ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Faculty of Engineering, Computer Science and Psychology
Databases and Information Systems Department

jQAssistant: A QA Tool for Definition and Validation of Software Architecture Rules

Bachelor's thesis at Universität Ulm

Submitted by:
Laura Robien Baldrich
robien.baldrich@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert

Supervisor:
Michael Zimoch
Dr. Jan Scheible

2018

Version from November 27, 2018

Abstract

One main problem in software development is technical debt, meaning anything that delays or stops development. This is often caused by imprecise or missing software architecture. Static software architecture defines not only a project's structure, but also relationships and responsibilities of the structural elements. While many approaches exist for documenting static software architecture, there is no formalised and standardised technique to do it. With continously growing and aging software, code can become confusing and turn into a big ball of mud, straying from the defined architecture. Although many approaches to correct this problem have been made, there is still a need for an automatised process to check for derivations and cyclic dependencies. The jQAssistant„ a QA tool for defintion and validation of software architecture, is one approach to fullfill the need for automatised compliance checking.

This thesis focuses on two parts: the definition of static software architecture rules and the automatised compliance check from the start of development. The documentation shall be easily accessible, human readable and also flexible for adjustments. Because of the abundance of different project structures, this thesis focuses on an exemplary Java web application. Using custom annotations, the architecture is integrated within the code, using predefined and application based annotations. The automatised compliance check is accomplished with jQAssistant.

Acknowledgment

First and foremost, I'd like to thank everybody who offered me advice, listened to my thoughts or otherwise contributed to this thesis.

A special thank you to TRANSPOREON GmbH, for giving me the chance to write this thesis and offering me support and guidance. Regarding this, I'd like to especially thank Dr. Jan Scheible for introducing me to this topic and being there whenever I got my words confused. Thank you for turning this into a worthwhile challenge!

I'd also like to thank my supervisor Michael Zimoch for providing me with feedback and support. Thank you especially for your kind words and encouragement.

Contents

1 Introduction

Technical debts, a term used to describe a variety of elements that delays or stops development, are one of the main problems in software development [1]. Technical debt can origin in lack of planning or violations against planned architecture. It often ends with unwanted, cyclic dependencies and a *big ball of mud*; "a casually, even haphazardly, structured system" [2]. Even if architectural requirements exist at the beginning of development, too often they are outdated or lost, the more the system is expanded. There is a clear need for proper architecture documentation, but also for automated checks of the architecture. It is important to check for violations against software architecture directly at the beginning of development and make that documentation easily accessible and understandable to anyone involved in the development process.

1.1 Problem Statement

Although these problems are known and several approaches, such as UML or ArchJava (see chapter 3), to fix those issues have been developed, they are mostly not applied in practice. Using diagramms to describe the architecture may help understand dependencies and responsibilities between code elements, but are in danger of being forgotten one the actual implementation begins. Software architecture documentation is oftentimes neglected in favour of fast implementation. At best, it may be stored digitally in an organised fashion, at worst it may be written on a whiteboard to be erased afterwards. Due to missing requirements for software architecture documentation, automatised checks for architecture compliance face the challenge of having to be designed for an individual project without hindering the development process. Therefore, software architecture

documentation has to be adjusted to fit the need for automatised architecture compliance checks.

1.2 Objective

The objective of this thesis is to develop an automated check for static software architecture compliance using *jQAssistant* [3]. Although the protoype's scope is a simple Java web application, it is adjustable for any kind of application. It can be integrated directly at the start of development right up to deployment. Due to the already mentioned correlation between documentation and automated checking, the prototype requires a specific architecture documentation.

1.3 Structure of the Thesis

In chapter 2, an overview of the theoretical background for this thesis is given, e.g., important vocabulary and the basis for the prototype. This also includes the description of the jQAssistant, a tool for definition and validation of software architecture rules. In chapter 3 related works in view of this thesis's objective are discussed, while the methodical approach to creating a prototype for defining and checking static software architecture is described in chapter 4. The presentation of said prototype is presented in chapter 5, while conclusion and discussion of future work can be found in chapter 6.

2

Foundation

In this chapter, an overview of the foundations of this thesis is given. The term software architecture within the thesis' scope is explained and furthur used terms are defined. Furthermore, a short description of the tool Pocketsaw, which inspired the used documentation of software architecture requirements is given. Additionally, the tool jQAssistant, which is used to develop a prototype for documenting and automatic checking of software architecture compliance is described.

2.1 Software Architecture

The term software architecture is used in varying contexts, reaching from hardware design to design decisions for a software project. This thesis deals with only the static software architecture, using the definition by [4] :

> *"(System) fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution."*

According to this definition, software architecture not only defines the structure of a system, but also on a higher level the various elements and their associated responsibilities and relationships. It specifies the entire construct and serves as a guideline on how to structure the project. Implementing a qualitative project requires a good software architecture, but also an accessible one [5]. As software ages ([6]), the architecture might no longer apply and be in need of adjustment. Thus, there is a need for easy

adaptation and flexibility in the documention. There are several approaches to document software architecture, such as UML or ADL, which are an attempt at standardising software architecture documentation [7][8][5].

Although these approaches are being successfully practiced, there is still need for a better documentation for the application the prototype is developed for. The requirements for this thesis' documentation of software architecture are not only flexibility, but especially accessibility. Therefore, it is desired to have a documentation within the code, that can be used to describe the architectural requirements to developers and also to automatically check for compliance. Accordingly, it is neccessary to have a clear definition of the terms used to describe static elements and their relationships, which is done in section 2.2.

2.2 Building Blocks

In this thesis, the term *building blocks* from Dr. Carola Lilienthal [9] is used to describe the static make-up of a software project, specifically a Java web application. Building blocks are software elements that make up a project, such as classes and packages. They can be grouped by responsibility and contain other building blocks as well, similiar to the package structure in Java projects.

By defining building blocks and relationships between them, the static software architecture of a project is defined. Responsibilities and individual responsibilities can be defined for either a groupd or a singular building block in order to give a more detailed descirption of the building block's purpose.

Applying building blocks to a software architecture documentation can help with understanding the construction of a project, but proves to be additional effort. To avoid repetetive and redundant building blocks, another level is added, the *predefined building blocks*. They represent a group of building blocks and define the requirements, that are valid for al building blocks associatiated with a respective predefined building block. A comprehensive definition can be found in chapter 5.

Two essetial relationships between building blocks are the `uses` and the `contains` relationship [10]. The relationship `uses` is a conceptual dependency between building blocks, that is only defined in the architecture documentation. The `contains` relationship between the indivdual building blocks or between groups of building blocks is also part of the documentation, but can be found within the implementation as well. Both relationships are part of the *architectural requirements*.

After building blocks are defined for the exemplary web application in chapter 4, the *uses* relationship is applied to define a dependency graph (see section 4.1). It is the foundation of the automatised compliance check of the architectural requirements.

According to Lilienthal [10], architectural elements, which are essentially the building blocks discussed previously, can be grouped by type. Lilienthal establishes three general rules for such elements in her dissertation:

- Element Rules
  - Only valid for one type of element. In this case, the rules only concern one predefined building block.
- Imperative Rules
  - Define certain relationships between elements, that are mandatory.
- Prohibition Rules
  - Forbid certain relationships between different types of elements.

Accordingly, this thesis defines several rules for the predefined building blocks in section 4.1.2 and translates them into constraints for the jQAssistant (see section 2.6) to analyse in chapter 5.

2.3 Pocketsaw

Pocketsaw [11] is a tool that can be used to analyse a project's package structure and visualise package dependencies as a graph. It is inspired by *Jabsaw* which "expresses

static package level dependencies"[12]. Root packages in projects that can be analysed with *Pocketsaw* must each contain one descriptive building block, called SubModule: a class annotated by `@SubModule` (see Listing A.1), that defines "uses" relations to other SubModules. A uses relationship between two SubModules can be translated to a code-dependency between classes belonging to the SubModules, as shown in 2.1

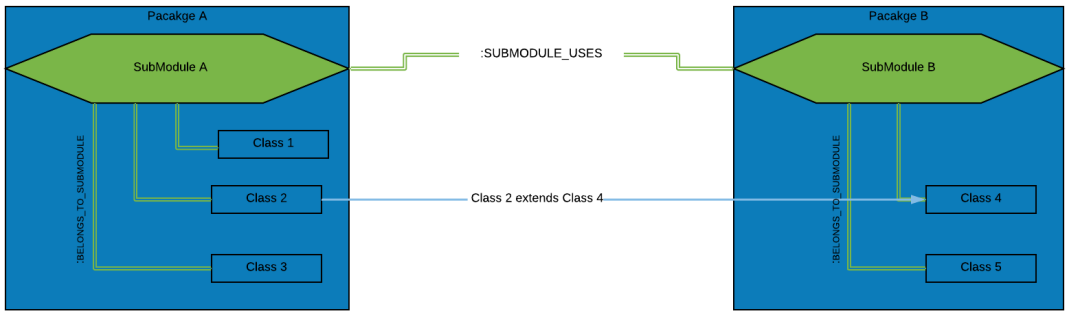

Figure 2.1: Abstract model of relationships and interactions between packages, classes, and SubModules

Pocketsaw is able to match all SubModules and their conceptual and code dependencies and show the group package structure of the project. The resulting graph visualises any violations against the defined architecture within the SubModules.

2.4 Maven

"Apache Maven is a software project management and comprehension tool. Based on the concept of a *project object model (POM)*, Maven can manage a project's build, reporting and documentation from a central piece of information" [13]. The project in this thesis' scope is a Maven project, as jQAssistant itself is a plugin for Maven.

2.5 Neo4j Graph Database

The company Neo4j Inc has designed a graph databse, which is a database that places importance on the connection between data, instead of a predefined model [14]. The elements in the database are

- Nodes
  - Represent the main data alements. They are connected to other nodes with relationships and can have properties and labels.
- Relationships
  - Directional connection of two nodes. They can also have properties.
- Properties
  - Named values that can be indexed and constrained.
- Labels
  - Applied to nodes or relationships to refer to those elements.

Figure 2.2 shows the elements and their relations.

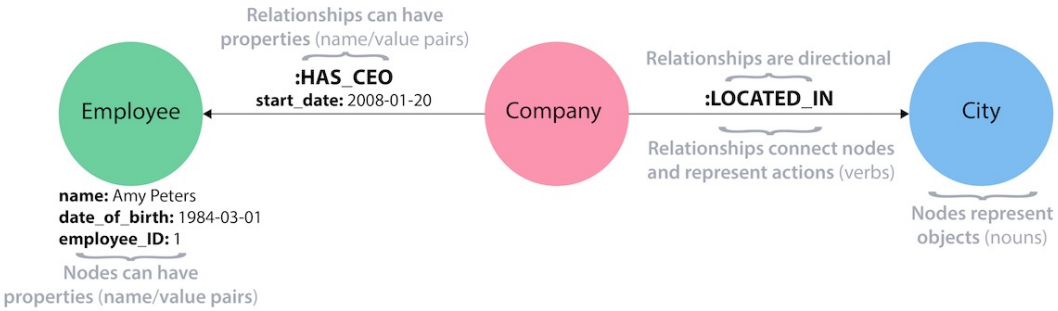

Figure 2.2: An overview of graph database elements and their properties[15]

2.5.1 Cypher

Cypher is a declarative query language that is used to query the elements stored in the graph database. A full glossar of the Cypher syntax can be found at [16]. For this thesis, it is important to understand the following syntax:

Nodes are referred to within parantheses, e.g., `(Node1)`. Relationships are capitalised letters with a leading colon within square brackets, e.g. `[:DEPENDS_ON]`. Relation ships connect two points, either direct with `<-[]-` or `-[]->` or indirect *-[]-*. An example of two connected nodes looks like this: `(Node1)-[:DEPENDS_ON]->(Node2)`. This indicats a dependency of Node1 on Node2. To reverse the dependency, the query can be written as `(Node1)<-[:DEPENDS_ON]-(Node2)`. Any labels from the database can be added within the node parantheses with a leading colon (`(Node1:Label)`. To search for Nodes with that declared dependency, the keyword `MATCH` is used and to return results, `RETURN`.

A valid Cypher is the following:

```
MATCH
(Node1)-[:DEPENDS_ON]->(Node2:Class)
RETURN
Node1 AS Node
```

2.6 jQAssistant

jQAssistant (jQA), is an open source tool for the definition and validation of structural project rules developed by Buschmais starting in 2013 [3]. The jQA scans generated artifacts and extracts the structural information into a Neo4j graph database. This data can then be analysed with already implemented or user defined rules using Cypher queries. Any violations can be automatically reported during and stop the build. There are pre-defined sets of labels and relationships already available, but it also allows for custom extensions. It also contains an integrated Neo4j server, which visualises the stored data as a graph [3].

There are several plugins already available for jQA, which can help aggregate and filter the data. In this thesis, no additional plugins are used.

2.6.1 Concepts

The Cypher queries used to enrich the data, can be grouped together as concepts in an XML file. One example of such a concept is demonstrated in Listing 2.1.

```
<concept id="example:ReturnSuperclasses">
    <requiresConcept refId="example:LabelSuperclass"/>
    <description>Returns all Classes with Superclass Label contained in
        root package.</description>
    <cypher><![CDATA[
        MATCH
            (p:Package)-[:CONTAINS]->(c:Superclass)
        WHERE
                p.name = {rootPackage}
        RETURN
            c
        ORDER BY
            c.fqn
     ]]></cypher>
</concept>
```

Listing 2.1: Example of a Concept

A concepts consists of three mandatory xml elements - id, description, cypher query - and an optional fourth element, a required concept. The id is individually assigned to every concept. Using a two-parts id, such as `example:SuperClasses`, allows the user to group the concepty by the first part - example- and indivualise with the second part. The description allows for a human readable description of what the query does or for what it is used. The cypher query can be added with the usual cypher syntax. If another concept is required, it is referenced by id with the *requiresConcept* tag.

The concept in Listing 2.1 returns all super classes contained in the package named root and orders them by fully qualified name(fqn). It requires another concept with the

id `"example:LabelSuperClass"` which first labels all superclasses, so they can be referenced as they are. Concepts can require multiple concepts

By using such id's, the concepts can be grouped by purpose, such as architecture compliance check or for testing a small portion of the project. Several concept groups can be applied at the same time, so it is not neccessary to have duplicate groups for different stages in development. The concepts only add to the database, they do not automatically check for compliance, although they are essential for constraints.

2.6.2 Constraints

Constraints are Cypher queries with the purpose of compliance checking. They can rely on multiple concepts, which have to be analysed before the constraint application. Constraints can also be grouped by purpose as concepts. This allows for checking different aspects of architecture, e.g. static rules. An example of a constraint is Listing 2.2.

```
<constraint id="example:CountSuperClasses">
        <requiresConcept refId="example:LabelSuperclass"/>
        <description>Returns superclasses</description>
        <cypher><![CDATA[
         MATCH (c:Superclass)
         RETURN c
        ]]></cypher>
      <verify>
         <rowCount max="10"/>
      </verify>
</constraint>
```

Listing 2.2: Example of a Constraint

The constraint returns all superclasses, but detects a violation, if there are more than ten.

2.6.3 Integrating jQAssistant

The jQAssistant can be integrated into a Maven project as a plugin or used as a stand alone via command line. For the command line the only requirement is a Java Runtime Environment/Java Development Kit 7 or later. After downloading the tool, the user can scan the project via command line and open the integrated Neo4j Browser to see the visualisation of the project's elements. Concepts and Constraints can also be applied via commandline, but have no influence on the build process.

To use jQA as a Maven plugin, the dependencies and configurations need to be added to the POM. An exemplary configuration is shown in section 5.4. Using jQA as a plugin ensures an automated check during the build. Any violations lead to breaking the build, not applied concepts only issue a warning. In this thesis, both approaches are used. First the command line during development to avoid configuration errors in early phases. Later on, the prototype is applied to a web application using the plugin.

3

Related Works

This chapter gives an overview of related works, specifically approaches to automatising architecture compliance checks and documenting software architecture within the implementation.

3.1 Automatising Architecture Compliance Checks

In [17], the three main *architecture compliance* checking approaches for static software architecture are identified and evaluated: *Reflexion models* compare two system's models (the plannend model and the source code model) and detect deviations. *Relation conformance rules* allow for automated checking of deviations from allowed and forbidden relations between components. The third approach, *Component access rules*, is used to specify component ports that can be called by other components. Evaluation using one of the three approaches leads to either *convergence*, *divergence* or *absence* of rules. However, reuse on a different project might not be possible or only with added effort.

Pattern-Lint presented in [18] offers a design compliance check from the start of development. However, it is limited by the many possibilities of architectural models, which cannot all be checked.

ArchUnit [19] is a library that can be used to check dependencies between static elements in Java code. It can be added to Maven projects as a dependency and requires no specific infrastructure.

3.2 Documenting Software Architecture

ArchJava is an extension for Java, which allows the developer to add language constructs in order to describe static architecture in the source code [20]. Unfortunately, the project is no longer active.

In [21], languages, such as the *expression language*, are designed to express architectural requirements. While a formalised and generic language can be useful for automatised compliance checks, the approach of using an algebra based on sets may require additional tranlsation of the implementation into the expression language.

The *Unified Modelling Language* (UML) is a graphic modelling langauge that can be used to sepcify and document objects, its properties and static relations [22]. While it is very useful for defining the required architecture, it has the disadvantage of high maintenance and being excluded from the implementation itself.

4

Method

This chapter describes the methodical aspects and preliminary considerations of the prototype from chapter 5. It is inspired by Pocketsaw from section 2.3.

4.1 Dependency Graph

The objective of this section is to develope an approach to automatising architecture compliance checks. Therefore, a dependency graph is needed to compare conceptual and code dependencies between the static elements, which are established as building blocks.

With the focus on the static software architecture it is neccessary to identify all dependencies within the application. The dependency graph shows a visualisation of all building blocks with the correlating dependencies. It can be used to detect cycles or unwanted dependencies between blocks. Figure 4.1 shows the extract of a dependency graph from chapter 5.

Theelements in the dependency graph can be grouped as conceptual and code building blocks.

4.1.1 Conceptual Building Blocks

The descriptive building blocks are manually added elements, which serve a purely conceptual purpose. They define allowed relations and responsibilities for the code

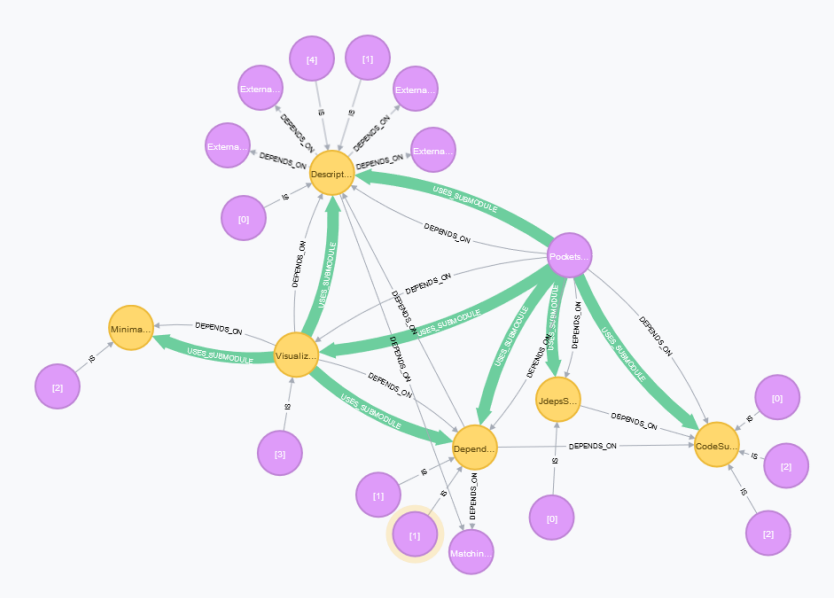

Figure 4.1: Dependency graph showing building blocks and their relations

building blocks. An example of the checked responsibilites in this thesis is listed in section 4.1.2.

4.1.2 Code Building Blocks

The code building blocks are the implemented elements of a project, which in case of this thesis is a Java web application.

They each belong to a conceptual building block and are supposed to adapt to their regulations. Through the connection to the conceptual building blocks it is possible to find code-dependencies that are either unwanted or even cyclic.

The dependency graph shows both conceptual and code building blocks, which makes it possible to detect any derivations from the planned architecture.

Annotations

The building blocks need to be easily identifiable, while at the same time stay adjustable, therefore a flexible yet effortless solution is needed.

The use of custom annotations, as used in Pocketsaw, is convenient since both the conceptual and the code building blocks can be annotated. The conceptual building blocks are custom annotations and annotate several code building blocks, marking them as part of their block. Within the annotations, the responsibilities for each block can be defined. It is possible to preset defaults or allow for individual modification, as explained in chapter 5.

Requirements

This list contains requirements defined for a static web application.

- Acyclic Dependencies
  - The domain components are allowed to depend on each other, as long as no cyclic dependencies are created.
  - Other components are not allowed to have dependencies between themselves.
- Cycles
  - No cycles are allowed anywhere.
- Annotations
  - Entity and Repository annotations are only allowed in Store and Domain components. Everywhere else the usage is forbidden.
  - JmsListenery, RestController, and Controller annnotations are only allowed in Port components. Everywhere else the usage is forbidden.

4.2 Static Web Application Structure

The descriptive building blocks have the major advantage of being reusable, as well as being predefined. With them, it is possible to define the different components of the application, translate them into descriptive building blocks and document the corresponding responsibilities. The building blocks can then be used in any application of the same architecture, while still allowing for changes to the responsibilities.

The predefinitional aspect is an ensurance of consistency; while it remains easy to adjust to new architectural guidelines, it is also consistent enough to not be overwritten out of mistake or convenience. For the scope of this thesis, building blocks are predefined for a static web application, according to Figure 4.2:

The arrows represent uses relationships between predefined building blocks. These predefined building blocks are chosen as small as possible to be able to define general rules for their blocks, but also have only a small amount of them to keep the process of documentation simple. The use of the descriptive building blocks is described in chapter 5.

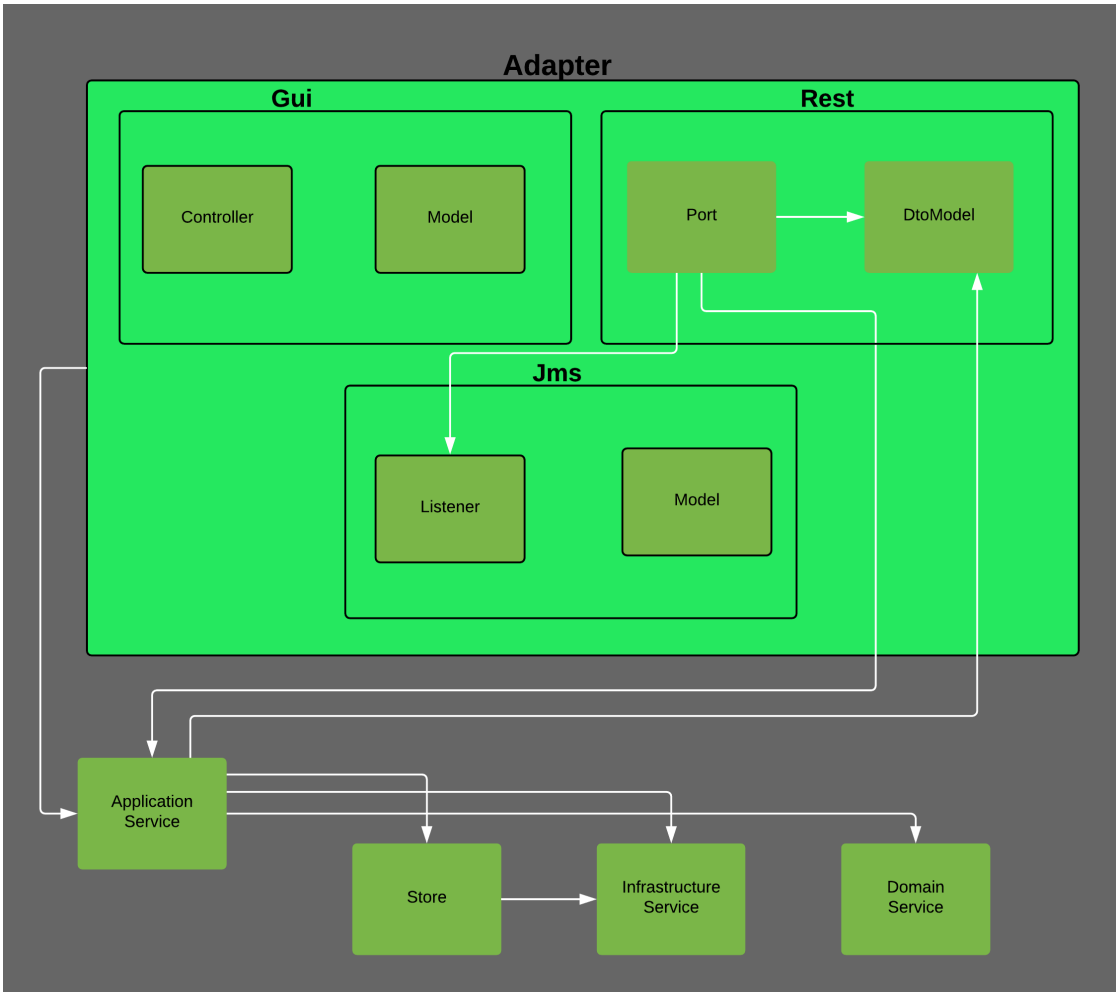

Figure 4.2: Building Blocks for a static web application

5

Prototype

This chapter deals with applying predefined building blocks from chapter 4 to a Java web application. These are then used to construct a dependency graph in order to check for architecture compliance. Section 5.1 explains the reasons behind using annotations for documentation. Sections 5.2 and 5.3 describe the approach and implementation of concepts and constraints to expand the Neo4j graph database and check for compliance as per the user defined rules.

5.1 Software Architecture Documentation

A seperation of code and documentation is undesirable, as stated in chapter 2. While software gets changed and modified, the corresponding documentation unfortunately does not [6]. It is often the case that documentation is incomplete or even imprecise to begin with[6]. To be able to have a precise documentation that is integrated within the code, annotations are used based on the *Pocketsaw @SubModule annotation* (see listingA.1). As explained in section 4.1.1, building blocks group and define the application's static architectural elements, which are defined in chapter 4. For the documentation, the building blocks are implemented as annotations, which can be divided into two groups:

- `@BuildingBlock`
- `@PredefinedBuildingBlocks`

The `@BuildingBlock` annotation (see Listing A.4) identifies an element as a conceptual building block of any kind. The `@PredefinedBuildingBlock` annotation (see

Listing A.3) is used for the predefined building blocks, such as in Figure 4.2. They define the rules for the annotated components and are later on used to check for compliance.

Building block instances (see Listing A.2) are added to the logical components, e.g., packages. Each package contains one instance of a predefined building block and can be used to create the dependency graph as well as define and check for the responsibilities mentioned in section 4.1.2.

5.2 Constructing a Dependency Graph

Constructing a dependency graph of the predefined building blocks helps detecting unwanted dependencies. Firstly, the graph database needs to be expanded by information about these blocks. As detailed in section 2.6, jQA uses concepts to enrich the data. This section explains the approach to add labels to the databse and define relationships between the building blocks.

5.2.1 Labels

After scanning a project, jQA is able to offer several dependencies by established labels, such as package or class. For the purpose of analysing not only code, but also conceptual dependencies, it is neccessary, to add further labels and relationships to the graph database.

First, the predefined building blocks and their respective annotations need to be identified. Figure 5.1 illustrates the relationships between the blocks. Using a top-down approach, the `@BuildingBlock` annotations are labeled first and then used to identify the `@PredefinedBuildingBlock` annotation.

After the *@BuildingBlock* annotations are labeled, the `@PredefinedBuildingBlock` annotations are identified with an extended query, which then leads to their instances. The used queries are shown in Listing A.5. For a better understanding of how the indivudal parts of the queries in the concepts work, the line number of the Listing A.5 is

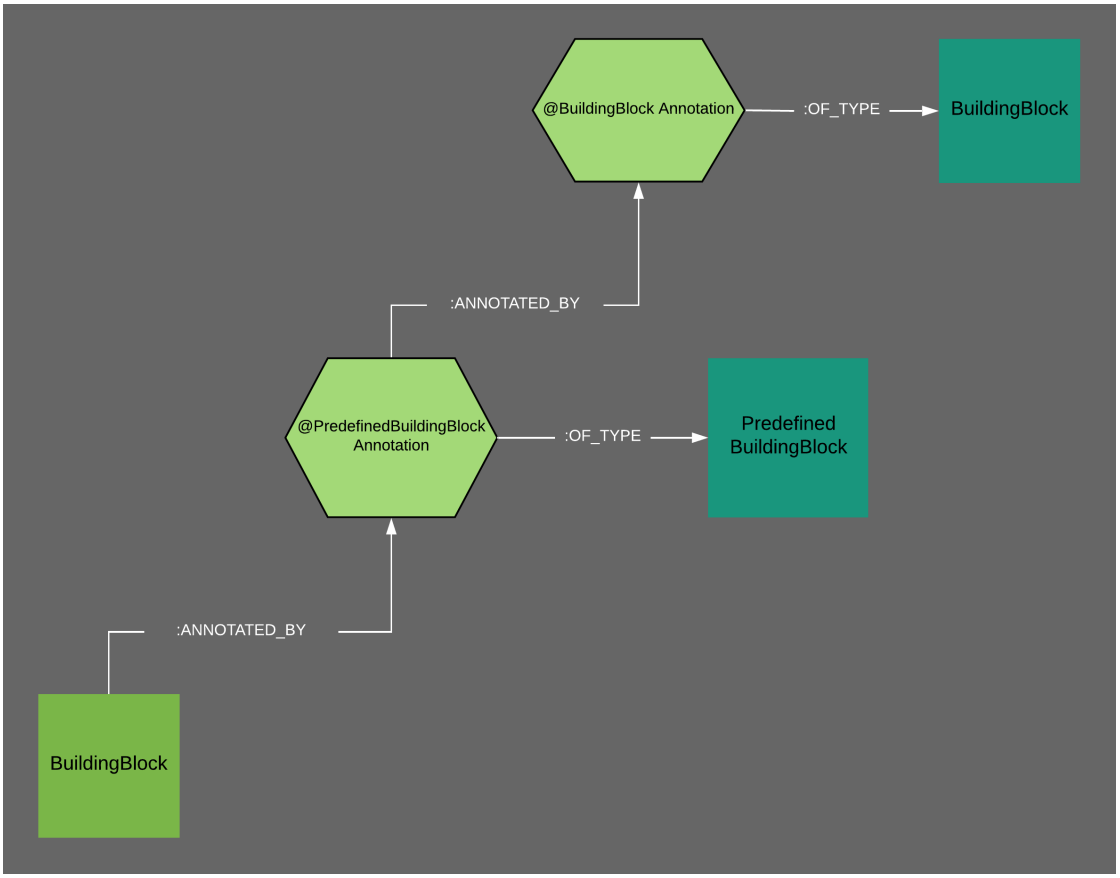

Figure 5.1: The @BuildingBlock and @PredefinedBuildingBlock annotations and a simple annotated building block element

given in parentheses at the end.
The first concept in Listing A.5 matches all annotations of the given type(line 4) and then adds the label to the annotations(line 5), before returning the building block annotations. The second concept is similiar, except for the label, which is added to the type itself. Since the building block annotations are now identified, it is possible to find the predefined building blocks, as they are the types annotated by these annotations. It is important to note, that the predefined building block annotation is the one that is annotated by the building block annotation (as shown in Figure 5.1). The concept starting in line 19 works like the on in line 1, with the exception that a full qualified name for the type is no longer neccessary, as we use the previously added label for building block annotations (line 25). All types annotated by the predefined building block annotation are matched. To shorten the query in the future, a label for the type predefined building block is set (line 26). Both annotations from section 5.1 are now labeled and easily queried for.

The last element missing from Figure 5.1 are the regular building blocks. To address these instances of the predefined building blocks, the query in line 31 is executed. It labels any types annotated by the `@PredefinedBuildingBlock` annotation, while the concept in line 43 labels `PredefinedBuildingBlocks`.

With the concepts from Listing A.5, the static architectural elements of the web application are added to the graph database. To complete the dependency graph, the conceptual and code dependencies need to be applied between the building blocks, which is described in section 5.2.2.

5.2.2 Relationships

As explained in section 4.1, the dependency graph differentiates between conceptual and code dependencies, which results in two sets of jQA concepts for the relationships between building blocks.

Conceptual Dependencies

The conceptual relationships are defined within the predefined building blocks, e.g., through a 'uses' relationship as implemented in Listing A.3. The predefined building block can use the attribute to define an allowed relationship to another predefined building block. To ensure a consistent architecture, the only building blocks allowed to define uses relationships are the predefined building blocks. Changes to the architecture are therefore easily made, as only one building block has to be adjusted to changes. As a result of the established dependency between building blocks and their respective predefined building blocks, the allowed uses relationships are automatically prescribed for building blocks as well. To give an example, alls building blocks that belong to the predefined building block A can depend on building blocks belonging to the predefined building block B as long as A defines a uses relationship to B.

The queries in Listing A.6 show the concepts required to add the conceptual relationships to the database. As stated above, it is essential to establish a dependency between building blocks and their predefined building blocks. The first concept in line 1 of Listing A.6 creates a relationships from building blocks to predefined building blocks, which is named `:BELONGS_TO_PREDEFINED`. Now it is possible to map requirements defined in the predefined building blocks to their building blocks.

The `uses` relationships between predefined building blocks, is split into `:PREDEFINED_USES` and `:PREDEFINED_ALLOWS_USE`. The first one shows any conceptual usage between predefined building blocks, while the second one is applied to the building blocks to show allowed conceptual relationships between building blocks. Since the information is implemented within the predefined building block annotation (see Listing A.3), it is necessary to get the value of the String in the `uses` attribute. For that purpose, it is important to know how jQA stores the data.

Figure 5.2 shows the graph of an annotation with such a uses attribute and its value. The first node represents the predefined building block, which is annotated by the

`@PredefinedBuildingBlock` annotation. This node is connected to the uses attribute with a `[:HAS]` relationship, which contains another node that has an `[:IS]` relationsip to a predefined building block. The relationships in white above the nodes represent the query that is used to get the value of the attribute used in the annotation. The green relationship under the nodes represent the new custom relationship between the outer nodes that can be used for future queries.

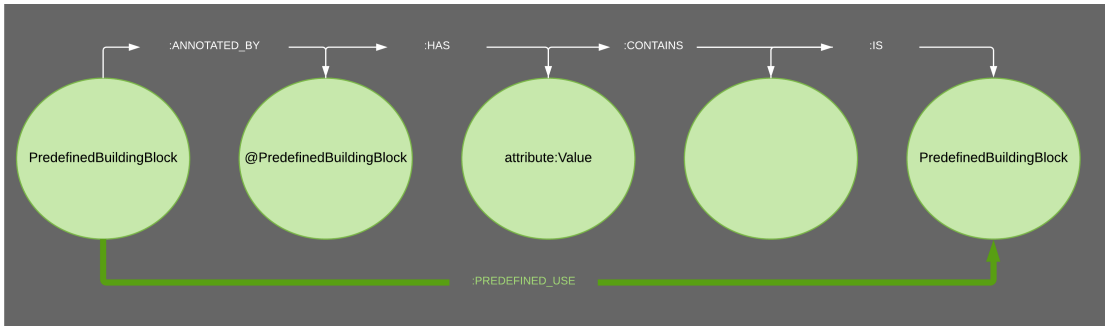

Figure 5.2: Comparison of default relationship between predefined building blocks (white) and user created relationship (green)

The query of the `PredefinedUses` concept in line 13 of Listing A.6 uses the same relations shown in the Figure 5.2 and shortens it to a new relationship `PREDEFINED_USES` between the two building blocks marked in green. The newly created relationship maps the architectural structure from section 4.2 to the dependency graph. The dependency graph now shows all building blocks and their static structure as intended by the architecture.

Code Dependencies

To complete the dependency graph, the code dependencies are created with the queries in Listing A.7. As Listing A.3 shows, a building block can be defined for several subpackages. By default the boolean value of `includeSubPackages` is set to true. The way jQA stores these information, as discussed previously in section 5.2.2, two concepts (line 1 to 24) are required to account for that value. Firstly, a property `includeSubPackages` is added to the building block and set to false, if the boolean in the predefined building

block is set to false. Then (line 15), the second concept matches all building blocks without the previously set property and adds the property with the value set to true. The order must be kept, as the jQA concepts can only match for existing, meaning explicitly set, attributes. The default values can not be matched currently.

Depending on the set property value, a relationship `:BELONGS_TO_BB` is created between types and building blocks. For a better understanding, if a building block is added to a package, all other classes/types contained in the same packages then belong to that building block. As of now, the dependency graph contains information about the predefined building blocks, the building blocks, code elements, e.g. classes belonging to a building block and their conceptual relationships. The relationships showing code dependencies between all these elements are created in the concepts in line 50 and 67. First, code dependency relationships `CODE_DEPENDENCY_BB` is created between building blocks, by using code dependencies between the type belonging to the building blocks. The query first matches two different types that belong to different building blocks (line 56) and then creates said relationship (line 63).

The code dependency relationship `CODE_DEPENDENCY_PREDEFINED` is created between predefined building blocks with the same method.

5.2.3 Requirements

The requirements defined in section 4.1.2 need additional data in the database to be checked for compliance, which is added with the concepts in Listing A.8. For checking for cycles where acyclic dependencies are allowed, a label `'AllowsAcyclic'` is added. The predefined building blocks declare a boolean value `'allowsAcyclic'` (see Listing A.4), which is false by default.

To check compliance for correct annotations, the relationship `:USES_ANNOTATION` is added between the building blocks and any annotations used. At the same time, relationships for allowed and forbidden use are created between the predefined building

blocks and annotations. Whenever a predefined building block allows the use of a specific annotation, the others automatically forbid the use if they do not explicitely list that annotation as allowed. An examplary implementation can be found in Listing A.3. The predefined building block allows the annotation `RestController`, `@Controller`, `and @JmsListener`, therefore, its building blocks can use these annotations. Every other predefined building block that does not define the same annotations automatically creates

5.3 Constraints

Although the concepts are essential for the compliance check, the constraints are responsible for breaking the build in case of any violations. Using the three rules from section 2.2, the constraints are grouped as element, imperative and prohibition constraints. The element constraints are listed in Listing A.9, the imperative in Listing A.10 and the prohibition in Listing A.11.

The responsibilities from section 4.1.2 can now be checked during the build and actions taken in case of non-compliance.

5.4 Applied on reference architecture

So far, the concepts and constraints are applied via command line. To test the practicality of the prototype, jQA is integrated as a Maven plugin and the building block annotations are imported as Maven module. Figure 5.3 shows the required project structure. It is possible to use the prototype in a multi module project, but not necessary. The Listing A.12 shows configuration parts, which have to be added to the parent POM. If any additional concept or constraint groups are added to building-blocks-rules.xml, they need to be added to the <groups> element for execution. Alternately, they can be removed for testing purposes. The additional concept `classpath:Resolve` is needed for working across multiple JARs.

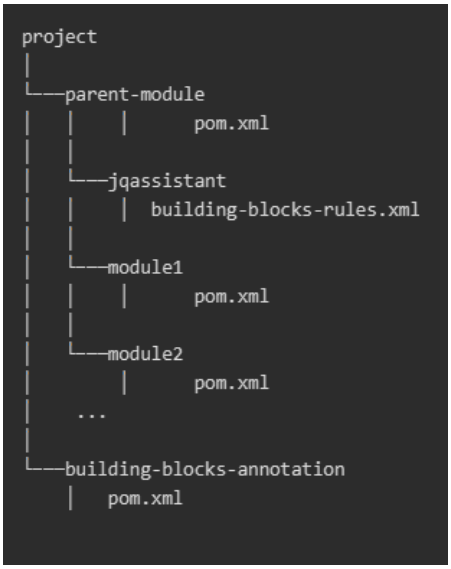

Figure 5.3: Required project structure for working with jQA plugin and building block annotations module)

Figure 5.4 shows that although both modules are scanned and detected, they have no relation between them. Therefore, a scan with jQA would not lead to the desired results. Instead, it would scan the two modules and show two unconnected jars.

Currently, it is possible to resolve two jars with the configuration mentioned previously, so any dependencies are detected. The requirements in the predefined building blocks however, need to be matched to the project that is checked for architectural compliance.

The solution is to add building blocks to the project and annotate them with the predefined building block annotations. Now the two modules are connected and the defined requirements can be checked.

The building blocks added to the project are of a conceptual nature, as already mentioned earlier. There is no need for any additional implementation except for the predefined building block annotation. Figure 5.5 shows the added building blocks and their relationships to the predefined building blocks from the annotation module. A scan now

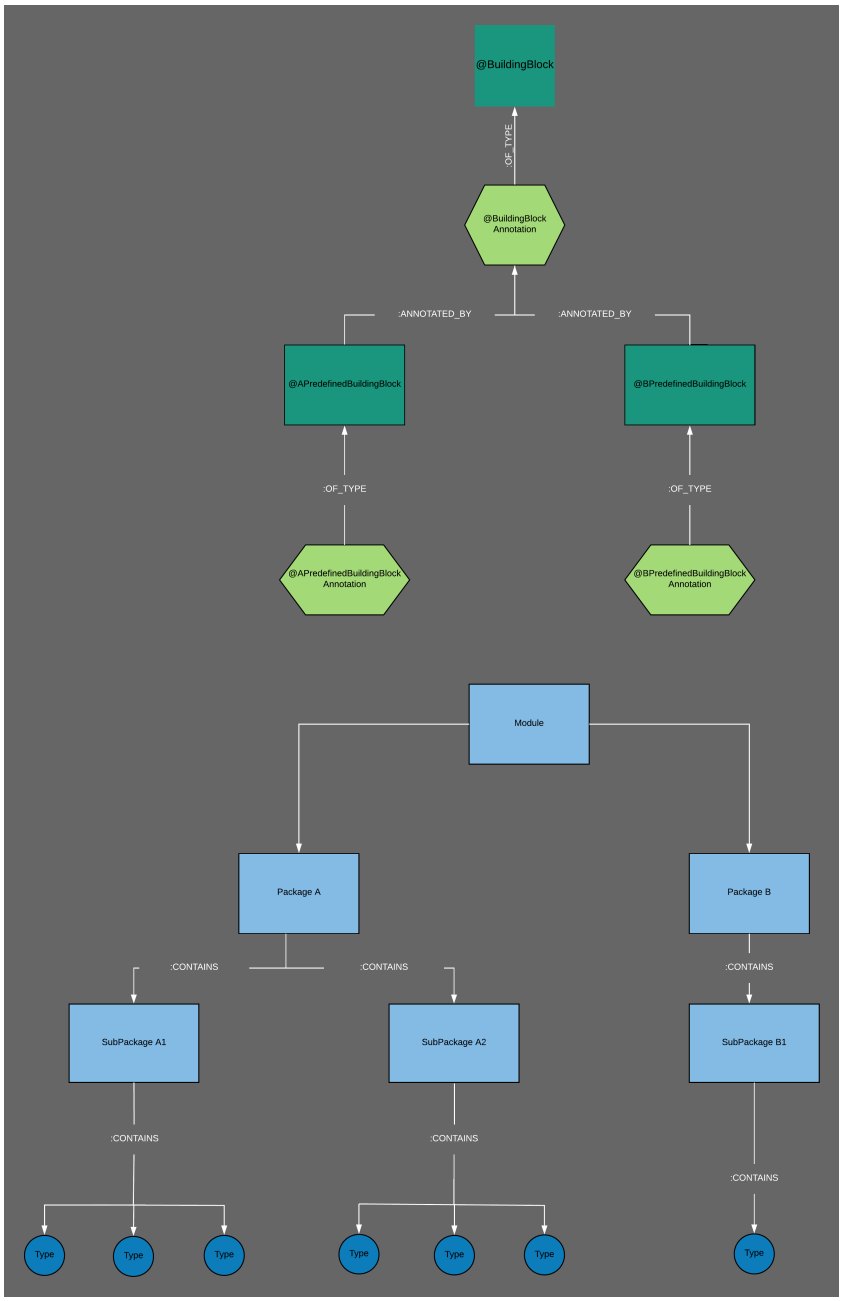

Figure 5.4: The two modules before any connection through additional building blocks is made

shows the two modules connected through the added building blocks and it is possible to execute the concepts from section 5.2.3.

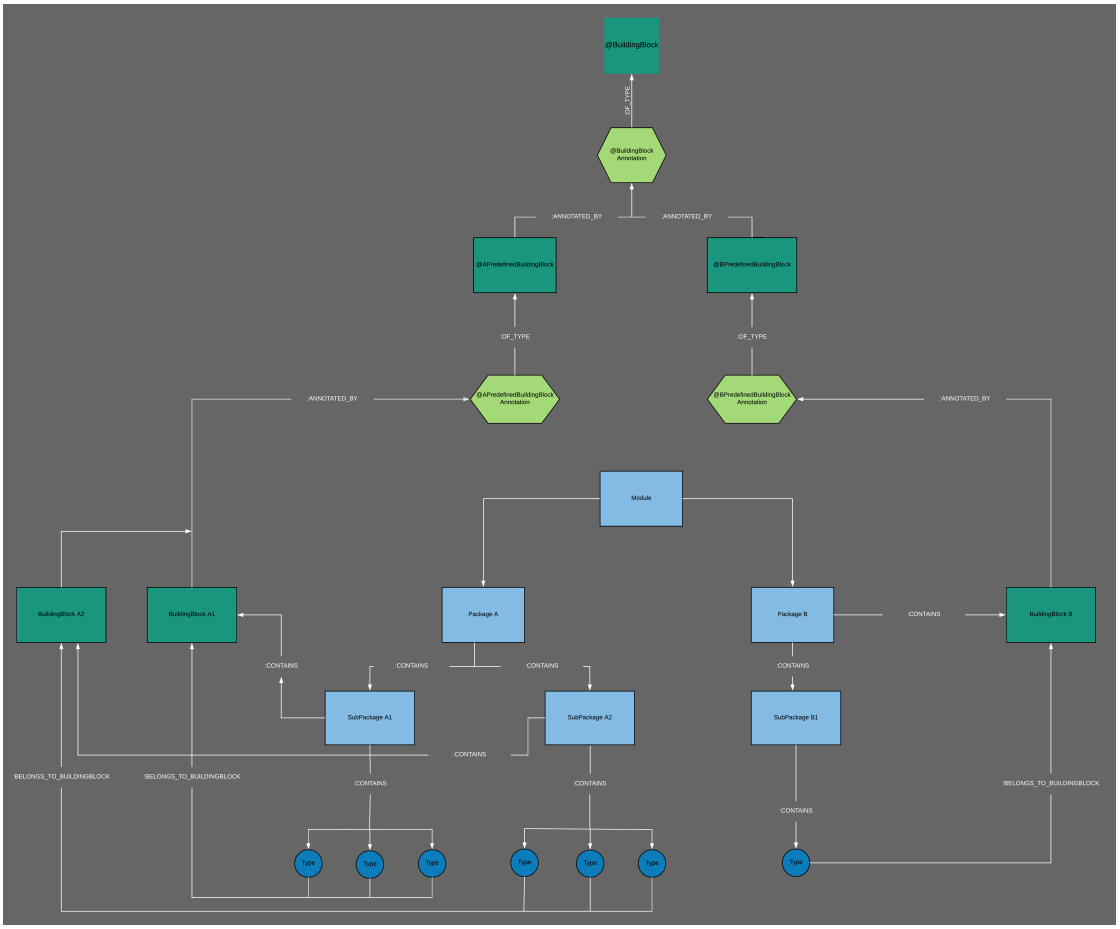

Figure 5.5: The two modules connected through annotated building blocks added to the project.

6

Conclusion

There is a correlation between "good" software architecture documentation and automatic compliance checking. In order to have such an automated process, the application's architectural requirements have to be machine-readable. At the same time it is just as important for the developers to understand the requirements in order to sucessfully write code in compliance.

To address both problems, the developed prototype serves as an approach to implementing static software architecture documentation within the code itself and using it for automatic compliance checking with the jQAssistant.

First, the software architecture documentation had to be implemented within the code. Looking at the static architecture, the application was divided into building blocks. Using the idea of custom annotations, predefined building blocks were implemented as annotations, that represent the required static architecture and also define attributes, such as allowed dependencies.
Further attributes were added, to allow for element specific rules, such as acyclic dependencies or allowed annotoations within the building block. The building blocks of the project were annotated by the respective predefined building block annotation.

Using jQAssistant's concepts and constraints, a dependency graph of the building blocks was implemented and used to check for any architectural violations. The resulting prototype, which consists of the `@BuildingBlock` and `@PredefinedBuildingBlock` annotations was then applied as a Maven module to a web application and tested.

The prototype developed in this thesis was succesfully applied on a real Java web application. Although it already contains several predefined building blocks and rules designed for a web application. Depending on the architecture, it would still need to be adapted to a new project to allow for changed or additional architectural requirements.

Nevertheless, with the additional manual it should be easily adjusted to a new project. The concepts and constraint can be reused as long as the idea of using the `@BuildingBlock` and `@PredefinedBuildingBlock` annotations are still used. The maintenance for the prototype is relatively low, as concepts and constraints don't need to be adjusted and only building blocks need to be added. A static software architecture documentation should already contain the information needed to add predefined building block annotations.

All in all, the prototype may offer advantages when it comes to continuous and early on compliance checking, but needs to be expanded for universal applicability. It is also thinkable to develop tests for the concepts and constraints, to avoid non-applied concepts that do not produce a warning during the build. Since the scope of the prototype is limited to only static software architecture, specifically the existence of static elements and their relationships and some responsibilities, there is still room for improvement.

6.1 Future Work

For the future, the prototype can be further developed to match other kinds of applications as well, instead of only web applications. With the possibility to apply concepts and constraint groups seperately by adjusting the configuration in the POM, the additional predefined building blocks can just be added. Depending on the kind of application, the groups(e.g. concepts:WebApplication, constraints:WebApplication ...) can be added or removed.
Right now the building blocks need to be manually added to the project. It might be time saving to also automise this process, as the building blocks have no other use as to

serve as a conncetion to the predefined building blocks.

The scope for this thesis was only the static architecture, but did not cover all its aspects. It would be possible, to also check for naming conventions. Regarding dynamic software architecture, with the concepts of jQAssistant it is possible to enforce additional requirements, such as checking of test coverage. Such requirements can be added to the predefined building blocks or added through additional annotations. It may also be helpful to implement rules to check for more advanced architectural rules, such as detecting business logic or naming conventions.

In view of early-on compliance checking, further concepts can be implemented to check for recent changes to the code and apply only the constraints that are applicable to that part again, instead of always applying the constraints for the whole project.

A very good addition would be a better visualisation than the Neo4j server. Although it is very well suited for showing relationships, it would be helpful to directly be shown any violations or to be able to compare two graphs to each other.

The detected Cycles need a better output, or rather, the constraint detecting the cycle needs to be able to match the cycles directly to the classes where the dependencies ocur, instead of only the predefined building blocks.

The existing prototype serves a a good basis for automatic compliance checking of static software architecture requirements, but can be extended to be applicable to any kind of project.

Bibliography

[1] Kruchten, Philippe and Nord, Robert L and Ozkaya, Ipek: Technical debt: From metaphor to theory and practice. Ieee software **29** (2012) 18–21

[2] Foote, B., Yoder, J.: Big ball of mud. Pattern languages of program design **4** (1997) 654–692

[3] Buschmais: jQAssistant User Manual (2018-05-13) http://buschmais.github.io/jqassistant/doc/1.4.0/, last accessed on 2018-10-17.

[4] : Systems and Software Engineering - Architecture Description . Standard, International Organization for Standardization, Geneva, CH (2011)

[5] Garlan, D.: Software architecture: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering, ACM (2000) 91–101

[6] Parnas, D.L.: Software aging. In: Proceedings of the 16th international conference on Software engineering, IEEE Computer Society Press (1994) 279–287

[7] Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on software engineering **26** (2000) 70–93

[8] Zöner, S.: Software-Architekturen dokumentieren und kommunizieren. Carl Hanser Verlag GmbH Co KG (2015)

[9] Lilienthal, C.: Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen. dpunkt. verlag (2017)

[10] Lilienthal, C.: Komplexität von Softwarearchitekturen, Stile und Strategien. (2008)

[11] Scheible, Jan: Pocketsaw (2018) https://github.com/janScheible/pocketsaw, last accessed on 2018-10-17.

[12] Steinmann, Ruedi : Jabsaw (2018) https://github.com/ruediste/jabsaw, last accessed on 2018-10-17.

[13] The Apache Software Foundation: Apache Maven Project (2018) https://maven.apache.org, last accessed on 2018-10-17.

[14] Neo4j, Inc.: Neo4j (2018) https://neo4j.com/, last accessed on 2018-10-17.

[15] Neo4j, Inc.: What is a Graph Database? (2018) https://neo4j.com/developer/graph-database/, last accessed on 2018-10-17.

[16] Neo4j, Inc.: Chapter 3. Cypher (2018) https://neo4j.com/docs/developer-manual/3.4/cypher/, last accessed on 2018-10-17.

[17] Knodel, J., Popescu, D.: A comparison of static architecture compliance checking approaches. In: Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on, IEEE (2007) 12–12

[18] Sefika, M., Sane, A., Campbell, R.H.: Monitoring compliance of a software system with its high-level design models. In: Proceedings of the 18th international conference on Software engineering, IEEE Computer Society (1996) 387–396

[19] TNG Technology Consulting GmbH: ArchUnitr (2018) https://github.com/TNG/ArchUnit, last accessed on 2018-11-26.

[20] Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. In: Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on, IEEE (2002) 187–197

[21] Van Ommering, R., Krikhaar, R., Feijs, L.: Languages for formalizing, visualizing and verifying software architectures. Computer languages **27** (2001) 3–18

[22] Fowler, M., Scott, K.: Uml konzentriert. Addison Wesley, Boston, USA **2** (1998)

A

Sources

This appendix contains important source code snippets.

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.TYPE)
3  public @interface SubModule {
4
5      /**
6       * Alias for uses();
7       */
8      Class<?>[] value() default {};
9
10     /**
11      * Alias for value().
12      */
13     Class<?>[] uses() default {};
14
15     Class<?>[] allowedAnnotations() default {};
16
17     boolean includeSubPackages() default true;
18
19     String color() default "orange";
20 }
```

Listing A.1: Submodule Annotation

```
1  @AdapterPredefinedBuildingBlock
2  public class AdapterBuildingBlock {
3
4  }
```

Listing A.2: BuildingBlock with PredefinedBuildingBlock Annotation

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.TYPE)
3 @BuildingBlock(uses = {ServicePredefinedBuildingBlock.class,
      InfrastructurePredefinedBuildingBlock.class}, allowedAnnotations = {
      RestController.class, Controller.class, JmsListener.class})
4 public @interface AdapterPredefinedBuildingBlock {
5     Class<?>[] allowedAnnotations() default {};
6 }
```

Listing A.3: PredefinedBuildingBlock for an Adapter

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.TYPE)
3  public @interface BuildingBlock {
4
5      Class<?>[] value() default {};
6
7      Class<?>[] uses() default {};
8
9      Class<?>[] allowedAnnotations() default {};
10
11     boolean allowAcyclic() default false;
12
13     boolean includeSubPackages() default true;
14
15     boolean springBootApplicationAnnotation() default false;
16 }
```

Listing A.4: BuildingBlock Annotation

```
1 <concept id="static:LabelBuildingBlockAnnotation">
2     <description>Set Label for annotation @BuildingBlock </description>
3     <cypher><![CDATA[
4     MATCH (a:Annotation)-[:OF_TYPE]->(type:Java {fqn:'com.tpgroup.
          buildingblocks.annotation.BuildingBlock'})
5     SET a:BuildingBlockAnnotation
6     RETURN a AS Annotation
7     ]]></cypher>
8 </concept>
```

```
9
10 <concept id="static:LabelBuildingBlockAnnotationType">
11     <description>Set Label for type of annotation @BuildingBlock </
           description>
12     <cypher><![CDATA[
13     MATCH (a:Annotation)-[:OF_TYPE]->(type:BuildingBlockAnnotation)
14     SET type:BuildingBlockAnnotationType
15     RETURN a AS Annotation
16     ]]></cypher>
17 </concept>
18
19 <concept id="static:LabelPredefinedBuildingBlockAnnotation">
20     <requiresConcept refId="static:LabelBuildingBlockAnnotation"/>
21     <description>Set Label PredefinedBuildingBlockAnnotation on predefined
           building blocks.</description>
22     <cypher><![CDATA[
23     MATCH (t:Type)-[:ANNOTATED_BY]->(annotation)
24     -[:OF_TYPE]->(predefinedBuildingBlock)
25     -[:ANNOTATED_BY]->(s:BuildingBlockAnnotation)
26     SET annotation:PredefinedBuildingBlockAnnotation
27     RETURN t
28     ]]></cypher>
29 </concept>
30
31 <concept id="static:LabelBuildingBlock">
32     <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
33     <description>Set Label BuildingBlock on classes with a @PredefinedBB
           annotation.</description>
34     <cypher><![CDATA[
35     MATCH
36         (t:Type)-[:ANNOTATED_BY]->()
37         -[:OF_TYPE]->(p:PredefinedBuildingBlock)
38         SET t:BuildingBlock
39         RETURN t AS BuildingBlock
40     ]]></cypher>
41 </concept>
42
43 <concept id="static:LabelPredefinedBuildingBlock">
```

```
44     <requiresConcept refId="static:LabelBuildingBlockAnnotation"/>
45     <description>Set Label PredefinedBuildingBlocks on classes with a
           @BuildingBlock annotation.</description>
46     <cypher><![CDATA[
47     MATCH (t:Type)-[:ANNOTATED_BY]->(annotation:BuildingBlockAnnotation)
48     SET t:PredefinedBuildingBlock
49     RETURN t AS PredefinedBuildingBlock
50     ]]></cypher>
51 </concept>
```

Listing A.5: Concepts that apply labels to the building blocks

```
1  <concept id="green:BelongsToPredefined">
2      <requiresConcept refId="static:LabelBuildingBlock"/>
3      <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
4      <requiresConcept refId="static:LabelPredefinedBuildingBlockAnnotation"
           />
5      <description>Create relationship from bb to predefinedBuildingBlock</
           description>
6      <cypher><![CDATA[
7      MATCH (b:BuildingBlock)-[:ANNOTATED_BY]->(
           p:PredefinedBuildingBlockAnnotation)-[:OF_TYPE]->(
           predefinedBuildingBlock:PredefinedBuildingBlock)
8      CREATE (b)-[:BELONGS_TO_PREDEFINED]->(predefinedBuildingBlock)
9      RETURN b AS BuildingBlock, predefinedBuildingBlock AS Predefined
10     ]]></cypher>
11 </concept>
12
13 <concept id="green:PredefinedUses">
14     <requiresConcept refId="static:LabelBuildingBlockAnnotation"/>
15     <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
16     <description>Create relationship between PredefinedBuildingBlocks. (
           green arrow)</description>
17     <cypher><![CDATA[
18     MATCH (predefinedBuildingBlock:PredefinedBuildingBlock)-[:ANNOTATED_BY
           ]->(annotation:BuildingBlockAnnotation)-[:HAS]->(attribute:Value)-[
           :CONTAINS]->()-[:IS]->(s)
19     WHERE attribute.name ='uses' OR attribute.name = 'value'
20     CREATE (predefinedBuildingBlock)-[:PREDEFINED_USES]->(s)
```

```
21     RETURN predefinedBuildingBlock,s
22     ]]></cypher>
23 </concept>
24
25 <concept id="green:PredefinedAllowsUse">
26     <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
27     <requiresConcept refId="green:PredefinedUses"/>
28     <requiresConcept refId="green:BelongsToPredefined"/>
29     <description>Creates a relationship between building blocks, if their
           predefinedBuildingBlocks have a uses-relationship</description>
30     <cypher><![CDATA[
31     MATCH (t:Type)-[:BELONGS_TO_PREDEFINED]->(p1:PredefinedBuildingBlock)
32     -[:PREDEFINED_USES]->(p2:PredefinedBuildingBlock)<-[
           :BELONGS_TO_PREDEFINED]-(t2:Type)
33     CREATE (t)-[:PREDEFINED_ALLOWS_USE]->(t2)
34     RETURN t, t2
35     ]]></cypher>
36 </concept>
```

Listing A.6: Concepts for adding conceptual relationships and labels

```
1 <concept id="blue:IncludeSubpackagesFalse">
2     <requiresConcept refId="static:LabelBuildingBlock"/>
3     <requiresConcept refId="static:LabelPredefinedBuildingBlockAnnotation"
          />
4     <description>Set property includeSubPackages = false to building block
           if subpackages are not included</description>
5     <cypher><![CDATA[
6     MATCH (element:Type)-[:ANNOTATED_BY]->(annotation)
7     -[:OF_TYPE]->(predefinedBuildingBlock)
8     -[:ANNOTATED_BY]->(b:BuildingBlockAnnotation)-[:HAS]->(attribute:Value
          {name:'includeSubPackages'})
9     WHERE attribute.value = false
10    SET element.includeSubPackages = false
11    RETURN element AS includeSubPackagesFalse
12 ]]></cypher>
13 </concept>
14
15 <concept id="blue:IncludeSubpackagesTrue">
```

```
16    <requiresConcept refId="blue:IncludeSubpackagesFalse"/>
17    <description>Set property includeSubPackages = true to building block
          if subpackages are included</description>
18    <cypher><![CDATA[
19    MATCH (element:BuildingBlock)
20    WHERE NOT EXISTS (element.includeSubPackages)
21    SET element.includeSubPackages = true
22    RETURN element AS IncludeSubpackagesTrue
23    ]]></cypher>
24 </concept>
25
26 <concept id="blue:BelongsToBBFalse">
27    <requiresConcept refId="static:LabelBuildingBlock"/>
28    <requiresConcept refId="blue:IncludeSubpackagesFalse"/>
29    <description>Create relationship from class to their building block in
           case include building block is false.</description>
30    <cypher><![CDATA[
31    MATCH (b:BuildingBlock)<-[:CONTAINS]-(p1:Package)-[:CONTAINS]->(t:Type
          )
32    WHERE b.includeSubPackages = false
33    CREATE (t)-[:BELONGS_TO_BB]->(b)
34    RETURN b AS BuildingBlock
35    ]]></cypher>
36 </concept>
37
38 <concept id="blue:BelongsToBBTrue">
39    <requiresConcept refId="static:LabelBuildingBlock"/>
40    <requiresConcept refId="blue:IncludeSubpackagesTrue"/>
41    <description>Create relationship from class to their building blocks
          in case includesubmodule is true</description>
42    <cypher><![CDATA[
43    MATCH (b:BuildingBlock)<-[:CONTAINS]-(p1:Package)-[:CONTAINS*0..42]->(
          p2:Package)-[:CONTAINS]->(t:Type)
44    WHERE b.includeSubPackages = true
45    CREATE (t)-[:BELONGS_TO_BB]->(b)
46    RETURN b AS BuildingBlock
47    ]]></cypher>
48 </concept>
```

```
49
50 <concept id="blue:CodeDependencyBB">
51     <requiresConcept refId="static:LabelBuildingBlock"/>
52     <requiresConcept refId="blue:BelongsToBBTrue"/>
53     <requiresConcept refId="blue:BelongsToBBFalse"/>
54     <description>Matches code-dependecy between BuildingBlocks (blue arrow
           )</description>
55     <cypher><![CDATA[
56     MATCH (s1:BuildingBlock)<-[:BELONGS_TO_BB]-(t1:Type)-[:DEPENDS_ON]->(
           t2:Type)-[:BELONGS_TO_BB]->(s2:BuildingBlock)
57     WHERE s1 <> s2
58     AND NOT t1:BuildingBlock
59     AND NOT t2:BuildingBlock
60     AND NOT s1:ApplicationAnnotation
61     AND NOT s2:ApplicationAnnotation
62     MERGE (s1)-[:CODE_DEPENDENCY_BB]->(s2)
63     RETURN s1 AS BuildingBlock1, s2 AS BuildingBlock2
64     ]]></cypher>
65 </concept>
66
67 <concept id="blue:CodeDependencyPredefined">
68     <requiresConcept refId="static:LabelBuildingBlock"/>
69     <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
70     <requiresConcept refId="blue:CodeDependencyBB"/>
71     <requiresConcept refId="green:BelongsToPredefined"/>
72     <description>Matches code-dependecy between PredefinedBuildingBlocks (
           blue arrow)</description>
73     <cypher><![CDATA[
74     MATCH (p1:PredefinedBuildingBlock)<-[:BELONGS_TO_PREDEFINED]-(
           s1:BuildingBlock)-[:CODE_DEPENDENCY_BB]->(s2:BuildingBlock)-[
           :BELONGS_TO_PREDEFINED]->(p2:PredefinedBuildingBlock)
75     WHERE p1 <> p2
76     AND NOT s1:ApplicationAnnotation
77     AND NOT s2:ApplicationAnnotation
78     MERGE (p1)-[:CODE_DEPENDENCY_PREDEFINED]->(p2)
79     RETURN p1 AS PredefinedBuildingBlock1, p2 AS PredefinedBuildingBlock2
80     ]]></cypher>
81 </concept>
```

Listing A.7: Concepts for adding code dependencies

```
<concept id="resp:CreateAllowsAnnotationsRel">
    <requiresConcept refId="static:LabelBuildingBlockAnnotation"/>
    <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
    <description>Finds allowed annotations from building block Annotation
        and creates a relationship between them. Also sets LABEL to allowed
         annotation </description>
    <cypher><![CDATA[
    MATCH (p:PredefinedBuildingBlock)-[:ANNOTATED_BY]->(
        ba:BuildingBlockAnnotation)
    -[:HAS]->(attribute:Value {name:'allowedAnnotations'})
    -[:CONTAINS]->()-[:IS]->(allowedAnno)
    SET allowedAnno:AllowedAnnotation
    MERGE (p)-[:ALLOWS_ANNOTATION]->(allowedAnno)
    RETURN p AS PredefinedBuildingBlock, allowedAnno AS AllowedAnnotations
    ]]></cypher>
</concept>

<concept id="resp:ForbidsAnnotationsRel">
    <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
    <requiresConcept refId="resp:CreateAllowsAnnotationsRel"/>
    <description>Creates a relationship between building blocks and
        annotations, if the annotation are forbidden to be used in said
        building block</description>
    <cypher><![CDATA[
    MATCH (p1:PredefinedBuildingBlock)-[:ANNOTATED_BY]->()
    -[:OF_TYPE]->(type:Java)
    <-[:OF_TYPE]-()<-[:ANNOTATED_BY]-(p2:PredefinedBuildingBlock)
    -[:ALLOWS_ANNOTATION]->(a)
    WHERE NOT (p1)-[:ALLOWS_ANNOTATION]->(a)
    MERGE (p1)-[:FORBIDS_ANNOTATION]->(a)
    RETURN p1 AS PredefinedBuildingBlock, a AS ForbiddenAnnotation
    ]]></cypher>
</concept>

<concept id="resp:UsesAnnotationsRel">
```

```
31     <requiresConcept refId="static:LabelBuildingBlock"/>
32     <description>Creates a relationship between building block and
           annotations, if the annotations are used, regardless of forbidden
           or allowed</description>
33     <cypher><![CDATA[
34     MATCH (b:BuildingBlock)<-[:BELONGS_TO_BB]-(t:Type)
35     -[:ANNOTATED_BY]->()-[:OF_TYPE]->(a)
36     MERGE (b)-[:USES_ANNOTATION]->(a)
37     RETURN b AS BuildingBlock, a AS UsedAnnotation
38     ]]></cypher>
39 </concept>
40
41 <concept id="resp:UsesAnnotationsBB">
42     <requiresConcept refId="static:LabelBuildingBlock"/>
43     <description>Creates a relationship between building block and
           annotations, if the annotation are used by block itself, regardless
            of forbidden or allowed</description>
44     <cypher><![CDATA[
45     MATCH (b:BuildingBlock)
46     -[:ANNOTATED_BY]->()-[:OF_TYPE]->(a)
47     CREATE  (b)-[:USES_ANNOTATION]->(a)
48     RETURN b AS BuildingBlock, a AS UsedAnnotation
49     ]]></cypher>
50 </concept>
51
52 <concept id="resp:AllowsAcyclic">
53     <requiresConcept refId="static:LabelBuildingBlockAnnotation"/>
54     <description>Labels PBB that allow acyclic dependencies within
           themselves</description>
55     <cypher><![CDATA[
56     MATCH (t:Type)-[:ANNOTATED_BY]->(annotation)
57     -[:OF_TYPE]->(predefinedBuildingBlock)
58     -[:ANNOTATED_BY]->(b:BuildingBlockAnnotation)-[:HAS]->(attribute:Value
           {name:'allowAcyclic'})
59     SET predefinedBuildingBlock:AllowsAcyclic
60     RETURN t
61     ]]></cypher>
62 </concept>
```

Listing A.8: Concepts that add additional data for requirements

```
<constraint id="constraintElement:CheckForDomainCycles">
    <requiresConcept refId="resp:AllowsAcyclic"/>
    <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
    <requiresConcept refId="static:LabelBuildingBlock"/>
    <requiresConcept refId="green:BelongsToPredefined"/>
    <requiresConcept refId="blue:CodeDependencyBB"/>
    <description>Returns any cycles between domains </description>
    <cypher><![CDATA[
        MATCH
        (p:PredefinedBuildingBlock:AllowsAcyclic)<-[:BELONGS_TO_PREDEFINED
            ]-(b1:BuildingBlock)-[:CODE_DEPENDENCY_BB]->(b2:BuildingBlock)-[
            :BELONGS_TO_PREDEFINED]->(p),
        path=shortestPath((b2)-[:CODE_DEPENDENCY_BB*]->(b1))
    WHERE
        b1<>b2
    RETURN
        b1 AS BuildingBlock, EXTRACT(p IN nodes(path) | p.fqn) AS Cycle
     ORDER BY
            BuildingBlock.fqn
    ]]></cypher>
    <verify>
        <rowCount max="0"/>
    </verify>
</constraint>
```

Listing A.9: Constraints for element rules

```
<constraint id="constraintImperative:CodeMatchesDescriptorBB">
    <requiresConcept refId="green:PredefinedAllowsUse"/>
    <requiresConcept refId="blue:CodeDependencyBB"/>
    <description>Returns buildingblocks with a code-dependency without a
        conceptual dependency</description>
    <cypher><![CDATA[
        MATCH
            (b1:BuildingBlock)-[:CODE_DEPENDENCY_BB]->(b2:BuildingBlock)
        WHERE NOT
```

```
9           (b1)-[:PREDEFINED_ALLOWS_USE]->(b2)
10          AND NOT b1:ApplicationAnnotation
11          AND NOT b2:ApplicationAnnotation
12      RETURN
13          b1 AS ForbiddenOutgoingDependency, b2 AS
                ForbiddenIncomingDependency
14  ]]></cypher>
15  <verify>
16      <rowCount max="0"/>
17  </verify>
18 </constraint>
19
20 <constraint  id="constraintImperative:CodeMatchesDescriptor">
21  <requiresConcept refId="green:PredefinedUses"/>
22  <requiresConcept refId="blue:CodeDependencyPredefined"/>
23  <description>Returns predefined buildingblocks with a code dependency
        without a conceptual relationship</description>
24  <cypher><![CDATA[
25      MATCH
26          (p1:PredefinedBuildingBlock)-[:CODE_DEPENDENCY_PREDEFINED]->(
                p2:PredefinedBuildingBlocks)
27      WHERE NOT
28          (p1)-[:PREDEFINED_USES]->(p2)
29      RETURN
30          p1 AS ForbiddenOutgoingDependency, p2 AS
                ForbiddenIncomingDependency
31  ]]></cypher>
32  <verify>
33      <rowCount max="0"/>
34  </verify>
35 </constraint>
36
37 <constraint  id="constraintImperative:EveryClassBelongsToBB">
38  <requiresConcept refId="blue:BelongsToBBFalse"/>
39  <requiresConcept refId="blue:BelongsToBBTrue"/>
40  <requiresConcept refId="green:BelongsToPredefined"/>
41  <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
42  <requiresConcept refId="static:LabelBuildingBlockAnnotationType"/>
```

```
43     <description>Returns classes that do not belong to a Building Block</
           description>
44     <cypher><![CDATA[
45         MATCH
46             (p:Package)-[:CONTAINS]->(t:Type)
47         WHERE NOT
48             (t)-[:BELONGS_TO_BB]->()
49         AND NOT
50             (t)-[:BELONGS_TO_PREDEFINED]->()
51         AND NOT
52             t:PredefinedBuildingBlock
53         AND NOT
54             t:BuildingBlockAnnotationType
55         RETURN t AS ClassWithoutBuildingBlock
56     ]]></cypher>
57     <verify>
58         <rowCount max="0"/>
59     </verify>
60 </constraint>
```

Listing A.10: Constraints for imperative rules

```
1 <constraint id="constraintProhibition:CheckForGreenCycles">
2     <requiresConcept refId="green:PredefinedUses"/>
3     <description>Returns PredefinedBuildingBlocks with a forbidden cycle
          in concept</description>
4     <cypher><![CDATA[
5     MATCH
6         (p1:PredefinedBuildingBlock)-[:PREDEFINED_USES]->(
              p2:PredefinedBuildingBlock),
7             path=shortestPath((p2)-[:PREDEFINED_USES*]->(p1))
8         WHERE
9             p1<>p2
10        RETURN
11            p1 AS PreBB, EXTRACT(p IN nodes(path) | p.fqn) AS Cycle
12        ORDER BY
13            PreBB.fqn
14    ]]></cypher>
15    <verify>
```

```
16        <rowCount max="0"/>
17    </verify>
18 </constraint>
19
20 <constraint  id="constraintProhibition:CheckForBlueCycles">
21    <requiresConcept refId="blue:CodeDependencyPredefined"/>
22    <description>Returns predefinedBuildingBlock with a forbidden cycle in
          code</description>
23    <cypher><![CDATA[
24        MATCH
25            (p1:PredefinedBuildingBlock)-[:CODE_DEPENDENCY_PREDEFINED]->(
                  p2:PredefinedBuildingBlock),
26             path=shortestPath((p2)-[:CODE_DEPENDENCY_PREDEFINED*]->(p1))
27        WHERE
28            p1<>p2
29        RETURN
30            p1 AS PreBB, EXTRACT(p IN nodes(path) | p.fqn) AS Cycle
31        ORDER BY
32            PreBB.fqn
33    ]]></cypher>
34    <verify>
35        <rowCount max="0"/>
36    </verify>
37 </constraint>
38
39 <constraint  id="constraintProhibition:CheckForForbiddenAnnotations">
40    <requiresConcept refId="resp:ForbidsAnnotationsRel"/>
41    <requiresConcept refId="resp:UsesAnnotationsRel"/>
42    <description>Returns used annotations, where predefinedBuildingBlock
          forbids them  </description>
43    <cypher><![CDATA[
44            MATCH
45                (p:PredefinedBuildingBlock)<-[:BELONGS_TO_PREDEFINED]
46                -(b:BuildingBlock)-[:USES_ANNOTATION]
47                ->(a)
48            WHERE
49                (p)-[:FORBIDS_ANNOTATION]->(a)
50            RETURN
```

```
51              b AS BuildingBlock, a AS ForbiddenAnnotation
52      ]]></cypher>
53      <verify>
54          <rowCount max="0"/>
55      </verify>
56  </constraint>
57
58  <constraint  id="constraintProhibition:UsesOnlyPredefined">
59      <requiresConcept refId="green:PredefinedUses"/>
60      <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
61      <description>Returns any classes that are descriptively used by
            PredefinedBuildingBlocks that are not plocks as well  </description
            >
62      <cypher><![CDATA[
63          MATCH
64              (p:PredefinedBuildingBlock)-[:PREDEFINED_USES]->(p2)
65          WHERE NOT
66              (p2:PredefinedBuildingBlock)
67          RETURN
68              p AS PredefinedUses, p2 AS WronglyUsedClass
69      ]]></cypher>
70      <verify>
71          <rowCount max="0"/>
72      </verify>
73  </constraint>
74
75  <constraint  id="constraintProhibition:UsesOnlyAllowedExceptions">
76      <requiresConcept refId="green:PredefinedUses"/>
77      <requiresConcept refId="static:LabelPredefinedBuildingBlock"/>
78      <requiresConcept refId="static:LabelBuildingBlock"/>
79      <requiresConcept refId="resp:CreateAllowsExceptionsRel"/>
80      <description>Returns exceptions, that are not allowed by Predefined</
            description>
81      <cypher><![CDATA[
82          MATCH
83              (p:PredefinedBuildingBlock)<-[:BELONGS_TO_PREDEFINED]-(
                    b:BuildingBlock)
```

```
84              <-[:BELONGS_TO_BB]-(t:Type)-[:DECLARES]->()-[:THROWS]->(
                    exception)
85          WHERE NOT
86              (p)-[:ALLOWS_EXCEPTION]->(exception)
87          RETURN
88              exception AS ForbiddenExcepion, t AS Offender
89          ]]></cypher>
90      <verify>
91          <rowCount max="0"/>
92      </verify>
93  </constraint>
94
95  <constraint  id="constraintProhibition:ReflexiveCycles">
96      <requiresConcept refId="green:BelongsToPredefined"/>
97      <requiresConcept refId="blue:BelongsToBBFalse"/>
98      <requiresConcept refId="blue:BelongsToBBTrue"/>
99      <description>Returns dependencies within the Packages(Masters), except
            for Domain</description>
100     <cypher><![CDATA[
101         MATCH
102             (p:PredefinedBuildingBlock)<-[:BELONGS_TO_PREDEFINED]-(
                    b:BuildingBlock)
103             <-[:BELONGS_TO_BB]-(t:Type)-[:DEPENDS_ON]->(t2:Type)-[
                    :BELONGS_TO_BB]->(b2:BuildingBlock)-[:BELONGS_TO_PREDEFINED
                    ]->(p)
104         WHERE NOT
105             p:AllowsAcyclic
106         RETURN t AS OutgoingDependency, t2 AS IncomingDependency
107     ]]></cypher>
108     <verify>
109         <rowCount max="0"/>
110     </verify>
111 </constraint>
```

Listing A.11: Constraints for prohibition rules

```
1  <dependency>
2      <groupId>com.tpgroup.buildingblocks.annotation</groupId>
3      <artifactId>building-blocks-annotation</artifactId>
```

```
4     <version>1.0-SNAPSHOT</version>
5 </dependency>
6
7 <build>
8     <plugins>
9         <plugin>
10            <groupId>com.buschmais.jqassistant</groupId>
11            <artifactId>jqassistant-maven-plugin</artifactId>
12            <version>1.4.0</version>
13            <executions>
14                <execution>
15                    <goals>
16                        <goal>scan</goal>
17                        <goal>analyze</goal>
18                    </goals>
19                    <configuration>
20                        <concepts>
21                            <concept>classpath:Resolve</concept>
22                        </concepts>
23                        <warnOnSeverity>MINOR</warnOnSeverity>
24                        <failOnSeverity>MAJOR</failOnSeverity>
25                        <groups>
26                            <group>static</group>
27                            <group>green</group>
28                            <group>blue</group>
29                            <group>responsibilities</group>
30                            <group>constraintElement</group>
31                            <group>constraintImperative</group>
32                            <group>constraintProhibition</group>
33                        </groups>
34                    </configuration>
35                </execution>
36            </executions>
37        </plugin>
38    </plugins>
39 </build>
40
41 <reporting>
```

```
42    <plugins>
43       <plugin>
44          <groupId>com.buschmais.jqassistant</groupId>
45          <artifactId>jqassistant-maven-plugin</artifactId>
46          <reportSets>
47             <reportSet>
48                <reports>
49                   <report>report</report>
50                </reports>
51             </reportSet>
52          </reportSets>
53       </plugin>
54    </plugins>
55 </reporting>
```

Listing A.12: jQA plugin configuration in POM

List of Figures

List of Tables

Name: Laura Robien Baldrich — Matriculation number: 888811

Honesty disclaimer

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.

Ulm, ...

Laura Robien Baldrich