UNIVERSITÄT KOBLENZ · LANDAU

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

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- to instance-level syntaxes of different manifestations

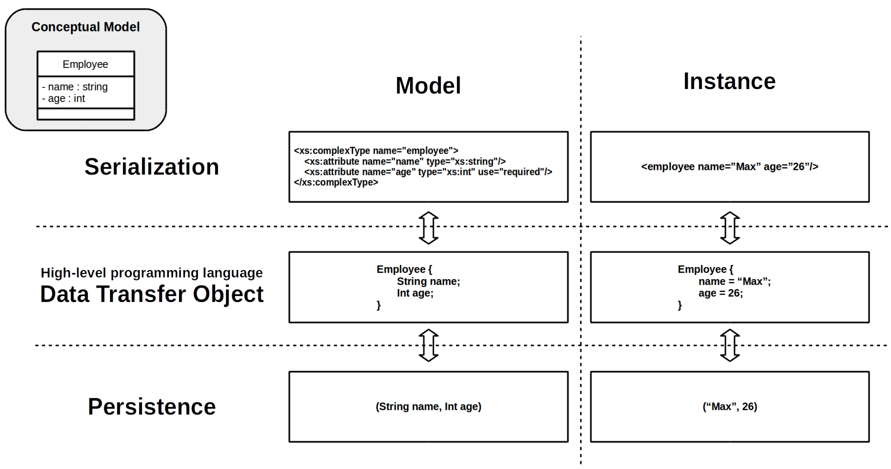


Figure 1.1 O/R/X-Mapping Manifestations

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. Figure 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:

```java
@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;

}
```

```xml
<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>
```

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

Figure 1.2 JAXB XML/XSD Mapping

- `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

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] [9] [7]. 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* [5]. In context of traceability, semantic links may be called *trace links*, however, both establish a relation between two entities denoting a certain meaning.

1.1 Approach

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

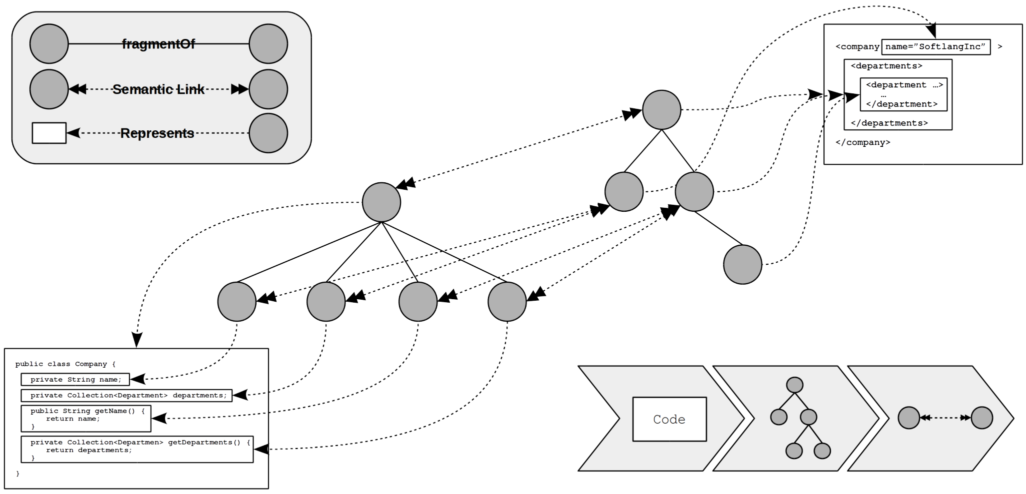


Figure 1.3 Recovery Approach

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

1.2 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 §1.1 (see §5).

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 §6 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 §7).

Non-Contribution 1 We do not contribute to the axiomatization of linguistic architectures as described in [9], [3], [8] and [7].

Chapter 2

Background

This chapter summarizes 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 [8] and refined in [7]. §2.1.1

Axioms are presented as First Order Logic and formalization is taken from [7]. 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 an essential relationship when reasoning about correspondence and conformance among artifacts within linguistic architectures [8] [7]. It describes the relation between entities and their constituent parts. The study of parthood and its derivatives is mereology [12] [11]. 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)

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

Parthood is usually considered to be reflexive, antisymmetric and transitive [12] [11], 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 normal parthood. Figure 2.1 shows a schematic illustration of proper parts. This Venn-style diagram depicts the sce-

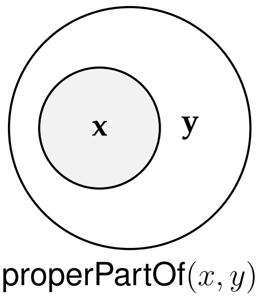


Figure 2.1 A schematic depiction of Proper Parthood

nario where 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 [12] [11], i.e. atomic parts which cannot be decomposed in further parts:

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

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 [7]. 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)

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

We emphasize the proper parthood here, since not all authors necessarily consider partOf to be reflexive [8] [7]. 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:

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

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:

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

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

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

The fragment nature propagates top-down alongside the order of parthood, i.e. if an entity is a fragment all parts are also fragments [7].

2.1.3 Correspondence

Correspondence

$$\begin{aligned}
&\textsf{represents}(r,e) \Rightarrow \textsf{Artifact}(r) \wedge \textsf{Entity}(e).\\
&\textsf{represents}(r,e) \Leftarrow r \text{ is a representation of } e.
\end{aligned}$$

$$\begin{aligned}
&\textsf{sameAs}(x,y) \Rightarrow \textsf{Artifact}(x) \wedge \textsf{Artifact}(y).\\
&\textsf{sameAs}(x,y) \Leftarrow \exists e.\textsf{Entity}(e) \wedge \textsf{represents}(x,e) \wedge \textsf{represents}(y,e).
\end{aligned}$$

Axiom 3 (correspondsTo)

$$\begin{aligned}
\textsf{\textit{correspondsTo}}(x,y) &\Rightarrow \textsf{\textit{Artifact}}(x) \wedge \textsf{\textit{Artifact}}(y).\\
\textsf{\textit{correspondsTo}}(x,y) &\Leftarrow (\forall px.\textsf{\textit{partOf}}(px,x) \Rightarrow \exists py.\textsf{\textit{partOf}}(py,y) \wedge \textsf{\textit{correspondsTo}}(px,py))\\
&\wedge (\forall py.\textsf{\textit{partOf}}(py,y) \Rightarrow \exists px.\textsf{\textit{partOf}}(px,x) \wedge \textsf{\textit{correspondsTo}}(py,px))\\
&\vee (\nexists p.\textsf{\textit{partOf}}(p,x) \vee \textsf{\textit{partof}}(p,y)) \wedge \textsf{\textit{sameAs}}(x,y).
\end{aligned}$$

2.1.4 Conformance

Axiom 4 (conformsTo)

$$\begin{aligned}
\textsf{\textit{conformsTo}}(a,d) &\Rightarrow \textsf{\textit{Artifact}}(a) \wedge \textsf{\textit{Artifact}}(d).\\
\textsf{\textit{conformsTo}}(a,a') &\Leftarrow (\forall p.\textsf{\textit{partOf}}(p,a) \wedge \exists p'.\textsf{\textit{partOf}}(p',a') \wedge \textsf{\textit{conformsTo}}(p,p'))\\
&\vee \exists t.\textsf{\textit{defines}}(a',t) \wedge \textsf{\textit{elementOf}}(a,t).
\end{aligned}$$

2.2 Traceability

[5]

Definition 1 (Trace) *content...*

Definition 2 (Trace Link) *content...*

Definition 3 (Traceability Recovery) *content...*

2.3 Megamodeling

[1] [3]

2.3.1 MegaL

[9] [2] [8]

2.3.2 MegaL/Xtext

[6]

2.4 Technical Background

2.4.1 Java Architecture for XML Binding (JAXB)

2.4.2 Hibernate

2.4.3 Another Tool For Language Recognition (ANTLR)

[10]

Chapter 3

Related Work

TBD.

Chapter 4

Methodology

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.

4.1 Example Driven Development Process

The Example Driven Development Process used to develop the recovery system for this thesis is shown in Figure 4.1. Given we acquired a suitable example corpus, which is outlined in detail in §4.2, we have to define conditions for acceptable results. These conditions can be seen as functional requirements, for instance, we want a Java class artifact to be linked via correspondence to its XML-serialized form:

```
public class Company {...} correspondsTo <company>...</company>
```

Then we implement a feature which produces the specified results and run it against the example corpus. If the result meets all conditions we are done. If not, we inspect the results to check whether our predefined conditions are to vague or wrong. If so, we refine our conditions, if not, we refine our feature implementation Either way the process starts anew.

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

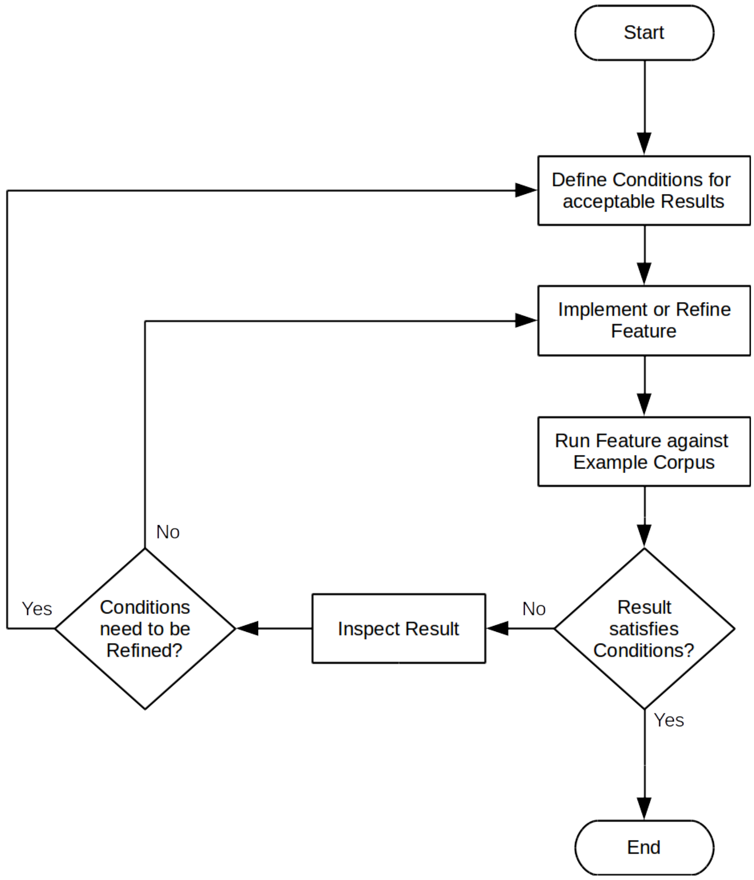


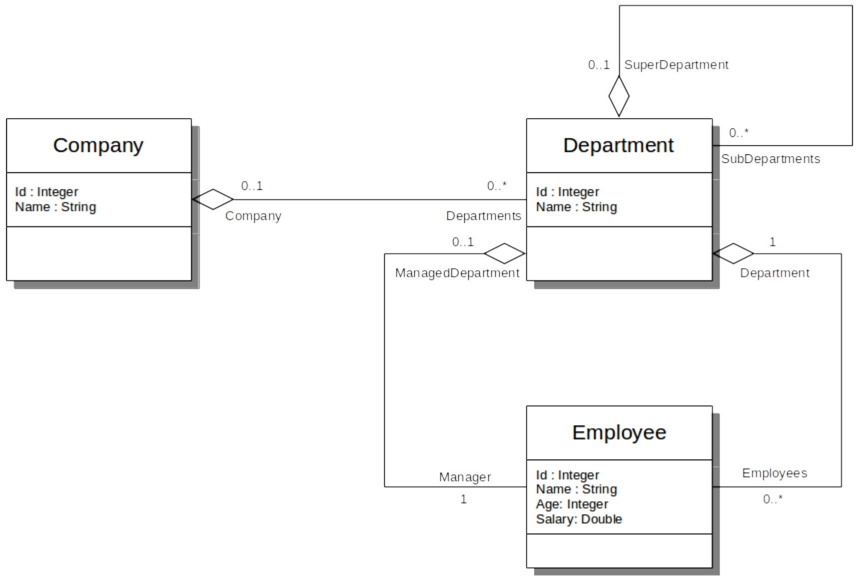
Figure 4.1 The Example Driven Development Process

4.2 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.

4.2.1 The 101HRMS Model

101wiki Human Resource Management System (101HRMS)[2] provides a simple model of a company with many departments and employees. Figure 4.2 shows an UML (Unified Modeling Language) class diagram of a variant of this model.



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 4.2 The 101 Human Resource Management System Model

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

4.2.2 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 Hi-

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

bernate mapping file and to SQL (Structured Query Language)/Data Definition Language (DDL) statements. Figure 4.3 shows a schematic illustration of the linguistic domains involved:

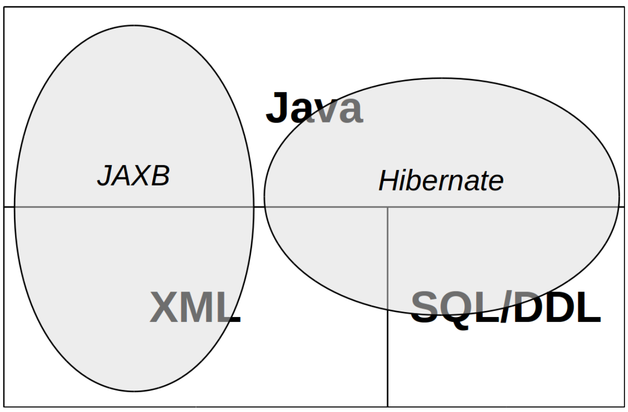
Java The language and technology used to implement the 101HRMS model.

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 (Object-Relational-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 4.3 Example Corpus Domains: Java O/R/X

The languages (Java, XML & SQL) in Figure 4.3 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 5

Design

This chapter summarizes the design of the recovery system developed for this thesis. §5.1 summarizes functional requirements of the developed system. §5.2 recapitulates design of the recovery system with respect to the specified requirements. §5.3 summarizes integration in the recovery system with the MegaL/Xtext environment .

5.1 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.

5.2 Recovery System Design

This section summarizes the design of the recovery system developed for this thesis. §5.2.1 will describe the all over process of the recovery system. §5.2.2 will describe the design core API and its components developed for the recovery system. §5.2.3 will describe the design of the actual system for recovering links among O/R/X-Mapping artifacts.

5.2.1 Recovery Process

The recovery process is a straight forward analysis of two artifacts. Figure 5.1 shows a flowchart depicting this. Given two artifacts as input, the recovery pro-

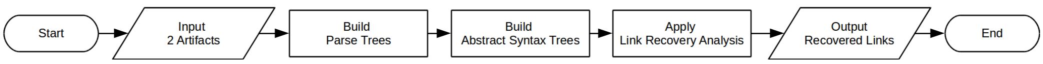


Figure 5.1 The Recovery Process

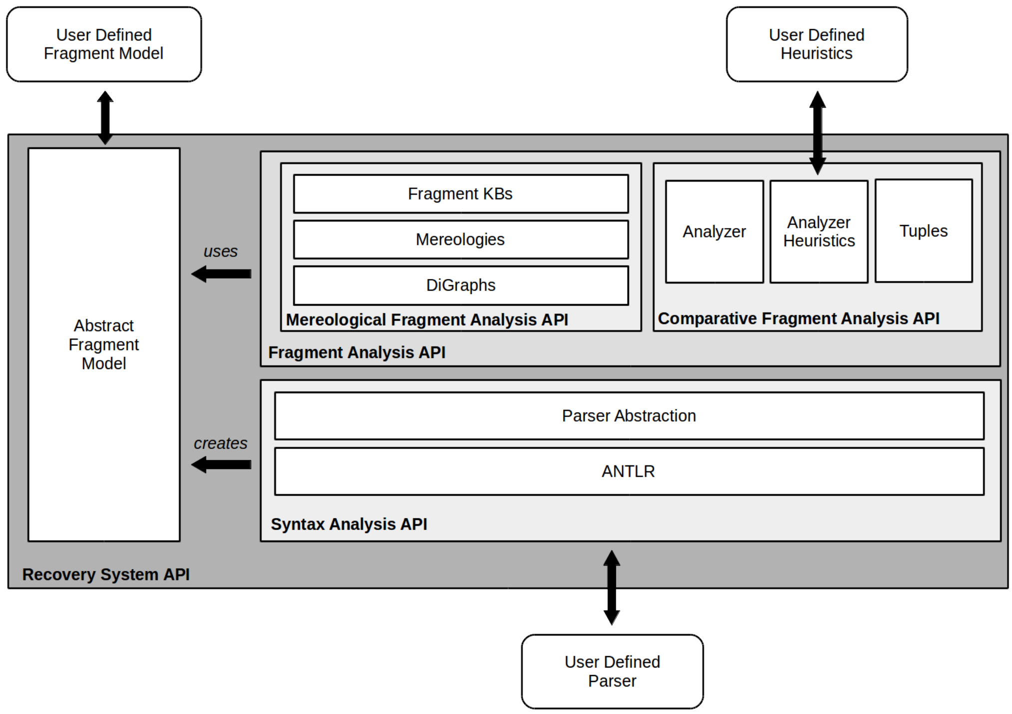
cess works as follows:

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.

5.2.2 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 5.2 The Recovery API

fragments. Figure 5.2 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 §5.2.2.1.

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 §5.2.2.2.

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 §5.2.2.3.

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 §5.2.2.4.

5.2.2.1 Abstract Fragment Model

The Abstract Fragment Model describes a tree in which each node represents an syntactically well-formed fragment. Figure 5.3 shows an UML 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.

5.2.2.2 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 5.4 shows the UML class diagram for the ANTLR backed implementation. One can see, that this implementation makes heavy use of the Abstract Factory Pattern [4]. This allows for a quick and easy definition of new parsers as shown in Figure 5.5.

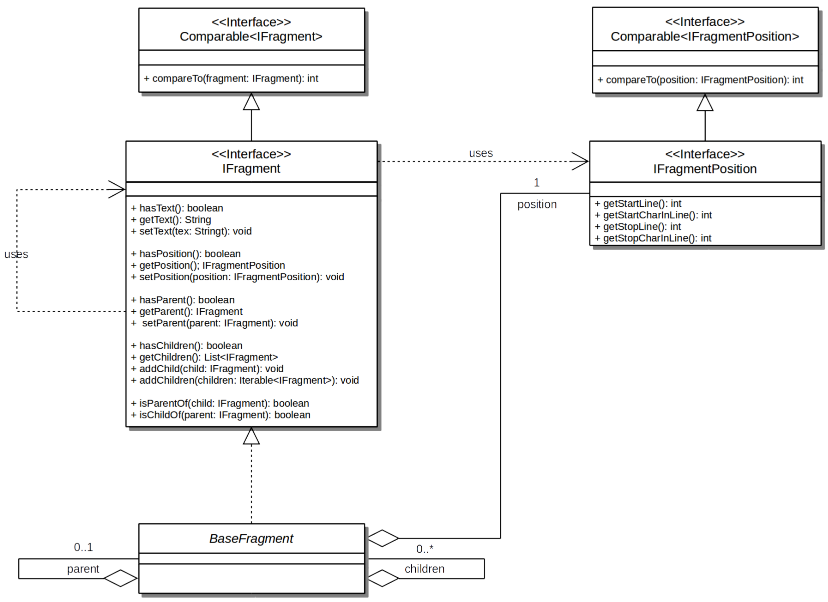


Figure 5.3 The Abstract Fragment Model

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 5.6.

Creation of ASTs or fragment trees is done using the `ParseTreeWalker` and `ParseTreeListener` infrastructure provided by ANTLR [10]. This is an variation of the Observer Pattern [4]. Listeners in conjunction with walkers are an alternative to the Visitor Pattern [4] 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 5.7.

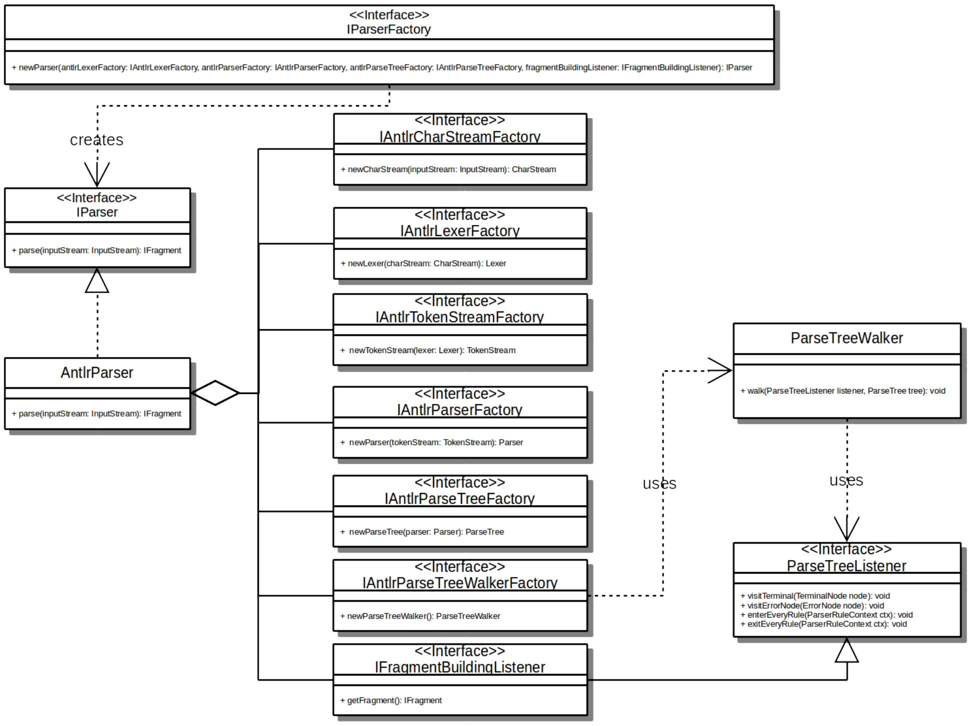


Figure 5.4 The Syntax Analysis API

```
...
IParser java8Parser = parserFactory.newParser(
    Java8Lexer::new,
    Java8Parser::new,
    Java8Parser::compilationUnit,
    Java8FragmentBuildingListener::new);
...
```

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 5.5 `IParserFactory` Usage Example

```
...
CharStream charStream = antlrCharStreamFactory.newCharStream(inputStream);
Lexer lexer = antlrLexerFactory.newLexer(charStream);
TokenStream tokenStream = antlrTokenStreamFactory.newTokenStream(lexer);
Parser parser = antlrParserFactory.newParser(tokenStream);
ParseTree parseTree = antlrParseTreeFactory.newParseTree(parser);
...
```

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

Figure 5.6 `AntlrParser` Parse Tree Creation

```
...
ParseTreeWalker parseTreeWalker = antlrParseTreeWalkerFactory.newParseTreeWalker();
parseTreeWalker.walk(fragmentBuildingListener, parseTree);
IFragment fragment = fragmentBuildingListener.getFragment();
...
```

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

Figure 5.7 `AntlrParser` Fragment Creation

5.2.2.3 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 5.8 shows an UML class diagram depicting the relevant classes and interfaces of the Mereological Fragment Analysis API. At its heart, the API uti-

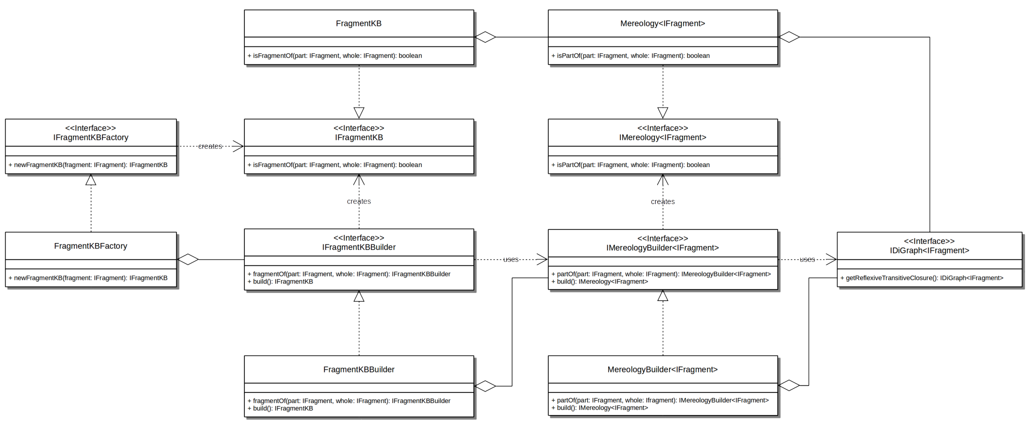


Figure 5.8 Mereological Fragment Analysis API

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 relations. Mereologys are constructed using the Builder Pattern [4]. 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.

5.2.2.4 Comparative Fragment Analysis API

The Comparative Fragment Analysis API provides components for deriving links from two `IFragment` ASTs generated from different artifacts. Figure 5.9 shows

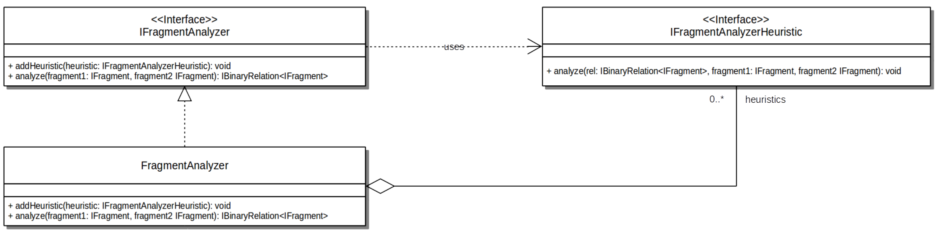


Figure 5.9 Comparative Fragment Analysis API

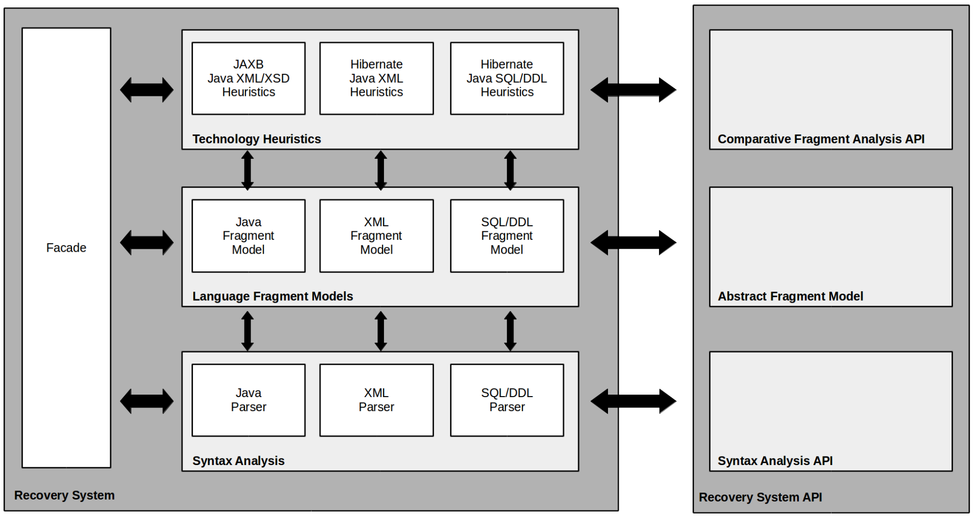
an UML class diagram depicting the relevant interfaces and classes of the API. It utilizes the Strategy Pattern [4] through the `IFragmentAnalyzerHeuristic` interface since concrete behavior of the recovery analysis is for the user to implement. In this context, strategies are called heuristics, because 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.

5.2.3 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 5.10 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 5.10 The Recovery System

The Recovery System utilizes the Syntactic Analysis API described in §5.2.2.2 for Java, XML and SQL/DDL. For this, the Abstract Fragment Model described in §5.2.2.1 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 §5.2.2.4 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.
- Java and SQL/DDL for Hibernate generated SQL artifacts, i.e. Hibernate uses Java annotations for O/R-Mapping

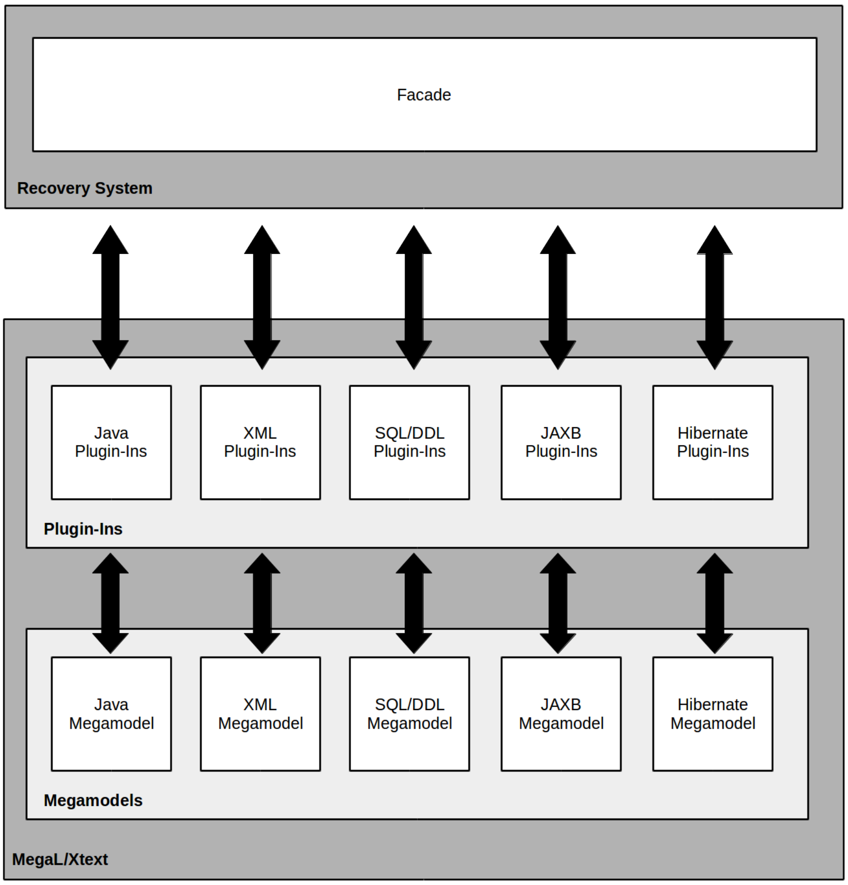
On top of the actual implementation, the Recovery System also utilizes the Facade Pattern [4]. This is to provide a single access point for all implemented analysis features.

5.3 Megal/Xtext Integration Design

This section summarizes the design of the recovery system's integration in the MegaL/Xtext environment. Figure 5.11 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 5.12 exemplifies the declaration of Plug-in withing MegaL.

The Recovery System or rather its facade (see §5.2.3) 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 5.11 Integration of the Recovery System int MegaL/Xtext

```
...
JavaFragmentRecoveryPlugin : Plugin
JavaFragmentRecoveryPlugin realizationOf Java
JavaFragmentRecoveryPlugin partOf FileFragmentRecoveryReasonerPlugin
JavaFragmentRecoveryPlugin = 'classpath:org.softlang.megal.plugins.impl.java.JavaFragmentRecoveryPlugin'
...
```

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

Figure 5.12 MegaL Plug-In Instantiation

Chapter 6

Implementation

This chapter summarizes the implementation of crucial parts of recovery system implemented for this thesis. §6.1 covers the implementation of fragment recovery. §6.2 covers the implementation of parthood link recovery. §6.3 covers the implementation of correspondence and conformance link recovery

6.1 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 6.1 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 §5.2.2.2 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 §6.1.1
3. eventually, nodes of the generated fragment AST are added to a megamodel as entities; the details are described in §6.1.2.

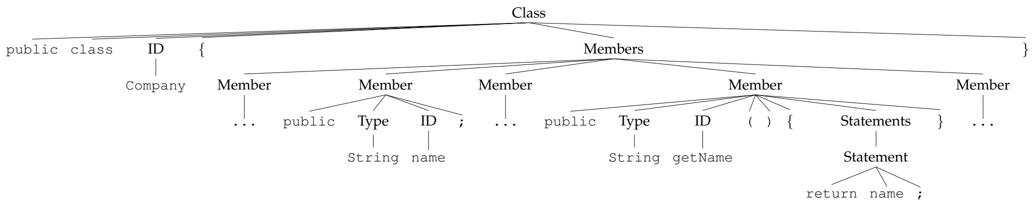
6.1.1 Concrete Fragment Models

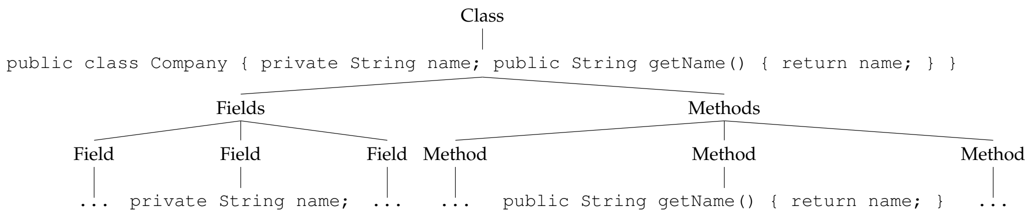
As mentioned in §5.2.2.1, Concrete Fragment Models are derivations of the Abstract Fragment Model of the Recovery System API. Figure 6.2 shows an UML class diagram of the implemented Fragment AST for Java. All fragment classes derive from `IFragment` through the base class `JavaFragment`. From here, 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 §6.1.

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






```
Company : Fragment
Company elementOf JavaClassFragments
Company fragmentOf artifact
...
Company.name : Fragment
Company.name elementOf JavaFieldFragments
Company.name fragmentOf Company
...
Company.getName : Fragment
Company.getName elementOf JavaMethodFragments
Company.getName fragmentOf Company
```

This picture shows an idealized recovery of Java fragments:

$$\text{code artifact} \rightarrow \text{Parse Tree} \rightarrow \text{fragment AST} \rightarrow \text{megamodel}$$

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

Figure 6.1 Idealized Recovery of Java Fragments

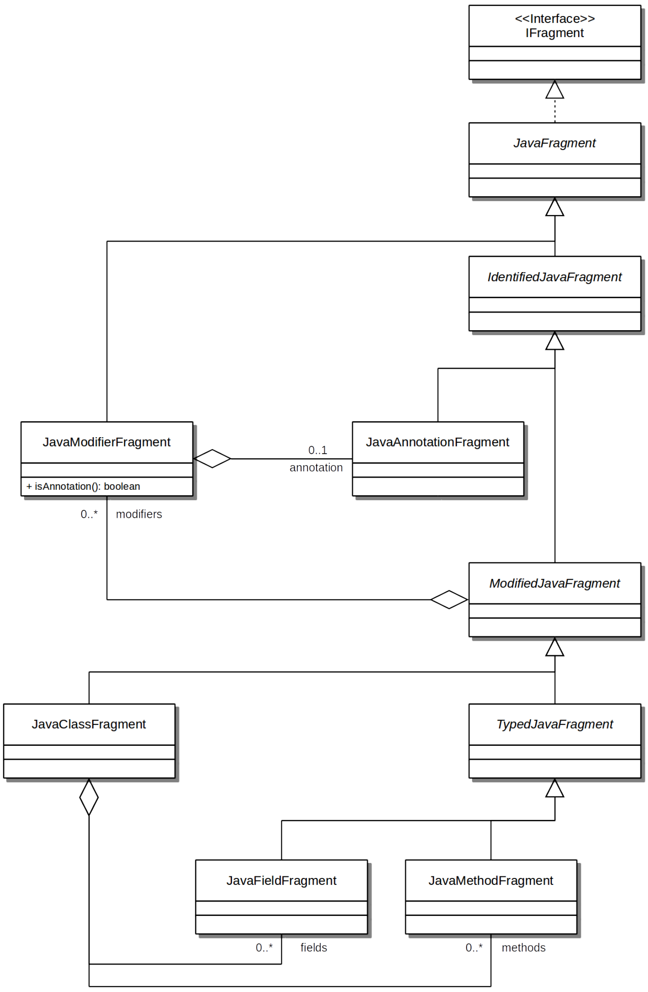


Figure 6.2 Java Fragment AST Model

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 §5.2.2.2. Figure 6.3 shows an excerpt from the listener implementation used to construct the the AST for Java fragments. Here, fragment instances are

```
...
@Override
public void exitMethodDeclaration(Java8Parser.MethodDeclarationContext ctx) {
    javaMethodFragments.push(java8FragmentFactory.newJavaMethodFragment(ctx, javaMethodModifierFragments));
}

@Override
public void exitNormalClassDeclaration(Java8Parser.NormalClassDeclarationContext ctx) {
    javaClassFragments.push(java8FragmentFactory.newJavaClassFragment(ctx, javaClassModifierFragments,
            javaFieldFragments, javaMethodFragments, declaredPackage));
}
...
```

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

Figure 6.3 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.

6.1.2 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.:

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

Further details of naming-scheme employed for fragment entities are explored in §6.1.2.2.

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 §6.1.2.1.

6.1.2.1 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 6.4 shows the megamodel for Java fragments. It introduces the special relationship type `fragmentLanguageOf`, denoting the

```
...
Language < Set
...
fragmentLanguageOf < Language * Language
...
Java : Language

JavaFragments : Language
JavaFragments fragmentLanguageOf Java

JavaClassFragments : Language
JavaClassFragments subsetOf JavaFragments

JavaFieldFragments : Language
JavaFieldFragments subsetOf JavaFragments

JavaMethodFragments : Language
JavaMethodFragments subsetOf JavaFragments
...
```

Figure 6.4 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:

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

6.1.2.2 Fragment Entity Identifiers

6.2 Recovering Parthood Links

This section summarizes the implementation of parthood link recovery.

6.3 Recovering Correspondence & Conformance Links

This section the implementation of correspondence and conformance link recovery

Chapter 7

Case Study

Chapter 8

Conclusion

TBD.

Glossary

101HRMS 101wiki[1] Human Resource Management System[2]. The model used by the 101wiki for its contributions. 14, 15, see also HRMS

Abstract Factory Pattern A creational GoF (Gang of Four) pattern used in software design to decouple instantiation from usage of objects. Hides the concrete nature of created instances. 19, 20, 23

ANTLR Another Tool For Language Recognition. 18–21, 28, 30

API Application Programming Interface. 4, 17–20, 22–25, 28, 37

artifact . 3, 4, 16, 29

AST Abstract Syntax Tree: A tree data structure representing the abstract syntax of a parsed text. This tree omits syntactic features like parentheses for grouping or semicolons for sequencing. 3, 4, 17–20, 22–24, 28–31

Builder Pattern A creational GoF pattern used in software design to prevent constructor parameters from piling up. 23

conformance The relation between . 4, 6, 7, 24, 27, 33

correspondence The relation between . 4, 6, 7, 9, 16, 24, 27, 33

CST Concrete Syntax Tree: A tree data structure representing the concrete syntax of a parsed text.. 17, 18

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

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

DDL Data Definition Language. Language or subset of a language used to describe structure and content of data. 15

DFS The algorithmic concept of traversing a tree or graph data structure 'top-down' until reaching the end of a path before backtracking and traversing another path. 17, 20

DTO Data Transfer Object. Objects with no relevant (business) logic of their own. Their sole purpose is to carry data between layers of a software system. 18

ER Model Entity-Relationship Model. 3

Facade Pattern A structural GoF pattern used in software design to simplify the usage of complex systems or APIs. It provides single access point for such system. Such access points are called facades. 25

fragment A syntactically well-formed piece of a possibly larger text. iv, 4, 8, 16, 27–32

GoF Gang of Four. A group of authors (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) publishing on the subject of object-oriented software design. The term may also refer to design patterns described in their book *Design Patterns: Elements of Reusable Object-Oriented Software* [4]. 36–39

Hibernate The Hibernate ORM Framework. 4, 13–15, 25, 30

HRMS Human Resource Management System. 12, 13

HTTP Hypertext Transfer Protocol. 1

Java The Java Programming Language and Platform. iv, 2, 4, 6, 8, 12–15, 21, 24, 25, 27–32

JAXB Java Architecture for XML Binding. 2, 4, 13–15, 25, 30

language . 3

linguistic architecture . 3

manifestation . 2, see representation

MegaL The megamodeling language developed by the Softlang Team at the University of Koblenz-Landau for descriptively and prescriptively modeling linguistic architectures of software systems. 25, 26, 38

MegaL/Xtext The Xtext implementation and eclipse IDE integration of MegaL. 16, 17, 25, 26

megamodel . iv, 3, 25, 28, 29, 31, 32

mereology . 6, 7, 22, 23

metamodel . 3

O/R-Mapping Object-Relational-Mapping. 15, 25

O/R/X-Mapping Object-Relational- and XML-Mapping. 1, 2, 4, 13–15, 17, 24, 30

O/X-Mapping Object-XML-Mapping. 15

Observer Pattern A behavioral GoF pattern used in software design to propagate state changes from one object to many dependent objects. 20

ontology . 3

ORM . 37, see O/R-Mapping

Parse Tree . iv, 3, 17, 20, 21, 28, 29, 31, see CST

parthood The relation between an entity and its constituent parts. 6–8, 16, 22–24, 27, 32, 33

representation . 2

SQL Structured Query Language. 15, 25

SQL/DDL The DDL subset of SQL. 13, 24, 25, 28

static program analysis . 3, 4

Strategy Pattern A behavioral GoF pattern used in software design to separate behavior from structure. It allows to encapsulate and reuse behavior as part of the configuration of larger constructs. 23

technology . 3

trace link . 3, 4

traceability . 3

traceability recovery . 3

UML Unified Modeling Language. 14, 19, 22, 23, 28

Visitor Pattern A behavioral GoF pattern used in software design to separate behavior from structure. Visitors facilitate the extension of behavior without modifying structure. The Visitor Pattern can be used to traverse object graphs. 20

XML Extensible Markup Language. 1, 2, 12–15, 24, 25, 28, 32

XSD XML Schema Definition. 2, 13, 14, 25

Bibliography

[1] Anya Helene Bagge and Vadim Zaytsev. "Languages, Models and Megamodels". In: Post-proceedings of the Seventh Seminar on Advanced Techniques ar 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 Langua 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] 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.

[5] 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.

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

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

[8] Ralf Lämmel. "Coupled software transformations revisited". In: Proceedings of the 2016 2016, pp. 239–252. URL: http://dl.acm.org/citation.cfm?id=2997366.

[9] Ralf Lämmel and Andrei Varanovich. "Interpretation of Linguistic Architecture". In: Modelling Foundations and Applications - 10th European Conference 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.

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

[11] 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/.

[12] 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.