Trace Link Recovery using Static Program Analysis

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

Maximilian Meffert

Erstgutachter:	Prof. Dr. Ralf Lämmel Institut für Informatik
Zweitgutachter:	Msc. Johannes Härtel Institut für Informatik

Koblenz, im Dezember 2017

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Maximilian Meffert)

Zusammenfassung

TBD.

Abstract

TBD.

Acknowledgements

TBD.

Contents

List of Theorems

List of Figures

Chapter 1

Introduction

We start with a motivational example: Common tasks in software development are implementation of serialization and persistence of domain models. For instance, consider a simple web service which serves data via HTTP (Hypertext Transfer Protocol) as XML (Extensible Markup Language) and stores it in a relational database. We call this an O/R/X-Mapping (Object-Relational- and XML-Mapping) scenario. Given such a system, the same conceptual data, i.e. the domain model, is transported through application tiers in different forms, that is, the same data is represented by various manifestations at a time. Each manifestation involves another software language and technology. Figure 1.1 opposes model-

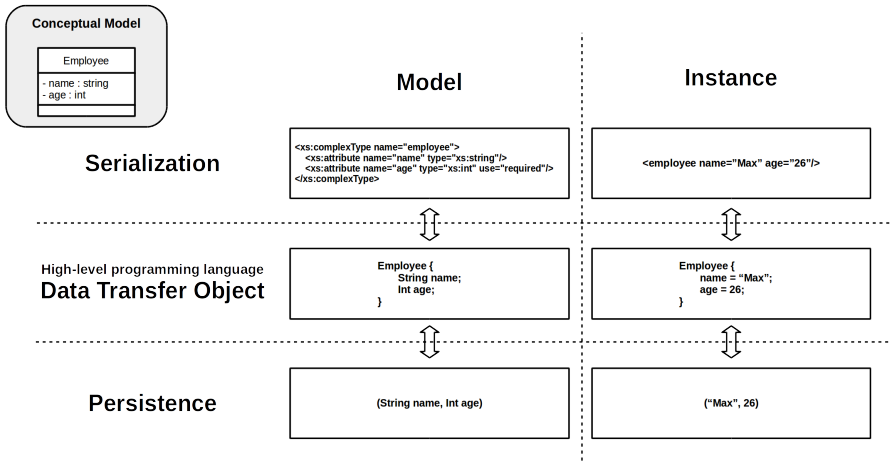


Figure 1.1 O/R/X-Mapping Manifestations

to instance-level syntaxes of different manifestations for a conceptual model for employees in an O/R/X-Mapping scenario. One employee only has a *name* and an *age* attribute. Persistence is presented in tuple notation, data transfer objects of an application tier are represented with a fictional high level programming language object notation and serialization is exemplified with XML and XSD (XML Schema Definition) syntax.

Although figure 1.1 shows a fictionalized example, one can observe structural similarities among the different model and instance representation syntaxes. Such similarities also occur in real-world software engineering through use of conventions which may be predefined in technologies for O/R/X-Mapping. Fig-

```
@XmlRootElement(name="employee")
@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {

    @XmlAttribute
    private int id;

    @XmlAttribute
    private String name;

    @XmlAttribute
    private int age;

    @XmlAttribute
    private double salary;

    private Department department;

    private Department managedDepartment;

}
```

```
<xs:complexType name="employee">
 <xs:attribute name="id" type="xs:int"
       use="required"/>

 <xs:attribute name="name" type="xs:string"/>

 <xs:attribute name="age" type="xs:int"
       use="required"/>

 <xs:attribute name="salary" type="xs:double"
       use="required"/>

 <xs:sequence>

   <xs:element ref="department" minOccurs="0"/>

   <xs:element name="managedDepartment"
         type="department" minOccurs="0"/>

 </xs:sequence>
</xs:complexType>
```

```
<employee name="Max" age="26" salary="55000.0"/>
```

Figure 1.2 JAXB XML/XSD Mapping

ure 1.2 displays an annotated Java class and XML/XSD output generated by JAXB (Java Architecture for XML Binding). Except for the root element all names for attributes and elements are taken from the Java class. Moreover, the XSD complex type and the Java class share a similar nested structure. Links through similarity can be found within the model-level representations (Java-XSD) and between model- and instance-level manifestations (Java-XML and XSD-XML), for instance:

- `public class Employee {...}` is linked with `<xs:complextType name="employee">...</xs:complexType>` and `<employee .../>`, the latter two are also linked with each other

- `private String name;` is linked with `<xs:attribute name="name" type="xs:string"/>` and `name="Max"`, the latter two are also linked with each other

1.1 Objectives

The aim of this thesis is to provide automated recovery of such similarities as semantic links. Recovered links are inserted into *megamodels* [1] [3] for *linguistic architectures* [2] [12] [10]. Megamodels are models providing a high level of abstraction with other models as modeling elements, e.g. a megamodel may describe the dependencies between metamodels, models and instances. Linguistic architectures intend to describe software systems from a language centric point of view. They model knowledge about software systems in terms of languages, artifacts, technologies, etc. Such entities are interrelated with relationships derived from common software engineering and theoretical computer science vocabulary providing special semantics, e.g. *defines*, *isA*, *instanceOf*, *represents*, *implements*, *realizationOf*, *elementOf*, *subsetOf*.

Linguistic architectures are related to ER Models (Entity-Relationship Models) and ontologies. Recovering semantic links as described above is related to the concept *traceability* and an application of *traceability recovery* [8] (see §2.2). In context of traceability, semantic links may be called *trace links*, however, both establish a relation between two entities denoting a certain meaning.

Trace links among software artifacts denoting that one artifact encodes the same information as the other are called *correspondence* links and denoted *correspondsTo* (see §2.1.3). Trace links between two artifacts denoting that one defines the other, in the sense of a metamodel defining a model, are called *conformance* links and denoted *conformsTo* (see §2.1.4). The main objective is to recover such trace links among well-formed, possibly partial, source code artifacts. We call well-formed partial source code artifacts *fragments* (see §2.1.2).

1.2 Approach

Our approach for recovering trace links utilizes *static program analysis*. We use it in the broad sense that we construct specialized ASTs (Abstract Syntax Trees)

for further analysis. Usually static program analysis is used with the purpose of formal verification or measurement of software engineering related properties of source code, e.g. cohesion and coupling metrics. We use static program analysis to semantically link code fragments and uncover knowledge of the analyzed artifacts. In that respect, our approach is related to computing software metrics.

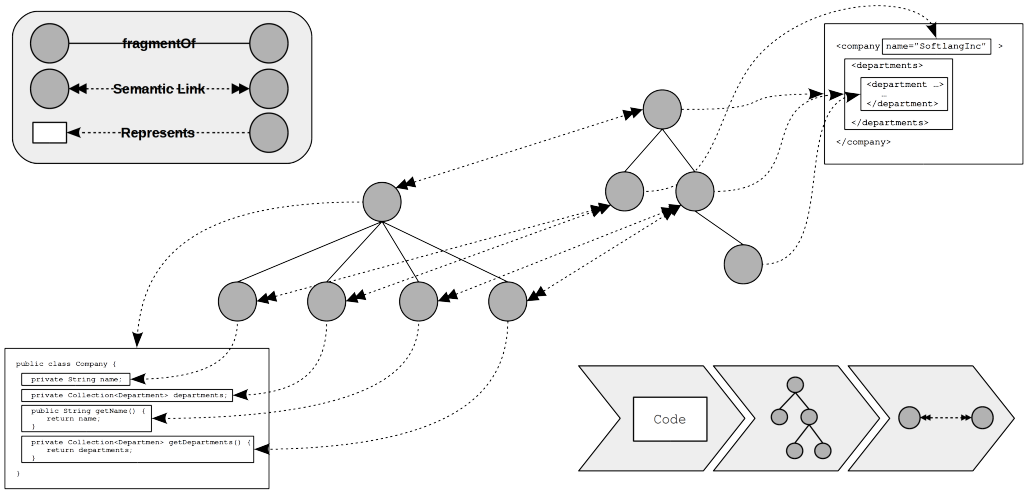


Figure 1.3 Recovery Approach

Figure 1.3 shows a schematic illustration of our recovery approach. It can be summarized with the following three steps:

1. we create two Parse Trees for both inputs respectively
2. Parse Trees are transformed to specialized ASTs; such ASTs are not intended to be used for compilation, instead each node represents a part of code which may be linked during further analysis; a child node represents a piece of code which is properly embedded the code represented by its parent
3. we apply a comparative analysis to both ASTs; while traversing through both trees we check whether each pair of nodes is a trace link

1.3 Contributions & Non-Contributions

This section lists critical contributions and non-contributions of this thesis:

Contribution 1 We contribute a Java based API (Application Programming Interface) for recovering trace links among artifacts and their constituent fragments, providing utilities for static program analysis as described in §1.2 (see §4).

Contribution 2 We contribute an implementation for correspondence and conformance link recovery among JAXB and Hibernate artifacts utilizing the created API and axioms for linguistic architectures (see §5 and §2.1 respectively).

Contribution 3 We contribute a small case-study applying the implemented recovery system to a minimal O/R/X-Mapping program (see §**??**).

Non-Contribution 1 We do not contribute to the axiomatization of linguistic architectures as described in [12], [3], [11] and [10].

Chapter 2

Background

This chapter provides the necessary theoretical an technological background topics of the thesis.

2.1 Axioms of Linguistic Architectures

This section summarizes axioms of linguistics architectures. Axioms are outlined in [11] and refined in [10]. §2.1.1 introduces the axiomatization for parthood §2.1.2

Axioms are presented in First Order Logic and formalization is taken from [10]. The universe to draw elements from is represented by the Entity-predicate:

$$\forall x.\mathsf{Entity}(x).$$

We assume it holds for everything of interest, hence such things are called *entities*.

Linguistic architectures intend to describe software from language centric point of view, thus we provide specializations of entities for languages:

$$\mathsf{Set}(x) \Rightarrow \mathsf{Entity}(x).$$
$$\mathsf{Language}(x) \Rightarrow \mathsf{Set}(x).$$

There are entities representing sets in a mathematical sense, and there are sets representing (formal-) languages in the sense of theoretical computer science.

On the other hand, we provide specializations for entities involved in software engineering terminology:

$$\mathsf{Artifact}(x) \Rightarrow \mathsf{Entity}(x).$$
$$\mathsf{File}(x) \Rightarrow \mathsf{Artifact}(x).$$
$$\mathsf{Folder}(x) \Rightarrow \mathsf{Artifact}(x).$$

There are entities representing all kinds of digital artifacts, e.g. files and folders. Files represent persistent data resources, locatable either trough file systems or web services. Folders represent locatable collections of files. The intended use and semantic of these predicates is not meant to differ from intuitive, every day use.

2.1.1 Parthood

Parthood is the essential relationship when reasoning about correspondence and conformance among artifacts within linguistic architectures [11] [10]. It describes the relation between entities and their constituent parts. The study of parthood and its derivatives is mereology [15] [14]. In the context of linguistic architectures we assume most entities to be composed of several conceptual or physical parts. In short, such entities are the sum of its parts. That is, programs may compiled from many files, systems consists of several disjoint but dependent components, a Java class is made up of methods and fields, etc. Furthermore such entities are considered to be *mereologically invariant*, i.e. if one part changes, the whole changes as well. Axiom 1 captures parthood at its most basic level.

Axiom 1 (partOf)

$$\mathit{partOf}(p, w) \Rightarrow \mathit{Entity}(p) \wedge \mathit{Entity}(w).$$
$$\mathit{partOf}(p, w) \Leftarrow p \textit{ is a constituent part of } w.$$

Parthood is usually considered to be reflexive, antisymmetric and transitive [15] [14], thus facilitating a partial order:

$$\mathsf{partOf}(p, p). \qquad \text{Reflxivity}$$
$$\mathsf{partOf}(p, w) \wedge \mathsf{partOf}(w, p) \Rightarrow p = w. \qquad \text{Antisymmertry}$$
$$\mathsf{partOf}(p, w) \wedge \mathsf{partOf}(w, u) \Rightarrow \mathsf{partOf}(p, u). \qquad \text{Transitivity}$$

The irreflexive parthood relationship is called proper:

$$\mathsf{properPartOf}(p, w) \Rightarrow \mathsf{Entity}(p) \wedge \mathsf{Entity}(w).$$
$$\mathsf{properPartOf}(x, y) \Leftarrow \mathsf{partOf}(x, y) \wedge \neg\mathsf{partOf}(y, x).$$

Proper parthood is the strict order induced by simple parthood. Figure 2.1 shows a schematic illustration opposing simple to proper parthood. This Venn-style

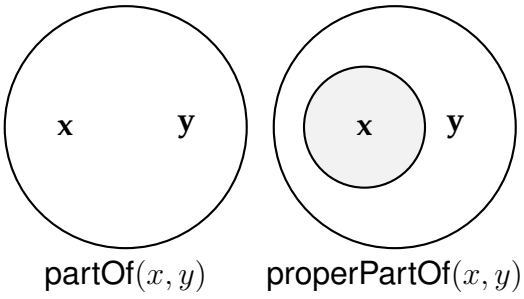


Figure 2.1 Simple vs. Proper Parthood

diagram depicts two scenarios

partOf(x,y) x may be a part of y or vice-versa; x and y cannot be distinguished

properPartOf(x,y) x is certainly a part of y, however y is not a part of x

In general proper parthood is a asymmetric relationship:

$$\mathsf{properPartOf}(x, y) \Rightarrow \neg\mathsf{properPartOf}(y, x). \qquad \text{Asymmetry}$$

Mereology also allows for the notion of atomicity [15] [14], i.e. atomic parts which cannot be decomposed in further parts:

$$\mathsf{atomicPart}(x) \Leftarrow \nexists p.\mathsf{properPartOf}(p, x).$$

Atomicity is later used to distinguish cases for correspondence and conformance (see §2.1.3 and §2.1.4).

2.1.2 Fragments

Fragments are a specialization of entities intended to capture the endogenous, mereological decomposition of artifacts [10]. Axiom 2 defines fragments as artifacts, which are neither files nor folders, and are properly embedded in at least one other artifact. For instance, consider the syntactical decomposition of Java classes, i.e. the class fragment contains method and field fragments.

Axiom 2 (Fragment)

$$\textit{Fragment}(f) \Rightarrow \textit{Artifact}(a) \wedge \neg(\textit{File}(f) \vee \textit{Folder}(f)).$$
$$\textit{Fragment}(f) \Rightarrow \exists a.\textit{Artifact}(a) \wedge \textit{properPartOf}(f, a).$$

We emphasize the proper parthood here, since not all authors necessarily consider partOf to be reflexive [11] [10]. However, we previously distinguished between reflexive and irreflexive parthood and if we were to use reflexive parthood for the definition of the Fragment-predicate it would be a tautology:

$$\mathsf{Fragment}(f) \Rightarrow \exists a.\mathsf{Artifact}(a) \wedge \mathsf{partOf}(f, a). \qquad \text{Tautology}$$

Since $\mathsf{Fragment}(f)$ implies $\mathsf{Artifact}(f)$ and partOf is reflexive there is always an artifact for a fragment the latter is a part of, that is the fragment itself.

From the specification of fragments we can first specialize the proper parthood predicate:

$$\mathsf{fragmentOf}(f, x) \Rightarrow \mathsf{Fragment}(f) \wedge \mathsf{Artifact}(x).$$
$$\mathsf{fragmentOf}(f, x) \Leftarrow \mathsf{Fragment}(f) \wedge \mathsf{Artifact}(x) \wedge \mathsf{properPartOf}(f, x).$$

This is just a restriction of domain and range facilitating a special semantic. Secondly, we can describe the process of fragmentation:

$$\mathsf{partOf}(p, w) \wedge \mathsf{Fragment}(w) \Rightarrow \mathsf{Fragment}(p).$$

The fragment nature propagates top-down alongside the order of parthood, i.e. if an entity is a fragment all parts are also fragments [10].

2.1.3 Correspondence

Correspondence is the relationship between to artifacts denoting both represent the same data in the sense that they are mereologically similar [10]. It usually occurs as exogenous relationship, i.e. the involved artifacts do not belong to the same language, for instance XML and JSON (JavaScript Object Notation) may contain the same information.

In order to axiomatize correspondence properly we need to introduce two other relationships first:

represents The relationship denoting that an artifact is a representation or manifestation of an entity.

$$\mathsf{represents}(a, e) \Rightarrow \mathsf{Artifact}(a) \wedge \mathsf{Entity}(e).$$
$$\mathsf{represents}(a, e) \Leftarrow a \text{ is a representation of } e.$$

sameAs The relationship denoting two artifacts represent the same entity.

$$\mathsf{sameAs}(x, y) \Rightarrow \mathsf{Artifact}(x) \wedge \mathsf{Artifact}(y).$$
$$\mathsf{sameAs}(x, y) \Leftarrow \exists e.\mathsf{represents}(x, e) \wedge \mathsf{represents}(y, e).$$

The axiomatization of correspondence from [10] is slightly altered with emphasis of irreflexive proper parthood, Otherwise it would not work with axiom 1.

Axiom 3 (correspondsTo)

$$\begin{aligned}&\mathit{correspondsTo}(x, y) \Rightarrow \mathit{Artifact}(x) \wedge \mathit{Artifact}(y).\\ &\mathit{correspondsTo}(x, y) \Leftarrow (\forall px.\mathit{properPartOf}(px, x) \Rightarrow \exists py.\mathit{properPartOf}(py, y) \wedge \mathit{correspondsTo}(px, py))\\ &\qquad \wedge (\forall py.\mathit{properPartOf}(py, y) \Rightarrow \exists px.\mathit{properPartOf}(px, x) \wedge \mathit{correspondsTo}(py, px))\\ &\qquad \vee (\nexists p.\mathit{properPartOf}(p, x) \vee \mathit{properPartOf}(p, y)) \wedge \mathit{sameAs}(x, y).\end{aligned}$$

Axiom 3 defines correspondence recursively:

Case I If both artifacts is atomic in the sense it cannot be decomposed in further parts, we check whether both artifacts represent the same data.

Case II Else, if at least one artifact is not atomic, then for each proper part of one artifact has to be a corresponding part in the other and vice versa.

This axiomatization demands a strict one-to-one correspondence in its recursive clause which occurs in reflexive cases, i.e. $\mathsf{correspondsTo}(x, x)$, but may be to strict for real-world applications [11].

2.1.4 Conformance

Conformance is the relationship between two artifacts denoting one satisfies the syntax specification given by the other [10]. This satisfaction is also meant to mereologically decomposable, i.e. there are parts of one artifact specified by a part of the other.

In order to axiomatize conformance properly we need to introduce two other relationships first:

defines The relationship simply denoting an artifact contains the definition of an entity.

$$\mathsf{defines}(a, e) \Rightarrow \mathsf{Artifact}(a) \wedge \mathsf{Entity}(e).$$
$$\mathsf{defines}(a, e) \Leftarrow a \text{ is a definition for } e.$$

elementOf The relationship denoting set-theoretic membership.[1]

$$\mathsf{elementOf}(e, s) \Rightarrow \mathsf{Entity}(e) \wedge \mathsf{Set}(s).$$
$$\mathsf{elementOf}(e, s) \Leftarrow e \in s.$$

The axiomatization of conformance from [10] is also slightly altered with emphasis of irreflexive proper parthood, Otherwise it would not work with axiom 1.

[1] We do not use the elementOfaxiom for artifacts and languages from [10], because it introduces the possibility for infinite mutual recursion:

$$\mathsf{elementOf}(a, l) \Rightarrow \mathsf{Artifact}(a) \wedge \mathsf{Language}(l).$$
$$\mathsf{elementOf}(a, l) \Leftarrow \exists s.\mathsf{defines}(s, l) \wedge \mathsf{conformsTo}(a, s)$$

Axiom 4 (conformsTo)

$$\textit{conformsTo}(a, d) \Rightarrow \textit{Artifact}(a) \wedge \textit{Artifact}(d).$$
$$\textit{conformsTo}(a, d) \Leftarrow (\forall pa.\textit{properPartOf}(pa, a) \wedge \exists pd.\textit{properPartOf}(pd, d) \wedge \textit{conformsTo}(pa, pd)) \vee \exists l.\textit{defines}(d, l) \wedge \textit{elementOf}(a, l).$$

Axiom 4 defines conformance recursively:

Case I If both artifacts is atomic in the sense it cannot be decomposed in further parts, we check whether one artifact defines a language the other artifact is an element of.

Case II Else, if at least one artifact is not atomic, then for each proper part of one artifact has to be a proper part in the other artifact it conforms to.

2.2 Traceability

This section provides brief background on the topic of traceability, i.e. its objectives and terminology (see §2.2.1 and §2.2.2 respectively).

2.2.1 Traceability Objectives

Traceability, as a desired property of software systems and their development processes, originates from requirement engineering, i.e. how do we validate or verify that all requirements are met; or more precisely: on what data can we base such validation or verification? [16]

An answer to these questions is motivated by observations on development processes of software systems. Because such a process usually has an iterative and incremental nature, we can reasonably assume engineering activities to leave traces which reflect the performed action.

Modern day traceability objectives are not limited to validation of requirement compliance. It is also regarded as a desirable property for supporting engineering activities, e.g. change impact analysis or dependency analysis [8] [6]. This feeds to the overall topic of program comprehension. We can informally define traceability as *"[...] the ability to [...] interrelate uniquely identifiable entities in a way that matters."* [4] [16].

2.2.2 Traceability Terminology

This section recapitulates the necessary excerpt of definitions on traceability terms given by [8]. An excessive glossary of traceability terminology can be found in [7].

Definition 1 (Trace)

- ***(Noun)*** *A specified triplet of elements comprising: a source artifact, a target artifact and a trace link associating the two artifacts. [8]*
- ***(Verb)*** *The act of following a trace link from a source artifact to a target artifact or vice-versa. [8]*

Definition 2 (Trace Artifact, Source Artifact, Target Artifact) *A traceable unit of data (e.g., a single requirement, a cluster of requirements, a UML class, a UML class operation, a Java class or even a person). A trace artifact is one of the trace elements and is qualified as either a source artifact or as a target artifact when it participates in a trace:*

- ***Source Artifact*** *The artifact from which a trace originates.*
- ***Target Artifact*** *The artifact at the destination of a trace.*

[8]

Definition 3 (Trace Artifact Type) *A label that characterizes those trace artifacts that have the same or a similar structure (syntax) and/or purpose (semantics). For example, requirements, design and test cases may be distinct artifact types. [8]*

Definition 4 (Trace Link) *A specified association between a pair of artifacts, one comprising the source artifact and one comprising the target artifact. Trace links are effectively bidirectional. [8]*

Definition 5 (Trace Link Type) *A label that characterizes those trace links that have the same or similar structure (syntax) and/or purpose (semantics). For example, "implements", "tests", "refines" and "replaces" may be distinct trace link types. [8]*

Definition 6 (Traceability) *The potential for traces to be established and used. Traceability (i.e., trace "ability") is thereby an attribute of an artifact or of a collection of artifacts. [8]*

Definition 7 (Traceability Creation) *The general activity of associating two (or more) artifacts, by providing trace links between them. This can be done manually, automatically or semi-automatically, and additional annotations can be provided as desired to characterize attributes of the traces. [8]*

Definition 8 (Trace Capture) *A particular approach to trace creation that implies the creation of trace links concurrently with the creation of the artifacts that they associate. These trace links may be created automatically or semi-automatically using tools. [8]*

Definition 9 (Trace Recovery) *A particular approach to trace creation that implies the creation of trace links after the artifacts that they associate have been generated and manipulated. These trace links may be created automatically or semi-automatically using tools. [8]*

2.3 Technical Background

2.3.1 MegaL/Xtext

[1] [3]
[12] [2] [11]
[9]

2.3.2 Java Architecture for XML Binding (JAXB)

2.3.3 Hibernate

2.3.4 Another Tool For Language Recognition (ANTLR)

[13]

Chapter 3

Related Work

TBD.

Chapter 4

Design

This chapter summarizes the design of the recovery system developed for this thesis. §4.1 summarizes functional requirements of the developed system. §4.2 recapitulates design of the recovery system with respect to the specified requirements. §4.3 summarizes integration in the recovery system with the MegaL/Xtext environment .

4.1 Requirements

This section summarizes functional requirements for the recovery system developed as part of this thesis. All presented requirements are must haves, so no priority differentiation is applied.

Requirement 1 *Fragmen Recovery*. The recovery system has to recover fragments, i.e. syntactically well-formed parts of artifacts (see §2.1.2).

Requirement 2 *Parthood Recovery*. The recovery system has to recover parthood links of fragments (see §2.1.1 and §2.1.2)).

Requirement 3 *Correspondence Recovery*. The recovery system has to recover correspondence links, i.e. links capturing the relation between fragments of artifacts denoting a predefined similarity holds (see §2.1.3).

Requirement 4 *Conformance Recovery*. The recovery system has to recover conformance links, i.e. links capturing the relation between artifacts or fragments denoting that one defines the other (see §2.1.4).

Requirement 5 *MegaL/Xtext Integration*. The recovery system has to run within MegaL/Xtext, i.e. the implementations of requirements 1, 2, 3 and 4 must be compatible and integrated with the MegaL/Xtext plug-in-system.

4.2 Recovery System Design

This section summarizes the design of the recovery system developed for this thesis. §4.2.1 will describe the all over process of the recovery system. §4.2.2 will describe the design core API and its components developed for the recovery system. §4.2.3 will describe the design of the actual system for recovering links among O/R/X-Mapping artifacts.

4.2.1 Recovery Process

The recovery process is a straight forward analysis of two artifacts. Figure 4.1 shows a flowchart depicting this. Given two artifacts as input, the recovery pro-

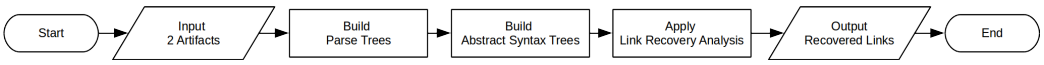


Figure 4.1 The Recovery Process

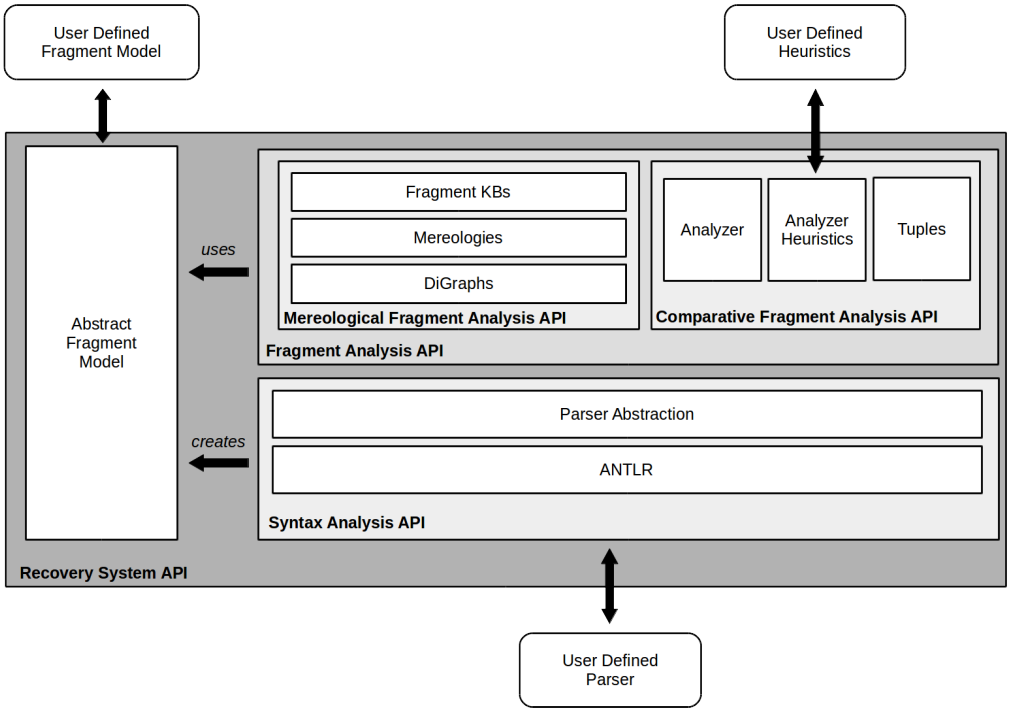
cess works as follows:

1. From each artifact a Concrete Syntax Tree (CST) or Parse Tree is constructed,
2. each Parse Tree is further refined into an AST,
3. both ASTs are compared with each other, i.e. both trees are traversed in a Depth-First Search (DFS) fashion and each pair of nodes is checked whether it can be recovered as link.

Eventually, the set of recovered links serves as output of the process.

4.2.2 Recovery API

The Recovery API is the core of the developed recovery system. It provides generalized data structures and methods for syntactic analysis of artifacts and their



This block diagram depicts the functional outline of the Recovery System API (note, that this does not necessarily correspond to the package outline of its actual implementation).

Figure 4.2 The Recovery API

fragments. Figure 4.2 depicts the API as block diagram. The components of the API are:

Abstract Fragment Model The Abstract Fragment Model serves as base model for all ASTs. It can thought of as Data Transfer Object (DTO) for the analysis components. As user of the API, i have to derive a specific AST from this model. A detailed description follows in §4.2.2.1.

Syntax Analysis API The Syntax Analysis API is the abstraction layer for parsing and AST construction. It is currently backed by ANTLR (Another Tool For Language Recognition), but its internal design is loosely coupled, so other parser libraries can be used. As user of the API, i have to implement a parser constructing an AST deriving the Abstract Fragment Model. However, if one uses ANTLR, only AST construction from a CST is required. A detailed description follows in §4.2.2.2.

Fragment Analysis API The Fragment Analysis API consists of two components:

Mereological Fragment Analysis API The Mereological Fragment Analysis API provides components for deriving parthood links between syntactically well-formed fragments. As user of the API, i only have to apply its components to constructed ASTs. A detailed description follows in §4.2.2.3.

Comparative Fragment Analysis API The Comparative Fragment Analysis API provides components for deriving links between different artifacts and their fragments. As user of the API, i have to implement one or more specialized heuristics for deciding which links can be recovered. A detailed description follows in §4.2.2.4.

4.2.2.1 Abstract Fragment Model

The Abstract Fragment Model describes a tree in which each node represents an syntactically well-formed fragment. Figure 4.3 shows an UML (Unified Modeling Language) class diagram of the model. The resulting tree data structure is doubly-linked, i.e. an `IFragment` node aggregates references to its children an to its parent, given it is not the root node. Each fragment `IFragment` contains the text it represents. In order to distinguish fragments which represent the same text, each node also carries the text's position in the artifact through `IFragmentPosition` instances. Positions inside the text are determined by the start and ending line number as well as the corresponding first and last character inside the line. Most of `IFragment`'s relevant code for trees is pre-implemented in the `BaseFragment` abstract class.

4.2.2.2 Syntax Analysis API

The Syntax Analysis API provides abstraction for AST construction. The API itself is really small, it only consists of the `IParser` and `IParserFactory` interfaces. However, its implementation for ANTLR is designed to reduce ANTLR-specific boilerplate code. Figure 4.4 shows the UML class diagram for the ANTLR backed implementation. One can see, that this implementation makes heavy use of the Abstract Factory Pattern [5]. This allows for a quick and easy definition of new parsers as shown in Figure 4.5.

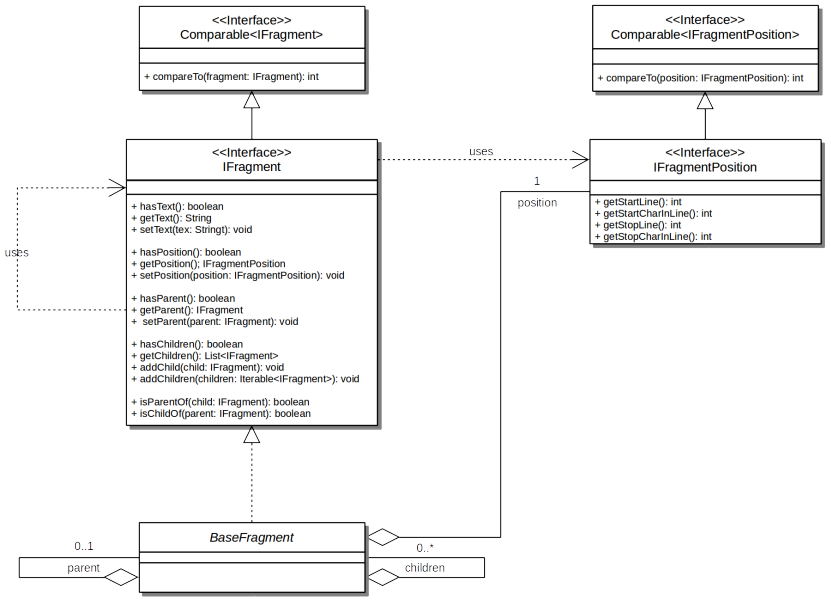


Figure 4.3 The Abstract Fragment Model

The other advantage of the Abstract Factory Pattern is that the instantiation of all relevant classes necessary for the creation of ANTLR Parse Trees takes place in the predefined `AntlrParser` class. This is exemplified by Figure 4.6.

Creation of ASTs or fragment trees is done using the `ParseTreeWalker` and `ParseTreeListener` infrastructure provided by ANTLR [13]. This is an variation of the Observer Pattern [5]. Listeners in conjunction with walkers are an alternative to the Visitor Pattern [5] provided by ANTLR. An instance of `ParseTreeWalker` traverses a Parse Tree using DFS. During traversal, a designated method of `ParseTreeListener` is executed. For AST creation, an API user has to implement the `IFragmentBuildingListener` interface, which is an extension of the `ParseTreeListener` interface. The creation of a fragment tree by `AntlrParser` is exemplified in Figure 4.7.

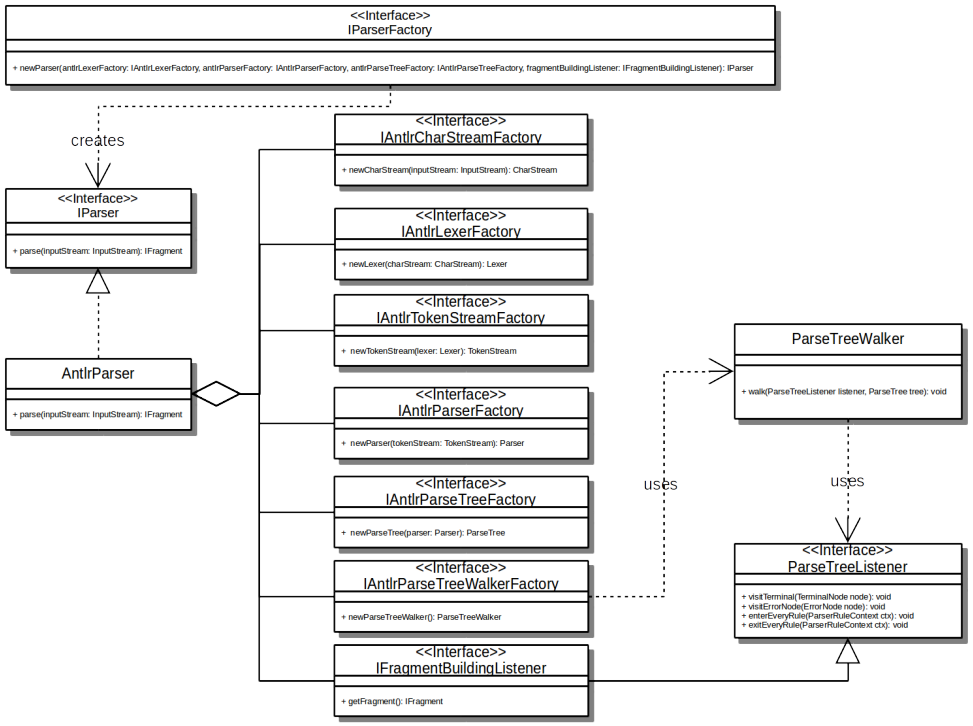


Figure 4.4 The Syntax Analysis API

```
1 ...
2 IParser java8Parser = parserFactory.newParser(
3     Java8Lexer::new,
4     Java8Parser::new,
5     Java8Parser::compilationUnit,
6     Java8FragmentBuildingListener::new);
7 ...
```

This example demonstrates the creation of a new parser using an `IParserFactory` instance with Java8 features. Lexer and Parser classes are generated by ANTLR. A suitable listener has to be implemented.

Figure 4.5 `IParserFactory` Usage Example

```
1 ...
2 CharStream charStream = antlrCharStreamFactory.newCharStream(inputStream);
3 Lexer lexer = antlrLexerFactory.newLexer(charStream);
4 TokenStream tokenStream = antlrTokenStreamFactory.newTokenStream(lexer);
5 Parser parser = antlrParserFactory.newParser(tokenStream);
6 ParseTree parseTree = antlrParseTreeFactory.newParseTree(parser);
7 ...
```

This example demonstrates the creation of an ANTLR `ParseTree` instance as implemented by the `AntlrParser` class.

Figure 4.6 `AntlrParser` Parse Tree Creation

```
1 ...
2 ParseTreeWalker parseTreeWalker = antlrParseTreeWalkerFactory.newParseTreeWalker();
3 parseTreeWalker.walk(fragmentBuildingListener, parseTree);
4 IFragment fragment = fragmentBuildingListener.getFragment();
5 ...
```

This example demonstrates the creation of an `IFragment` AST instance as implemented by the `AntlrParser` class.

Figure 4.7 `AntlrParser` Fragment Creation

4.2.2.3 Mereological Fragment Analysis API

The Mereological Fragment Analysis API provides components for deriving parthood links between syntactically well-formed fragments. Direct parthood links are recovered from parent/child relationships within an `IFragment` AST. However, because parthood is transitive, we also need to recover these links. This is done using a digraph data structure upon which we can compute its reflexive transitive closure.

Figure 4.8 shows an UML class diagram depicting the relevant classes and interfaces of the Mereological Fragment Analysis API. At its heart, the API uti-

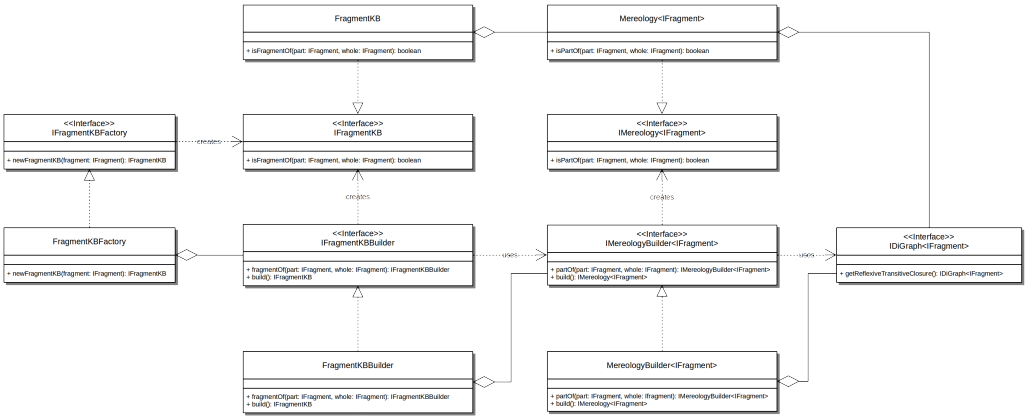


Figure 4.8 Mereological Fragment Analysis API

lizes the generic `IDiGraph` interface whose implementations provide means for interrelating comparable objects, i.e. `IFragment` instances.

This digraph is wrapped by generic `IMereology` implementations, a data structure providing query methods on the topic of mereology, i.e. parthood rela-

tions. Mereologys are constructed using the Builder Pattern [5]. An `IMereology-Builder` instance adds nodes and edges into its digraph with semantically named methods allowing a descriptive programming style.

`IMereology`'s are then further wrapped by `IFragmentKB` (a Knowledge Base over `IFragment`) implementations in order to avoid dealing with generics throughout the system. This is also done utilizing the Builder Pattern.

The recovery of parthood links is encapsulated through `IFragmentKBFactory` using the Abstract Factory Pattern, which computes `IFragmentKB` instances from an `IFragment` AST as input.

4.2.2.4 Comparative Fragment Analysis API

The Comparative Fragment Analysis API provides components for deriving links from two `IFragment` ASTs generated from different artifacts. Figure 4.9 shows

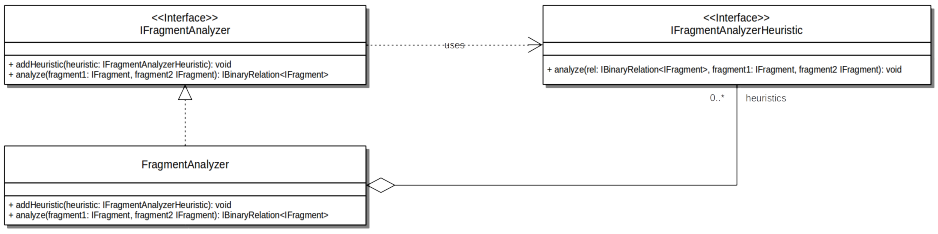


Figure 4.9 Comparative Fragment Analysis API

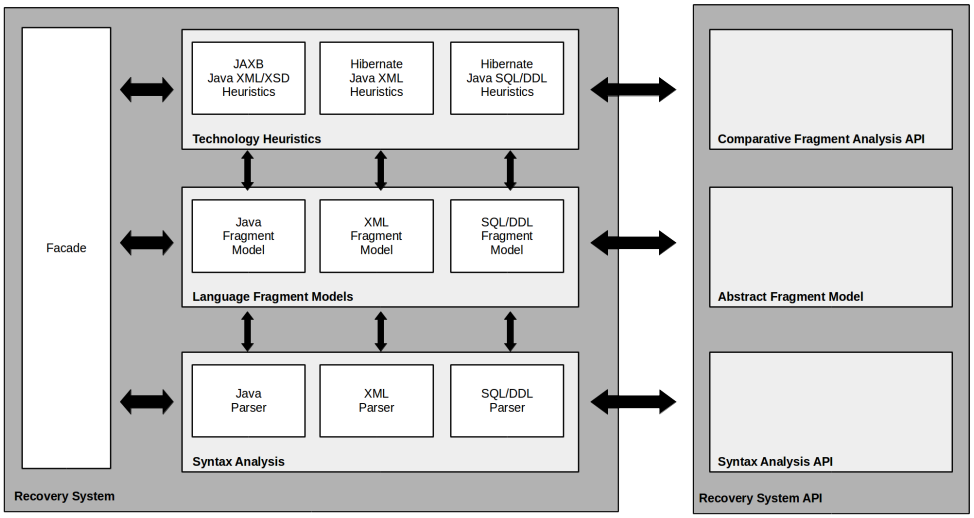
an UML class diagram depicting the relevant interfaces and classes of the API. It utilizes the Strategy Pattern [5] through the `IFragmentAnalyzerHeuristic` interface since concrete behavior of the recovery analysis is for the user to implement. In this context, strategies are called heuristics, because recovery of links is not necessarily based on optimal, i.e. absolutely correct, solutions. Instead strategies may also implement practical solutions with reasonably sufficient results.

`IFragmentAnalyzer` objects allow one to apply multiple heuristics, since there may be more than one practical approach to achieve a goal. Also this allows strategies to keep a relatively small implementation footprint.

Recovered links are captured and stored in an `IBinaryRelation` instance which works like a set of pairs.

4.2.3 Recovery System

The Recovery System is the actual implementation of parthood, correspondence and conformance link recovery for Java based O/R/X-Mapping artifacts. Figure 4.10 shows a block diagram outlining the functional components of the system.



This block diagram depicts the functional outline of the Recovery System and its dependencies to the Recovery System API (note, that this does not necessarily correspond to the package outline of its actual implementation).

Figure 4.10 The Recovery System

The Recovery System utilizes the Syntactic Analysis API described in §4.2.2.2 for Java, XML and SQL/DDL. For this, the Abstract Fragment Model described in §4.2.2.1 is implemented. Note, fragment models do not implement AST suitable for compilation of the targeted language. Syntactic features unnecessary for the intended analysis, like local variable declarations, arithmetic expressions or invocations, are omitted. Fragment models rather focuses on structural features of the languages at hand.

The Comparative Fragment Analysis API described in §4.2.2.4 is implemented with focus on artifacts of technologies, namely JAXB and Hibernate. It implements heuristics for link recovery between:

- Java and XML for JAXB artifacts, i.e. Java models are serialized as XML
- Java and XSD for JAXB artifacts, i.e. Java models are serialized as XSD
- XML and XSD for JAXB artifacts, i.e. the two previous scenarios occurred
- Java and XML for Hibernate mapping artifacts, i.e. Hibernate uses XML meta-data for O/R-Mapping (Object-Relational-Mapping).
- Java and SQL/DDL for Hibernate generated SQL (Structured Query Language) artifacts, i.e. Hibernate uses Java annotations for O/R-Mapping

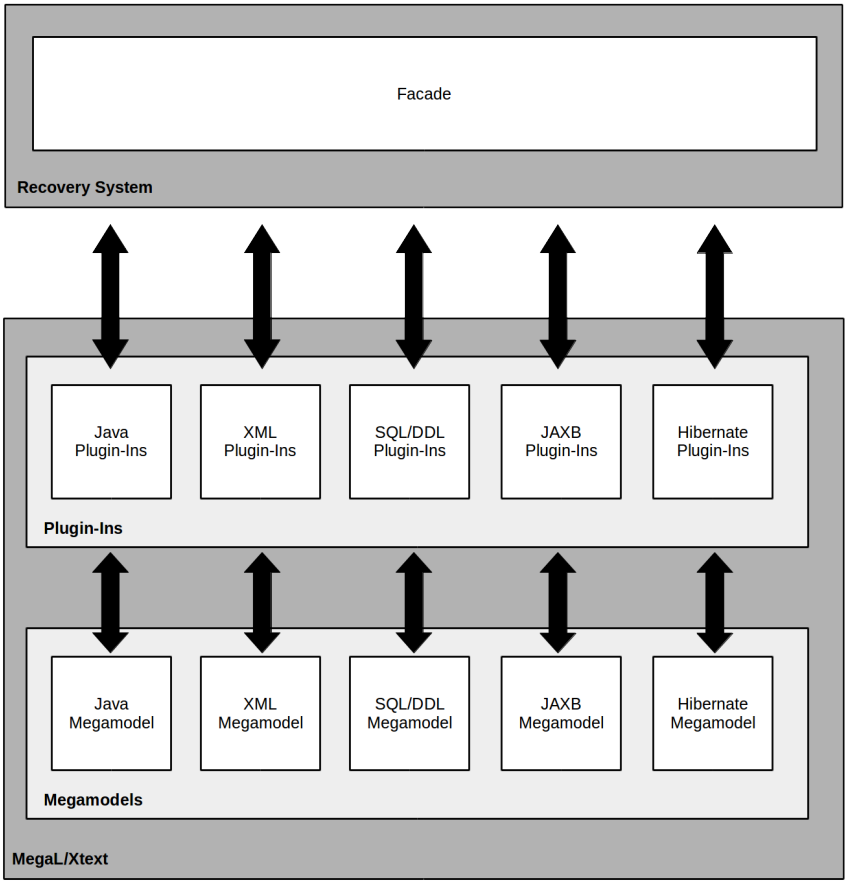
On top of the actual implementation, the Recovery System also utilizes the Facade Pattern [5]. This is to provide a single access point for all implemented analysis features.

4.3 Megal/Xtext Integration Design

This section summarizes the design of the recovery system's integration in the MegaL/Xtext environment. Figure 4.11 shows a block diagram depicting the outline this integration. Note, the Recovery System is implemented as separate project, which is then included as reference (through a `.jar` file) in a MegaL/Xtext project.

MegaL/Xtext provides a plug-in system, which allows to bind special Java classes to `Plugin` entities. Code of such classes is then executed during the evaluation of a megamodel. **ToDo: needs reference** Figure 4.12 exemplifies the declaration of Plug-in withing MegaL.

The Recovery System or rather its facade (see §4.2.3) is used to implement MegaL plug-ins. These Plug-ins are then in turn bound within megamodels, which are divided by language and technology, i.e. there are separate MegaL modules for Java, XML, SQL, Hibernate and JAXB.



This block diagram depicts the functional outline of the Recovery System's integration into the MegaL/Xtext environment (note, that this does not necessarily correspond to the outline of its actual implementation).

Figure 4.11 Integration of the Recovery System int MegaL/Xtext

```
1 ...
2 JavaFragmentRecoveryPlugin : Plugin
3 JavaFragmentRecoveryPlugin realizationOf Java
4 JavaFragmentRecoveryPlugin partOf FileFragmentRecoveryReasonerPlugin
5 JavaFragmentRecoveryPlugin = 'classpath:org.softlang.megal.plugins.impl.java.JavaFragmentRecoveryPlugin'
6 ...
```

This snippet demonstrates the instantiation of `JavaFragmentRecoveryPlugin` as part of `FileFragmentRecoveryReasonerPlugin` in MegaL.

Figure 4.12 MegaL Plug-In Instantiation

Chapter 5

Implementation

This chapter summarizes the implementation of crucial parts of recovery system implemented for this thesis. §5.1 covers the implementation of fragment recovery. §5.2 covers the implementation of parthood link recovery. §5.3 and §5.4 covers the implementation of correspondence and conformance link recovery

5.1 Recovering Fragments

This section summarizes the implementation of fragment recovery. Fragments are syntactically well-formed pieces of code. For instance, consider the following Java class:

```
public class Company {
   ...
   private String name;
   ...
   public String getName() {
      return name;
   }
   ...
}
```

It consists, among others, of the following fragments:

- a field declaration fragment for `name`:

```
private String name;
```

- a method declaration fragment for an accessor-method of `name`:

```
public String getName() {
   return name;
}
```

- the return statement of the accessor-method:

```
return name;
```

- the class declaration itself can also be considered a fragment.

The task of recovering fragments is to add entities for each of such code pieces into a megamodel. Figure 5.1 exemplifies the recovery of Java[1] fragments in an idealized form.

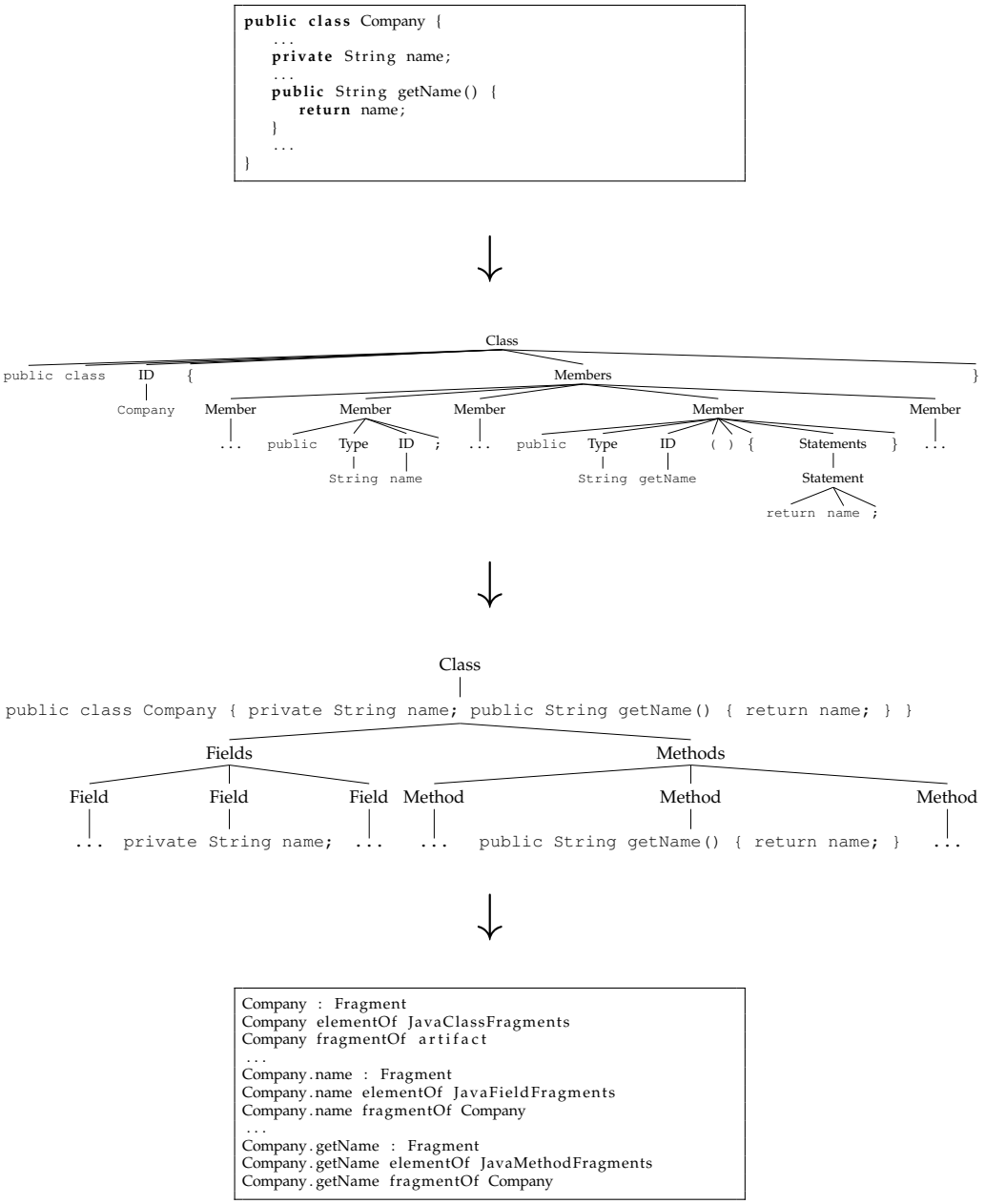
1. a Parse Tree is generated from an artifact; this is done via ANTLR as described in §4.2.2.2 and does not require further explanation, since it is only an application of the most common ANTLR use case
2. then the Parse Tree is transformed into an concrete fragment AST model; this is also done relying on ANTLR tree traversal utilities and described in detail in §5.1.1
3. eventually, nodes of the generated fragment AST are added to a megamodel as entities; the details are described in §5.1.2.

5.1.1 Concrete Fragment Models

As mentioned in §4.2.2.1, Concrete Fragment Models are derivations of the Abstract Fragment Model of the Recovery System API. Figure 5.2 shows an UML class diagram of the implemented Fragment AST for Java. All fragment classes derive from `IFragment` through the base class `JavaFragment`. From here, several specializations are introduced:

- `IdentfiedJavaFragment` for constructs with identifiers
- `ModifiedJavaFragment` for constructs with modifiers like `private`, `public`, `final`, `abstract`, `static`, etc. or annotation meta-data
- `TypedJavaFragment` for constructs with distinct (return-) type like fields, methods or variables

[1] Recovery for XML and SQL/DDL is implemented in a similar fashion. If there is a noteworthy difference for other languages it will be explored, otherwise we keep using Java as example domain for the remaining sections of §5.1.



This picture shows an idealized recovery of Java fragments:

code artifact → Parse Tree → fragment AST → megamodel

Both Parse Tree and AST are depicted in a simplified, schematic form.

Figure 5.1 Idealized Recovery of Java Fragments

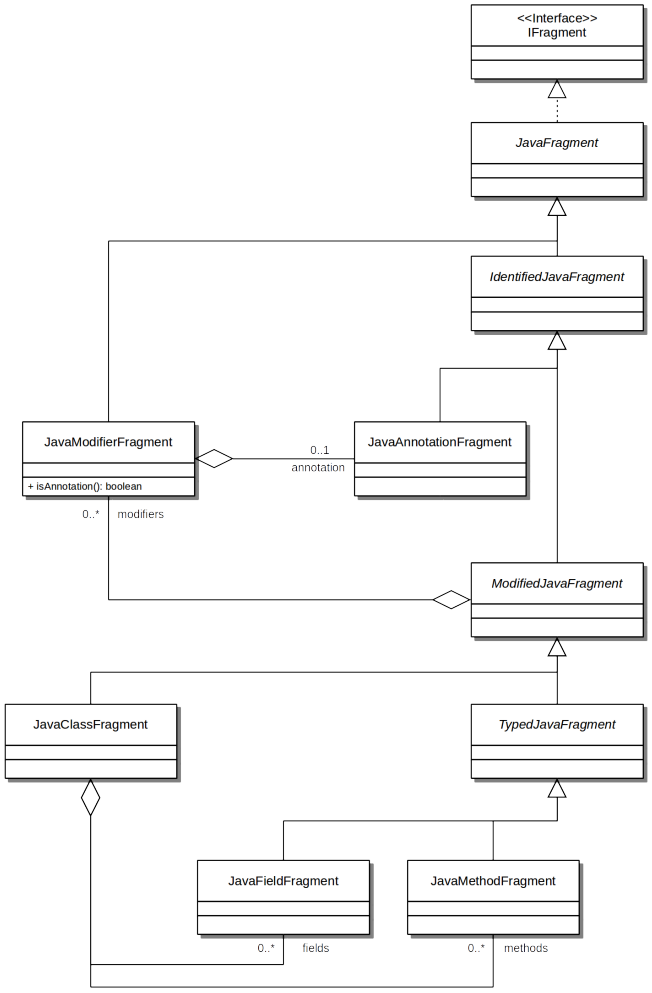


Figure 5.2 Java Fragment AST Model

The fragment model for Java only focuses on structural syntactical features of the language. Everything below method signature level is omitted, because fragments of this level are not important for the scope of this thesis. On the other hand, annotations are captured, since they provide significant information for further recovery analysis. Hibernate and JAXB rely on annotations for O/R/X-Mapping.

The AST is constructed using ANTLR `ParseTreeListener` approach as described in §4.2.2.2. Figure 5.3 shows an excerpt from the listener implementation used to construct the the AST for Java fragments. Here, fragment instances are

```
1  ...
2  @Override
3  public void exitMethodDeclaration(Java8Parser.MethodDeclarationContext ctx) {
4      javaMethodFragments.push(java8FragmentFactory.newJavaMethodFragment(ctx, javaMethodModifierFragments));
5  }
6
7  @Override
8  public void exitNormalClassDeclaration(Java8Parser.NormalClassDeclarationContext ctx) {
9      javaClassFragments.push(java8FragmentFactory.newJavaClassFragment(ctx, javaClassModifierFragments,
           javaFieldFragments, javaMethodFragments, declaredPackage));
10 }
11 ...
```

This snippet shows an excerpt of the listener used to construct Java fragment ASTs.

Figure 5.3 Construction of Java Fragment ASTs

created when the listener leaves a certain node of the Parse Tree. Creation is delegated to a factory. The listener pushes fragments onto stacks, making the overall implementation work like a pushdown automaton or stack machine.

5.1.2 Megamodeling Fragments

In order to add fragments into a megamodel a suitable linguistic model has to be declared first. Consider the following instance level megamodel of recovered fragments:

```
File < Artifact
Fragment < Artifact
...
elementOf < Set * Set
fragmentOf < Fragment * Artifact
...
aJavaFile : File
aJavaFile = 'Company.java'
...
aJavaFile.Company : Fragment
aJavaFile.Company elementOf JavaClassFragments
aJavaFile.Company fragmentOf aJavaFile
...
aJavaFile.Company.name : Fragment
aJavaFile.Company.name elementOf JavaFieldFragments
aJavaFile.Company.name fragmentOf aJavaFile
aJavaFile.Company.name fragmentOf Company
...
aJavaFile.Company.getName : Fragment
aJavaFile.Company.getName elementOf JavaMethodFragments
aJavaFile.Company.getName fragmentOf aJavaFile
aJavaFile.Company.getName fragmentOf Company
```

This megamodel shows fragments of a Java file `Company.java`. It contains entities denoting a class declaration, a field declaration and a method declaration

fragment. All are declared instances of the entity type `Fragment`. Entity names use dot-notation for implying parthood, i.e.:

$$a.b \Rightarrow b\text{partOf}a$$

Further details of naming-scheme employed for fragment entities are explored in §5.1.2.2.

We use `elementOf` for further specialization and/or generalization of the entitie's kind[2]. However, the right-hand side of these relationships used to specialize fragments are not simply sets, they are actually modeled as languages as we will see in §5.1.2.1.

5.1.2.1 Fragment Languages

As mentioned in §**??**, a fragment alone cannot be element of the language the artifact it originated from belongs to, e.g. a Java method alone cannot be accepted by the Java grammar, nor can XML attribute alone be accepted by the XML grammar. Therefore, in order to cleanly add fragments into a megamodel, we need to model fragment languages. Figure 5.4 shows the megamodel for Java fragments. It introduces the special relationship type `fragmentLanguageOf`, denoting the

```
1  ...
2  Language < Set
3  ...
4  fragmentLanguageOf < Language * Language
5  ...
6  Java : Language
7
8  JavaFragments : Language
9  JavaFragments fragmentLanguageOf Java
10
11 JavaClassFragments : Language
12 JavaClassFragments subsetOf JavaFragments
13
14 JavaFieldFragments : Language
15 JavaFieldFragments subsetOf JavaFragments
16
17 JavaMethodFragments : Language
18 JavaMethodFragments subsetOf JavaFragments
19 ...
```

Figure 5.4 A Megamodel for Java Fragments

left-hand side is the language containing all possible fragments the right-hand

[2]We use the term *kind* to avoid confusion with entity types, although we use it to refer to fragment types.

side's language elements can be deconstructed in. This also implies a super-set relation between the two languages as fragmentOfis reflexive like partOf, so:

$$A \texttt{ fragmentLanguageOf } B \Rightarrow B \subseteq A$$

5.1.2.2 Fragment Entity Identifiers

5.2 Recovering Parthood Links

This section summarizes the implementation of parthood link recovery.

5.3 Recovering Correspondence Links

This section the implementation of correspondence and conformance link recovery

5.4 Recovering Conformance Links

This section the implementation of correspondence and conformance link recovery

Chapter 6

Mini Case-Study

This chapter summarizes the methodology used to develop the recovery system for this thesis. In short, the recovery system is developed example-driven using the model of a fictional Human Resource Management System (HRMS) used by the 101wiki[1] for its contributions.

6.1 Example Corpus

The example corpus used to develop the recovery system for this thesis consists of artifacts implementing a fictional HRMS within an O/R/X-Mapping scenario using Java technologies. The model is implemented using plain Java. It is then mapped to plain XML/XSD with JAXB and to SQL/DDL statements using Hibernate mapping files and/or annotations.

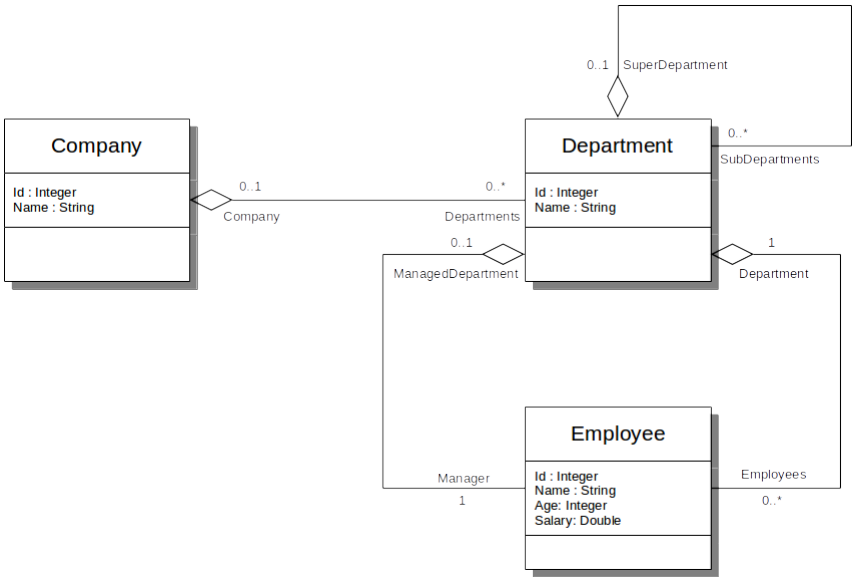
6.1.1 The 101HRMS Model

101wiki Human Resource Management System (101HRMS)[2] provides a simple model of a company with many departments and employees. Figure 6.1 shows an UML class diagram of a variant of this model.

The 101HRMS model consists of companies attributed with a name. Each company accumulates departments. Each department is also attributed with a

[1] https://101wiki.softlang.org/ (retrieved 12th November, 2017)

[2] https://101wiki.softlang.org/101:@system (retrieved 12th November, 2017)



This UML class diagram depicts the model of the 101HRMS. It consists of simple companies with nested departments and employees mapped to the latter.

Figure 6.1 The 101 Human Resource Management System Model

name, aggregates employees and has one employee acting as manager. Departments can further be refined into sub-departments. Each employee is attributed with a name, an age and a salary. Each entity is also attributed with an ID.

6.1.2 Linguistic Domains of the Example Corpus

The example corpus used to develop the recovery system contains artifacts implementing the 101HRMS model generated or used by Java technologies for O/R/X-Mapping, i.e. a Java model is mapped to plain XML/XSD with JAXB, to a Hibernate mapping file and to SQL/Data Definition Language (DDL) statements. Figure 6.2 shows a schematic illustration of the linguistic domains involved:

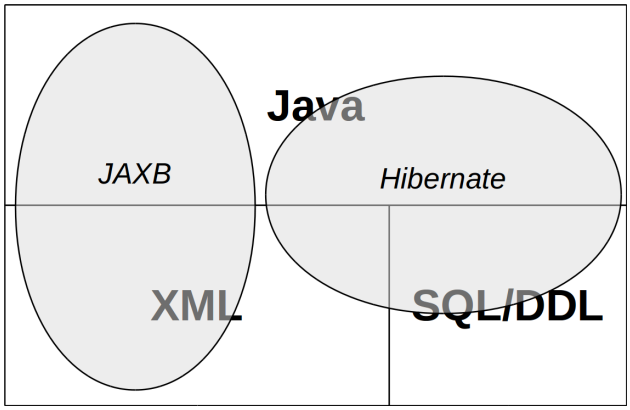
Java The language and technology used to implement the 101HRMS model.

XML The language used to serialize the 101HRMS model.

SQL/DDL The language used to persist the 101HRMS model.

JAXB The technology used to implement O/X-Mapping (Object-XML-Mapping) of the 101HRMS model.

Hibernate The technology used to implement O/R-Mapping of the 101HRMS model.



This schematic illustration depicts the interrelation among linguistic domains of example corpus used. It depicts languages and technologies for O/R/X-Mapping with Java.

Figure 6.2 Example Corpus Domains: Java O/R/X

The languages (Java, XML & SQL) in Figure 6.2 are displayed as disjoint square sets. Technologies (JAXB & Hibernate) are displayed as oval sets intersecting languages. This is due to their linguistic nature, e.g. JAXB produces specific Java- and XML-Code which does not necessarily intersect with code produced by other technologies. Hibernate intersects all three languages. It uses XML files or Java-Annotations for describing O/R-Mapping of a data-model and generates SQL artifacts according to that mapping. In this sense, technologies create technology-specific subsets of a languages.

Chapter 7

Conclusion

7.1 Future Work

Bibliography

[1] Anya Helene Bagge and Vadim Zaytsev. "Languages, Models and Megamodels". In: *Post-proceedings of the Seventh Seminar on Advanced Techniques and Tools for Software Evolution, SATToSE 2014, L'Aquila, Italy, 9-11 July 2014*. 2014, pp. 132–143. URL: http://ceur-ws.org/Vol-1354/paper-12.pdf.

[2] Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. "Modeling the Linguistic Architecture of Software Products". In: *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*. 2012, pp. 151–167. DOI: 10.1007/978-3-642-33666-9_11. URL: http://dx.doi.org/10.1007/978-3-642-33666-9_11.

[3] Jean-Marie Favre and Tam Nguyen. "Towards a Megamodel to Model Software Evolution Through Transformations". In: *Electr. Notes Theor. Comput. Sci.* 127.3 (2005), pp. 59–74. DOI: 10.1016/j.entcs.2004.08.034. URL: http://dx.doi.org/10.1016/j.entcs.2004.08.034.

[4] Richard Freeman Paige et al. "Building Model-Driven Engineering Traceability Classifications". In: (Jan. 2010).

[5] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[6] O. C. Z. Gotel and C. W. Finkelstein. *An analysis of the requirements traceability problem*. Apr. 1994. DOI: 10.1109/ICRE.1994.292398.

[7] Orlena Gotel et al. “Appendix A: Glossary of Traceability Terms (v1.0)”. In: *Software and Systems Traceability*. 2012, pp. 413–424.

[8] Orlena Gotel et al. “Traceability Fundamentals”. In: *Software and Systems Traceability*. 2012, pp. 3–22. DOI: 10.1007/978-1-4471-2239-5_1. URL: https://doi.org/10.1007/978-1-4471-2239-5_1.

[9] Lukas Härtel. “Linguistic architecture on the workbench”. Bachelor Thesis. University of Koblenz-Landau, Oct. 2015.

[10] Marcel Heinz, Ralf Lämmel, and Andrei Varanovich. *Axioms of linguistic architecture*. 9 pages. 2017.

[11] Ralf Lämmel. “Coupled software transformations revisited”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. 2016, pp. 239–252. URL: http://dl.acm.org/citation.cfm?id=2997366.

[12] Ralf Lämmel and Andrei Varanovich. “Interpretation of Linguistic Architecture”. In: *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*. 2014, pp. 67–82. DOI: 10.1007/978-3-319-09195-2_5. URL: http://dx.doi.org/10.1007/978-3-319-09195-2_5.

[13] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999, 9781934356999.

[14] Achille C. Varzi. “Mereology”. In: *The Stanford encyclopedia of philosophy*. Ed. by Edward N. Zalta. Winter 2016 ed. retrieved 27. July 2017. 2016. URL: https://plato.stanford.edu/archives/win2016/entries/mereology/.

[15] Achille C. Varzi. “Parts, Wholes, and Part-Whole Relations: The Prospects of Mereotopology”. In: *Data Knowl. Eng.* 20.3 (1996), pp. 259–286. DOI: 10.1016/S0169-023X(96)00017-1. URL: http://dx.doi.org/10.1016/S0169-023X(96)00017-1.

[16] Stefan Winkler and Jens Pilgrim. "A Survey of Traceability in Requirements Engineering and Model-driven Development". In: *Softw. Syst. Model.* 9.4 (Sept. 2010), pp. 529–565. ISSN: 1619-1366. DOI: 10.1007/s10270-009-0145-0. URL: http://dx.doi.org/10.1007/s10270-009-0145-0.