Department of Informatics and Mathematics
Programming Group

Diploma Thesis

A Description Language for Feature-Oriented Software Development

Author:

Sebastian Scharinger

12th June 2009

Advisors:

Prof. Christian Lengauer, Ph.D.,
Dr.-Ing. Sven Apel

University of Passau
Department of Informatics and Mathematics
94030 Passau, Germany

Scharinger, Sebastian:
A Description Language for Feature-Oriented Software Development
Diploma Thesis, University of Passau, 2009.

Contents

List of Figures iii

Table of listings iv

1 Introduction 1
- 1.1 Problems of Object Oriented Software Design in Product-Line Development . . . 2
- 1.2 Motivation . . . 3
- 1.3 Outline . . . 5

2 Background 7
- 2.1 Features and Feature Composition . . . 7
  - 2.1.1 General Feature Structure Tree . . . 8
  - 2.1.2 Feature Composition . . . 9
  - 2.1.3 Feature Interactions . . . 10
- 2.2 XML-Schema Basics . . . 12

3 The FST Description Language 15
- 3.1 Describing the General FST-Model via XML . . . 15
  - 3.1.1 Type Definition for Terminal and Non-Terminal Nodes . . . 15
  - 3.1.2 Definition of the Program Description . . . 16
- 3.2 Extending the General FST-Model for Java Source Code . . . 18
  - 3.2.1 Mapping of Packages, Classes and Interfaces to Non-terminal Nodes . . . 18
  - 3.2.2 Mapping of Fields, Constructors and Methods to Terminal Nodes . . . 19
- 3.3 Extending the XML-Description . . . 20
  - 3.3.1 Several Extensions . . . 20
  - 3.3.2 Type definition for Packages, Classes and Interfaces . . . 22
  - 3.3.3 Type definition for Fields, Constructors and Methods . . . 22
  - 3.3.4 Example . . . 24

4 Implementation of a Code Generator for FST-DL-Documents 26
- 4.1 General Approach to Parse Source Code Artefacts . . . 27
- 4.2 Parser Tier . . . 27
- 4.3 Converter Tier . . . 28
- 4.4 Generator Tier . . . 32
  - 4.4.1 Data Transformations . . . 33

5 The Application of FST-DL 35
5.1 Representing an XML-Document in a Tree Model 35
5.2 Graphical Visualization of the FST 35
5.2.1 Displaying FST as a Tree 35
5.2.2 Displaying FST as a Graph and Feature Interactions . . . 36
5.2.2.1 Finding Cyclic References between Features . . . 37
5.3 Composition of Features 38
5.4 Case Studies 39
5.4.1 GUIDSL 39
5.4.2 GPL 40
5.4.3 Violet 40

6 Summary 42
6.1 Conclusion 42
6.2 Future Works 42

Bibliography 44

A Appendix 49
A.1 Additional Graphics 49
A.2 Additional Listings 51

List of Figures

2.1 Implementation and FST of the feature *StackBase* 9
2.2 A FST superimposition *Empty* • *Base* = *EmptyBase* 10
2.3 Pseudo-code of features f_1,f_2 and f_3 11
2.4 Pseduo-code of Programs p_1 and p_2 12
2.5 Parts of an XML-Schema 14
2.6 The Typesystem of XML Schema 14

3.1 Both ways of mapping 20
3.2 The FST-DL type system 23
3.3 The Document Structure 23
3.4 The FSTs of the *StackBase* feature and its extension the *StackElementCount* feature. 24
3.5 Implementation of the *StackBase* feature and its extension the *StackElementCount* feature. 25

4.1 The state diagramm of the used automaton 29

5.1 The Tree Representation of a FST-DL Document 36
5.2 The Graph Representation of a FST-DL Document 37
5.3 The Cyclic Reference between Features 38
5.4 Cyclic reference in GUIDSL 40

A.1 The extract of the UML model of the state machines' classes . . . 49
A.2 The Graph Representation of a FST-DL Document with 88 Features 50
A.3 Representation of GUIDSL features 50

Table of listings

2.1 An example XML document describing the programm from figure 2.1 12
2.2 The schema used in listing 2.1 13

3.1 Complex type definition of a non-terminal node 16
3.2 Complex type definition of a terminal node 16
3.3 Modelling the StackBase elements 16
3.4 Definition of the root element 17
3.5 An XML-Document describing the *StackBase-example* from figure 2.1 by using the further introduced types and the root element . . 17
3.6 The Typedefinition of the *FeatureNodeType* 21

4.1 Introduction of the *GDeep* supercall statement to the parser description 28

5.1 The *compose* method implemented by the composer. 38

A.1 The generated program description for the two FSTs defined in figure 3.5 51
A.2 Part of the generated program description for GarphJak example shown in figure 5.1 52

Chapter 1

Introduction

Todays software engineering encounters a lot of new facades compared to years ago. On the one hand lots of software products increase in terms of their functionality, which is often only used to a minor degree by a user. On the other hand mobile computing and embedded systems assign new tasks, by limitation of hardware resources like memory capacity, to a software engineer. Both of these challenges implie the goal to establish customized software products by each of the mentioned needs. Additionally a high reusability of program components should be achieved by modularisation and cohesion of these components.
Regarding software products like an comercial database management system (DBMS) it is outstanding, that these applications fulfill a bunch of different requirements due to their assignments. In terms of mobile computing sometimes lots of the features of these systems are not needed and should be held out, because of limitation of memory. For example features like transaction management and query optimazation could be held out because the amount of data stored in these systems does not justify the amount of resources these features need.
With standard techniques of software engeneering this leads to different implementations of the same functionality to meet different requests like:

- optimization according to reduce consumption of electricity,
- optimization in performance and
- optimization in use of resources.

One possible solution to this problem is the theory of *product-lines and families.* A product-line can be described by[1]:

> *A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

Software Engeineering Institute
Carnegie Mellon University

There are several works, like [Bat00], [Gri00a] and [Gri00b], which are dealing with creating product lines according to the following principles:

[1] This description is taken from the website *http://www.sei.cmu.edu/productlines/index.html.*

- A code basis with the necessary functionality for all products.
- A set of different *features* which contains different functionalities and can be added to the code basis.
- A feature can be necessary, optional or alternatively used.
- A certain product of the SPL can be configuered and generated by selecting different features.

The problem is that there is no appropriate technologie to compose the SPL in the desired quality. The next section deals with this problem in object-oriented software design.

1.1. Problems of Object Oriented Software Design in Product-Line Development

To understand the problems which have to be faced in Product-Line Development two fundamental principles in software design have to be mentioned: *separation of concern* and *cohesion and modularity*

Separation Of Concern
The article [Par72] by D.L. Parnas was one of the first which dealt with the concept of partitioning software in different modules. According to this article these modules can be written only with little knowlege about the other modules in a complex software system and this leads to the following benefits:

- **Managerial:** because there is no need knowlege about the other modules in the system, the development can take place in seperate groups with less need of communication. This should reduce the amount of time it will take to create the complete system.
- **Product Flexibility:** modules can internally be changed, without any influence on the other modules
- **Comprehensibility:** each module can be tested and reviewed at once. This makes it easier to track problems.

Parnas did not use the expression of *Separation of Concerns* in his article, propably E.W. Dijkkstra was the first who formed this expression in [Dij82].
Through the years this was one of the main principles of software development, although it has changed a little bit in its characteristics. Nowadays the focus on structuring software into modules moved from strictly technical aspects to more problem oriented aspects. This means that parts of the software which are designed to a certain intention should be grouped in an individual module. For example logging the actions of a user throughout a whole application should be done by a dedicated module.

Cohesion and Modularity
According to [SMC79] cohesiveness is achieved by decoupling the different modules. To decouple modules it is necessary to reduce the relationship (or bindings) between different modules, whereas the binding within a certain module should be maximized. Modularity is then achieved by building coherent and encapsulated units. These units can be understood as mostly independent modules.

These principles of software design cause several problem in SPL development. The main problems are:

Crosscutting Concerns:
describe a concern which can not be encapsulated in a module. One example for a crosscutting concern is logging a method call. Even if the logic which does the logging, e.g. make a console output, can be modularized. It needs extension to all the methods which should be logged. These methods usually spread throughout several other modules.

Feature Traceability Problem:
goes along with crosscutting concerns. Because the parts of a crosscutting concerns' source code is placed in different modules of the program. It can hardly be determined where to find the source code of a concern. On the other side it is hard to determine whether a certain piece of code belongs to the module where it is found or to a crosscutting concern.

Extensibility Problem:
is dealing with the problem that common design patterns like *composite-* or *visitor*-pattern[2] can be extended only in one dimension. Every additional dimension demands changes throught different modules.

Preplanning Problem:
means that increments of a modules can force changes to other modules or even in the same and therefore must be preplanned.

1.2. Motivation

Due to the problems mentioned in section 1.1 a different way to standard *object-oriented* software design must be choosen to develop efficiently SPLs. There are lots of successful approaches in composing software artefacts by using superimposition[3] throughout different parts of software design and implementation [OH92] [Kat93] [VN96] [Pre97a] [HO93] [MH03]. But all these approaches are engaged only in very explicit parts of software aspects in contrast to the globality of superimposition [AL08].

---

[2] A description of these patterns can be found in [GHJV95].

[3] Section 2.1 gives a brief introduction to feature-oriented programming and also to superimposition.

With [ALB+07] a global definition of feature composition using superimposition is presented. To build a practical toolset based on this definition an description language is needed which recognizes the global theorie. This language should be both independent from the software artefacts which it describes and independent of applications which make use of it.
With XML[4] these goals can be accomplished as XML is designed as a exchange format for a variety of data [W3Cb]. There are a lot of benefits and usage types for XML, especially the ones mentioned by [WK03] make it essential for this purpose:

- **Descriptive markup**
  This point fulfills two necessary requirements for an exchange format (or language) which is needed. On the one hand XML is a markup language so that text (or information[5]) is always embedded in a certain markup structure. On the other hand the markup names can freely be choosen which makes a documents' content and structure both self describing and understandable by people and machines.

- **Hierarchical structuring**
  As mentioned previously the text in XML documents is structured by markup tags. Every begin and end markup tag with the same name forms an *element*, e.g. ⟨*myElement*⟩⟨/*myElement*⟩ forms an element called *myElement*[6]. All elements belong to a root element (or *document element*) and each element can contain other elements itself. This leads to a tree like structure of elements in an XML document. For this reason XML is perfect to describe structures like the FST.

- **Extensibility and Adaptivity**
  XML does not force one to use a certain vocabulary, in contradiction to *HTML*, so everyone can define himself a vocabulary on which his documents are based on. So it is possible to define a vocabulary especially on the terms of FST[7].

- **Standardization and Openness**
  XML itself is an *open standard* so everyone can use it. It is a *meta language* to define different types of documents, but each of these documents can be handled by the same tools[8]. This makes it possible to describe the information about an FST independently from any target application, which might process the information for further usage.

[4] Refer to [BPSM+] for a brief introduction.

[5] As XML originally was designed for text publishing it contains only markup and text. Text and information are used synonymously in this context.

[6] In this case *myElement* is also the name of this element.

[7] This is usually done by *Document Type Definition (DTD)* [W3Cc] or by *XML-Schema* [W3Cd].

[8] Not every tool can handle the information contained in a certain document, but every tool which is able to handle documents itself can handle every type of document.

- **Separation of Content and Presentation**
  XML was designed to capture semi structured text by using a self defined vocabulary for structuring. Each initial representation of an XML-Document leads to a tree like representation of its structure. The way how this information may be processed depends only on the application which uses this representation. The same XML-Document can be represented or processed in different ways, e.g. visualization of an FST in section 5.2 or composing features in section 5.3.

An XML-Document itself does not necessarily need an external defined vocabulary or grammar. But without defining one it, is only possible to check if such a document is *well formed*[9]. The notion of a well formed document includes only the fact that it is syntactically and structurally correct in respect of using the predefined symbols for markup and using a tree structure for the elements. Wellformness of documents does not suffice to use XML as an exchange format. An exact grammar and vocabulary for a document is needed so that every application can check if a document satisfies the needs of the application.
Such grammars can be defined by XML-Schema and each document can then be checked if it is valid against an XML-Schema. XML-Documents can only be valid against a DTD or XML-Schema. If none of them exists one can not determine whether a document is valid or not. There is also a connection between well formness of a document and validity of a document:

- A not valid document can be a well formed document regardless validity can not be determined or it is actually not valid.
- A not well formed document can never be a valid document.

Because of the mentioned benefits of XML as an exchange language and the ability to easily verify the validity of a given document with existing XML-Schema definition the desicion was made to desgin a formal description language in a XML-Schema definition. This definition, its extension to program language specific elements and the implementation of tools generating and using this language are subject of this work.

1.3. Outline

This work is structured as follows:

Chapter 2 introduces the main idea of features and feature composition. As well a basic introduction to XML-Schema is given in this chapter.

Chapter 3 shows how the formal definition of feature structure trees can be mapped into an XML-Schema definition. It also shows

[9] Refer to [W3Cb] 2.1 for an exact definition of well formed documents.

how this definition can be extended to capture the language specific elements of the programming language Java.

Chapter 4 explains how the XML-Shema definition can be implemented into a parser to generate FST-DL-Documents out of Java source code artefacts.

Chapter 5 shows some example applications build on top the generated FST-DL-Documents. It also presents three case studies which were made with the introduced applications.

Chapter 6 summarizes the whole work and gives a short outline for future works based on this one.

Chapter 2

Background

This chapter introduces the main principles and techniques on which this work is based.
The first section shows the main ideas of features their composition and its appliance in software engineering. Also the idea of feature interactions and their impact on feature oriented software design are pointed out.
The second section then introduces the basics of XML and XML-Schema which form the basis of the introduced description language and its applications.

2.1. Features and Feature Composition

Throughout different works related to feature-oriented programming there are many different definitions of what a feature is or should be like[1]:

> *A feature is a product characteristic that users and customers view as important in describing and distinguishing members of the product-line. A feature can be a specific requirement, a selection amongst optional or alternative requirements, or related to certain product characteristics, such as functionality, usability, and performance, or implementation characteristics, such as size, execution platform or standards compliance.*

Martin L. Griss
Implementing Product-Line Features By Composing Component Aspects

This definition captures the need for features in product-line development and what a feature should contain. But it does not define how a feature influences a program or another feature.
There are several other related researches which are dealing with feature refinements[2] like [Pre97a] [BSR03] [BO92] [HO93] and [KIL+97], but there is no overall definition of a feature. Because this work gives an introduction of an overall description language for feature-oriented software development, a feature definition is needed which is independent from the underlying refinement technologies and captures what a feature is. In [ALB+07] such a definition is given by:

[1] This quote is taken from [Gri00a].
[2] A feature refinement describes the way how features are modifying underlying programs.

> *A feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement a design decision, and to offer a configuration option.*
>
> Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner
> An Algebra for Features and Feature Composition

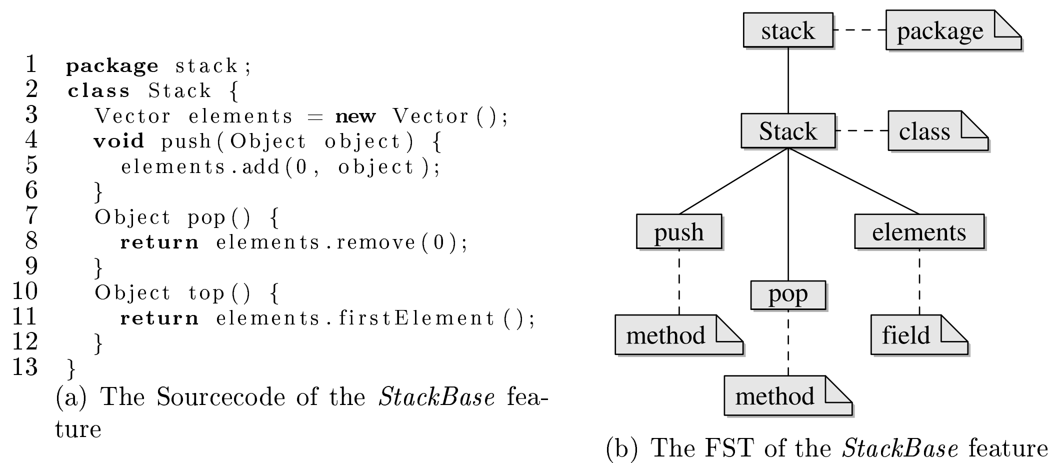
It must be said that a program is the aplliance of two or more features and the resulting program itself can as well be regarded as a feature. This leads to the mathematic feature description[3]:

$$\bullet : F \times F \rightarrow F \qquad p = f_n \bullet f_{n-1} \bullet \ldots \bullet f_2 \bullet f_1$$

2.1.1. General Feature Structure Tree

Regarding the recursive definition of a feature one can see a feature itself contains a lot of structural information. Therefore a feature is a structure which must be able to hold the information about modifying the structure of a given program. Because a program itself can be regarded as a feature it must hold the structure information about this feature or feature composition.
Although a program does not exist of source code, or source code artifacts in this case, more than on other artifacts [BSR03] [CHOT99] [AGM+06], this work is based on describing the structure of source code artifacts as an example for a program representation which is *feature-ready*[4]. The language which is used in this work, is mainly the object-oriented program language Java 1.4 [5] with some limitations and some extensions.
The fact that source code artifacts contain hierarchically ordered structure elements, classes belonging to packages, methods belonging to classes and so on, leads to a tree like representation. This representation is called *feature structure tree* (FST). There is a strong connection between an FST, at least if it is representing code artifacts, and the abstract syntax tree of this artifact. Unlike the abstract syntax tree the detailed level of information contained in the FST depends mainly on the further processing [6]. Independent of the the detail level there is always a tree structure composed of inner nodes and leave nodes.
Figure 2.1 shows the implementation of a simple Java class called *Stack* with two methods and a field. Additionally the associated FST is shown on the right side of this figure. In this exapmle each node of the FST has a name and is labled with a type (*package, class, method and field*), these are necessary informations for further processing and determination of tree nodes.

[3] $\bullet$ denotes the feature composition operator, F denotes a set of features f_n and p the result program.
[4] A feature-ready artefact is a hierarchly structured artefact [ALB+07].
[5] All informations about the programming language Java and its distributions can be found at [Sun08].
[6] In chapter 5 one can see different applications where each needs a different level of information from the FST.



(a) The Sourcecode of the *StackBase* feature

(b) The FST of the *StackBase* feature

Figure 2.1.: Implementation and FST of the feature *StackBase*

2.1.2. Feature Composition

Now by knowing what a feature and its structual representation, the FST, is it is possible to define the composition of features via their FSTs.
[ALB+07] proposes to compose features by FST *superimposition*[7], where two FSTs are superimposed by recursivly superimposing the subtrees starting with the root. How the superimposition takes place is defined informal by [ALB+07]:

> *The basic idea is that two trees are superimposed by superimposing their subtrees, starting from the root and proceeding recursively. Two nodes are superimposed to form a new node (a) when their parents have been superimposed previously or both are root nodes and (b) when they have the same name and type. If two nodes have been superimposed, the whole process proceeds with their children. If not, they are added as separate child nodes to the superimposed parent node. This recurses until all leaves have been processed.*

Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner
An Algebra for Features and Feature Composition

Figure 2.2 shows a feature composition using the *StackBase*-example. The intention is to add a new method *isEmpty* which indicates whether the stack is empty or not and refines the method *pop*, for example to throw an exception when the stack is empty. The package *stack* and the class *Stack* are each merged into one node containing the superimposition of their child nodes. The new method *isEmpty* is added while the refinement and the base method *pop* lead to one new node in the resulting FST. To model this behaviour it is necessary to distinguish between two types of nodes in a FST: *non-terminal* and *terminal* nodes[8].

[7] Refer to [Bos99] [BSR03] [CM86] [OH92] for further informations about superimposition and techniques.

[8] In [ABKR] these nodes are called *compound* and *atomic*. These expressions are synonymously used to *non-terminal* and *terminal*.

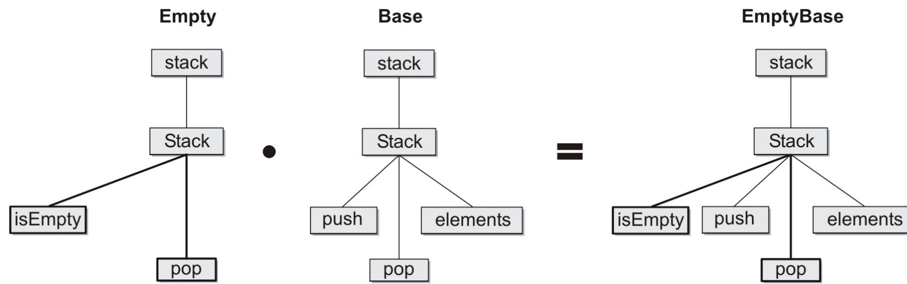


Figure 2.2.: A FST superimposition *Empty* • *Base* = *EmptyBase*

Non-terminal nodes are the inner nodes of an FST, like *Stack* in figure 2.2. A non-terminal node does not contain any content except its implementation structure, modelled by the subtree rooted at the non-terminal node and its name and type.

Terminal nodes are the leaves of an FST. Additionally to its name and type a terminal node can have, and usually does have, some content which is not modelled in the FST. It does not necessarily mean that they do not have any implementation structure below, like a non-terminal node, but in the current representation of the FST this structure is hidden.
In the Java example packages and classes are modelled as non-terminal nodes which are merged when their name and type is the same. Fields and methods are modelled as terminals, for each terminal there must be a specific rule how to superimpose these terminals.

The decision whether a specific implementation artifact is represented by a non-terminal or a terminal node depends on the granularity of the FST and is one subject of chapter 3. The formal algebraic model of features and their composition can be found in [ALB+07], where also a full description of features and composition in general can be found.

2.1.3. Feature Interactions

When composing features the order in which features are composed is very importent [ALB+07] [LBL06] [Bel03] [Pre97b]. The resulting programs p_1 and p_2 can be different even when they are composed by the same features f_1, f_2 and f_3 where $f_3 \bullet f_2 \bullet f_1 = p_1$ and $f_2 \bullet f_3 \bullet f_1 = p_2$. Feature f_1 introduces a class A with a method *getX()* and a field x, pseudo-code shown in figure 2.3(a). Feature f_2, pseudo-code shown in figure 2.3(b), adds another method *setX* to class A and feature f_3, pseudo-code shown in figure 2.3(c), refines both methods and adds a tracing output statement. It depends on the composition mechanism to handle the fact that a method may not be present. For this example it is assumed that the • operator is able to refine only one method if the other is not present.
The different composing results p_1 and p_2 are shown in figure 2.4. While both methods *getX* and *setX* produce an output when they are called in program p_1,

```
class A {
  int x;

  int getX() {
    return x;
  }
}
```
(a) Feature f_1

```
class A {

  void setX(int newX) {
    x = newX;
  }
}
```
(b) Feature f_2

```
class A {

  int getX() {
    System.out.println("called getX");
    Super;
  }

  int setX(int newX) {
    System.out.println("called setX");
    Super;
  }
}
```
(c) Feature f_3. The keyword *Super* denots that the code from the refined method replaces this keyword.

Figure 2.3.: Pseudo-code of features f_1,f_2 and f_3

only the method *getX* produces an output in program p_2.
Interactions between features can be *dynamic* or *static*:

Dynamic interaction: always occours at runtime, when data is exchanged between different features. A feature can refine a method and change the return value of a method, this will affect other features when using this method.

Static interaction: always occurs on implementation level. Features are referencing implementation artefacts which are introduced by other features. [ABKR] defines an additional distinction within static interactions. *Reference* interactions occur when one feature references another feature[9]. *Structural* interactions occur when one feature refines another feature. These interactions can be *unidirectional*, only feature A references feature B, or biderctional, feature A references feature B and vice versa.

There are several works, like [CKMRM03] [LBN05], dealing with feature interactions and resolving these interactions. While resolving interactions is not in the scope of this work, it has to be mentioned, that feature interactions are the cause of many problems in feature-oriented programming like the *feature optionality problem*[10] and the fact that the amount of interactions grows exponentially with the number of features in an SPL.

[9] For example a method call from one method in feature A to a method in feature B.
[10] Refer to section 5.2.2 for further information on this problem.

```
class A {
  int x;

  int getX() {
    System.out.println("called getX"
        );
    return x;
  }

  void setX(int newX) {
    System.out.println("called setX"
        );
    x = newX;
  }
}
```

(a) Program p_1

```
class A {
  int x;

  int getX() {
    System.out.println("called getX"
        );
    return x;
  }

  void setX(int newX) {
    x = newX;
  }
}
```

(b) Program p_2

Figure 2.4.: Pseduo-code of Programs p_1 and p_2

2.2. XML-Schema Basics

According to [Sch] the purpose of an XML Schema is defined as:

> An XML Schema
>
> - defines elements that can appear in a document.
> - defines attributes that can appear in a document.
> - defines which elements are child elements.
> - defines the order of child elements.
> - defines the number of child elements.
> - defines whether an element is empty or can include text.
> - defines data types for elements and attributes.
> - defines default and fixed values for elements and attributes.

These things are achieved by defining elements and their types in the schema which is the base of a schema definition. Therefore this section only deals with explaining the way to define elements and types, all other possibilities are held out.

Listing 2.1: An example XML document describing the programm from figure 2.1

```
<?xml version="'1.0" encoding='ISO-8859-1'>
<ProgrammDescription ProgrammLanguage="Java"
                     xmlns="http://www.scharinger.de/FSTDLSchemata"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                     xsi:schemaLocation="http://www.scharinger.de/FSTDLSchemata.
                         xsd">
  <Class Name="Stack" Package="stack">
    <Variable>elements</Variable>
    <Method>push</Method>
    <Method>pop</Method>
    <Method>top</Method>
  </Class>
</ProgrammDescription>
```

Listing 2.2: The schema used in listing 2.1

```
<?xml version="'1.0" encoding='ISO-8859-1'>
<xsd:schema xmlns:xsd:="http://www.w3.org/2001/XMLSchema"
            targetnamespace="http://www.scharinger.de/FSTDLSchemata">
  <xsd:element name="ProgrammDescription" type="ProgrammDescriptionType"/>

  <xsd:complexType name="ProgrammDescriptionType">
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="Class" type="ClassType"/>
    </xsd:sequence>
    <xsd:attribute name="ProgrammLanguage" type="ProgrammLanguageType"/>
  </xsd:complexType>

  <xsd:complexType name="ClassType">
    <xsd:sequence>
      <xsd:element name="Variable" type="xsd:string" minOccurs="0" maxOccurs="
          unbounded"/>
      <xsd:element name="Method" type="xsd:string" minOccurs="0" maxOccurs="
          unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="Name" type="xsd:string" use="required"/>
    <xsd:attribute name="Package" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:simpleType name="ProgrammLanguageType">
   <xs:restriction base="xs:string">
    <xs:enumeration value="Java"/>
  </xs:restriction>

  </xsd:simpleType>
</xsd:schema>
```

Listing 2.1 shows how the structure of the stack programm in figure 2.1 could be modelled in an XML-Document. The listing 2.2 shows the schema which is the basis for this programm description. It contains some element and attribute declarations as well as type definitions which are explained below.
An XML-Schema document consists mainly of two parts, the element and attribute declaration and the type definition.
An **element or attribute declaration** designates a name and a type to an element or attribute. This declaration can be global or local. Global elements are declared within the *<xsd:schema>* element and local elements within a type definition.
In listing 2.2 the element *ProgrammDescription* is global whereas all other elements and attributes are local. In terms of elements this is necessary, because at least one global element is needed to have a root element in instance documents[11].
A **type definition** like *ProgrammDescriptionType* describes the content of an element of this type. The associativity of an element to a type is done within an element declaration by the type attribute, e.g. *<xsd:element name="Programm-Description" type="ProgrammDescriptionType"/>* declares an element with the name *ProgrammDescription* and type *ProgrammDescriptionType*.
The Typesystem of XML-Schema is devided in *simple types* and *complex types*.
A **simple type** allows charactersets and listings or unions of them to be content of the element. Therefore there are a lot of built-in datatypes defined in XML-

[11] A document is called an instance document of a certain schema, if the document is valid against the schema specified in the attribute *schemaLocation* of the document's root node.

Schema, for example string used in *<xsd:element name="Variable" type= "xsd:string" ... />*.
There is also a mechanism to extend or restrict simple types shown in *<xsd:simpleType name="ProgrammLanguageType">* where the primitive type *string* is restricted to the only value *Java*.
A **complex type** in contrast allows, that an element is composed of character content as well as markup content.

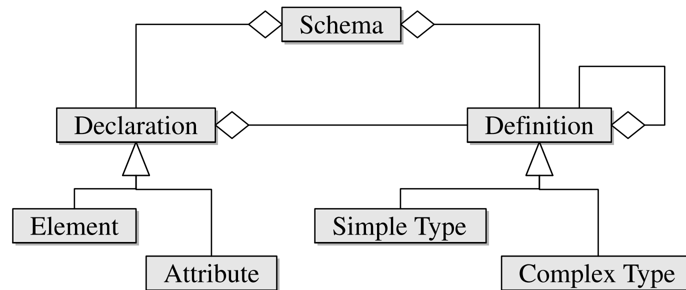


Figure 2.5.: Parts of an XML-Schema

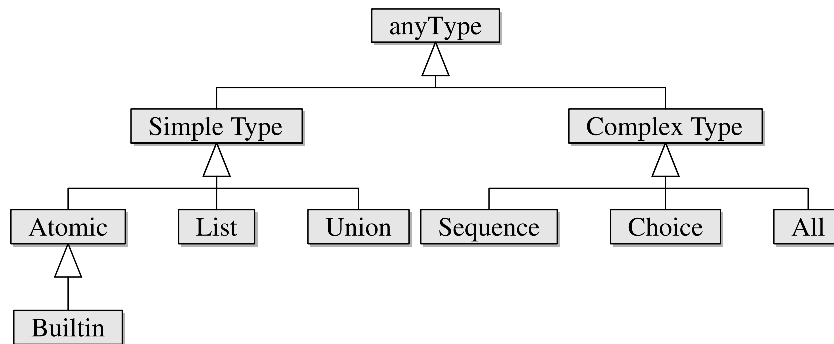


Figure 2.6.: The Typesystem of XML Schema

Chapter 3

The FST Description Language

This chapter introduces the idea of an *feature structure tree (FST)* and its XML-Representation.
In section 2.1.1 one can find a general definition about what an FST is like and how it can be applied to some specific feature languages. This is the basis of the following sections.
Sections 3.2 and 3.3 show how to extend the general idea of FSTs to a program-language and application specific description language called *FST-DL*[1].
At the end of this chapter one should be able to understand the basic idea of an FSTs' XML representation. These are the basics of the prototypic implementation of a parser, shown in chapter 4 to parse Java source code into an FSTDL-Document.

3.1. Describing the General FST-Model via XML

The general model of an FST, described in the last preceding section, allows it to build suitable data models for any application dealing with feature oriented programing or software engineering. As also mentioned in the preceding section, this general FST-Model fits to any program artifact which is feature ready. So the missing thing is an exchange language which expresses the FST-Model of a certain program artifact and also can be used in every type of application not constrained by the techniques used in a special application. In order to meet all these requirements, or at least most of them, an exchange format is necessary which is independent of a target application and also extensible enough to model the theory about FST. According to this, the FST-DL was embedded in an XML-Schema definition.

3.1.1. Type Definition for Terminal and Non-Terminal Nodes

The definition of the feature structure tree in [ALB+07] is keept quite simple by dividing the different parts in two classes: *terminal and non-terminal nodes.* Reproducing this in an XML-Schema is quite simple in the first approximation. To model a basic FST, only two *complex type* definitions are necessary:

[1] FST-DL is used as a shortcut for *Feature Structure Tree Description Language.*

- **Non-Terminal**

  Listing 3.1: Complex type definition of a non-terminal node

```
<xs:complexType name="NonterminalNodeType">
  <xs:sequence minOccurs="0" maxOccurs="1">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="Non-terminal" type="tns:NonterminalNodeType"
          minOccurs="0" maxOccurs="1">
      </xs:element>
      <xs:element name="Terminal" type="tns:TerminalNodeType" minOccurs="0"
          maxOccurs="1">
      </xs:element>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="Name" type="xs:string" use="required" />
</xs:complexType>
```

  This definition ensures that each element has a name-attribute and it ensures that it can have an unlimited amount of child nodes bth of *NonterminalNodeType*- and *TerminalNodeType*-type.

- **Terminal**

  Listing 3.2: Complex type definition of a terminal node

```
<xs:complexType name="TerminalNodeType">
  <xs:attribute name="Name" type="xs:string" use="required" />
</xs:complexType>
```

  An element of the *TerminalNodeType*-type can therefore only have a name-attribute but no child elements.

With these two type definitions it is possible to model all feature structure trees defined in [ALB$^+$07]. The FST displayed in 2.1 could be expressed with the following elements[2]:

Listing 3.3: Modelling the StackBase elements

```
<tns:Nonterminal Name:"stack" xsi:type="tns:NonterminalNodeType">
  <tns:Nonterminal Name:"Stack" xsi:type="tns:NonterminalNodeType">
    <tns:Terminal Name:"push" xsi:type="tns:TerminalNodeType"/>
    <tns:Terminal Name:"pop" xsi:type="tns:TerminalNodeType"/>
    <tns:Terminal Name:"elements" xsi:type="tns:TerminalNodeType"/>
  </tns:Nonterminal>
</tns:Nonterminal>
```

3.1.2. Definition of the Program Description

To build documents representing a FST the definition of the root element is still missing. There is one decission to make, when defining the type of the root element:

- Each *Feature Structure Tree* should be modelled in its own document.

[2] Note that the listing does not describe an XML-Document, only the elements included in a document. To be an XML-Document a root element and the header must be inserted.

Listing 3.4: Definition of the root element

```
<xs:element name="ProgramDescription">
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="Feature" minOccurs="1" maxOccurs="unbounded" type="
          tns:NonterminalNodeType"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

- *Feature Sructure Trees* belonging to a certain *Software Product Line* are combined in one document to describe the whole product line.

There are several pros and cons for both ways modeling this type. On the one hand it is much comfortable to build a program based on one single document then referencing different documents describing whole product lines. On the other hand the reusability of features would be much better if each is encapsulated in its own document.
In XML-Schema there is no limitation to define only one element which then can be a root element of all instance documents. So both ways can be modelled as legal root elements for instance documents. The first intention of this work was to define a structured language to supply other processing programs with the necessary informations. Therefore only the second way is in the scope of this work.
The root element is called *ProgramDescription* to indicate that it contains all the *Feature Structure Trees* that are necessary to describe the whole program. The definition in [ALB+07] does not care about bundling several trees in a description[3].
To stay compliant with the definition of a FST, the root element definition shown in listing 3.4 is keept quite simple as well. It only consists of a list of *NonTerminalNodeType*-elements which are called *Feature*. Note that in the original definition there is no parent node of a FST, but when describing a set of *Features* in one document it is easier to distinguish them by making them explicit.
The whole description of the *StackBase-example* in figure 2.1 is shown in listing 3.5.

Listing 3.5: An XML-Document describing the *StackBase-example* from figure 2.1 by using the further introduced types and the root element

```
<?xml version="1.0" encoding="iso-8859-1"?>
<tns:ProgramDescription xmlns:tns="http://www.scharinger.de/
    FOPClassRepresentation" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
    ">
  <tns:Feature Name:"StackBase" xsi:type="tns:NonterminalNodeType">
    <tns:Nonterminal Name:"stack" xsi:type="tns:NonterminalNodeType">
      <tns:Nonterminal Name:"Stack" xsi:type="tns:NonterminalNodeType">
        <tns:Terminal Name:"push" xsi:type="tns:TerminalNodeType"/>
        <tns:Terminal Name:"pop" xsi:type="tns:TerminalNodeType"/>
        <tns:Terminal Name:"elements" xsi:type="tns:TerminalNodeType"/>
```

3 The description has nothing to do with the technical way to build up a program using several *Feature Structure Trees* which is explained in [ALB+07].

```
      </tns:Nonterminal>
    </tns:Nonterminal>
  </tns:Nonterminal>
</tns:ProgramDescription>
```

3.2. Extending the General FST-Model for Java Source Code

To do some further processing on a *Feature Structure Tree*, some additional information about the structure is comfortable or somtimes necessary. Taking the *pop()* terminal from the *StackBase-example* in figure 2.1 it is not evident that this represents a method. Even with regard to using only Java examples it can not be optained if this terminal represents a method or a constructor of the underlying class. There has to be an additional check to the name of the parent element, to make sure that this can only be a method and no constructor[4].
The following extensions to the basic definition of a *Feature Structure Tree* are not part of [ALB+07]. This section shows a way to map several parts of Java source code to the model of FST.

3.2.1. Mapping of Packages, Classes and Interfaces to Non-terminal Nodes

As mentioned earlier, a non-terminal represents an inner node of the FST with a name and a type. During the process of superimposition the substructure of these nodes are subject of the recursive process. The node itself does not take any changes. In case of packages this perception maps perfect, because the package structure serves only as a path to find the object in the object structure. Regarding classes and interfaces they represent thier own types and the signature, e.g. the methods or fields they contain, are differenting factors in terms of object oriented programming. From the point of view of feature composition they are only containers. So the mapping will work as following:

- **Package:** The non-terminal node representing a package is of the type *Package* and gets the name of the representing package.
- **Class or Interface:** The non-terminal node representing a class or interface is of the type *Class* or *Interface* and gets the name of the representing class or interface.

All other criteria of classes or interfaces, like the modifiers or their implements list and so on, are left out in this view. Whether these parts are modelled as terminal nodes or keept as additional information depends on the underlying application or rule set which is used. It is possible to restrict the superimposition process on not manipulating the modifiers of a class, so adding them as a terminal node will

[4] A Constructor has always the name of the the class in Java.

make no sense. In the scope of this work they are modelled as terminal nodes, but not explicit mentioned anymore.

3.2.2. Mapping of Fields, Constructors and Methods to Terminal Nodes

A terminal node's content is subject of modification during the superimposition process, so these nodes are represented by their type name and content. In the Java language it is not possible to declare fields with different types and equal names on the same level. Therefore it is enough to take the name of the field as the name of its representing node. In case of constructors and methods, the signature[5] must be unique during the whole class or interface. Therefore the name of representing node is assigned to the signature. Fields, Constructors and Methods are mapped as:

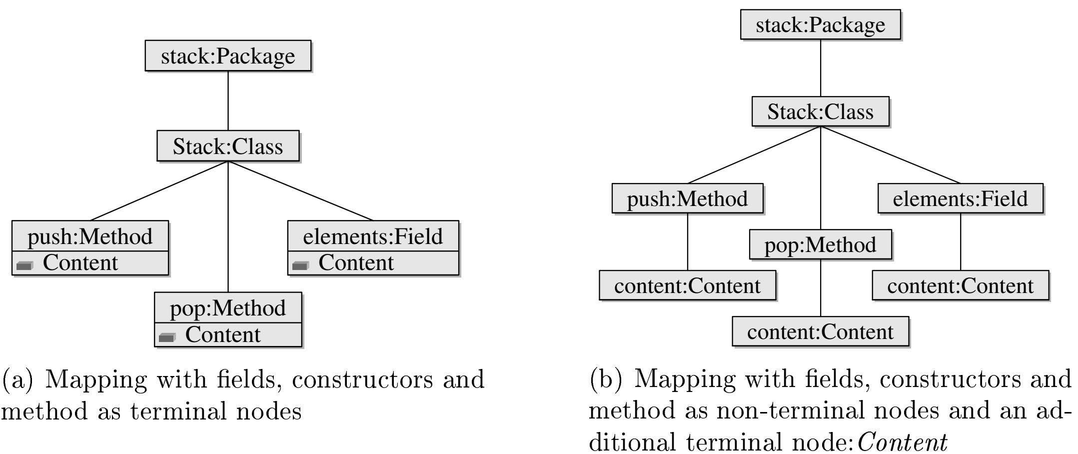
- **Field:** The terminal node representing a field is of the type *Field* and gets the name of the field. The content of this node is the whole field declaration, e.g *String field = "field";*.
- **Constructors and Methods:** The terminal node representing a constructor or method is of the type *Constructor* or *Method* and gets the signature as its name. The content of this node is the whole body of the representing constructor or method.

There are several problems with asigning these objects to terminal nodes:

- These objects also have modifiers which are modelled as terminal nodes when regarding classes or interfaces.
- Methods and constructors can have a list of exceptions which are thrown inside the body. These list can also be a matter of change and therefore must be modelled as a terminal node.
- The real subject of change in the superimposition process is the content of these objects and not the object itself[6].

Due to this the mapping seems not to be correct and only the content of these objects can be a terminal node. The objects themselfs must be represented by non-terminal nodes. As mentioned before it depends only on the granularity which the underlying application needs. To keep the examples clearly arranged during the rest of this work, the mapping of fields, constructors and methods to terminal nodes will be keept.
Figure 3.1 shows both ways of mappings[7]. On the left there is the mapping where fields, methods and constructors are terminal nodes and the content is included in this node. On the right hand each content is a terminal node, while the others are modelled as non-terminal nodes.

[5] In the Java language the signature contains the name and the parameter values.
[6] Even this depends on the strategy how the composing is implemented.
[7] The name of each node is consists of *"name":"type"*.



(a) Mapping with fields, constructors and method as terminal nodes

(b) Mapping with fields, constructors and method as non-terminal nodes and an additional terminal node:*Content*

Figure 3.1.: Both ways of mapping

3.3. Extending the XML-Description

Extending the XML-Description is analogous extending the general FST-Model. All newly introduced type definitions are derivations of the type definitions for non-terminal and terminal nodes. There are several additional requests on the structure of FST by the Java language which had to be modelled as well, e.g. a method body can not contain a regular class definition. There are also requests by the definition of the root element of a description element and the FST, e.g. the root element's children must be *feature nodes* which are modelled as non-terminal nodes, but a *package node* can not have *feature nodes* as its children. Also, as mentioned before the lists of exceptions, which a method can throw, are not part of the terminal and non-terminal definitions, but essential for the further processing. They must be a part of an elements description.
So this section starts with some extensions on the existing types to ensure a complete and correct representation of the underlying source code. Later on the extensions of terminal and non-terminal nodes are made to map the different types in Java. In the end the whole picture of the type structure will be shown and the StackeBase example will be decoded into the FST-DL.

3.3.1. Several Extensions

To ensure that the allowed types for children are determined by its parent, e.g. a class can not have a package as its child, some structure types were introduced:

- FirstOrderStructureType
- SecondOrderStructureType
- ClassOrInterfaceStructureType

While *FirstOrderStructureType* is a direct subtype of *NonterminalNodeType* and remaining types are subtyped along the order in the above list, every subtype of

Listing 3.6: The Typedefinition of the *FeatureNodeType*

```
<xs:complexType name="FeatureNodeType">
  <xs:complexContent>
    <xs:extension base="tns:FirstOrderStructureType">
      <xs:sequence minOccurs="1" maxOccurs="1">
        <xs:element name="NonterminalNode" type="tns:FirstOrderStructureType"
            minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="JavaType" type="xs:string" fixed="Feature" use="
          required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

these types is again a *NonterminalNodeType*. This makes it possible to control the structure and to keep the semantic of *FST Nodetype System*, by allowing only child elements of a certain structure type.

Since a feature is a central in the program description and a necessary element while modeling several FSTs in one document it was assumed to the type system. Listing 3.6 shows the definition of the *FeatureNodeType*. It is an extension to *FirstOrderStructureType* and therefore also a *NonterminalNodeType*. It can have an unlimited amount of child nodes which must be at least of the *FirstOrderStructureType*. This means all other nonterminal nodes are allowed as children, while using the *StructureTypes*.
The attribute *JavaType* is equivalent to the type information which every nonterminal and terminal node has in the FST.
All other types presented in the following are desgined equally, so this is the example for all other type definitions and only massive differences are shown explicit during the rest of this chapter.

To capture all informations included in the described source code some additional listing types where added. They can be found in some other type definitions like *ClassNodeType* and so on.

- **ImportListType** For all imports in a class's header.
- **ExtendsListType** All extends references of a certain class[8].
- **ImplementsList** All implemented interfaces.
- **ModifierListType** All modifiers of the different objects.
- **ThrowsListType** All exceptions listed in a throws clause.
- **InputListType** All input parameters belonging to a constructor or method.

[8] Allthough Java does not allow multiple inheritence, this list type was included to capture the needs of other languages.

There are also three *SupercallTypes* implemented which are used to indicate whether a constructor or method contains a supercall. The supercall mechanism is used by the composing tool, refer to section 5.3 on further information, because they indicate the point where the composition should take place.

- **JavaSupercallType** To indicate that the terminal contains a supercall statement to its parent class.
- **AheadSupercallType** To indicate that the terminal contains a supercall statement which is used by the AHEAD-Tool Suite to compose features.
- **GDeepSupercallType** To indicate that the terminal contains a supercall statement which used by GDeep to compose terminals.

3.3.2. Type definition for Packages, Classes and Interfaces

Packages, Classes and Interfaces are all non-terminal nodes in terms of the FST model, but to capture the Java semantics they are classified in several *StructureTypes*.

PackageNodeType is derived from *SecondOrderStructureType* and allows only children of this type. This ensures that no *FeatureNodeType* can be a child element of a package, but the package itself can be child of a *FeatureNodeType*. This implicates also, that classes and interfaces must be at least from this type otherwise they can not be children of a package and this will not match with the Java language.

ClassNodeType is derived from *ClassOrInterfaceStructureType* and so it is possible to make a class node a child of a package or feature node. The *ClassNodeType* allows only children of the type *ClassOrInterfaceStructureType*, because it is possible to define so called inner classes or interfaces. Also the type *TerminalNodeType*, in order that fields, constructors and methods can be children of a class node is allowed

InterfaceNodeType is designed nearly like the *ClassNodeType* except that it is not allowed to add a child of the type *ConstructorNodeType*. Of course the *JavaType*-attribute is also different.

3.3.3. Type definition for Fields, Constructors and Methods

Because terminal nodes do not have any child elements, the type definitions do not differ much from the original definition. Just the additional information, like the throws list, is varying.

FieldNodeType is derived from *TerminalNodeType* and allows two child elements. The first element is the content element, where the whole declaration ist stored, the second is the type element where the exact Javatype can be stored.

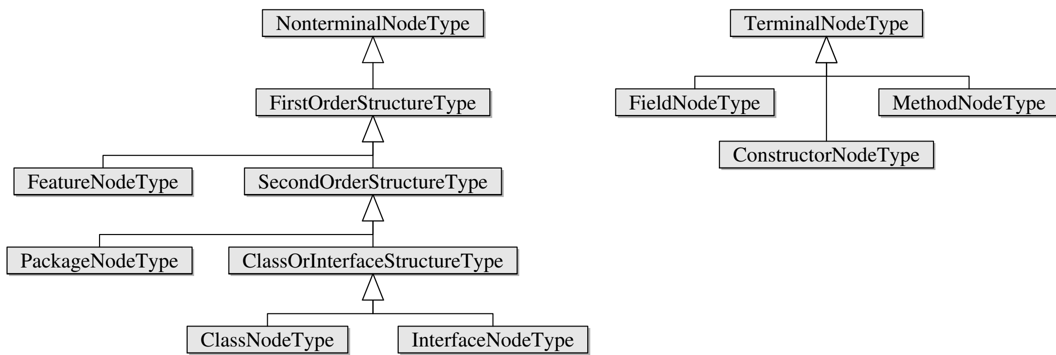


Figure 3.2.: The FST-DL type system

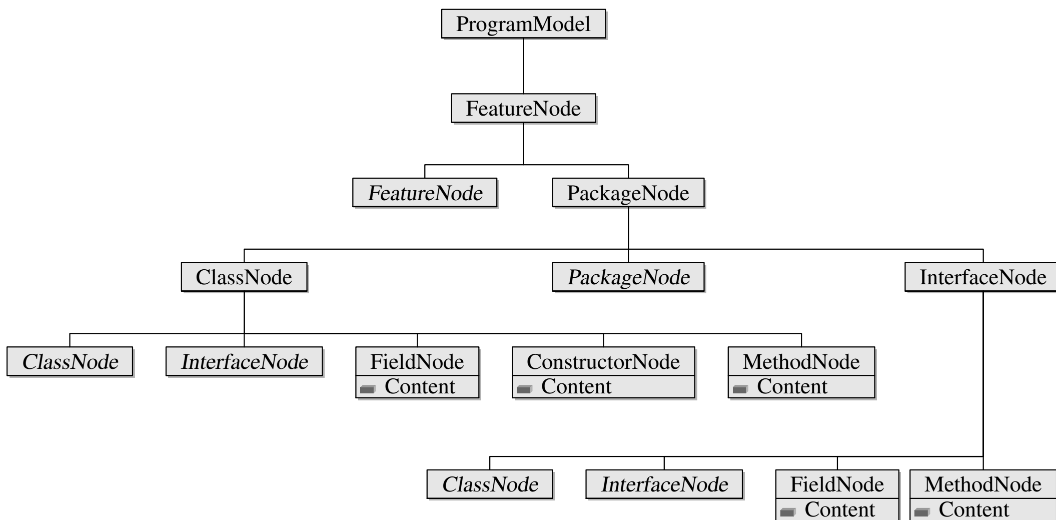


Figure 3.3.: The Document Structure

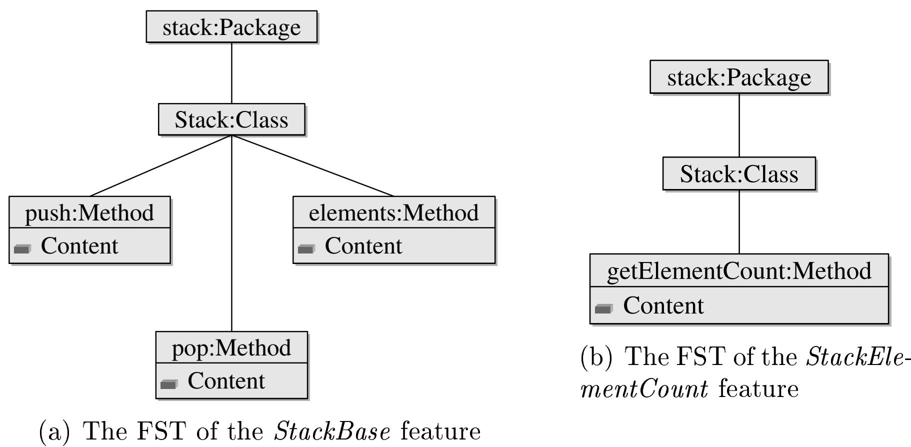
ConstructorNodeType is derived from *TerminalNodeType* and allows several list elements as children: *ModifierList*, *ThrowsList* and *InputList*. The *Content*-child element contains the whole body of the constructor. There is also a list of *SupercallType* elements.

MethodNodeType is derived from *TerminalNodeType* and allows several list elements as children: *ModifierList*, *ThrowsList* and *InputList*. The *ReturnType*-child elements contain the Java type-reference of the method's return type. The *Content*-child element contains the whole body of the constructor. There is also a list of *SupercallType* elements.

Figure 3.2 shows a simplified representation of the introduced type system. Figure 3.3 shows the layout the element structure of possible instance documents according to the FST-DL schema. All recursive steps ar left out and marked by italicised names.

3.3.4. Example

For this example the *StackBase*-example from figure 2.1 is extended by a second FST which will introduce a new method to the class *Stack*. At this point, there is no discussion how this introduction will take place, it only shows the document layout of the description document. Figure 3.4 shows the two FSTs which are used for this example, where *(a)* is the FST for the *StackBase* feature and *(b)* the FST for the *StackElementCount* feature. The *StackElementCount* feature will add a single method, called *getElementCount()* to the *Stack* class which is used to get the current number of elements stored in the *elements* field. The Source of



(a) The FST of the *StackBase* feature

(b) The FST of the *StackElementCount* feature

Figure 3.4.: The FSTs of the *StackBase* feature and its extension the *StackElementCount* feature.

these features is shown in figure 3.5, where *(a)* belongs to the *StackBase* feature and *(b)* to the *StackElementCount*.
Listing A.1 shows the complete program description for the example FSTs, generated by the parser introduced in the next section. What has not been mentioned yet ist the *ID*-attribute in each element. This attribute is used by XML to reference elements throughout the whole document. It can be used by parser to jump from one element description to another, so it has no impact on the introduced FST-DL-Description. What can be done is to define some key definitions on this attribute to ensure referential integrity which is used in the FST-DL description to check references made in the element *DissolvedReference*, a child element of *ReferenceList*. These elements are used resolve feature interactions.
What might be confusing is the fact, that there are only used a few different element names for the "main" elements *Nonterminal* and *Terminal*. This was done to keep the connection to the *General Feature Structure Tree* model where only these two types exist. The produced documents can therefore easily be mapped on this original definition. Just leaving out the additional information, like the return type of a method, and removing the *xsi:type* attribute, leads to a valid document against the general definition.

```
1  package stack;
2  class Stack {
3    Vector elements = new Vector();
4    void push(Object object) {
5      elements.add(0, object);
6    }
7    Object pop() {
8      return elements.remove(0);
9    }
10   Object top() {
11     return elements.firstElement();
12   }
13 }
```

(a) The Sourcecode of the *StackBase* feature

```
1 package stack;
2 class Stack {
3   int getElementCount() {
4     return elements.size();
5   }
6 }
```

(b) The Sourcecode of the *StackElementCount* feature

Figure 3.5.: Implementation of the *StackBase* feature and its extension the *StackElementCount* feature.

An XML-Parser[9] is not able to distinguish between the different types when using the same element names for different children. For example the child elements of the *Stack* class are all *Terminal*-elements, but they do have different types. To gain the ability to resolve the exact type of this element the attribute *xsi:type* is used. This attribute refers to the type definition in the underlying schema which then is used to evalute the validity of this element declaration.
The whole listing containing the example can be found in the appendix section A.2 listing 3.5, which contains the example.

[9] like Xerces parser [Gro08].

Chapter 4

Implementation of a Code Generator for FST-DL-Documents

This chapter describes the exact implementation of the code generator (framework), which is currently used in the *FST-Composer*[1]. The main goal of this implementation was to create a plugable framework, which produces an FST-DL document for further processing. The process, resulting in the document generation, passes through three phases:

- parsing the code-artefacts written in Java language *parser tier*,
- collecting all the parsing informations and convert them *converter tier* and
- further data processing and generation of the *FST-DL* document *generator tier*.

Each of this phases should be done by different modules of the framework, so one can easily replace the implementation of a phase. Therefore the framework is concieved in a three tier architecture, where each tier models exactly one phase of the generation process.
The first tier is realized by a modified Java parser, the second tier by a mealy machine[2] and the third tier then by a tree model. To accomplish the separation of the parser tier and the converter tier the facade pattern[3] was used. This pattern ensures that the different tiers communicate by a so called facade class and therefore it is possible to replace the whole tier by another implementation without changing the implementation of the other tiers. The facade object is called *Java2XMLConverter* it is called by the parser and the passes all the parser informations to the underlying objects of the state machine model.
The converter tier and the gernerator tier are divided by using two different object models which do the desired work. So the state machine objects in the converter tier create several *TreeModel* objects and pass the processed information to them. The *TreeModel* objects then can be used to do further transformations on the passed data and finally produce the XML-Document describing the parsed source code-artefacts.

[1] Refer to [Ape] for additional information about the FST-Composer project.
[2] Refer to [Mel55] or [HMU00] for further informations on mealy machine and automata theory
[3] Refer to [GHJV95] p.185 ff. for further information on the facade pattern.

4.1. General Approach to Parse Source Code Artefacts

Before introducing the different tiers it must be mentioned how the parser collects the different sourcecode artefacts and builds up one document with all the included FSTs.
To start the parsing process, an expression file must be created which is passed to the parser engine. This file contains names of all features which should be parsed in this style:f_1 f_2 $f_2 ... f_n$[4]. The name of a feature in this file must exactly match by a file system directory and the directory must be a subdirectory of the directory where the expression file is located. The features themself mus be represented by *containment hierarchies* [BSR03]. All artefacts (code and noncode) must be contained in a file system directory and its subdirectories.
The order of the features in the expression file is necessary to use the resolve reference option, refer to section 4.4.1 to gain more information on this option, and also defines the order in which the features are composed by the composer.
The parser then starts to parse each artefact that can be found in the feature directory and its subdirectories by processing the different feature directories in the given order. Within a folder it simply parsers the existing artefacts in alphabetical order.

4.2. Parser Tier

To produce an XML-Description of given source code artefacts, it is necessary to extract the required information. This information is likely the information in the abstract syntax tree of this artefact but not in complete deepth. For further processing only this information is needed:

- The definition of a class with its package declaration, import list, modifiers, name, extends clause and implements list.
- The definition of a interface with its package declaration, importlist, modifiers, name extends clause and implements list.
- The definition of a field with its modifiers, type, name and the assignement.
- The definition of a constructor with its modifiers, input parameter list and the complete body.
- The definition of a method with its modifiers, return type, name, input parameter list and the complete body.

Especially when parsing fields, constructors and methods the information about the structure of their bodies[5] is not necessary. To parse the given Java source

[4] This is called the *feature expression*.
[5] In case of fields the body is the assignment of the field.

Listing 4.1: Introduction of the *GDeep* supercall statement to the parser description

```
void OriginalMethodCallExpression():
{
}
{
  "original" {converter.gdeepSuperConstructBeginRead();converter.readString("
      original");} Arguments() {converter.gdeepSuperConstructEndRead();}
}
```

code the *javaCC 4.0* parser generator[6] was used. This parser generator comes in with a complete parser description for the *Java 1.5 language specification*, which was used to generate the parser for this work. Allthough this work basses on the *Java 1.4 language specification*, this doesn't matter, because all 1.5 specific parts are ignored by the underlying tiers.
To gain the ability of parsing also *AHEAD* source code artefacts, it was necessary to change the parser to accept the *refines-* keyword, *layer* statement and the specific supercall statements[7].
To notice *GDeep* supercall statement *original()*, it was necessary to introduce a new statement to the parser description. Listing 4.1 shows a part of the extension to the parser description to recognize the supercall statement. Since the statement *original(...)* is like a "normal" methodcall to the parser, it will accept this statement without this extension. It was only made to be able to indicate this call in the outcomming document and prevent additional parsing in further applications.
Listing 4.1 shows also, how the communication with the second tier takes place. The parser only calls methods on the object *converter* which is defined in the parsers constructor and is the entry class to the *DFA*.
Except the extensions on the language structure which is accepted by the parser, all other extensions are only method calls on the *converter* object. This is why no specific parser implementation is needed, only the communication interface must be implemented and called on the desired parts of source code parsing. This is also the point where it can be controlled how much information will be included in the generated XML-Description. When there is no need for the concrete body of methods, then these communication calls are just omitted.

4.3. Converter Tier

Since this tier is used to convert the parsed information into a tree representation, the states of the used state machine correspond to the states of a parser accepting the artefact language. Because the *FST-DL* does not contain the complete structural information of the source code artefact, some states are not necessary. Figure 4.1 shows the complete state diagramm of the implemented automaton.

6 Refer to the project Homepage [Jav08] for futher information.
7 Refer to [AHE08] for further information on the AHEAD Tool Suite.

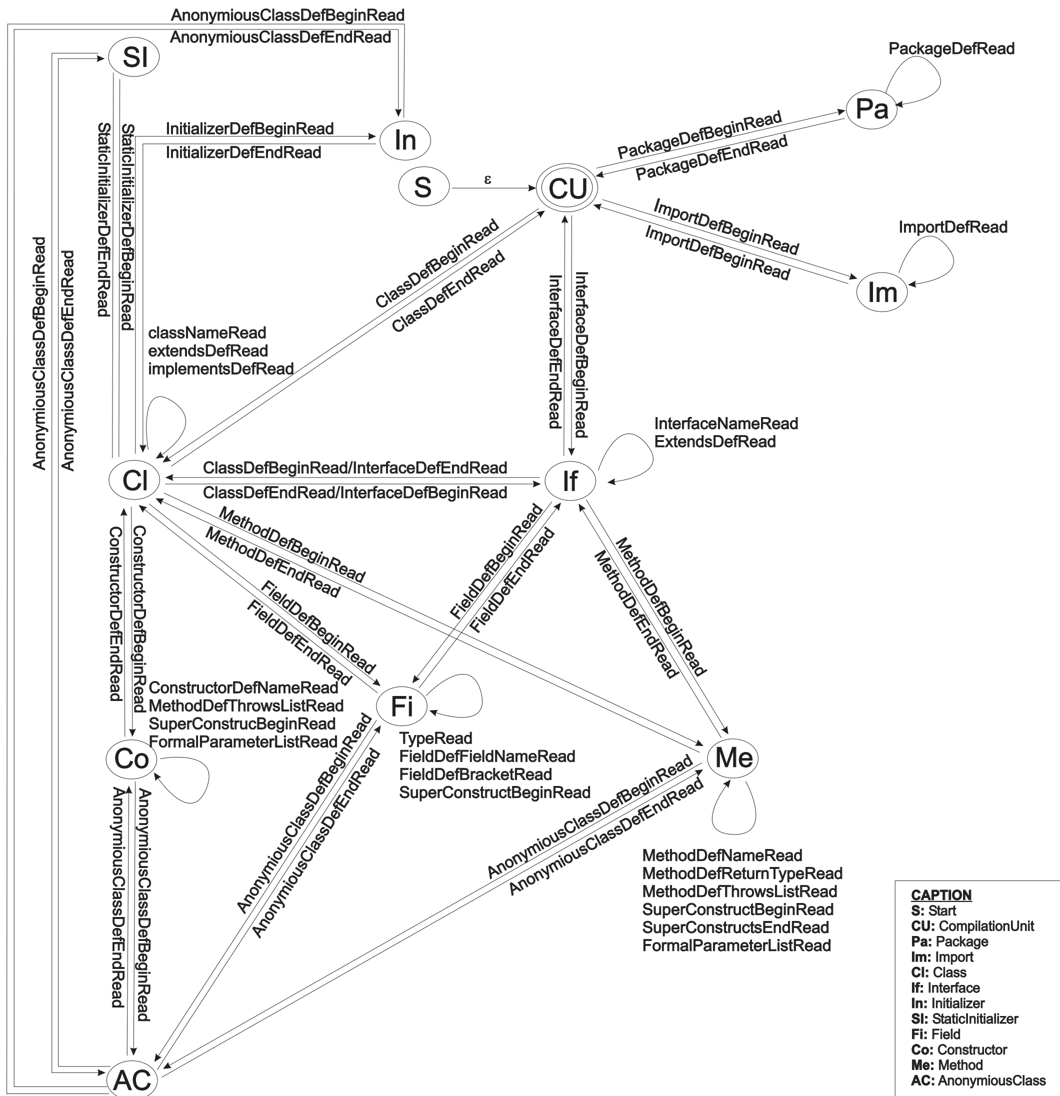


Figure 4.1.: The state diagramm of the used automaton

To be able to easily add new states and transitions a special design was used to model the states. An abstract class *AbstractAutomatNode* contains all transition methods which can be used in the whole state machine. All other objects which model a concrete state of this state machine, must extend this class. Per default each transition method in the *AbstractAutomatNode* throws in an *UnsupportedNodeOperationException*, so that a concrete state implementation does not need to care about transitions which are not allowed in this state. All transitions which are allowed in a concrete state have to be overridden by this class.
Figure A.1 shows an extract of the automaton class model, it does not contain all the additional fields and methods each class can have. Also the figure shows only the model for two subclasses of the *AbstractAutomatNode*. It should just illustrate how the transition methods are implemented.
The *AbstractAutomatNode* also contains some fields and methods used by all subclasses, which can be divided into two groups. The first group are fields and methods which are used to avoid the parsers necessary *lookahead*, the second group is to keep information persistend over statechanges.

Methods to avoid the *lookahead*
Some informations are passed through the parser before knowing to which state they will belong. The best example are the *modifiers* which are passed through and it is not known whether they will belong to a class definition or a field definition. Therefore these informations are collected in the *AbstractAutomatNode* in static fields, so if the state changes, the following state can access these informations and process them. The fields and methods belonging to this group are:

- The field *String currentModifier* and the method *String getCurrentModfier()*. There is also the method *void modifierRead(String modifier)* which is defined as a transition method in each state to itself, but it is used as described above to store the modifier information belonging to the following state.

- The field $Vector < String > importedObjects$ and the methods $Vector < String > getImportedObjects()$, *void addImportedObject(String importedObject)* and *void clearLoadedImports()*. This field and its access methods are used to manage the imports at the beginning of class and interface declarations.

- The field *boolean packageRead* and the method *boolean packageRead()* which indicates that a package definition has been read. This information is necessary when a class or a interface are processed, because a so called *Default Package* has to be inserted if a class or interface does not have a package declaration.

All these fields and their access methods are modelled with a *static* modifier, so they can be accessed by each subclass and it is not necessary to pass this information from one state to another by referencing the following state directly.

Fields and methods to keep information persistend over statechanges

- A field *AbstractAutomatNode father* and the *AbstractAutomatNode getFather()* method.
- A *static* field *AbstractTreeNode currentTreeNode* and the *AbstractTreeNode getCurrentTreeNode()* method.
- A field *ProgrammTreeModel treeModel* and the *ProgrammTreeModel getTreeModel()* method.

To understand the meaning and the functioning of these fields and methods in this state machine, a closer look at the design of the whole state machine structure is necessary.
As one can see in figure 4.1 the shown automaton is not deterministic. For example the *field* state has two transitions *FieldDefEndRead* to different states *Interface* and *Class*. This is also where another specific design of the automaton class modelling snaps in. The automaton has no static structure which is produced at startup and then held throughout the whole process. It has a tree like representation where the different objects, representing the states, can appear in different instances. To come upon a deterministic automaton each state where the following state can not be determined is cloned as often as the same transitions are leading away from this state to another. So for example the *field*-state exists in two instances and the *AnonymiousClass*-state in five instances. To realize this behaviour, each implemented state knows its possible upcoming states, which are reached by a *xxxDefBeginRead* transition. The *xxxDefEndRead* transition is then realized by calling the method *getFather()* which returns the state from which this state was reached.
Since the automaton holds only one instance of each following state in an internal field it is necessary to keep several references to the generated *ProgrammTreeModel*. For example the processing of the whole class definition splitts up on different states. So after the class name, its modifiers and so on have been read, the parser might deliver a constructor definition. This will force the state machine to switch to the state *Constructor* where the upcomming input of the parser is processed. The constructor information and all upcomming informations about methods and flieds have to be added to the *ClassTreeNode* produced in the *Class*-state, therfore it is necessary to obtain all the nodes generated in the different states.
To get access to the complete *ProgrammTreeModel* at the current time, note that this model changes throughout the whole process. Besides the *AbstractTreeNode*, which is modified at a certain time, these fields and methods were implemented. These fields and their access methods represent the internal memory structure of this automaton. Each state generates a specified *TreeNode* element which is added to the *ProgrammTreeModel* as a child of the *currentTreeNode* and then sets the *currentTreeNode* to the newly generated element. At some places, like the transition form the *Method*-state back to the *Class*-state, the *currentTreeNode* is not generated new by the *Class*-state, but rather the formaly generated node replaces the current.

4.4. Generator Tier

The last phase of the *FST-DL* document generation process is modelled by a tree model. This model is produced by the *Converter* tier and than can be used to do some further transformations on the given input from the parser. The *ProgrammTreeModel* represents the complete FST structure of the given source code artefacts and contains also the additional informations like the different *Supercall* statement. Each element of the tree model represent exactly one FST node or additional information element. So the process of producing the XML-Document is straight forward. Each node knows its representation in the XML-Document and only fills in the parsed information where it is necessary.
The technology to produce the XML-Document which used in this tier, is the Document Object Model defined by the W3C in [W3Ca]. It defines how XML-Documents can be accessed by in object oriented program languages. Java therfore defines a set of interfaces within the *org.w3c.dom* package. To process XML-Documents by the DOM api with Java an additional parser is needed which is accessible by the Java *javax.xml.parsers.DocumentBuilderFactory*. In this work the xerces parser provided by the *Appache Software Foundation* [Fou08] is used to generate the XML-Documents. Allthough this parser implementation is used, the implementation is not limited to this parser because all types which are used, are taken from the standard java api. Every implementation of a DOM parser which uses the Java specification can also be used. The Java Api provides a *DocumentBuilderFactory* which is called to get an implementation of the *DocumentBuilder* interface. Which implementation the factory returns can be controlled by several ways described in the api description of this method.
Not all available parsers will deliver the full function set described in the Java api. The Xerces parser was choosen, because with this parser it is possible to create XML-Documents which are validated against a given XML-Schema.This ensures, that all documents produced by this tier are valid FST-DL documents, otherwise the parser will generate an exception and will not produce the document.
For the generation process of the FST-DL-Document, only two types from the api are necessary:

- **Document** Which represents a whole XML-Document.
- **Element** Which represents a single XML-Element.

The *Document* object is produced by the *DocumentBuilder*, it contains only the header of a XML-Document. To insert an *Element* in this document, it must first be created with the *createElement* methods in a document object. After filling the *Element* with the necessary information it can be directly added to the *Document*-object. Than it is the root element of this document, or added to an other *Element*, in both ways the method *appendChild(...)* is used. If the parser is turned on validation, then it would generate exceptions at this point when violating element constraints[8].

[8] For further information on DOM-Parsing in Java please refer to [McL00] or [WK03].

The generation process starts at the root of the tree steps recursivly through the whole document. Each node adds its element representation to the document and then calls its children. The position where the new element should be added is determined by the method *getElementById()* in the document object. The *ID* attribute is defined in the FST-DL as the feature name and the full qualified object name. This is unique over the whole document per definition, otherwise when inserting an element with an existing id, the parser will encounter an error.

4.4.1. Data Transformations

As mentioned earlier, the data is transformed as well in this phase.
One transformation which is implemented is the *resolve reference option* which is used to make static feature interactions visible. The idea is to resolve all type and method references within all features which are contained in the given expression. References to types and methods belong to code artefacts which are not located in the given features, for example the standard Java library, are regarded as unresolvable references. That is because a feature can not refine an artefact out of a third party library, otherwise the source of this library would be a feature itself and must be declared in the feature expression.
To reduce the complexity of this option, only used types in field declaration[9],used types in constructor or in method signatures as well as used types in import lists, implements clauses, refines clauses and throws clauses are regarded. The parser in its current implementation does not parse any method body, constructor body or field assignment, which would be necessary to resolve all references. The complete reference resolving is out of the scope of this work beacuse it should only show a different appliance of the FST-DL.
All the above mentioned type declarations are collected during the parsing process in the *Converter Tier*, and passed to the *Generation Tier* when creating the different tree nodes. The basis is a simple ruleset based on the rules for feature composition in [ALB+07] and the Javaconvention on source code, like uniqueness of qualified names in Java:

- The first appearance of a type definition is the base definition. The first appearance results from the parsing order, which is given by the feature expression.
- All other type definitions with the same full qualified Java name must be refinements of the base definition.
- Every use of a certain type reference to the base definition. If no such base definition exists it is a unresolvable reference.
- Each resolvable reference gets an entry into the FST-document with an object-reference to the base definition[10].

[9] field assignment is also not included.
[10] Therefore every XML-Object in document needs a unique ID.

Its not necessary to explain the whole implementation in detail, the main concept are static datastructures which are accessable from every tree node object and which are containing unique entries for each type definition. Since this parser is processing artefacts within a feature in alphabetical order and each directory in a deepth first search manner, it is possible that some type references can occur while the definition has not been parsed[11]. This makes it necessary that the whole reference resolving process can only be finished after the parsing process has finished.

[11] It is even possible, that a feature f_n references a type definition, which is introduced later in an feature f_{n+x}.

Chapter 5
The Application of FST-DL

To show the appliance of the above introduced FST-DL and the implemented parser, some applications are included to the FST-Composer.
All of this shown applications use a tree representation of the FST-DL XML-Document which is introduced in section 5.1.
The first applications in this section present a graphical visualization of the FST. While the tree representation gives a good access to all of the components of the whole FSTs given by a certain feature expression. The graph representation gives a first impression how the different features interacts with each other. This section shows also a way to visualize feature interactions and presents a way to compute, whether a given feature expression is valid or not.
The prece section shows the first application which is able to compose features represented in FST-DL.
At the end of this chapter three case studies are shown where all the implemented applikations were tested with.

5.1. Representing an XML-Document in a Tree Model

To represent an XML-Document in Java the same techniques to create this document, which were used in chapter 4, can be used again. The DOM-Model of the document is again produced with the xerces parser. To add all additional functionality each DOM-Element[1] is encapsulated in a so called *AbstractFSTDL-TreeNodeAdapter*, following the *Adapter Pattern*[2]. Each Type which occurs in the *type attribute* of a non-terminal or terminal node element in the FST-DL is represented by a concrete class extending the abstract Adapter. So it is possible to enrich the different adapter with individual methods used for example to display it.

5.2. Graphical Visualization of the FST

5.2.1. Displaying FST as a Tree

Figure 5.1 shows the tree representation of an FST-DL document modeling the FSTs of *GraphJak*, an example delivered with the FST-Composer. On the left (1)

[1] These are the objects which contain the information of the XML-Element.
[2] Refer to [GHJV95] p.139 ff. for further information on the adapter pattern.

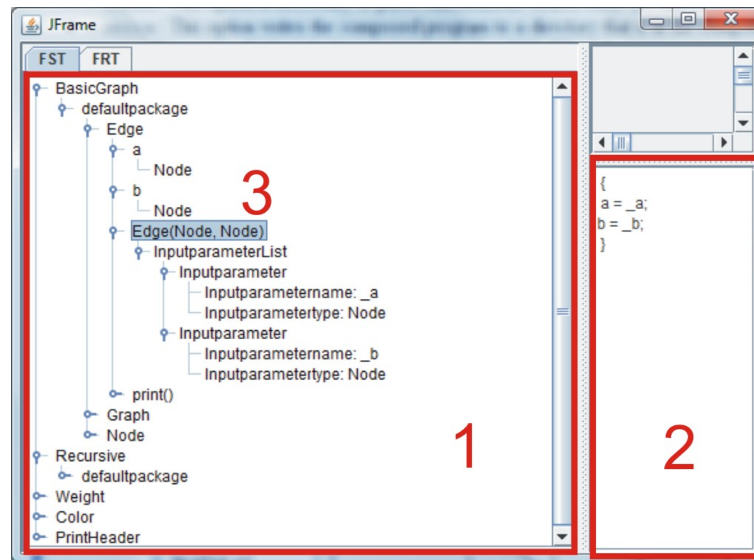


Figure 5.1.: The Tree Representation of a FST-DL Document

one can see the tree representation of the features:*BasicGraph*, *Recursive*, *Weight*, *Color*, *PrintHeader*. A part of the generated FST-DL can be seen in listing A.2 in the Appendix. While the first feature is spread out one can see different components of this feature and on the right (2) the content of the selected method node (3).

5.2.2. Displaying FST as a Graph and Feature Interactions

The graph representation is used to show the dependencies between different features. As figure 5.2 (1) shows, each node of the graph represents a feature and each directed edge represents one or more references for example from feature *Recursive* to feature *BasicGraph*. In fact there are several references from *Recursive* to *BasicGraph* as it can be seen at the bottom (2). Additionally to the graph there is a new tree representation (3), which shows the references between one feature, in the first layer of the tree, and another, the child of a first level feature.
To display the FST-DL Document as a feature an additional library[3] was necessary. Addtionally to this a new datamodel is needed which encapsultes the tree model in a graph model. In the underlying model the references computed as explained in the section 4.4.1 are added to the tree[4]. This graph has a very high complexity and no particial information kann be found while regarding the graphical representation. Even the choosen representation becomes confusing when there are a lot of features as figure A.2 shows[5]. To reduce complexity all

[3] The library used to display a graph in a Java application was the jung-1.7.6.jar found at *http://jung.sourceforge.net/*.

[4] Listing A.3 shows a part of the generated FST-DL Document, where the references can be found.

[5] Note that these graphs are containing only a fraction of all references.

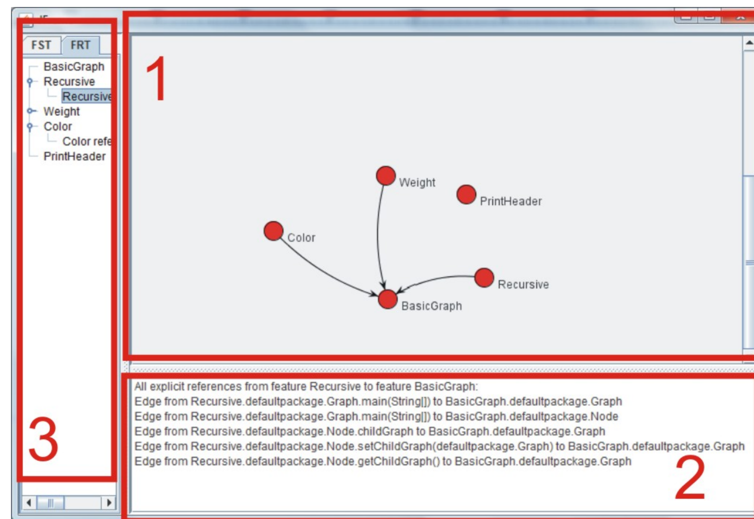


Figure 5.2.: The Graph Representation of a FST-DL Document

nodes where consolidated to one node with the name of the feature.
To display the edges and keep the information, which specific references contain in one edge. The references where also moved to the consolidated node keeping the additional information which former start- and endpoint they had. All edges are then consolidated to one edge between two feature nodes, but keeping the access to the single edges they contain.

5.2.2.1. Finding Cyclic References between Features

To compute the cyclic references of a feature graph basicly a *Depth First Search* algorithm with classification of edges is used as explained in [CLRS01]. By using the DFS timestamps and edge classification it is possible to find cycles in a feature graph. If such cycles exist in a graph, then a partital order on the features can not be obtained, this leads to the *feature optionality problem* [ABKR] [LBL06] [Pre97a]. This means that features are recommended in a SPL for syntatic reasons, the absence of one feature will cause a compile time error, rather than for a semantic reason.
Even though the reference resolving gathers a fraction of all references it was possible to find such a cycle in the FST-Composer examples. Figure 5.3 shows the cyclic reference(2) between the features *Base* and *UndirectedOnlyVertices* in the *GPL* example. (2) indicates by turning red, that this reference is one part of the cycle, whereas (3) lists all involved features. The cyclic reference in this example is not very critical, because the *Base* feature is always required and as a base feature it has always references leading to it. The reference from *Base* to *UndirectedOnlyVertices* results from an import clause in *Base* which imports a type introduced in *UndirectedOnlyVertices* but it is never used throughout *Base*.

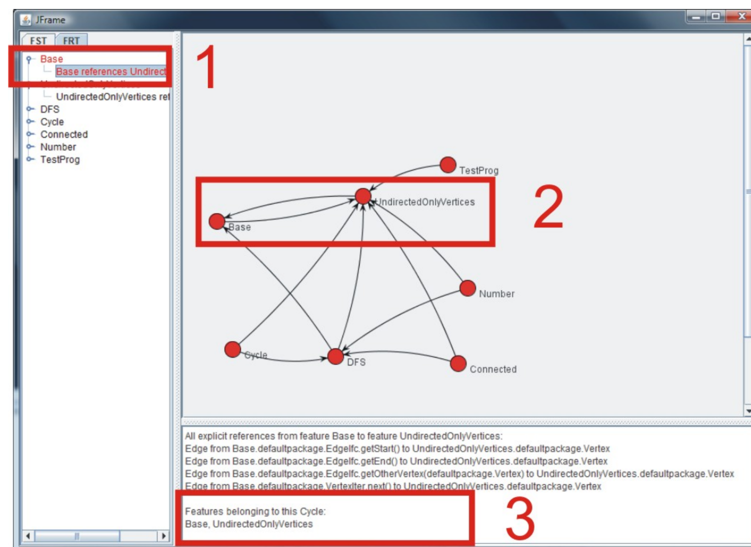


Figure 5.3.: The Cyclic Reference between Features

5.3. Composition of Features

The composition of features was implemented by Sven Apel [AL08]. The input of the composition tool is the generated FST-DL by the parser described in chapter 4. The tool does not take any advantage of resolved reverences, so this optional functionality can be turned off for composition. To compose Java artefacts the different rules for terminal composition (mentioned in section 2.1) are implemented.

The composer takes two FSTs and then it composes them recursivly with the *compose method* shown in listing[6] 5.1. To walk through the trees the method *findChild* is used to find a child in subtree, located at the current position, with the same name and type. These two nodes are then composed by the method *compose*, if there is no matching child node found the node is added to the result tree.

Listing 5.1: The *compose* method implemented by the composer.

```
static Tree compose(Tree treeA , Tree treeB ) {
  Node newNode = treeA.node().composeNode(treeB.node());
  if (newNode != null) {
    Tree newTree = new Tree(newNode);
    for (Tree childA : treeA.children()) {
      Tree childB = treeB.findChild (childA.name(), childA.type());
      if (childB != null) newTree.addChild(compose(childA , childB));
      else newTree.addChild(childA.copy());
    }
    for (Tree childB:treeB.children ()) {
      Tree childA = treeA.findChild(childB.name(), childB.type());
      if (childA == null) newTree.addChild(childB.copy());
    }
    return newTree ;
  } else return null;
}
```

[6] The implementation is taken from [AL08].

The composition of two nodes is done by the method *composeNode*. Several distinctions according to the node type (terminal or non-terminal) and according to the java type (class, method, field etc.) must be made in this method. The exact composition rules and implementations of all types can be found in [AL08].

5.4. Case Studies

The case studies which always were made in association with Sven Apel and his implementation of the composer tool were introduced in section 5.3. The intention was not just to produce FST-DL-Documents above verifying that the produced documents can be taken as base to compose the different programs. Because of the strong connection to the composer tool during thie case studies there were made no other tests concerning the correctness of the composition as in [AL08] ,so these are not mentioned here again.
The first program that was examined was *GUIDSL*[7] which is a graphical programming tool developped by Batory [Bat05]. Secondly the *Graph Product-Line* (GPL) was examined which is a collection of graph applications, implemented by Herrejon and Batory [LHB01]. The last study was made on a graphical UML editor called *Violet*[8] which was refactored into several features by a student.

5.4.1. GUIDSL

GUIDSL is a graphical tool to specify product line configurations and constraints [Bat05]. It consists of 26 different features with only one valid configuration. Different feature expressions will also lead to a compilable program, but without any useful functionality. The used GUIDSL artifacts which were used are all written in the *JAK*[9] which were composed into a Java program.
The resulting Java program was composed by using all 26 features of the GUIDSL tool. It consists of 9,050 lines of composed Java code[10]. The generated FST-DL-Document contains 34,819 lines. Figure A.3 in section A.1 shows the complete graphical representation of the GUIDSL features.

While studying this example it points out that the GUIDSL feature description contains also cyclic references which are shown in figure 5.4. In this cyclic reference three features are involved *kernel*, *genbali* and *dgram*. There is one bidirectional reference between *kernel* and *genbali* and also a cyclic reference because *genbali* references *dgram* and *dgram* references *kernel*.

[7] The GUIDSL-homepage can be found under http://www.cs.utexas.edu/users/dsb/fopdocs/guidsl.html.

[8] http://sourceforge.net/projects/violet/.

[9] JAK is a special Java dialect used by the AHED tool suite [AHE08] for feature-oriented programming.

[10] For comparability of the lines-of-code metric, the composed code was formatted using a standard Java pretty printer (http://www.jindent.com/). Unlike [AL08] also lines containing just one character were counted but no comments. Because another pretty printer was used the lines of codes differ. This was necessary to compare it with the lines of code of the FST-DL-Dokument (http://www.csc.calpoly.edu/ jdalbey/SWE/PSP/LOChelp.html).

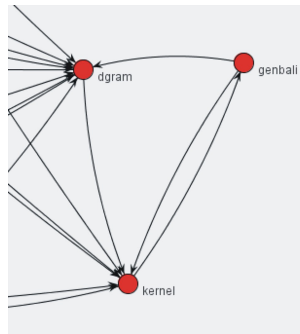


Figure 5.4.: Cyclic reference in GUIDSL

The complete parsing process, resolving references, produce the FST-DL output and set up the gui with its different datamodels was done in about 5 seconds.

5.4.2. GPL

The code artefacts of the *Graph Product-Line* are again written in JAK. It cointains 26 features, but only 17 contain source code artefacts, mainly divided into basic graph features and graph algorithms. Basic graph features are for example directed and undirected graphs or weighted and unweighted edges. Included graph algorithms are depth-first search, breadth-first search, cycle checking, computation of strongly connected components and so on. Because the different algorithms requires different types of graphs and there are a lot of different configurations.
For the case study the same testcases as in [AL08] were used but without the addaptions to process other artefacts than source code. Of the 10 testcases each includeds seven to nine features with about 200 - 400 lines of code in the composed programm. The lines of the FST-DL-Description varied from 1,300 to 2,200. In each testcase the lines in the description were more than five times the lines of code in the composed program.
Each test case could be done in less than 3 seconds.

5.4.3. Violet

The last case study was the *Violet*, a graphical UML editor written in Java. It has been refactored by a student as a class project at the University of Texas in Austin[11]. The refactored version of *Violet* was divided into 88 features with several functions like different UML diagram types, drag-and-drop and look-and-feel functionality. The used code base contains 157 classes, implemented by 5,220 lines of Java code. The 10 used test cases included 51 - 88 features and generated programs with 3,700 to 4,800 lines of code. The produced FST-DL-Documents had from 14,500 to 19,000 lines. Each test case took about ten seconds with

[11] The project was done in the course of the 2006 FOP class at the Department of Computer Sciences of the University of Texas in Austin.

parsing, resolving references, generating the FST-DL output and starting up the gui. Figure A.2 shows the visualization of the FST using all 88 features.

Chapter 6

Summary

At the end of this work all the results of this work are summarized and a short perspective on possible future researchs is given.

6.1. Conclusion

This work shows a general mapping from FSTs into a XML-Schema which contains all the necessary elements to reflect the structure and behaviour of an FST. Furtheron a first extension to this mapping was done by adding language specific elements to it. So it was possible to represent an FST based on the program language JAVA in FST-DL.
With the implementation of a parser to produce FST-DL-Documents out of Java sourcecode a basis for further processing of these FSTs was founded. On top of the parser implementation several applications were build to enhance feature-oriented software design.
With the case studies it was shown that the approach with FST-DL is useable even with programs consisting of 10,000 lines of code. Although the reference resolfing was implemented with minimized functionality the case studies pointed out that feature interactions can be found and also some of the problematic cases can be detected.

6.2. Future Works

Because of the XML implementation of FST-DL it is possible to create a composer using XML related toolsets like XML-Stylesheet. Which will be a step forward to make it complete language independent while composing source code artefacts. Another advantage would be that with XML-Databases the complete SPL descriptions could be stored. Tehy form a huge feature repository. This repository then can be queried with standard XML functionality from this composer.
The minimized reference resolving implementation in this work could also be a major field of research. Especially with regards on resolving feature interactions on base of circle elimination in graphs. Therefore a complete reference detection mechanism must be found.
With the presented gui to visualize FSTs a first step in direction of an editor was done. The gui could be enhanced to make modifications in the feature structure

or could even build a whole SPL.
This work only introduces a parser for Java language and its' dialect JAK. Extending the parser and therefore the FST-DL to other languages will be a main goal for future researches.

Bibliography

[ABKR] APEL, Sven ; BATORY, Don ; KÄSTNER, Christian ; ROSENMÜLLER, Marko: *Handling Large-Scale Feature Interactions*

[AGM+06] ALVES, Vander ; GHEYI, Rohit ; MASSONI, Tiago ; KULESZA, Uirá ; BORBA, Paulo ; LUCENA, Carlos: Refactoring product lines. In: *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–237–2, S. 201–210

[AHE08] *The AHEAD Tool Suite Homepage*. Website http://www.cs.utexas.edu/users/schwartz/ATS.html, 2008

[AL08] APEL, Sven ; LENGAUER, Christian: Superimposition: A Language-Independent Approach to Software Composition. In: PAUTASSO, Cesare (Hrsg.) ; TANTER, Éric (Hrsg.): *Software Composition, 7th International Symposium, SC 2008, Budapest, Hungary, March 29-30, 2008. Proceedings* Bd. 4954, Springer, 2008 (Lecture Notes in Computer Science), S. 20–35

[ALB+07] APEL, Sven ; LENGAUER, Christian ; BATORY, Don ; MÖLLER, Bernhard ; KÄSTNER, Christian: An Algebra for Feature-Oriented Software Development / Department of Informatics and Mathematics University of Passau. 2007 (MIP-0706). – Forschungsbericht

[Ape] APEL, Sven: *FST-Composer Homepage*. Website http://www.infosun.fim.uni-passau.de/cl/staff/apel/FSTComposer/. http://www.infosun.fim.uni-passau.de/cl/staff/apel/FSTComposer/

[Bat00] BATORY, Don: Product-line architectures, aspects, and reuse (tutorial session). In: *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–206–9, S. 832

[Bat05] BATORY, Don: Feature Models, Grammars, and Propositional Formulas. In: *Software Product Line Conference (SPLC)*, 2005

[Bel03] BELTAGUI, Fatima: Features and Aspects: Exploring feature-oriented and aspect-oriented programming interactions / Lancaster University Computing Department. 2003. – Forschungsbericht

[BO92] BATORY, Don ; O'MALLEY, Sean: The design and implementation of hierarchical software systems with reusable components. In: *ACM Trans. Softw. Eng. Methodol.* 1 (1992), Nr. 4, S. 355–398. http://dx.doi.org/http://doi.acm.org/10.1145/136586.136587. – DOI http://doi.acm.org/10.1145/136586.136587. – ISSN 1049–331X

[Bos99] BOSCH, Jan: *Superimposition: A Component Adaptation Technique.* 1999

[BPSM+] BRAY, Tim ; PAOLI, Jean ; SPERBERG-MCQUEEN, C. M. ; MALER, Eve ; YERGEAU, François: *Extensible Markup Language (XML) 1.0 (Fourth Edition).* Website http://www.w3.org/TR/xml/. http://www.w3.org/TR/xml/

[BSR03] BATORY, Don ; SARVELA, Jacob N. ; RAUSCHMAYER, Axel: Scaling step-wise refinement. In: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering.* Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–1877–X, S. 187–197

[CHOT99] CLARKE, Siobhán ; HARRISON, William ; OSSHER, Harold ; TARR, Peri: Subject-oriented design: towards improved alignment of requirements, design, and code. In: *SIGPLAN Not.* 34 (1999), Nr. 10, S. 325–339. http://dx.doi.org/http://doi.acm.org/10.1145/320385.320420. – DOI http://doi.acm.org/10.1145/320385.320420. – ISSN 0362–1340

[CKMRM03] CALDER, Muffy ; KOLBERG, Mario ; MAGILL, Evan H. ; REIFF-MARGANIEC, Stephan: Feature interaction: a critical review and considered forecast. In: *Comput. Netw.* 41 (2003), Nr. 1, S. 115–141. http://dx.doi.org/http://dx.doi.org/10.1016/S1389-1286(02)00352-3. – DOI http://dx.doi.org/10.1016/S1389–1286(02)00352–3. – ISSN 1389–1286

[CLRS01] CORMEN, T. H. ; LEISERSON, C. E. ; RIVEST, R. L. ; STEIN, C.: *Introduction to Algorithms.* Second. MIT Press, 2001 http://mitpress.mit.edu/catalog/item/default.asp?tid=8570\&ttype=2

[CM86] CHANDY, Mani ; MISRA, Jayadev: An example of stepwise refinement of distributed programs: quiescence detection. In: *ACM Trans. Program. Lang. Syst.* 8 (1986), Nr. 3, S. 326–343. http://dx.doi.org/http://doi.acm.org/10.1145/5956.5958. – DOI http://doi.acm.org/10.1145/5956.5958. – ISSN 0164–0925

[Dij82] DIJKSTRA, Edsger W.: On the role of scientific thought. In: *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, 1982, S. 60–66

[Fou08] FOUNDATION, The Appache S.: *Homepage.* Website `http://www.apache.org/`, 2008

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Resusable Object-Oriented Software.* Addison-Wesley Professional, 1995

[Gri00a] GRISS, Martin: Implementing Product-Line Features By Composing Component Aspects. In: *First International Software Product-Line Conference*, 2000

[Gri00b] GRISS, Martin L.: Implementing Product-Line Features with Component Reuse. In: *ICSR*, 2000, S. 137–152

[Gro08] GROUP, The Xerces P.: *The Xerces Project Hompage.* Website `http://xerces.apache.org/`. `http://xerces.apache.org/`. Version: 2008

[HMU00] HOPCROFT, John E. ; MOTWANI, Rajeev ; ULLMAN, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation (2nd Edition).* Addison Wesley, 2000 `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0201441241`. – ISBN 0201441241

[HO93] HARRISON, William ; OSSHER, Harold: Subject-oriented programming: a critique of pure objects. In: *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications.* New York, NY, USA : ACM, 1993. – ISBN 0–89791–587–9, S. 411–428

[Jav08] *The JavaCC Project Hompage.* Website `https://javacc.dev.java.net/`, 2008

[Kat93] KATZ, Shmuel: A superimposition control construct for distributed systems. In: *ACM Trans. Program. Lang. Syst.* 15 (1993), Nr. 2, S. 337–356. `http://dx.doi.org/http://doi.acm.org/10.1145/169701.169682`. – DOI http://doi.acm.org/10.1145/169701.169682. – ISSN 0164–0925

[KIL^{+}97] KICZALES, Gregor ; IRWIN, John ; LAMPING, John ; LOINGTIER, Jean-Marc ; LOPES, Cristina V. ; MAEDA, Chris ; MENDHEKAR, Anurag: *ASPECT-ORIENTED PROGRAMMING.* 1997

[LBL06] Liu, Jia ; Batory, Don ; Lengauer, Christian: Feature oriented refactoring of legacy applications. In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–375–1, S. 112–121

[LBN05] Liu, Jia ; Batory, Don S. ; Nedunuri, Srinivas: Modeling Interactions in Feature Oriented Software Designs. In: *Feature Interactions in Telecommunications and Software Systems VIII, ICFI 05, 28-30 June 2005, Leicester, UK*, 2005

[LHB01] Lopez-Herrejon, Roberto E. ; Batory, Don S.: A Standard Problem for Evaluating Product-Line Methodologies. In: *GCSE '01: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*. London, UK : Springer-Verlag, 2001. – ISBN 3–540–42546–2, S. 10–24

[McL00] McLaughlin, Brett: *Java and XML*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2000. – ISBN 0–596–00016–2

[Mel55] Melay, GH: A Method to Synthesizing Sequential Circuits / Bell System Technical J. 1955. – Forschungsbericht

[MH03] McDirmid, Sean ; Hsieh, Wilson C.: Aspect-oriented programming with Jiazzi. In: *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. New York, NY, USA : ACM, 2003. – ISBN 1–58113–660–9, S. 70–79

[OH92] Ossher, Harold ; Harrison, William: Combination of inheritance hierarchies. In: *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*. New York, NY, USA : ACM, 1992. – ISBN 0–201–53372–3, S. 25–40

[Par72] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. In: *Commun. ACM* 15 (1972), Nr. 12, S. 1053–1058. `http://dx.doi.org/http://doi.acm.org/10.1145/361598.361623`. – DOI http://doi.acm.org/10.1145/361598.361623. – ISSN 0001–0782

[Pre97a] Prehofer, Christian: Feature-Oriented Programming: A Fresh Look at Objects. In: *Proc. Ann. European Conf. Pbject-Oriented Programming*, 1997

[Pre97b] Prehofer, Christian: An Object-Oriented Approach to Feature Interaction. In: *In Fourth IEEE Workshop on Feature Interactions in Telecommunications networks and distributed systems*, 1997, S. 313–325

[Sch] School, W3C: *XML Schema Tutorial*. Website http://www.w3schools.com/schema/default.asp. http://www.w3schools.com/schema/default.asp

[SMC79] Stevens, W. ; Myers, G. ; Constantine, L.: Structured design. (1979), S. 205–232. ISBN 0–917072–14–6

[Sun08] Sun: *Java Hompage*. Website http://java.sun.com, 2008

[VN96] VanHilst, Michael ; Notkin, David: Using role components in implement collaboration-based designs. In: *SIGPLAN Not.* 31 (1996), Nr. 10, S. 359–369. http://dx.doi.org/http://doi.acm.org/10.1145/236338.236375. – DOI http://doi.acm.org/10.1145/236338.236375. – ISSN 0362–1340

[W3Ca] W3C: *Document Object Model (DOM)*. Website http://www.w3.org/DOM/. http://www.w3.org/DOM/

[W3Cb] W3C: *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. Website http://www.w3.org/TR/REC-xml/. http://www.w3.org/TR/REC-xml/

[W3Cc] W3C: *Guide to the W3C XML Specification("XMLspec")DTD, Version 2.1*. Website http://www.w3.org/XML/1998/06/xmlspec-report. http://www.w3.org/XML/1998/06/xmlspec-report

[W3Cd] W3C: *XML-Schema Part0: Primer Second Edition*. Website http://www.w3.org/TR/xmlschema-0/. http://www.w3.org/TR/xmlschema-0/

[WK03] Wittenbrink, Heinz ; Köhler, Werner: *XML (Wissen, dass sich auszahlt)*. 2003

Appendix A

Appendix

A.1. Additional Graphics

Figure A.1.: The extract of the UML model of the state machines' classes

Figure A.2.: The Graph Representation of a FST-DL Document with 88 Features

Figure A.3.: Representation of GUIDSL features

A.2. Additional Listings

Listing A.1: The generated program description for the two FSTs defined in figure 3.5

```
<?xml version="1.0" encoding="iso-8859-1"?>
<tns:ProgrammDescription xmlns:tns="http://www.scharinger.de/
    FOPClassRepresentation" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
    ">
  <tns:Nonterminal ID="StackBase" JavaType="Feature" Name="StackBase" xsi:type="
      tns:FeatureNodeType">
    <tns:Nonterminal ID="StackBase.stack" JavaType="Package" Name="stack"
        xsi:type="tns:PackageNodeType">
      <tns:Nonterminal ID="StackBase.stack.Stack" JavaType="Class" Name="Stack"
          xsi:type="tns:ClassNodeType">
        <tns:Terminal ID="StackBase.stack.Stack.elements" JavaType="Field" Name=
            "elements" xsi:type="tns:FieldNodeType">
          <tns:ReferenceList>
            <tns:UndissolvedReference Referencing="Vector"/>
          </tns:ReferenceList>
          <tns:Type>Vector</tns:Type>
          <tns:Content> =  new Vector (   ) ;</tns:Content>
        </tns:Terminal>
        <tns:Terminal ID="StackBase.stack.Stack.push(Object object)" JavaType="
            Method" Name="apush(Object object)" xsi:type="tns:MethodNodeType">
          <tns:ReferenceList>
            <tns:UndissolvedReference Referencing="Object"/>
          </tns:ReferenceList>
          <tns:InputParameterList>
            <tns:InputParameter>
              <tns:ParameterName>object</tns:ParameterName>
              <tns:ParameterType>Object</tns:ParameterType>
            </tns:InputParameter>
          </tns:InputParameterList>
          <tns:ReturnType>void</tns:ReturnType>
          <tns:Content>{
    elements.add(0, object);
  }
           </tns:Content>
        </tns:Terminal>
        <tns:Terminal ID="StackBase.stack.Stack.pop()" JavaType="Method" Name="
            pop()" xsi:type="tns:MethodNodeType">
          <tns:ReferenceList>
            <tns:UndissolvedReference Referencing="Object"/>
          </tns:ReferenceList>
          <tns:InputParameterList>
          </tns:InputParameterList>
          <tns:ReturnType>Object</tns:ReturnType>
          <tns:Content>{
    return elements.remove(0);
  }
           </tns:Content>
        </tns:Terminal>


        <tns:Terminal ID="StackBase.stack.Stack.top()" JavaType="Method" Name="
            top()" xsi:type="tns:MethodNodeType">
          <tns:ReferenceList>
            <tns:UndissolvedReference Referencing="Object"/>
          </tns:ReferenceList>
          <tns:InputParameterList>
          </tns:InputParameterList>
          <tns:ReturnType>Object</tns:ReturnType>
          <tns:Content> {
    return elements.firstElement();
  }
           </tns:Content>
        </tns:Terminal>
```

```
      </tns:Nonterminal>
    </tns:Nonterminal>
  </tns:Nonterminal>
  <tns:Nonterminal ID="StackElementCount" JavaType="Feature" Name="
      StackElementCount" xsi:type="tns:FeatureNodeType">
    <tns:Nonterminal ID="StackElementCount.stack" JavaType="Package" Name="stack
        " xsi:type="tns:PackageNodeType">
      <tns:Nonterminal ID="StackElementCount.stack.Stack" JavaType="Class" Name=
          "Stack" xsi:type="tns:ClassNodeType">
        <tns:Terminal ID="StackElementCount.stack.Stack.getElementCount()"
            JavaType="Method" Name="getElementCount()" xsi:type="
            tns:MethodNodeType">
          <tns:ReferenceList>
            <tns:UndissolvedReference Referencing="int"/>
          </tns:ReferenceList>
          <tns:InputParameterList>
          </tns:InputParameterList>
          <tns:ReturnType>int</tns:ReturnType>
          <tns:Content>{
    return elements.size();
  }
            </tns:Content>
        </tns:Terminal>
      </tns:Nonterminal>
    </tns:Nonterminal>
  </tns:Nonterminal>
</tns:ProgrammDescription>
```

Listing A.2: Part of the generated program description for GarphJak example shown in figure 5.1

```
  <tns:Nonterminal ID="BasicGraph" JavaType="Feature" Name="BasicGraph" xsi:type
      ="tns:FeatureNodeType">
    <tns:Nonterminal ID="BasicGraph.defaultpackage" JavaType="Package" Name="
        defaultpackage" xsi:type="tns:PackageNodeType">
      <tns:Nonterminal ID="BasicGraph.defaultpackage.Edge" JavaType="Class" Name
          ="Edge" xsi:type="tns:ClassNodeType">
        <tns:Content/>
        <tns:Terminal ID="BasicGraph.defaultpackage.Edge.a" JavaType="Field"
            Name="a" xsi:type="tns:FieldNodeType">
          <tns:ReferenceList>
            <tns:UndissolvedReference Referencing="Node"/>
          </tns:ReferenceList>
          <tns:Type>Node</tns:Type>
          <tns:Content>;</tns:Content>
        </tns:Terminal>
        <tns:Terminal ID="BasicGraph.defaultpackage.Edge.b" JavaType="Field"
            Name="b" xsi:type="tns:FieldNodeType">
          <tns:ReferenceList>
            <tns:UndissolvedReference Referencing="Node"/>
          </tns:ReferenceList>
          <tns:Type>Node</tns:Type>
          <tns:Content>;</tns:Content>
        </tns:Terminal>
        <tns:Terminal ID="BasicGraph.defaultpackage.Edge.Edge(defaultpackage.
            Node, defaultpackage.Node)" JavaType="Constructor" Name="Edge(Node,
            Node)" xsi:type="tns:ConstructorNodeType">
          <tns:ReferenceList>
            <tns:UndissolvedReference Referencing="Node"/>
          </tns:ReferenceList>
          <tns:InputParameterList>
            <tns:InputParameter>
              <tns:ParameterName>_a</tns:ParameterName>
              <tns:ParameterType>Node</tns:ParameterType>
            </tns:InputParameter>
            <tns:InputParameter>
              <tns:ParameterName>_b</tns:ParameterName>
              <tns:ParameterType>Node</tns:ParameterType>
```

```
            </tns:InputParameter>
          </tns:InputParameterList>
          <tns:Content> {
 a = _a;
b = _b;
 }
 </tns:Content>
        </tns:Terminal>
```

Listing A.3: Part of the generated program description for GarphJak example shown in figure 5.2

```
<tns:Nonterminal ID="Recursive" JavaType="Feature" Name="Recursive" xsi:type="
    tns:FeatureNodeType">
  <tns:Nonterminal ID="Recursive.defaultpackage" JavaType="Package" Name="
      defaultpackage" xsi:type="tns:PackageNodeType">
    <tns:Nonterminal ID="Recursive.defaultpackage.Graph" JavaType="Class" Name
        ="Graph" xsi:type="tns:ClassNodeType">
      <tns:Content/>
      <tns:Terminal ID="Recursive.defaultpackage.Graph.main(String[])"
          JavaType="Method" Name="main(String[])" xsi:type="tns:MethodNodeType
          ">
        <tns:ReferenceList>
          <tns:DissolvedReference Referencing="BasicGraph.defaultpackage.Graph
              "/>
          <tns:DissolvedReference Referencing="BasicGraph.defaultpackage.Node"
              />
          <tns:UndissolvedReference Referencing="String"/>
          <tns:UndissolvedReference Referencing="main(String[])"/>
        </tns:ReferenceList>
        <tns:ModifierList>
          <tns:Modifier>public</tns:Modifier>
          <tns:Modifier>static</tns:Modifier>
        </tns:ModifierList>
        <tns:InputParameterList>
          <tns:InputParameter>
            <tns:ParameterName>args</tns:ParameterName>
            <tns:ParameterType>String[]</tns:ParameterType>
          </tns:InputParameter>
        </tns:InputParameterList>
        <tns:ReturnType>void</tns:ReturnType>
```

Erklärung

Hiermit versichere ich eidesstattlich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe. Die Arbeit wurde von mir in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Sebastian Scharinger
Passau, 12th June, 2009