATIONUNIT* will result in true, otherwise in false. However, when a project is specified as additional condition, the

existential operator will result in true, only if there is an entity named as specified in the project being named as second parameter. In all other cases it results in false.

EX_PROJECT

This existential operator checks the existence of a Java project in the current workspace. The parameter defines the project name to be searched for. It will only return true if there is a project named as described.

EX_SUBTYPERELATION

Like checking the existence of programming entities as projects, packages or classes, it is also possible to check for the existence of a specific relation between a type A and a type B. Therefor the operator *EX_SUBTYPERELATION* takes two parameters, whereby it verifies whether the first parameter is a subtype of the second parameter. If so, the function will result in true, otherwise it will return false.

EX_DEPENDENCY

As the whole communication across the components of the Register Factory® is managed by the Spring-Framework, the existential operator *EX_DEPENDENCY* parameterized with *springDependency* has been added to the description language. This operator enables the verification of the spring communication channels by checking the parameter injection, which is automatically done by the Spring-Framework. Hence, the first parameter describes the type, in which the second type parameter will be injected, such that the first type can access the functionality of the second type. Important to know is, that the verification checks against the type injected and not against the type which is expected in the target attribute. Most likely the type of such attributes is a super type of the injected one. The operator *EX_DEPENDENCY* will only result in true if the expected dependency is injected by the Spring-Framework.

Next to the verification of Spring related dependency, it is also possible to check the existence of dependencies introduced by the programming level of the Java language itself. Therefore, the keyword *JavaDependency* has been established for *EX_DEPENDENCY*. A Java dependency will exist if the first type knows the second type. This again will be valid if the first type declares a variable of the second type or if the first type calls a method on an object of the second type. Analogous to the parameterization with *springDependency*, *EX_DEPENDENCY* parameterized with *JavaDependecy* will be valid if the Java dependency exists as stated before. Otherwise, it is false.

EX_SPRINGBEAN

Additionally to the last existence operator, *EX_SPRINGBEAN* checks the existence of the communication entities of the Spring-Framework called Spring-Beans. This enables the verification of

component interfaces, which should be provided by a given implementation using Spring. Currently, there are three possible signatures of EX_SPRINGBEAN in order to be able to check different Spring-Bean definitions allowed by the Spring-Framework. First, there is the value *id* which is a unique identifier for Spring-Beans, such that it can only be sufficient to pass this as parameter, e.g., *EX_SPRINGBEAN(id=BeanID)*. Second, there is the possibility not only to check the existence of the intended Spring-Bean, but also the Java type associated with the given Spring-Bean. This can be done by adding an additional parameter called *type*: *EX_SPRINGBEAN(id=BeanID && type=namespace.type)*. Nevertheless, it is also possible to define anonymous Spring-Beans, which have no id set. Exclusively for this type of beans, there is the possibility to check a Spring-Bean that represents a given type by only defining the *type* parameter in the signature. In the first case the existential operator *EX_SPRINGBEAN* will return true if a Spring-Bean is registered with the given *id* in the Spring configuration. In the second case the operator will result in true if the Spring-Bean with the given *id* is also associated with the given type. The last case will result in true if there is an anonymous bean for the given *type* declaration. In all other cases *EX_SPRINGBEAN* result in false.

5.1.2.6 Data Types

SET

As mentioned in the description of the *FORALL*-operator, sets are used for defining values over which can be iterated. This can be useful when verifying different components against the same behavior by iterating of their names and generate package and class names from this information. A set can be adapted by two operators: The operator '\' removes all values of the second set from the first set, the operator '|' unifies both sets.

COMPONENT

In order to enable the notion of components, another level of abstraction has been established. Each component describes a set of types identified by a set of namespaces, which the component consists of. The components also have to be defined in a globally valid dictionary. In order to get a notion of how to specify components in detail, Section 6.4 introduces the data management. Similar to the *SET* definition, *COMPONENT* definitions can be adapted using the set operators defined above. Furthermore, *COMPONENT* definitions can be adapted by other *COMPONENT* definitions. Then the set operators will be executed on the set of types the components consist of.

VARIABLE

Like the *SET* construct, also the *VARIABLE* construct currently is only necessary when using the *FORALL*-operator. It defines the iterating variable over a given *SET*. Variables will be substituted by their values before the evaluation. According to the formatting of the variable name in each usage of

the variable, the value will be formatted as follows. It has exposed, that there are three necessary formats, which have to be supported. The normal formatting, e.g., <variable> inserts the value as defined. Besides, there are two possibilities of reformatting the value. Either the value can be capitalized by <Variable> or completely written in upper case by <VARIABLE>. This enables adaptions to variables according to their occurrences in class or package names.

PROPERTY

The concept of properties is a rudimentary approach for pattern parameterization. Properties are syntactic similar to variables besides the $ sign at the beginning of the property name. The property values are stored in a globally valid dictionary in order to enable reuse, e.g., for namespaces and to have a single point of edit for adaptions to be made. Contrary to variables, properties are substituted recursively in order to enable encapsulated property usages in properties. More technical issues about the definition of properties will be discussed in Section 6.4.

DATA

The DATA construct is a wrapper for all potential data containers as variables or properties.

5.2 Pattern Specification

After defining patterns and a therefor well-suited pattern description language, the following lists the specification of each introduced pattern transformed into the previously defined pattern description language. In addition the properties and components defined for the patterns are described in the next Section. The properties are mainly used to have short descriptions and enable a high degree of adaptability of the patterns.

Remark: It might be possible that class names are specified in German language as the Register Factory® is only defined for the German governmental software development.

5.2.1 Pattern Preferences

5.2.1.1 Properties

```
external_components={auskunft,datenpflege,meldung,mitteilungen}

registerProject=register_xyz
register=registerxyz
rootPackage=de.bund.bva.xyz.<$register>

component_core=<$rootPackage>.core.<component>
componentInterface=<$component_core>.<Component>

component_service_10=<$rootPackage>.service.<component>.httpinvoker.v1_0
component_service_11=<$rootPackage>.service.<component>.httpinvoker.v1_1

component_http_project_10=<$registerProject>-httpinvoker-sst-<component>-v1.0
component_http_project_11=<$registerProject>-httpinvoker-sst-<component>-v1.1

mitteilungen_accessRestrictions_validExceptionalCases=
{<$rootPackage>.core.meldung.impl.MeldungMitteilungErzeuger,
<$rootPackage>.core.fristenkontrolle.impl.FristenkontrolleMitteilungErzeuger,
<$rootPackage>.core.datenpflege.impl.DatenpflegeMitteilungErzeuger,
<$rootPackage>.core.auskunft.impl.AuskunftMitteilungErzeuger,
<$rootPackage>.core.common.AbstractMitteilungErzeuger}
```

5.2.1.2 Component Definitions

```
auskunft=<$rootPackage>.core.auskunft
datenpflege=<$rootPackage>.core.datenpflege
meldung=<$rootPackage>.core.meldung
mitteilungen=<$rootPackage>.core.mitteilungen,<$rootPackage>.batch.mitteilungen

auskunft_persistence=<$rootPackage>.persistence.auskunft
datenpflege_persistence=<$rootPackage>.persistence.datenpflege
meldung_persistence=<$rootPackage>.persistence.meldung
mitteilungen_persistence=<$rootPackage>.persistence.mitteilungen

auskunft_service=<$rootPackage>.service.auskunft
datenpflege_service=<$rootPackage>.service.datenpflege
meldung_service=<$rootPackage>.service.meldung
mitteilungen_service=<$rootPackage>.service.mitteilungen
```

5.2.1.3 Universe Adaptations

```
excludeNamespaces=test.de.bund.bva
```

5.2.2 Program Structures

5.2.2.1 Component-Façade

```
ForAll. [<component> in <$external_components>] (
   Ex. [class] (<$component_core>.<Component>.java inProject <$registerProject>)
   && Ex. [class] (<$component_core>.impl.<Component>Impl.java inProject
      <$registerProject>)
   && Ex. [subtypeRelation] (<$component_core>.impl.<Component>Impl subtypeOf
      <$component_core>.<Component>)
   && Ex. [springBean] (id=<component> &&
      type=<$component_core>.impl.<Component>Impl)
)
```

5.2.2.2 Exception-Façade for Component Interfaces

```
ForAll. [<component> in <$external_components>\{auskunft}] (
   Ex. [class] (<$component_service_10>.impl.<Component>Service.java inProject
      <$registerProject>)
   && Ex. [class] (<$component_service_10>.impl.<Component>ServiceImpl.java
      inProject <$registerProject>)
   && Ex. [class] (<$component_service_10>.impl.<Component>ExceptionFassade.java
      inProject <$registerProject>)
   && Ex. [springBean] (type=<$component_service_10>.impl.<Component>ServiceImpl)
   && Ex. [springBean] (id=<component>ExceptionFassade &&
      type=<$component_service_10>.impl.<Component>ExceptionFassade)

   && Ex. [subtypeRelation] (<$component_service_10>.impl.<Component>ServiceImpl
      subtypeOf <$component_service_10>.impl.<Component>Service)

   && Ex. [springDependency] (<$component_service_10>.impl.<Component>ServiceImpl
      to <$component_core>.impl.<Component>Impl)
   && !Ex. [javaDependency] (<$component_service_10>.impl.<Component>ServiceImpl
      to <$component_core>.impl.<Component>Impl)
   && Ex. [javaDependency] (<$component_service_10>.impl.<Component>ServiceImpl to
      <$component_core>.<Component>)

   && Ex. [springDependency]
      (<$component_service_10>.impl.<Component>ExceptionFassade to
      <$component_service_10>.impl.<Component>ServiceImpl)
   && !Ex. [javaDependency]
      (<$component_service_10>.impl.<Component>ExceptionFassade to
      <$component_service_10>.impl.<Component>ServiceImpl)
   && Ex. [javaDependency]
      (<$component_service_10>.impl.<Component>ExceptionFassade to
      <$component_service_10>.impl.<Component>Service)
)
```

The component “auskunft” will be checked by only replacing the <$component_service_10> with the <$component_service_11> property as this service already is implemented in version 1.1.

5.2.2.3 HTTP-Interface for Components

```
ForAll. [<component> in <$external_components>\{auskunft}] (
   Ex. [class] (<$component_service_10>.impl.<Component>ExceptionFassade.java
      inProject <$registerProject>)
   && Ex. [class] (<$component_service_10>.<Component>RemoteBean.java inProject
      <$component_http_project_10>)
   && Ex. [subtypeRelation]
      (<$component_service_10>.impl.<Component>ExceptionFassade subtypeOf
      <$component_service_10>.<Component>RemoteBean)
)
&& ForAll. [<component> in {auskunft}] (
   Ex. [class] (<$component_service_11>.impl.<Component>ExceptionFassade.java
      inProject <$registerProject>)
   && Ex. [class] (<$component_service_11>.<Component>RemoteBean.java inProject
      <$component_http_project_11>)
   && Ex. [subtypeRelation]
      (<$component_service_11>.impl.<Component>ExceptionFassade subtypeOf
      <$component_service_11>.<Component>RemoteBean)
)
```

5.2.3 Implementation Facets

5.2.3.1 Exception-Façade Security

```
ForAll.[<component> in <$external_components>\{auskunft}] (
   isExceptionFacade(<$component_service_10>.impl.<Component>ExceptionFassade
      .java)
)
&& ForAll.[<component> in {auskunft}] (
   isExceptionFacade(<$component_service_11>.impl.<Component>ExceptionFassade
      .java)
)
```

5.2.4 Component Structures

5.2.4.1 Access Restriction to Component-Façades

```
ForAll.[<component> in <$external_components>\{mitteilungen}] (
   ForAll.[<type> not in <component>|<component>_service] (
      ForAll.[<componentType> in <component>\{<$componentInterface>}] (
         !Ex.[javaDependency](<type> to <componentType>)
      )
   )
)
&& ForAll.[<component> in {mitteilungen}] (
   ForAll.[<type> not in <component> | <component>_service |
   <$mitteilungen_accessRestrictions_validExceptionalCases>] (
      ForAll.[<componentType> in <component>\{<$componentInterface>}] (
         !Ex.[javaDependency](<type> to <componentType>)
      )
   )
)
```

5.2.4.2 Data Sovereignty

```
ForAll.[<component> in <$external_components>\{mitteilungen}] (
   ForAll.[<type> not in <component>|<component>_persistence] (
      ForAll.[<persistenceType> in <component>_persistence] (
         !Ex.[javaDependency](<type> to <persistenceType>)
      )
   )
)
&& ForAll.[<component> in {mitteilungen}] (
   ForAll.[<type> not in <component> | <component>_persistence |
   <$mitteilungen_accessRestrictions_validExceptionalCases>] (
      ForAll.[<persistenceType> in <component>_persistence] (
         !Ex.[javaDependency](<type> to <persistenceType>)
      )
   )
)
```

6 Architecture Verification Engine (AVE)

The Architecture Verification Engine is a prototype and thus, driven by very customized and specific requirements. Nevertheless, it implements a wide spectrum of different architectural structures and entities.

As an eclipse plug-in, the Architecture Verification Engine currently focuses on the code developed in the IDE. Therefore, the scope of source code to be verified is defined by the current Eclipse workspace, it is running in. Furthermore, only types are considered, which sources are defined in an open project of the current workspace, such that external libraries and other Java classpath imports are not in scope. In summary, the type universe is defined by all types in the workspace, which exist as source files in source folders of open java projects. Unfortunately it turned out, that the type universe has to be designed as a customizable value in order to meet the requirements of large companies as the Capgemini Holding GmbH. Test classes can be implemented in the same project as the source code, they are testing. This leads to problems, when checking, e.g., access rights of components using a set inversion over the type universe, since test classes generally should not be restricted to any interfaces. Thus, a universe adaption mechanism has been integrated, which enables the exclusion of namespaces from universe. Where to define such adaptations will be described in Section 6.4.

After defining the surrounding preconditions, the AVE works in, the next Section introduces the architecture of the AVE.

6.1 Architecture

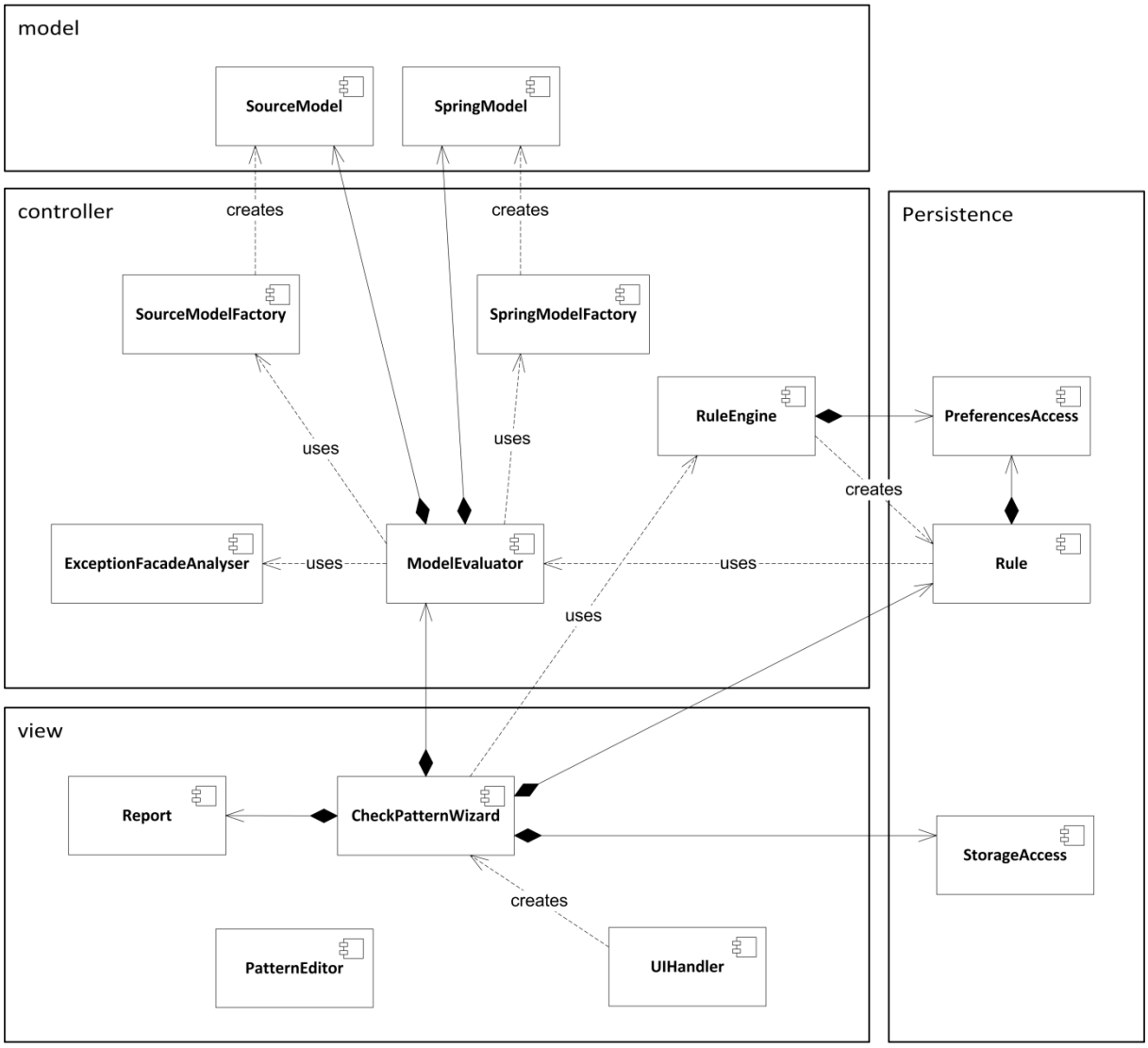

Figure 6: Architecture of the Architecture Verification Engine

The architecture of the AVE can basically be divided into four layers as shown in Figure 6. The three layers model, controller and view are based on the well-known Model-View-Controller pattern of the GoF. Besides, there is the persistence layer containing components, which have to be accessible from more than one layer. On the one hand it contains file access logic e.g. for pattern and preferences files, on the other hand all rule transfer objects are located in this layer.

Driven by the Eclipse framework, the *UIHandler* defines the entry point for the system. The handler will be invoked by the click event on the newly established functionality of the PackageExplorer, see in Section 6.5. After invocation, the handler creates the *CheckPatternWizard* for further user interactions. If the pattern verification is triggered, the *CheckPatternWizard* will create the *ModelEvaluator* and let the *RuleEngine* parse the given pattern into the rule transfer objects. Furthermore the rule will be invoked to evaluate itself by using the newly created *ModelEvaluator*.

The *ModelEvaluator* is responsible of creating of all necessary models and provides a high level model evaluation interface. Next to the *UIHandler*, there is also a *PatternEditor*, which is registered in the Eclipse framework and will be automatically used, if a file is opened ending on *.pattern.

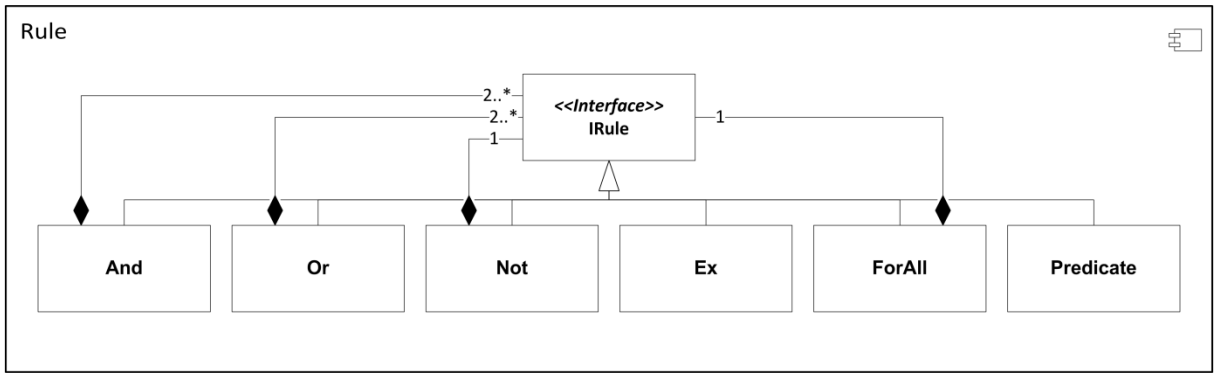

Figure 7: Rule transfer objects

The rule transfer objects are structured in a natural object-oriented way, as described in Figure 7. All rule entities have to implement the *IRule* interface, which provides a generic access type and assures the functionality of evaluation. The *And* and *Or* operator are binary operators, which can be concatenated to arbitrary length, such that one of these operators can contain two or more *IRules* to be evaluated in concatenation. Similar to *Not*, the *ForAll* operator also is a kind of unary operator with respect to the *IRule* interface as parameters. Therefore both have to have exactly one *IRule* as evaluation parameter. The last two operators *Ex* and *Predicate* are the elementary operators, which can be evaluated without evaluating any sub ordered *IRule*.

6.2 Verification Processing

When invoking the Architecture Verification Engine to check the current implementation against a specified pattern, first of all the plain text rule will be preprocessed into the rule transfer objects. This step assures the right syntax of the pattern description and extracts all necessary information for the evaluation. Next, the AVE has to set up the source code and spring model of the current available implementation. How this is done in detail is described in the next two following sections. After that, the evaluation can be easily done on all extracted data or the Abstract Syntax Tree given by the JDT framework.

6.2.1 Source Code Model Extraction

The JDT framework uses the Abstract Syntax Tree for static compilation of java source code. In order to benefit from this build in static compiler, the source code model of the AVE is extracted from the AST. For the current state of this Master's Thesis, there are two mandatory aspects to be extracted in order to build the source code model.

On the one hand, all types in scope have to be extracted. Fortunately, the AST also provides a very simple mechanism to get all super types of each type, such that this information can also be extracted easily. According to the AST JLS4 two different AST Nodes have to be visited for a complete coverage of all type definitions. First the *TypeDeclaration* has to be visited which combines the declarations of classes and interfaces.

```
TypeDeclaration:
              ClassDeclaration
              InterfaceDeclaration
 ClassDeclaration:
      [ Javadoc ] { ExtendedModifier } class Identifier
                      [ < TypeParameter { , TypeParameter } > ]
                      [ extends Type ]
                      [ implements Type { , Type } ]
                      { { ClassBodyDeclaration | ; } }
 InterfaceDeclaration:
      [ Javadoc ] { ExtendedModifier } interface Identifier
                      [ < TypeParameter { , TypeParameter } > ]
                      [ extends Type { , Type } ]
                      { { InterfaceBodyDeclaration | ; } }
```

The second node, which has to be visited, is the *EnumDeclaration* declaring enum types.

```
EnumDeclaration:
     [ Javadoc ] { ExtendedModifier } enum Identifier
         [ implements Type { , Type } ]
         {
         [ EnumConstantDeclaration { , EnumConstantDeclaration } ] [ , ]
         [ ; { ClassBodyDeclaration | ; } ]
         }
```

On the other hand, type dependencies have to be evaluated separately. Therefore, the AVE visits all variable declarations and method invocations of each type. A type will have a dependency to another type if the first type declares a variable of the second type or if the first type calls a method on the second type. The last case also includes concatenated method calls which may leak object references in between of the method call chain. For the analysis of such dependencies there are also two SST nodes, which have to be visited to gain all information. The first one is the *VariableDeclarationStatement* node, which combines all variable declarations in itself.

```
VariableDeclarationStatement:
    { ExtendedModifier } Type VariableDeclarationFragment
        { , VariableDeclarationFragment } ;
```

The second node is the *MethodInvocation* node, which represents all method invocations in the Java source code.

```
MethodInvocation:
     [ Expression . ]
         [ < Type { , Type } > ]
         Identifier ( [ Expression { , Expression } ] )
```

6.2.2 Spring Model Extraction

Beside the source code model, there is also the necessity for deriving a spring model, containing all available Spring Beans and their dependencies. All information about the Spring configuration is stored in the Spring configuration file ".springBeans" in the root of a project folder in Eclipse. Therefore, the spring parser of the AVE searches in each java project of the workspace the Spring configuration file. If it exists, it will be parsed using the XMLBeans framework of Apache, such that the object representation of the Spring configuration is generated on the bases of the Spring Beans XSD version 2.5. All referenced Spring configuration files (beyond the tag <configs>) will be treated for further information extraction.

The extraction of all Java Beans will be done in a recursive manner, as it is possible to set Spring Beans as a dependency not only by stating the Bean's id, but also by declaring a new anonymous Bean directly. For available patterns needs, there are currently three different aspects which have to be extracted from the Spring Bean configuration. First, each Bean should be identifiable by its id if existent (not for anonymous Bean's). Second, each Bean has to have an implementation type, which describes the bridge from the Spring framework to the java implementation. This information will be highly important, if patterns prescribe a dependency between two types established over the Spring framework. The last aspect describes the type dependencies defined in Spring. They have to be analyzed by using the information of each bean and the set bean dependencies. Therefore, the identification of Beans using their ids is necessary, as described in the first step.

6.2.3 Realization of Predicate isExceptionFacade

Beside all other operators, which are only describing logical well-known predicates, the isExceptionFacade is currently the only one covering special implementation logics. As similarly to the source code extraction, it is necessary to visit the Abstract Syntax Tree again. This time the implementation has to cover all source code of a given type or at least of the given method defined in a given type.

For this analysis, the AST entry point is the *MethodDeclaration* node which combines method and constructor declarations.

```
MethodDeclaration:
    [ Javadoc ] { ExtendedModifier }
                [ < TypeParameter { , TypeParameter } > ]
        ( Type | void ) Identifier (
        [ FormalParameter
                  { , FormalParameter } ] ) { [ ] }
        [ throws TypeName { , TypeName } ] ( Block | ; )
 ConstructorDeclaration:
    [ Javadoc ] { ExtendedModifier }
                [ < TypeParameter { , TypeParameter } > ]
        Identifier (
                [ FormalParameter { , FormalParameter } ] )
        [throws TypeName { , TypeName } ] Block
```

As shown by the grammar, the body of a method is defined by a *Block,* whereas *Blocks* are defined by a set of *Statements*. Unfortunately, the *Statement* and also often occurring *Expression* type cannot be accessed in a generic way, such that it is not possible to handle all different *Statements* or *Expressions* simultaneously. Therefore, the current state of implementation only evaluates a fraction, which exposes to be sufficient for two implementations of registers. As the most names of the available *Statements* and *Expressions* are self-explanatory, the following gives an overview of all entities, which will be handled by the AVE (green). The occurrence of other *Statements* or *Expressions* will lead to the invalid result of isExceptionFacade.

```
Statement:
    AssertStatement,
    Block,
    BreakStatement,
    ConstructorInvocation,
    ContinueStatement,
    DoStatement,
    EmptyStatement,
    ExpressionStatement,
    ForStatement,
    IfStatement,
    LabeledStatement,
    ReturnStatement,
    SuperConstructorInvocation,
    SwitchCase,
    SwitchStatement,
    SynchronizedStatement,
    ThrowStatement,
    TryStatement,
    TypeDeclarationStatement,
    VariableDeclarationStatement,
    WhileStatement
    EnhancedForStatement
```

```
Expression:
    Annotation,
    ArrayAccess,
    ArrayCreation,
    ArrayInitializer,
    Assignment,
    BooleanLiteral,
    CastExpression,
    CharacterLiteral,
    ClassInstanceCreation,
    ConditionalExpression,
    FieldAccess,
    InfixExpression,
    InstanceofExpression,
    MethodInvocation,
    Name,
    NullLiteral,
    NumberLiteral,
    ParenthesizedExpression,
    PostfixExpression,
    PrefixExpression,
    StringLiteral,
    SuperFieldAccess,
    SuperMethodInvocation,
    ThisExpression,
    TypeLiteral,
    VariableDeclarationExpression
```

Whereas the *ExpressionStatement* only is a wrapper type for embedding *Expressions* into *Statements*, the notion of handling this type refers to the list of Expressions, which can be evaluated at the current state. It is important to note, that there are currently only three syntax elements leading to an invalid result of isExceptionFacade: first, method invocations and constructor invocations outside of a try-catch block and second, the absence of the catch(Throwable) clause in an existing try-catch block. There is a high potential for many improvements, as not all Statements and Expressions have been threatened yet. Nevertheless it has been shown, that it is possible to establish such a predicate, which covers the code on a very high level of detail.

6.3 Handling of Components

The namespaces defined by the naming conventions of the Register Factory® encode all necessary information about components and component related implementation entities. Using this as a framework for the realization of the component abstraction, the notion of components is first of all reduced to a set of namespaces. According to that, a component is defined by all implementation units defined in a set of different namespaces. One step closer to the implementation, the focus will be on types. Now it is possible to relate a component name to a set of types, which are implemented in the defined namespaces of the component. Consequently, in the context of the Register Factory® components can be handled as sets.

This component abstraction generally works fine. Nevertheless, there are also exceptions breaking with this concept. One example for this is the verification of the data sovereignty, as there are factory types directly accessing data objects of other components. As such exceptional behavior cannot be covered using the abstraction through namespaces, the patterns, e.g., for access rights have to be adapted to their needs using set operators.

6.4 Data Management

The AVE stores all information about patterns, component definitions and properties in a project called “Architectural Patterns”. First invocation of any AVE functionality would create all necessary files, e.g., the project and preferences files if they have not been existent yet. Within project one file is created for each pattern specification using the pattern’s name and the file ending “.pattern” as shown in Figure 8. The component definitions and global properties are stored separately in a subfolder called “preferences”. Currently there are three different preferences files.

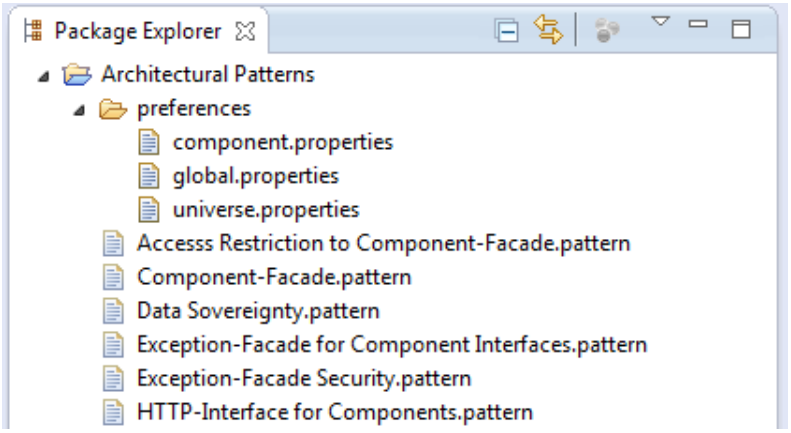

Figure 8: File management of the AVE

The “component.properties” file stores information of the namespaces of each component. The contents have to be defined as a Java property file, such that each line describes a property and the syntax “key”=”value” holds in each line. The value of a component definition should be described by a comma separated list of different namespaces. Thus, all types within the defined namespaces will be part of the component identified by its key.

In the “global.properties” file all properties have to be defined, which are used in patterns or component definitions. This file also uses the Java property file syntax. Properties for the AVE can be recursively defined, such that a generic property can be refined by other properties. This helps maintaining the property base and also leads to very adaptable patterns, which can be easily ported to other code bases.

The “universe.properties” file has currently only one predefined value to be set. The “excludeNamespaces” value can be set to a comma separated list of namespaces, which have to be excluded from the type universe. Types excluded from universe are also not in scope of the pattern verification.

6.5 Usage

Installing the Architecture Verification Engine enables a new functionality called “Check architectural Patterns” in the context menu of the PackageExplorer. The PackageExplorer is part of the JDT interface and provides access to all file system resources necessary for developing in Java. Right after installation, this is the unique entry point in order to work with the AVE. Invoking this command, the pattern selection mask shows up.

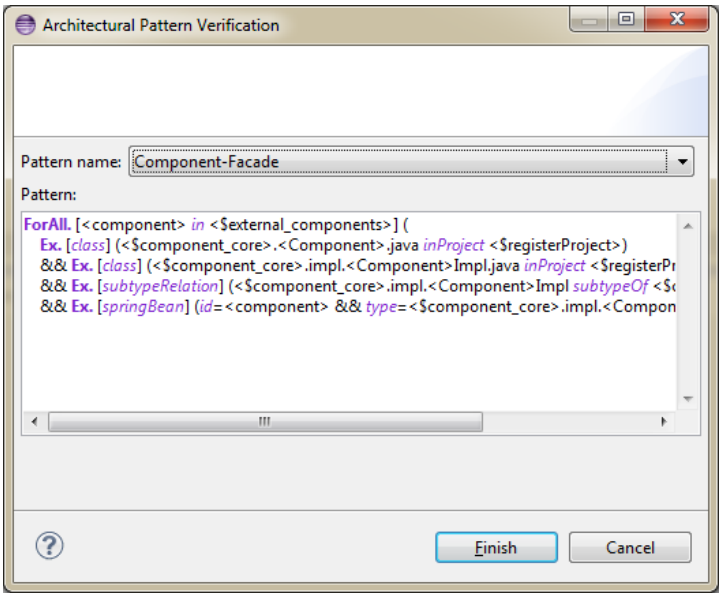

Figure 9: Selection mask for pattern verification

It provides a dropdown box to select a previously defined pattern, which will be displayed in the text box right below after selection. This text box is read only in order to see the pattern specification before verifying the workspace against it. After selecting a pattern in the dropdown box, it can be checked against the current workspace by clicking the “Finish” button at the bottom. The results will show up in a new window a few seconds later. The results window initially only displays atomic checkable entities—all leafs of the syntax tree—which cannot be checked successfully. In some cases, it might be necessary to display additional information of the cause of the failed check. Thus, a checkbox is provided on the bottom of the window, which can be enabled to get further information right below each entry. This feature is not generally implemented for each checkable entity, as it has not been in focus of the evaluation. Nevertheless, it has been nice to know for developers, where, for example, access rights are ignored in detail. The filter mechanism, initially only displaying invalid results, is called “Only display invalid results” and can be found at the left bottom of the results window. Disabling this filter will also bring valid results into focus.

The creation of patterns has to be done manually by creating a new file in the “Architectural Patterns” project, whereby the file has to end on *.pattern in order to use the provided pattern editor automatically. After creation, patterns can be modified using the pattern editor. It provides a rudimentary syntax highlighting. Thus, all correctly spelled keywords will be highlighted, such that it supports the architect defining architectural patterns for the Architecture Verification Engine.

7 Conclusions

As one of the biggest cost drivers of today's Software Engineering, maintenance has to be supported by effective and efficient usable tools. One tool supporting compliance checking between an architecture description and its implementation has been prototypically developed in this Master's Thesis as part of an evaluation.

The evaluation addresses the feasibility of automatically checking architectural compliance on any implementation in the context of the Register Factory®. Therefore, a set of architectural patterns had been derived from the latest architecture description of the Register Factory®. The patterns were selected as is in order to gain a representative set of patterns according to their abstraction level. In a next step the patterns were translated into a machine readable pattern description language, which had been developed just for the purpose of evaluation. Afterwards, these patterns were set as the requirements for the Architecture Verification Engine—the compliance checking tool developed in this Master's Thesis.

The implementation of the AVE shows, that it is possible to verify architectural patterns against a given implementation automatically. Furthermore, the evaluation encompasses patterns of different architectural abstraction levels. Thus, it could be shown, that there are architectural aspects on each abstraction level which can be checked by using a compliance checking tool. Especially the usage of strict naming conventions enables also architectural verification on high component abstraction, as naming conventions are able to encode architectural context information into the implementation.

Nevertheless it turned out, that especially by checking access rights or data sovereignty, there are often exceptions from the architectural description. Since due to special requirements, instantiations of the Register Factory® have to adapt the meta-architecture to their needs. In order to focus on this problem in an efficient way, a more advanced pattern management system has to be developed, e.g., enabling pattern inheritance as described in the next Section. Furthermore, unlike to patterns for component structures or patterns for program structures, it is not easy to define new patterns for implementation facets without adapting the developed pattern description language with another predicate.

Apart from the original focus of this Master's Thesis—the feasibility analysis for an architecture compliance checking tool for the Register Factory®—further knowledge has been obtained about the possibility of an effective and efficient usable integration into the Eclipse IDE. The rudimentary developed user interface is well-integrated in the IDE, although it only consists of a few elements and it can support the user defining architectural patterns without getting to know a new environment.

7.1 Future Work

In order to manifest the obtained knowledge of the prototypical implementation of this Master's Thesis, further evaluations should be performed on a larger set of patterns. In addition, it might be useful to develop a more generalized pattern description language, as the current one has been developed for expressing the defined patterns of Section 5.2.

Beside the development of a more generic, but also readable pattern description language, there are other requirements driven by usability aspects of today's enterprises. The possibility of defining patterns in a machine readable way does not solve all maintenance problems of architectural compliance. It rather pulls them onto a new level, describing the compliance between the machine readable pattern representation and the plain architecture description. Therefore, in order to provide a comprehensive solution for the initial maintenance problem of keeping the implementation compliant to the architecture description, a pattern management system needs to be developed. Imaginable requirements might be enabling pattern inheritance or version management for such a management system. This will help to keep the maintenance effort for patterns in acceptable ranges. Recently, there might be the outlook for a pattern based automatic migration system, which enables code migration due to architecture changes, but this will also include the big topic of code generation from patterns.

Bibliography

[1] A. V. Deursen, P. Klint und C. Verhoef, „Research Issues in the Renovation of Legacy Systems," in *Proceedings of the Second Internationsl Conference on Fundamental Approaches to Software Engineering*, Amsterdam, Springer, 1999, pp. 1-21.

[2] R. Leonhard, „BVA Internet: Publikationen," 22 February 2012. [Online]. Available: http://www.bva.bund.de/cln_117/nn_2160356/DE/Aufgaben/Abt__I/RegisterFactory/Publikationen/Dokumente/Whitepaper__Register__Factory,templateId=raw,property=publicationFile.pdf/Whitepaper_Register_Factory.pdf. [Zugriff am 03 July 2012].

[3] L. Vogel, „Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model - Tutorial," 8 August 2012. [Online]. Available: http://www.vogella.com/articles/EclipseJDT/article.html. [Zugriff am 13 September 2012].

[4] T. Kuhn und O. Thomann, „Eclipse Corner Article: Abstract Syntax Tree," 20 November 2006. [Online]. Available: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html. [Zugriff am 13 September 2012].

[5] J. Aldrich, C. Chambers und D. Notkin, „ArchJava: connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering. ICSE*, Orlando, FL, USA, 2002.

[6] N. Medvidovic und R. N. Taylor, „A Classification and Comparison Framework for Software Architecture Description Languages," in *IEEE Transactions on Software Engineering*, 2000.

[7] M. L. Bernardi und G. A. Di Lucca, „Model-driven Detection of Design Patterns," in *Proc of 26th IEEE International Conference on Software Maintenance ICSM*, Timisoara, 2010.

[8] A. De Lucia, V. Deufenia, C. Gravino und M. Risi, „An Eclipse plug-in for the Detection of Design Pattern Instances through Static and Dynamic Analysis," in *Proc of 26th IEEE International Conference on Software Maintenance ICSM*, Timisoara, 2010.

[9] J. Dong, Y. Sun und Y. Zhao, „Design pattern detection by template matching," in *Proceedings of the 2008 ACM symposium on Applied computing - SAC*, New York, 2008.

[10] L. Passos, R. Terra, M. T. Valente, R. Diniz und N. Mendonca, „Static Architecture-Conformance Checking: An Illustrative Overview," *IEEE Software,* Bd. 27, Nr. 5, pp. 82-89, 2010.

[11] J. Knobel und D. Popescu, „A Comparison of Static Architecture Compliance Checking Approaches," in *The Working IEEE/IFIP Conference on Software Architecture*, Mumbai, Maharashtra, India, 2007.

[12] S. Herold, *Architectural Compliance in Component-Based Systems Foundations, Specification, and Checking of Architectural Rules,* TU Clausthal, 2011.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: elements of reusable object-oriented software, Amsterdam: Addison-Wesley Longman, 1994.

[14] N. Soundarajan and J. O. Hallstrom, "Responsibilities and rewards: specifying design patterns," in *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, 2004.

[15] J. K. H. Mak, C. S. T. Choy und D. P. K. Lun, „Precise Modeling of Design Patterns in UML," in *Proceedings. 26th International Conference on Software Engineering*, Edinburgh, 2004.

[16] A. H. Eden, Y. Hirshfeld und K. Lundqvist, „LePUS – Symbolic Logic Modeling of Object Oriented Architecture: A Case Study," in *NOSA '99 Second Nordic Workshop on Software Architecture*, University of Karlskrona/Ronneby, 1999.

[17] J. Gosling, B. Joy, G. Steele, G. Bracha and A. Buckley, "The Java™ Language Specification," 27 July 2012. [Online]. Available: http://docs.oracle.com/javase/specs/jls/se7/html/index.html. [Accessed 28 August 2012].

[18] Bundesverwaltungsamt, "BVA Internet: Register Factory," [Online]. Available: http://www.bva.bund.de/nn_2143576/DE/Aufgaben/Abt__I/RegisterFactory/node.html?__nnn=true. [Accessed 17 June 2012].

[19] H. M. Sneed, „Offering Software Maintenance as an Offshore Service," 30 September 2008. [Online]. Available: http://www.icsm2008.org/downloads/sneed.pdf. [Zugriff am 18 June 2012].

[20] P. Avgeriou und U. Zdun, „Architectural Patterns Revisited – A Pattern Language," in *In 10th European Conference on Pattern Languages of Programs. EuroPlop*, Irsee, 2005.

List of figures

Figure 1: System Application Architecture [2] 13
Figure 2: Steps of Refinement using the Register Factory® 15
Figure 3: Patterns for program structures: Component-Façade 25
Figure 4: Patterns for program structures: Exception-Façade for Component Interfaces 26
Figure 5: Patterns for program structures: HTTP-Interface for Components 27
Figure 6: Architecture of the Architecture Verification Engine 40
Figure 7: Rule transfer objects 41
Figure 8: File management of the AVE 46
Figure 9: Selection mask for pattern verification 47

Listings

Listing 1: Naming convention of the Register Factory® 16
Listing 2: Grammar of the pattern description language 29