ADONIS – A Case Study of a Legal Advisory System Using Adaptive Programming

by
Sven Bürgel
sven.buergel@informatik.tu-chemnitz.de

Advisor: Prof. Dr.-Ing. Petr Kroha

May 30, 2002
Abstract

Software evolution and maintenance have received great attention with the steadily increasing complexity of software systems. One recent approach in this field is adaptive programming, which focuses on the evolution of large class hierarchies. Its main objectives are to manage the change in evolutionary systems and to keep costs for adaptive maintenance low. In this thesis we present our experiences with the application of adaptive programming for modeling and implementing the legal advisory system ADONIS. Unlike most other information systems, ADONIS does not simply process data but regulations. Since regulations and laws are frequently a subject to change, we have chosen this domain as the basis for our practical research on adaptive programming.

Keywords: software engineering, object-oriented modeling, adaptive programming, legal advisory system
# Contents

Abbreviations  x

Acknowledgements  xi

1 Introduction  1
  1.1 The Project  1
  1.2 Organization  2
  1.3 Related Work  2

2 Adaptive Programming  4
  2.1 Main Principles  4
    2.1.1 Two-Dimensional Separation of Concerns  4
    2.1.2 Law of Demeter  5
    2.1.3 Visitor-Style Programming with Propagation Patterns  6
    2.1.4 Structure-Shy Programming  6
    2.1.5 Back to Adaptiveness  7
  2.2 Class Dictionaries  7
    2.2.1 Components  8
      2.2.1.1 Classes  8
      2.2.1.2 Relations  9
    2.2.2 Notations  10
      2.2.2.1 Textual Form  10
      2.2.2.2 Graphical Form  11
      2.2.2.3 UML Mapping  12
2.2.2.4 Example .......................... 12
2.2.3 Application-Specific Languages .......................... 14
2.3 Traversal Strategies .................................. 15
  2.3.1 Background .................................. 15
  2.3.2 Approach .................................. 16
  2.3.3 Formal Definition .................................. 16
    2.3.3.1 Class Graph .................................. 17
    2.3.3.2 Traversal Strategy and Traversal Strategy Graph 18
    2.3.3.3 Traversal Graph .................................. 20
  2.3.4 Traversing Object Graphs .................................. 21
    2.3.4.1 Example .................................. 23
  2.3.5 Notations .................................. 24
    2.3.5.1 Strategy-Graph Notation .................................. 24
    2.3.5.2 Strategy-Line Notation .................................. 25
    2.3.5.3 Compound Traversal Strategies .................................. 26
2.4 Adaptive Visitors .................................. 28
  2.4.1 The Visitor Design Pattern .................................. 28
  2.4.2 Classification of Wrappers .................................. 28
    2.4.2.1 Vertex and Edge Wrappers .................................. 29
    2.4.2.2 Prefix, Postfix and Around Wrappers .................................. 29
  2.4.3 Notation .................................. 30
3 Formalization of Legal Texts .......................... 32
  3.1 Used Terms .................................. 32
  3.2 Artifacts of a Formal Representation .................................. 34
    3.2.1 Class Diagrams .................................. 34
    3.2.2 Object Diagrams .................................. 35
    3.2.3 State Diagrams .................................. 35
      3.2.3.1 Events .................................. 36
      3.2.3.2 Guards .................................. 37
      3.2.3.3 Actions .................................. 37
    3.2.4 Object Constraint Language .................................. 37
CONTENTS

3.2.4.1 Invariant Conditions ........................................ 38
3.2.4.2 Pre- and Postconditions .................................... 38
3.2.4.3 Constraints on Collections ................................... 38
3.2.4.4 Adaptive OCL .................................................... 39
3.2.5 Sequence Diagrams ............................................... 39
3.2.5.1 Sequence Diagrams for Adaptive Operations .......... 40
3.3 From Legal Text to Formal Model ................................. 42
  3.3.1 Capturing Static Concepts ................................. 42
    3.3.1.1 Identifying Concepts .................................. 43
    3.3.1.2 Setting up Relationships ............................... 43
  3.3.2 Capturing the Dynamics of Processes ....................... 44
    3.3.2.1 Identifying and Assigning Processes ................ 44
    3.3.2.2 Deriving Transitions .................................. 45
    3.3.3 Constraining Properties .................................. 49
      3.3.3.1 Modeling Universal Conditions ..................... 49
      3.3.3.2 Modeling Legal Norms ................................. 50
3.4 Open Problems .................................................... 51
  3.4.1 Scoping ......................................................... 51
  3.4.2 Fuzzy Definitions ............................................. 52
  3.4.3 Incomplete or Ambiguous Clauses ........................... 52
  3.4.4 Legal Norms with Organizational Character ............... 52

4 System Architecture ................................................. 54
  4.1 The Idea ............................................................. 54
    4.1.1 Query and Result Notation ................................ 55
    4.1.2 Query Processing ........................................... 55
    4.1.3 Concept of Usage ............................................ 56
  4.2 Architectural Overview .......................................... 57
  4.3 A Look at the User Interface .................................. 59
  4.4 From Formal Model to Source Code ............................. 62
    4.4.1 Class Diagrams ................................................ 62
    4.4.2 Object Diagrams .............................................. 63
    4.4.3 State Diagrams ............................................... 63
    4.4.4 Object Constraint Language ............................... 64
    4.4.5 Sequence Diagrams .......................................... 65
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A   Object Graph Description Language</td>
<td>92</td>
</tr>
<tr>
<td>A.1  Syntax</td>
<td>93</td>
</tr>
<tr>
<td>A.2  Semantics</td>
<td>93</td>
</tr>
<tr>
<td>A.3  Interpreter</td>
<td>94</td>
</tr>
<tr>
<td>B   A Finite State Machine Implementation</td>
<td>95</td>
</tr>
<tr>
<td>B.1  Shortcomings</td>
<td>96</td>
</tr>
<tr>
<td>B.2  Notation Syntax</td>
<td>97</td>
</tr>
<tr>
<td>C   Specification and Source Code</td>
<td>99</td>
</tr>
<tr>
<td>Bibliography</td>
<td>100</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Textual class dictionary notation</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Class dictionary graph</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>UML class diagram</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>An arbitrary class graph</td>
<td>18</td>
</tr>
<tr>
<td>2.5</td>
<td>A traversal strategy</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>The computed traversal graph</td>
<td>21</td>
</tr>
<tr>
<td>2.7</td>
<td>An object graph corresponding to Figure 2.4</td>
<td>23</td>
</tr>
<tr>
<td>2.8</td>
<td>The traversal strategy from Figure 2.5 in strategy-graph notation</td>
<td>27</td>
</tr>
<tr>
<td>2.9</td>
<td>The traversal strategy from Figure 2.5 in strategy-line notation</td>
<td>27</td>
</tr>
<tr>
<td>2.10</td>
<td>Figure 2.5 as compound traversal strategy</td>
<td>27</td>
</tr>
<tr>
<td>2.11</td>
<td>The object traversal from Figure 2.7 with attached wrappers</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>Class diagram of an academic domain</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>State diagram of an examination process assigned to the Exam class</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>Sequence diagram of the <code>calculateFinalGrade</code> operation</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>Sequence diagram with superimposed TSG of the adaptive <code>calculateFinalGrade</code> operation</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>The system architecture of ADONIS</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Client-server interaction in MVC/Model 2</td>
<td>60</td>
</tr>
<tr>
<td>4.3</td>
<td>A user-system dialog in action</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Design pattern of an actor-initiated operation</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>Design pattern of an input-safe, actor-initiated operation</td>
<td>65</td>
</tr>
<tr>
<td>4.6</td>
<td>The adaptive <code>calculateFinalGrade</code> operation in DemeterJ</td>
<td>67</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Scenarios $B_1$ and $B_2$.</td>
<td>72</td>
</tr>
<tr>
<td>6.2</td>
<td>Scenario $C_1$.</td>
<td>72</td>
</tr>
<tr>
<td>6.3</td>
<td>Scenario $C_2$.</td>
<td>73</td>
</tr>
<tr>
<td>6.4</td>
<td>Cyclic CDG (a) and TSG (b).</td>
<td>79</td>
</tr>
<tr>
<td>6.5</td>
<td>Traversal graph of the CDG and TSG from Figure 6.4.</td>
<td>79</td>
</tr>
<tr>
<td>6.6</td>
<td>Correct TSG.</td>
<td>79</td>
</tr>
<tr>
<td>6.7</td>
<td>UML class diagram of the inlaws example.</td>
<td>80</td>
</tr>
<tr>
<td>6.8</td>
<td>Extended TSG for the inlaws problem.</td>
<td>81</td>
</tr>
<tr>
<td>6.9</td>
<td>Equivalent class graphs in UML (a) and CDG notation (b).</td>
<td>82</td>
</tr>
<tr>
<td>6.10</td>
<td>Object-equivalent semi-CDG (a) and flattened CDG (b).</td>
<td>83</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Relation-types and cardinality.</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Valid combinations of class- and relation-types.</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Mapping of class dictionary components to class dictionary graph symbols.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Traversal on the object graph from Figure 2.7 under guidance of the traversal graph from Figure 2.6.</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Classification of operations according to their initiating events.</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>A progressive user-system conversation.</td>
<td>58</td>
</tr>
<tr>
<td>5.1</td>
<td>The development platform of ADONIS.</td>
<td>69</td>
</tr>
<tr>
<td>6.1</td>
<td>Total number of classes in the legal model.</td>
<td>73</td>
</tr>
<tr>
<td>6.2</td>
<td>The results of each scenario.</td>
<td>75</td>
</tr>
<tr>
<td>6.3</td>
<td>The pros and cons of AP at a glance.</td>
<td>85</td>
</tr>
<tr>
<td>6.4</td>
<td>Code generation in DemeterJ.</td>
<td>86</td>
</tr>
</tbody>
</table>
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADONIS</td>
<td>Adaptive Object-Oriented Information System</td>
</tr>
<tr>
<td>AP</td>
<td>Adaptive Programming (or Adaptive Object-Oriented Programming)</td>
</tr>
<tr>
<td>ASL</td>
<td>Application-Specific Language</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer-Aided Software Engineering</td>
</tr>
<tr>
<td>CDG</td>
<td>Class Dictionary Graph</td>
</tr>
<tr>
<td>COOL</td>
<td>Coordination Language</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>JSP</td>
<td>JavaServer Pages</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OGDL</td>
<td>Object Graph Description Language</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>PO</td>
<td>Prüfungsordnung (examination regulations)</td>
</tr>
<tr>
<td>RIDL</td>
<td>Remote Invocation Description Language</td>
</tr>
<tr>
<td>SO</td>
<td>Studienordnung (study regulations)</td>
</tr>
<tr>
<td>TSG</td>
<td>Traversal Strategy Graph</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Acknowledgements

Many people have contributed to this thesis in some way. All of them I owe my deepest gratitude, but some of them I would like to acknowledge explicitly.

First of all I would like to thank my mentor Petr Kroha for his valuable feedback and guidance throughout the entire project. During the last year we have been sharing our knowledge in numerous discussions. I am convinced that this fruitful cooperation has improved the quality of this thesis to a great degree. Moreover, my research on this project was partially funded on his behalf.

Adaptive programming was a completely new field to me. I am grateful to Karl Lieberherr and his team, not only for inventing that amazing technology but also for providing an extensive range of publicly available information on their website. For me it has never been so easy to look for the information I wanted.

Special thanks to Gerhard Navratil who was so kind to send me an early draft of his upcoming PhD thesis. His findings gave me a first orientation and many useful insights in the issues related to the formalization of legal texts.

Writing in a non-native language can be a fairly challenging task. In this regard, I am much obliged to my friends Ines Brunner, Luo Wei, Jan Parthey, and Torsten Thau who have proofread and reviewed my written work. I owe you a big one. Last but not least, my family and girlfriend deserve my full appreciation for their never-ending support and enthusiasm during my years at university.

— Sven Bürgel —
Chapter 1

Introduction

Software has to meet a steady stream of changing requirements. New features are requested, old requirements are updated or dropped completely. Combined with the ever increasing complexity of software, the evolution and maintenance of software has become an important issue in Software Engineering.

Numerous studies show that the costs for software maintenance exceed half of the total budget. Hence, there is a need for new technologies and methodologies that improve the evolvability and maintainability of software in order to reduce development and maintenance costs. In this thesis we focus on one specific approach to software evolution termed adaptive programming (AP) [1]. AP concentrates on the evolution of the static model, i.e. the class graph. By separating the dynamic aspects from the static model, AP tries to minimize the negative effects of structural changes on the system behavior.

1.1 The Project

This thesis aims at the evaluation of AP from a practical perspective. Several theoretical papers about the concepts and advantages of AP have been published since its emergence in the beginning 1990s. The project website [2] also mentions a few success stories in commercial projects. However, we have not found case studies dealing with the application of AP in information systems. This shall be addressed by our contribution.

Next to AP itself, we will introduce the legal advisory system ADONIS in this thesis. ADONIS shall support students of our department with respect to the agenda of academic regulations and laws – especially with the examination regulations [3] and study regulations [4]. Regulations are frequently subject to changes which usually have political or administrative reasons. Consequently, a system that models legal norms 'suffers' from changing requirements in particular. We believe that the development and later maintenance of ADONIS
is representative to gain valuable insights into the evolutionary matters of AP. Hence, with the aid of ADONIS we should be able to give evidence on our central hypothesis, which states:

Adaptive programming improves the evolvability and maintainability of information systems.

ADONIS shall be implemented in DemeterJ [5]. DemeterJ is both a programming language and a development tool. We chose DemeterJ as target platform since it currently is the most advanced development platform for AP.

1.2 Organization

The rest of this thesis is organized as follows. We start with a general introduction of the concepts behind AP in Chapter 2 as we think a thorough understanding of AP is advantageous for the remainder of this thesis. Chapter 3 presents an original approach to the formalization of legal texts into an UML-based notation enriched by AP-related enhancements. After outlining the system architecture of ADONIS in Chapter 4, we give an overview of implementation-specific issues in Chapter 5. Our experiences with AP are discussed in Chapter 6. In this chapter we also evaluate AP with respect to the initially stated maintenance problem. Finally, we draw our conclusions in Chapter 7.

1.3 Related Work

Legal advisory systems have been in use for a while. Most of them, though, have not left the academic environment. The introduction of web-based legal information systems might raise user acceptance soon. In [6] the knowledge-based system WebShell is presented, which brings legal expertise to the World Wide Web. The way WebShell represents procedural knowledge, such as legal norms, is of particular interest for us. Contrary to ordinary rule-based system shells, WebShell uses a special form of decision trees called sequenced transition networks for that purpose.

The work of Navratil [7] only deals with the formalization of law. The author introduces a semi-automatic method that successively translates paragraphs into an algebraic model. Even though our system is based on the object-oriented paradigm, we find his work useful because, interestingly, an algebra and a class (in object-oriented terms) share some basic properties. Therefore, we are able to apply Navratil’s formalization technique to the construction of object-oriented legal models to a certain degree.
The gap between Knowledge Engineering and Software Engineering is addressed by Knublauch/Rose [8] and Cranefield/Purvis [9]. They investigate how conceptual models (ontologies) of knowledge-based systems can be represented in UML. Typically, ontologies are represented by specialized, high-level formalisms, which are only understood within the Artificial Intelligence community. Ontologies in UML are open to a much broader audience. They are well supported by CASE tools and integrate better into the overall software architecture. Both Knublauch and Cranefield base their efforts on a subset of UML chiefly comprising class diagrams, object diagrams, and OCL. Our formalism for legal models partially resembles ideas of their research.

Extracts from this thesis can also be studied in two other papers. In [10] we share our experiences with the application of AP for legal advisory systems. [11] focuses on the formalization of regulations into an adapted UML notation.
Chapter 2

Adaptive Programming

Adaptive programming (AP) [1] is a novel programming style that has been invented and developed by the Demeter Research Group at the Northeastern University in Boston. It provides a high-level interface to conventional object-oriented programming (OOP) aiming at the production of better evolvable and maintainable code.

In this chapter we will explain the basic characteristics of AP. Even though AP is a methodology rather than a concrete technology, we will also cover details from an application-dependent view. Accordingly, the terms and definitions stated in this chapter refer to both the AP methodology and the DemeterJ tools.\footnote{Even though the principles of AP have remained the same over the past years, the terminology has changed. Lieberherr’s book [1] illustrates the theoretical foundations of AP in terms of Demeter/C++. Since then the main effort of development has been put into DemeterJ. We will therefore introduce the more recent concepts of DemeterJ here.}

2.1 Main Principles

The term ‘adaptive’ already suggests that AP leads to software which is able to adjust to new contexts somehow. But how is that adaptiveness property achieved?

2.1.1 Two-Dimensional Separation of Concerns

A fundamental principle in software engineering is modularity [12]. Modularity decomposes complex problems into manageable modules. In well decomposed systems the modules show low coupling, i.e. interdependencies between modules
chapter 2. adaptive programming

are minimized, and high cohesion, i.e. a module’s responsibilities are highly-
related.

Modularity is a specialization of the separation of concerns principle [13] in that it separates software into components according to functionality and responsibility. Separation of concerns provides many benefits for software engineering, but it is not sufficient in regard to the modern requirements of software evolution and maintenance. The main critique is that the decomposition and composition mechanisms of current programming paradigms only support a single, dominant dimension of separation. For example, OOP organizes concerns into classes. A class encapsulates data and algorithms on these data. However, the functionality of complex operations, which involve the collaboration of several classes, cannot be cleanly encapsulated into just another concern. Instead operations are scattered throughout the collaborating classes, which complicates later evolution and maintenance noticeably. It would be beneficial if we could also separate different types or dimensions of concerns.

AP aligns the class graph and behavior of object-oriented software along two dimensions. It decomposes behavior from underlying class graphs. So, modifications can be made to either dimension without having negative side-effects on the other. However, the two dimensions of AP are not orthogonal, i.e. they overlap and hence are not completely separated. There must be some dependency between them because algorithms are always bound to data structures. AP ensures that these interdependencies are kept at a minimum. Generally speaking, AP implements ideas of multi-dimensional separation of concerns [14].

2.1.2 Law of Demeter

The Law of Demeter [15] is a generally accepted design-style rule. It was found by Holland during research on the Demeter project in 1987. The law states:

Each unit should have only limited knowledge about other units:
only units ‘closely’ related to the current unit.

In other words, the law promotes the principle of least knowledge, i.e. units shall not make assumptions about the whole environment but only about their immediate neighborhood. Because the law is formulated kind of vague, it can be applied to several contexts. With respect to object-oriented systems the law could be rewritten as follows.

An operation $O$ of class $C$ should only send messages to the following classes:

- the classes of the immediate attributes of $C$,
- the classes of the arguments of $O$ (including $C$ itself), and
- the classes of objects created by $O$.  

The idea is to avoid chains of method invocations, such as \( C.O_1.O_2.O_3 \), in the program code. In this example the class \( C \) would make unnecessary assumptions about the class graph. For instance, \( C \) assumes that the class returned by the operation \( O_1 \) declares an operation \( O_2 \). The code may break as soon as \( O_1 \) returns another class. Instead, the Law of Demeter demands \( O_1 \) to propagate the message to \( O_2 \), which will propagate it further to \( O_3 \).

As a side-effect the law adds a noticeable time and space overhead because it urges the programmer to write a lot of small wrapper methods that successively propagate messages to distant-related objects. However, in turn, the law improves adaptiveness.

### 2.1.3 Visitor-Style Programming with Propagation Patterns

In typical object-oriented code a high percentage of code fragments is only concerned with passing messages or transporting objects along a chain of objects. All these trivial wrapper methods share common behavior, which may be abstracted out if we find a way to specify those object chains. Traversal strategies [16] exactly accomplish this task.

All in all traversal strategies succinctly specify the collaborating classes of operations. Strictly speaking, they define traversals on class graphs. A traversal only relies upon a certain subgraph of the class graph, i.e. the subgraph holding all collaborating classes. We denote that subgraph as traversal graph. The original aspect of traversal strategies is that they do not enumerate every single collaborating class of an operation or traversal. As a matter of fact, a traversal strategy only sets some minimal constraints that a concrete class graph must meet so that a traversal graph can be constructed.

Although traversal strategies on their own are used to generate the traversal code, they cannot free us from writing non-trivial operation code, i.e. the actual behavior. This task is realized by so called adaptive visitors, which encapsulate the functionality of entire operations.

An adaptive operation typically consists of one or more adaptive visitors that are attached to precisely one traversal strategy. The combination of a traversal strategy and an adaptive visitor is also known as propagation pattern.

### 2.1.4 Structure-Shy Programming

Actually, all previously mentioned principles of AP operate under the term structure-shy programming.

The restrictions imposed by the Law of Demeter avoid information overload of methods. By applying this law, methods are only concerned about their local environment but not about the class graph as a whole.
While the decomposition of structure and behavior does not directly contribute to structure-shyness, it is required by AP’s approach to visitor-style programming. Traversal strategies take advantage of a separated class graph. By specifying traversals in a succinct way, traversal strategies support the structure-shyness property.

The same applies to adaptive visitors. Visitors carry out operations without changing the classes they work on. Generally speaking, the functionality of a complex operation is outsourced from the collaborating classes to a visitor. Adaptive visitors hence support the localization of functionality.

2.1.5 Back to Adaptiveness

The key to AP’s adaptiveness under evolving environments lies in structure-shy programming. An application, which has been developed with those principles in mind, is able to adapt to new class graphs more easily than another, more structurally dependent application.

In general, an AP-enabled software is compatible to a whole family of class graphs. That means, an adaptive application runs on different class graphs without modifications as long as each class graph belongs to the family.

The number of compatible class graphs mainly depends on the structural constraints given by all propagation patterns. Structure-shy applications impose less structural constraints. So, the better the structure-shyness of an application is, the larger is the family of compatible class graphs. Consequently, the adaptiveness property of an application is equivalent to the number of compatible class graphs.

2.2 Class Dictionaries

AP separates behavior from structure. Class dictionaries represent the structural concern of adaptive software, i.e. they specify class graphs. A class dictionary consists of a set of class definitions describing application classes and their relationships in an own notation.

Class dictionaries are also referred to as ‘customizers’ since they set the context which an adaptive program runs in. Whether a class dictionary is compatible with an adaptive program depends on the given structural constraints. An adaptive program and a class dictionary can be compiled to create an instance of the adaptive program if and only if those constraints are fully met.

\footnote{It should be mentioned that the application of traversal strategies is not necessarily bound to AP and separated class structures. Generally they might be used for computation tasks on labeled graphs.}
CHAPTER 2. ADAPTIVE PROGRAMMING

2.2.1 Components

Class dictionaries show similarities to data dictionaries. Both concepts model data structures. In AP the data structures are object-oriented. Thus, class dictionaries model classes and relationships, which are termed ‘relations’ in AP jargon.

2.2.1.1 Classes

We first list the three major class-types.

- *Construction classes* are defined as concrete classes. Concrete means they must not have any subclasses, i.e. they correspond to leaves in the inheritance hierarchy.

- *Alternation classes* act as placeholders for construction classes. In object-oriented terminology alternation classes would be called abstract superclasses. An alternation class must have at least one subclass of construction or alternation type.

- A *repetition class* represents a collection class. It contains a number of class instances. In DemeterJ repetition classes are implemented by simple, single-linked lists.

Next to the mentioned class-types, there are also two specializations.

- *Primitive classes* are a special case of construction classes. In DemeterJ they denote built-in terminal classes that represent primitive data types. Currently *Ident*, *String*, *Text*, *Integer*, and *Double* are supported as primitives.\(^3\)

- *Parameterized classes* are generic classes that will be instantiated by class arguments.\(^4\) All three major class-types can be parameterized independently of whether the underlying object-oriented programming language supports generic classes.

Strictly speaking, repetition and parameterized classes were just introduced for the sake of convenience. They can be substituted by construction and alternation classes.

Classes from third-party packages may also be imported to class dictionaries. Such classes are treated as terminal classes. This has implications on their use within traversal strategies. Terminal classes – whether primitive classes or imported classes is irrelevant – cannot be included in traversal strategies. Accordingly, the objects of terminal classes cannot be addressed by traversals.

---

\(^{3}\)These five primitives refer to the DemeterJ implementation. Other Demeter implementations may use different primitive classes.

\(^{4}\)Compare to templates in C++.
2.2.1.2 Relations

Class dictionaries support uni-directional relationships only. Altogether, there are five relation-types known, all of which correspond to the originating class-type.

- **Construction relations** reference the parts\(^5\) of construction and alternation classes. A construction class may reference many parts where each part corresponds to a certain class. In case of alternation classes, construction relations model the common parts, i.e. parts that are shared among all subclasses.

- **Alternation relations** express inheritance. Class dictionaries define inheritance in top-down fashion, i.e. the superclass enumerates all of its subclasses. This contradicts the most OOP languages, which usually prefer opposite is-a relationships.

- **A repetition relation** refers to the outgoing relation of repetition classes. A repetition class must have one repetition relation. The reason for this restriction is that a collection – as expressed by a repetition class – may only contain elements of one class and their subclasses.

Construction and repetition relations can also be made optional. Optional construction relations allow parts to contain null-values, and optional repetition relations permit collections to be empty. The cardinality of relationships is not so well developed in AP. Table 2.1 displays each relation-type with respect to cardinality. Many-to-many relationships are not supported directly, but they can be modeled by two reversed one-to-many relationships.

The semantics of the different class- and relation-types only permits certain valid combinations of classes and relations. Table 2.2 gives a summary on which class-types may be allowed as the ends of a relation-type.

\(^5\)Compare to ‘attributes’ in object-oriented terminology.

<table>
<thead>
<tr>
<th>Relation-Type</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>construction</td>
<td>1 : 1</td>
</tr>
<tr>
<td>optional construction</td>
<td>1 : 0..1</td>
</tr>
<tr>
<td>alternation</td>
<td>n/a</td>
</tr>
<tr>
<td>repetition</td>
<td>1 : 1..n</td>
</tr>
<tr>
<td>optional repetition</td>
<td>1 : 0..n</td>
</tr>
</tbody>
</table>

Table 2.1: Relation-types and cardinality.
2.2.2 Notations

The Demeter Research Group has developed two notations for the specification of class graphs. On the one hand, there is a textual way in form of class dictionaries. On the other hand, there are class dictionary graphs (CDGs), which represent class structures graphically. Both approaches share the same components, as introduced in Section 2.2.1. However, their applicability depends on the desired purpose.

CDGs are more intuitive and easier to understand by humans than class dictionaries because illustrations of complex models are usually perceived faster by the human brain. Nevertheless, they do not make the textual class dictionary notation obsolete. Class dictionaries are more expressive than their graphical counterparts.\(^6\) Besides that, they can be input into AP compilers directly.\(^7\)

2.2.2.1 Textual Form

In the following we define the grammar of class dictionaries in Extended Backus-Naur Form (EBNF). For the sake of compactness we focus on the basic components from Section 2.2.1 and hide any extended features for now.

A class dictionary comprises construction, alternation, and repetition class definitions. They can be mixed in any order.

\(^6\)See also Section 2.2.3.
\(^7\)This is not to say that class dictionary graphs cannot be processed by a computer. Visual development tools, such as AP Studio from the DemeterJ suite, may close that gap in future. However, the effort needed in order to process graphical data will always be higher.
CHAPTER 2. ADAPTIVE PROGRAMMING

\[
\langle \text{ClassDictionary} \rangle ::= ( \langle \text{ConstructionClassDef} \rangle \\
| \langle \text{AlternationClassDef} \rangle \\
| \langle \text{RepetitionClassDef} \rangle \\
)*
\]

Construction classes are represented by the equal sign. They may contain parts (attributes), which reference other classes by construction relations. Optional parts are enclosed in brackets. Every part must at least specify the class it references. Additionally, a part name may be given. If it is not defined, the part name will be derived from the class name by lowering its first letter.

\[
\langle \text{ConstructionClassDef} \rangle ::= \langle \text{ClassName} \rangle \"=\" \langle \text{Parts} \rangle \".\"
\langle \text{Parts} \rangle ::= \{ \langle \text{Part} \rangle | \"[\" \langle \text{Part} \rangle \"]\" \}* \\
\langle \text{Part} \rangle ::= \[ \"<\" \langle \text{PartName} \rangle \">\" \] \langle \text{ClassName} \rangle
\]

A colon followed by a list of subclasses distinguishes alternation classes. Common parts may be defined after the \textit{common} keyword.

\[
\langle \text{AlternationClassDef} \rangle ::= \langle \text{ClassName} \rangle \":\" \langle \text{SubClasses} \rangle \\
\[ \text{\"common\"} \langle \text{Parts} \rangle \] \".\"
\langle \text{SubClasses} \rangle ::= \langle \text{ClassName} \rangle \{ \"\" | \langle \text{ClassName} \rangle \}*
\]

Repetition classes are declared by the tilde sign. The collected class must appear once or twice depending on whether the collection can be empty. If the collected class is given in curly brackets only, then the collection may contain no elements, i.e. it refers to an optional repetition relation.

\[
\langle \text{RepetitionClassDef} \rangle ::= \langle \text{ClassName} \rangle \"\--\" \langle \text{CollectedClass} \rangle \".\"
\langle \text{CollectedClass} \rangle ::= \[ \langle \text{ClassName} \rangle \] \{" | \langle \text{ClassName} \rangle \}"
\]

The \textit{ClassName} and \textit{PartName} terminals are unqualified identifiers. Every class definition ends with a dot.

\subsection*{2.2.2.2 Graphical Form}

CDGs map class-types to vertices and relation-types to directed edges. The three class-types are expressed by different vertex symbols, and the relation-types are symbolized by different arrows. Labels can be assigned to vertices or construction edges to indicate class names or part names respectively. Table 2.3 summarizes all mappings.

Note that construction and repetition edges are both symbolized by plain arrows. Here the source vertex determines the type of the edge. If a plain arrow starts from a construction or an alternation vertex, it will be interpreted as a construction edge; otherwise it corresponds to a repetition edge. The same applies to optional construction and repetition edges.
### 2.2.2.3 UML Mapping

The class diagrams in UML have become very popular for the specification of class graphs. We will use class diagrams in later chapters. Therefore it is essential to introduce a mapping that relates UML class diagrams to CDGs. Such a mapping is proposed by Orleans et al. [17].

In that mapping construction classes are drawn like ordinary UML classes. Alteration classes differ from construction classes only in that they are stereotyped `abstract`. Repetition classes do not occur in class diagrams directly. Their semantics is represented by cardinalities.

Construction relations correspond to binary associations in UML. Their navigability is limited to one direction, though. The labels of construction relations, i.e. the part names, are expressed by role names. The cardinality is limited to 1 : 1 (construction relation) or 1 : 0..1 (optional construction relation). There is a special case with respect to construction relations that reference terminal classes. In that case the terminal class becomes an attribute of the referencing construction class.

Alternation relations are mapped to generalization relationships in UML. Contrary to alternation relations, generalizations always point to the superclass. So, special care should be taken here.

Repetition relations refer to binary associations with a cardinality of 1 : 1..n (repetition relation) or 1 : 0..n (optional repetition relation). In UML, collections can be visualized by using only one association in comparison to the two relations plus one repetition class in CDGs.

### 2.2.2.4 Example

Last but not least, we provide an exemplary class graph in the three introduced notations. See Figures 2.1, 2.2, and 2.3 for details.
University = <name> String <faculties> List(Faculty) 
  <uni_staff> List(AdministrativeStaff) .
Faculty = <name> String <head> String <fac_staff> List(Staff) .
Staff : TeachingStaff | AdministrativeStaff
  common <name> String <salary> Integer .
TeachingStaff = .
AdministrativeStaff = .
List(Element) - { Element } . // parameterized class

Figure 2.1: Textual class dictionary notation.

Figure 2.2: Class dictionary graph.

Figure 2.3: UML class diagram.
2.2.3 Application-Specific Languages

Class dictionaries are not exclusively used to describe class graphs. They can also be used to define languages for the representation of application objects. Such application-specific languages (ASLs) enable composite objects to be conveniently instantiated and printed from/to human-readable text.

Object representation in ASLs has one crucial advantage. Class graphs may evolve over time, but an ASL is robust against evolution as long as its grammar is not touched. That means, object descriptions will remain valid even if the class structure changes.

Under normal circumstances the class graph and the grammar of an ASL have to be maintained separately. In object-oriented programs we would specify all the classes and, furthermore, write a parser for the ASL. The parser would have to be updated manually everytime the class graph changes. AP combines both tasks into one place. It works as follows.

1. Each class definition of a class dictionary is interpreted as a grammar rule. The left-hand side, i.e. the defined class name, represents a non-terminal symbol, whereas every part on the right-hand side references another non-terminal or terminal symbol.\(^8\)

2. All class definitions may be enriched by some ‘syntactical sugar’ in order to make the grammar unambiguous. This is accomplished by keywords, which are placed into the right-hand side. Keywords are enclosed in quotation marks.

3. During compilation the Demeter tools derive an LL(1) grammar description from the information given by 1 and 2. Moreover, a special PrintVisitor class is generated which will print textual object representations according to the grammar.

4. A parser generator, such as Yacc or JavaCC\(^9\), reads that grammar description and creates corresponding parser code. Ambiguous grammar elements are reported at this stage. Ambiguity is a minor problem since there are simple disambiguating rules for Demeter ASLs known [18, Section 2.5.3].

5. Finally, the operations for parsing and printing object representations are provided.

One limitation remains though. The ASLs defined in class dictionaries can only be used to instantiate tree-like object structures. Tree-like means that an object is never referenced by more than one object. This restriction is caused

\(^8\)It should be obvious that the terminal classes are considered as terminal symbols.

\(^9\)JavaCC builds recursive-descent parsers that support even LL(k) languages. DemeterJ has therefore enhanced the class dictionary notation by the `lookahead` keyword.
by the parsing process. The generated ASL parser transforms a textual object representation into an abstract syntax tree. Broadly speaking, that syntax tree is structurally equivalent to the object structure to be instantiated. Hence, the object structure must conform to a tree-like structure, too.

2.3 Traversal Strategies

2.3.1 Background

A complex operation normally involves a group of collaborating objects. Such an operation solves a specific task by visiting each of the collaborating objects in some predefined order. We call this navigation through an object (sub)graph traversal.

In OOP each object encapsulates its own data. The concept of encapsulation forbids direct access to object data; instead objects provide interfaces to access internal data. Encapsulation eases the evolution of software because the internal data representation of an object can be substituted by other implementations without affecting the interface. Nevertheless, every object also defines relationships to other objects. Those objects that participate in an operation must have some implicit knowledge about their adjacent objects. Simply speaking, an object must know the objects it shall interact with.

Object interaction is realized by sending messages in OOP. In the context of traversals a number of messages is sent from object to object. On the code level the entirety of message sendings describe implicit structural knowledge paths. In OOP such structural knowledge paths are hardwired into the program. There is a high probability that some of these paths break as soon as the class structure is changed. As a consequence, the developer must fix the broken structural knowledge paths manually. This fact, in return, weakens the concept of encapsulation and essentially contributes to the software maintenance problem.

The Law of Demeter tackles the problem of software evolution and maintenance. By restricting object interaction to ‘closely related’ objects, this law ensures a low coupling of classes, which is generally considered to be good style.10 On the other hand, the Law of Demeter leads to a high number of small methods that are only concerned with propagating messages and transporting objects within a traversal. A study by Wilde et al. [19] takes a look at three large object-oriented systems. It shows that more than half of the examined member functions and methods have fewer than two lines of C++ code. To a large extent these small methods only propagate messages along the object graph. So, even though the Law of Demeter decreases the coupling of classes, it cannot reduce the implicit structural knowledge held by those small traversal methods. This is what Lopes et al. [20] denote structural anomaly:

\[\text{A strong coupling of classes introduces many dependencies between classes. It should be avoided with respect to reuse and evolution.}\]
While each class has minimal coupling and works only on its private data, the implementation of the operation has been spread across many classes and implicitly assumes one particular class organization that hard-wires structural knowledge paths into the program.

Traversal strategies present a solution to the structural anomaly.

### 2.3.2 Approach

The idea behind traversal strategies [16] is to define object traversals by using a succinct graph description called *traversal strategy graph* (TSG). Informally, a TSG explicitly mentions the key ‘milestones’ of a traversal rather than a detailed ‘roadmap’. Traversal strategies do not commit to a concrete class graph. Hence, they do not suffer from the structural anomaly.

The binding of a TSG to a concrete class graph takes place at compile-time, e.g. in case of DemeterJ, or even later at runtime\(^\text{11}\). The result of that binding process is a detailed traversal graph which contains all collaborating classes and relations of a traversal.

However, even if we have a detailed traversal graph at hand, we need some algorithm that performs the actual traversal. A traversal graph holds classes and relations while a traversal navigates through an object graph. A traversal does not visit classes but objects. Simply speaking, a traversal graph guides traversals in depth-first fashion. Starting from an object, the traversal graph is used to decide which paths are taken or pruned. This pattern recursively repeats for every visited object. In DemeterJ, for instance, traversal graphs are used to generate traversal code. That code is included in traversal methods which carry out the desired object traversals at runtime.

### 2.3.3 Formal Definition

In this section we formally introduce traversal strategies. A sound understanding of the theory behind traversal strategies is not mandatory to follow the remainder of this thesis. But, a deeper look at traversal strategies may considerably contribute to a better understanding of how adaptiveness is achieved in AP.

A comprehensive, formal definition and implementation of traversal strategies can be found in [16]. For the sake of compactness, we only cover the most useful definitions and statements from that paper here.

\(^{11}\)Systems with reflection capabilities allow TSGs to be bound to concrete class structures at runtime. Generally, reflection enables an application to realize its meta-level class structure. This information can be used to dynamically construct traversals from TSGs. The DJ library [21] is an example for traversal construction at runtime.
2.3.3.1 Class Graph

Definition 2.1 A directed labeled class graph is a triple $G = (V, E, L)$ where $V$ is a set of nodes, $L$ is a set of labels, and $E \subseteq V \times L \times V$ is a set of edges. If $e = (u, l, v) = u \xrightarrow{l} v \in E$, then $u$ is the source of $e$, $l$ is the label of $e$, and $v$ is the target of $e$.

We first define the notion of a class graph. A class graph represents the concrete class structure of an application. The directed labeled class graph $G$ required by traversal strategies must fulfill several conditions:

- $V \subseteq \mathcal{C}$, where $\mathcal{C}$ is a finite set of either abstract or concrete class names. That means the nodes of $G$ are class names.
- $L \subseteq L[\mathcal{F}]$, where $L$ is a finite set of attribute names. Edges labeled by an attribute name are called reference edges, and edges labeled by $\phi$ are called subclass edges.
- For each $v \in V$, the labels of all reference edges outgoing from $v$ are distinct.
- For each $v \in V$ such that $v$ is concrete, $v \xrightarrow{\text{self}} v \in E$.
- The set of subclass edges is acyclic, i.e. a class cannot be an indirect superclass of itself.

Multiple inheritance is allowed in class graphs as long as there are no conflicting reference edges declared in the superclasses of a class. Class graphs must be simple by definition. That means, all subclass edges start from abstract classes, and all reference edges start from concrete classes. Furthermore all subclass edges are incoming into concrete classes. This restriction does not cause a loss of generality because every arbitrary class graph can be transformed into an object-equivalent simple class graph by the Simplify algorithm.

The class dictionary graphs from Section 2.2 comply with the class graphs required by traversal strategies although it does not seem obvious. Class graphs do not know the repetition classes of CDGs, and the construction relations (reference edges) as well as alternation relations (subclass edges) do not meet all restrictions imposed by simple class graphs. The former problem is actually not an issue since repetition classes are translated to a number of concrete classes. The latter issue can be solved by applying class-graph simplification.

Figure 2.4 sketches an arbitrary class graph. We will use this class graph as the basis for our further explanations on traversal strategies.

---

12 See also the Single Inheritance Condition in [16].
13 See [16, Appendix A] for class-graph simplification.
CHAPTER 2. ADAPTIVE PROGRAMMING

A
B
c
D
e

Figure 2.4: An arbitrary class graph.

C
C

Figure 2.5: A traversal strategy with \( s = A \) and \( t = E \). The TSG nodes are mapped onto class-graph nodes using the default name map. The following element predicates are defined: \( B(e_1)(C) = \text{false} \) and \( B(e_1)(x) = \text{true} \) for all elements \( x \neq C \), \( x \in V \cup E \); \( B(e_2)(x) = \text{true} \) for all \( x \); and \( B(e_3)(C \leftarrow E) = \text{true} \) and \( B(e_3)(x) = \text{false} \) for all \( x \neq C \). Note that the edge labels \( e_1 \), \( e_2 \), and \( e_3 \) are only inserted for explanation. They do not belong to the TSG.

2.3.3.2 Traversal Strategy and Traversal Strategy Graph

Definition 2.2 A traversal strategy \( S \) is a triple \( S = (S; s; t) \), where \( S = (C; D) \) is a directed unlabeled graph called TSG, where \( C \) is the set of TSG nodes and \( D \) is the set of TSG edges, and \( s, t \in C \) are the source and target of \( S \), respectively.

Informally, a traversal strategy is represented by a certain TSG and two characteristic TSG nodes representing the source and target of a traversal. A TSG comprises nodes and edges, which do not directly refer to paths in the class graph. Instead, TSG nodes are seen as ‘milestones’ on the traversal paths from the source to target node. A directed TSG edge connects two milestones. A traversal strategy is displayed in Figure 2.5.

Definition 2.3 Let \( S = (C, D) \) be a TSG and let \( G = (V, E, L) \) be a class graph. A name map for \( S \) and \( G \) is a function \( N : C \rightarrow V \).

Name maps draw the connection between TSGs and class graphs in that they map TSG nodes to class-graph nodes. Under a name map each TSG edge \( a \rightarrow b \) expands to the set of paths in the class graph starting at node \( N(a) \) and ending at node \( N(b) \). The default name map is the identity function that maps TSG nodes to class-graph nodes one-to-one, i.e. \( N(c) \equiv c \) for all \( c \in C \subseteq V \).
Definition 2.4 Given a class graph $G = (V, E, L)$, an element predicate for $G$ is a predicate over $V \cup E$. Given a TSG $S$, a function $B$ mapping each edge of $S$ to an element predicate for $G$ is called a constraint map for $S$ and $G$.

Until now, traversal strategies do not place any negative constraints on the nodes and edges of a class graph. A TSG edge generally represents all class-graph paths between its mapped ends. However, it may sometimes be useful to explicitly exclude certain class-graph nodes or edges from path expansion. This is accomplished by element predicates and constraint maps. A constraint map assigns an element predicate to each TSG edge. The predicate decides whether a certain class-graph path is included in the set of expanded class-graph paths.

A path expansion $p'$ of some TSG path $p$ can be obtained by inserting nodes between the nodes of $p$. In traversal strategies path expansion is used to fill up the missing concrete classes between milestones. The ordinary path expansion assumes all TSG edges to be unconstrained, i.e. the element predicate evaluates to true for each TSG edge. In the following we define the path expansion for constrained TSG paths.

Definition 2.5 Let $S = (C, D)$ be a TSG, let $G$ be a class graph, let $N$ be a name map for $S$ and $G$, and let $B$ be a constraint map for $S$ and $G$. Given a TSG path $p = \langle a_0, a_1, \ldots, a_n \rangle$ with $a_i \in C$, we say that a class-graph path $p'$ is a satisfying expansion of $p$ with respect to $B$ under $N$ if there exist paths $p_1, \ldots, p_n$ such that $p' = p_1 \cdot p_2 \cdots p_n$ and:

1. For all $1 \leq i \leq n$, Source$(p_i) = N(a_{i-1})$ and Target$(p_i) = N(a_i)$.
2. For all $1 \leq i \leq n$, the interior elements of $p_i$ satisfy the element predicate $B(a_{i-1} \rightarrow a_i)$.

In other words, a TSG path $p$ is first segmented into its path segments $p_1, \ldots, p_n$, where $p_i = \langle a_{i-1}, a_i \rangle$ for all $1 \leq i \leq n$. All path segments $p_i$ are applied to a name map (Condition 1). Under the name map each $p_i$ represents the set of class-graph paths that span from $N(a_{i-1})$ to $N(a_i)$. That set of class-graph paths can be restricted by applying a constraint map (Condition 2). Eventually, the expanded path $p'$ is obtained by concatenating all path segments $p_i$ in natural order. Under the name map the TSG path $p$ and the expanded path $p'$ share the same source and target nodes.

Definition 2.6 Let $S = (S, s, t)$ be a traversal strategy, let $G = (V, E, L)$ be a class graph, let $N$ be a name map for $S$ and $G$, and let $B$ be a constraint map for $S$ and $G$. Then $S[G, N, B]$ is the set of concrete paths defined by

$$S[G, N, B] = \{ X(p') \mid p' \in P_G(N(s), N(t)) \text{ and } \exists p \in P_S(s, t) \text{ such that } p' \text{ is a satisfying expansion of } p \}.$$
This definition states an ‘instantiated’ traversal strategy. By ‘instantiated’ we mean that it corresponds to some form of traversal code. A traversal strategy is instantiated by applying a class graph, a name map and a constraint map. Basically, it specifies a set of class-graph paths $P_G$ that start in the source node $N(s)$, end in the target node $N(t)$, and were satisfyingly expanded from the set of TSG paths $P_S$.

Strictly speaking, an instantiated traversal strategy contains only concrete paths. A concrete path is a sequence of alternating concrete class names and labels. The mapping $X$ of class-graph paths to concrete paths excludes all abstract classes and subclass edges. We call this mapping natural correspondence. The natural correspondence of a class-graph path $p$ is written $X(p)$. Similarly, the natural correspondence of a class-graph path set $P$ is defined as $X(P) = \{X(p) | p \in P\}$.

2.3.3.3 Traversal Graph

In Section 2.3.2 we state that the binding of a traversal strategy to a concrete class graph results in a detailed traversal graph. Up to now, we have only mentioned what instantiates a traversal strategy and how this is done. In this part we close the circle in that we sketch the relationship between a traversal graph and the instantiated traversal strategy from Definition 2.6.

Simply speaking, an object traversal is guided by a set of concrete paths. At each traversal step, the traversal visits a specific object. The set of concrete paths determines which other object will be visited at the next step. By Definition 2.6 an instantiated traversal strategy $\mathcal{S}[S, N, B]$ is such a set of concrete paths. Consequently, an instantiated traversal strategy may be used to guide a traversal directly.

In reality, traversal graphs are constructed in order to perform traversals. A traversal graph $\mathcal{TG}(S, G, N, B)$ can be efficiently constructed from a traversal strategy, a class graph, a name map, and a constraint map. The basic idea is to substitute each TSG edge with a copy of an appropriately pruned class graph. Here the main difference between an instantiated traversal strategy and a traversal graph becomes visible: the traversal graph may still contain abstract classes and subclass edges, while the instantiated traversal strategy expresses only concrete paths. Therefore, a traversal graph is closer related to a class graph than an instantiated traversal strategy.

It can be shown that a traversal graph $\mathcal{TG} = \mathcal{TG}(S, G, N, B)$ is equal to a

\textsuperscript{14}Objects in an object graph always refer to concrete, i.e. instantiable, classes in a corresponding class graph. Hence, this set can only comprise concrete paths. All paths with abstract classes or subclass edges must be excluded.

\textsuperscript{15}See the traversal graph computation algorithm (Algorithm 1) in [16]. That algorithm, applied to the traversal strategy from Figure 2.5 and the class graph from Figure 2.4, outputs a traversal graph as in Figure 2.6.
traversal strategy \( S[S, N, B] \) by applying the natural correspondence.\(^\text{16}\)

\[
X(P_{T_G(T_s, T_f)}) = S[G, N, B]
\]

Note that \( T_s \) is the start set, and \( T_f \) is the finish set. The traversal graph computation algorithm possibly duplicates the original source and target node while it replaces TSG edges with pruned copies of the class graph. Thus, \( T_s \) and \( T_f \) are related to the source and target of the traversal strategy.

This clarifies that traversal graphs can be used to guide object traversals.

### 2.3.4 Traversing Object Graphs

In Section 2.3.2 we broadly sketched AP’s approach to structure-shy object traversals. Having formally defined the terminology of traversal strategies in Section 2.3.3, we can now take a more detailed look at how a traversal strategy can be compiled into executable traversal code.

Inputting a traversal strategy \( S = (S, s, t) \), a simple class graph \( G \), a name map \( N \) for \( S \) and \( G \), and a constraint map \( B \) for \( S \) and \( G \), we have a two-fold compilation algorithm:

1. Construct a traversal graph \( T_G(S, G, N, B) \).

\(^{16}\)See Lemma 5.2 in [16].
2. Attach a traversal method definition to each class in $TG$ such that an object graph (compatible to $G$) can be traversed with respect to $TG$.

The traversal graph computation (Item 1) was roughly outlined in Section 2.3.3. The algorithm is given in detail in [16, Algorithm 1]. An exemplary traversal graph is shown in Figure 2.6. We therefore focus on the traversal method definition (Item 2) from [16, Algorithm 2] here.

In this algorithm, a traversal method is denoted by $\text{Traverse}(T)$, where $T$ is a set of tokens. A token corresponds to a node in the traversal graph. Intuitively, a token placed on a traversal-graph node $v$ may be interpreted as ‘the traversal made so far places self on $v$.’ So, a token set maintains the synchronization between the currently visited object self in the object graph and the corresponding node(s) in the traversal-graph. A traversal method $\text{Traverse}(T)$ that is guided by a traversal graph $TG$ is defined as:

1. Define a set of traversal-graph nodes $T'$ by
   
   $$
   T' \leftarrow \{v \mid \text{Class}(v) = \text{Class}(\text{self}) \text{ and } \exists u \in T \text{ such that } u = v \text{ or } u \xrightarrow{\sim} v \text{ is an edge in } TG\}.
   $$

   $T'$ ensures that the token set only includes the nodes of the current object’s (self) concrete class. Any tokens in $T$ that correspond to abstract classes are moved to their subclass nodes.

2. If $T' = \emptyset$, return.

3. Call $\text{self}.\text{visitBefore}()$.

4. Let $Q$ be the set of labels which appear both on edges outgoing from a node in $T'$ in $TG$ and on edges outgoing from self in the object graph. For each label $l \in Q$, let
   
   $$
   T_l = \{v \mid u \xrightarrow{l} v \in TG \text{ for some } u \in T'\}.
   $$

5. Call $\text{self}.l.\text{Traverse}(T_l)$ for all $l \in Q$, ordered by ‘$<$’, the ordering of the labels.

6. Call $\text{self}.\text{visitAfter}()$.

The algorithm first verifies whether the current object lies on a traversal graph path. If not, $T' = \emptyset$, the traversal returns (Step 2). Otherwise, the prefix wrapper of the current object is invoked (Step 3). After that, the subsequent token sets $T_l$ are computed for each label $l$ outgoing from both the current object and the nodes in $T'$ (Step 4). Simply speaking, a token set $T_l$ represents the tokens of an object that is referenced by the current object via the attribute
Figure 2.7: An object graph corresponding to the class graph from Figure 2.4. Each object, denoted by a small letter, is an instance of the corresponding class in capitals, e.g. the object \(a\) is an instance of class \(A\). The shaded subgraph indicates the set of collaborating objects (or object traversal) given by the traversal graph from Figure 2.6 and this object graph.

The next step (Step 5) carries on the traversal with all objects that are referenced by the current object via some label \(l\) from the set of valid labels \(Q\). On return, the postfix wrapper is executed (Step 6). The visitor method in Step 3 corresponds to a preorder traversal, and the one in Step 6 refers to a postorder traversal.

2.3.4.1 Example

In the following we observe the traversal-method algorithm. We assume the object graph from Figure 2.7 and the traversal graph from Figure 2.6. We further assume an alphabetically ascending ordering of the labels. Our traversal starts at the object \(a\). The complete traversal with intermediary results is listed in Table 2.4.

Interestingly, the tokens sets \(T\) and \(T'\) are equal at every recursion step in our example. This is due to the fact that the guiding traversal graph does not contain any abstract classes. Moreover, the traversal is pruned in the Steps 1.1.1 and 1.2. In both cases the current object is of class \(C\). An unconstrained traversal would proceed to two objects of class \(D\) and \(E\). But, the one and only traversal-graph edge from the current token set \(T' = \{C^2, C^3\}\) is \(C^3 \rightarrow E^3\). Consequently, the traversal proceeds to an object of class \(E\) only.

\[\text{A token set } T_l \text{ may also contain tokens representing abstract classes. Step 1 of the algorithm corrects this circumstance by moving such tokens down to the appropriate, concrete subclasses.}\]
### Table 2.4: Traversal on the object graph from Figure 2.7 under guidance of the traversal graph from Figure 2.6.

<table>
<thead>
<tr>
<th>Step</th>
<th>Object</th>
<th>Tokens</th>
<th>Labels</th>
<th>Next Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>self</td>
<td>$T, T'$</td>
<td>$Q$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>1.1</td>
<td>$a$</td>
<td>${A^1, A^2}$</td>
<td>${b, c}$</td>
<td>$T_b = {B^1, B^2}$, $T_c = {C^2, C^3}$</td>
</tr>
<tr>
<td>1.1.1</td>
<td>$b$</td>
<td>${B^1, B^2}$</td>
<td>${c, d}$</td>
<td>$T_c = {C^2, C^3}$, $T_d = {D^1}$</td>
</tr>
<tr>
<td>1.1.1.1</td>
<td>$e_1$</td>
<td>${C^2, C^3}$</td>
<td>${e}$</td>
<td>$T_e = {E^3}$</td>
</tr>
<tr>
<td>1.1.1.2</td>
<td>$e_2$</td>
<td>${E^3}$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1.1.2</td>
<td>$d_2$</td>
<td>${D^1}$</td>
<td>${e}$</td>
<td>$T_e = {E^1}$</td>
</tr>
<tr>
<td>1.1.2.1</td>
<td>$e_3$</td>
<td>${E^1}$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>$e_2$</td>
<td>${C^2, C^3}$</td>
<td>${e}$</td>
<td>$T_e = {E^3}$</td>
</tr>
<tr>
<td>1.2.1</td>
<td>$e_5$</td>
<td>${E^3}$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
</tbody>
</table>

### 2.3.5 Notations

Having defined traversal strategies, we introduce two notations for describing traversal strategies in this section. It should be noted that traversal strategies are universally applicable to object-oriented structures. Here we only take those notations into consideration which are used in DemeterJ.

#### 2.3.5.1 Strategy-Graph Notation

The strategy-graph notation expresses a TSG by listing all TSG edges together with optional negative constraints. More formal, an edge in a TSG $S = (C, D)$ is a triple $(a, b, \mathcal{B}(a \rightarrow b))$ in strategy-graph notation where $a, b \in C$ are the start and end nodes, and $\mathcal{B}(a \rightarrow b)$ is an element predicate.

An element predicate evaluates to $\text{true}$ by default. If a TSG edge needs to be constrained, we must specify a negative constraint. The following negative constraints are available in DemeterJ:

- **bypassing** includes all elements except the one(s) specified, e.g., ‘bypassing $A$’ excludes the node $A$, and ‘bypassing $\rightarrow *_1, *$’ excludes all construction edges with the label $l$.

- **only-through** excludes all elements except the one(s) specified, e.g., ‘only-through $\rightarrow A, 1, B$’ excludes all elements except the edge $A \xrightarrow{l} B$.

A traversal strategy can be described by the following simplified grammar.
A name map can be provided optionally. If it is not present, TSG node names will be mapped one-to-one to class-graph node names. Figure 2.8 shows the textual representation of the traversal strategy from Figure 2.5 in strategy-graph notation.

### 2.3.5.2 Strategy-Line Notation

Beside the strategy-graph notation there exist a more intuitive notation called strategy-line notation. Traversal strategies that are given in strategy-line notation model a TSG that has only one path from the source to the target node. This path may consist of several path segments, but all path segments are arranged in sequential order.\(^{18}\)

A grammatical description of the strategy-line notation is provided next.

```latex
PathDirective ::= <SourceDirective>
                ( <NegativeConstraint> )? 
                ( <PathSegment> )* 
                <TargetDirective>
                ( "with" <NameMap> )? 

SourceDirective ::= "from" <Name> 
TargetDirective ::= ( "to" | "to-stop" ) <Name> 
PathSegment ::= <PositiveConstraint>
                ( <NegativeConstraint> )? 
```

Here the source and target directives name the source and target node, respectively. The difference between to and to-stop target directives is that a to-stop directive prunes all outgoing edges of the target class \(N(t)\). Such a pruned class will terminate a traversal even on cyclic object-graphs, while a to directive may lead to infinite traversals in that case. Moreover, path segments are separated by positive constraints. A positive constraint follows the form through or via plus some TSG node or edge specification.\(^{19}\) Each path segment (including the one between the source and target node) can be constrained by negative constraints. We point again to the traversal strategy from Figure 2.5. The corresponding description in strategy-line notation is displayed in Figure 2.9.

---

\(^{18}\)A graph with only one path looks like a list or line. Therefore, we call this notation strategy-line notation.

\(^{19}\)There does not seem to be a semantical difference between through and via. From the English language's point of view, through would probably be recommended in conjunction with TSG edges and via with TSG nodes.
As said before, the strategy-line notation is able to describe one TSG path at a time. We are therefore not able to describe a structurally equivalent TSG in Figure 2.9. Instead we obtain two TSGs. So, it can be concluded that the strategy-line notation is less powerful than the strategy-graph notation. However, in many cases the strategy-line notation is sufficient to express desired traversals. Beside that, the textual representations of traversal strategies in strategy-line notation are usually shorter and more convenient than their counterparts in strategy-graph notation.

2.3.5.3 Compound Traversal Strategies

Sometimes it could be useful if we composed several traversal strategies. For instance, the strategy-line notation allows us to specify one TSG path only. What if we could merge several paths into one TSG? Compound traversal strategies exactly serve that purpose. A compound traversal strategy composes a number of traversal strategies into one traversal strategy by applying one of the following graph operations.

- The join operator concatenates two traversal strategies $S_1$ and $S_2$, such that

$$
\text{join}(S_1, S_2) = \begin{cases} 
(S_1 \cup S_2, s_1, t_2) & \text{if } t_1 = s_2 \\
\text{empty} & \text{otherwise}
\end{cases}
$$

- The merge operator unites two traversal strategies $S_1$ and $S_2$, such that

$$
\text{merge}(S_1, S_2) = \begin{cases} 
(S_1 \cup S_2, s_1 \cup s_2, t_1) & \text{if } t_1 = t_2 \\
(S_1 \cup S_2, s_1, t_1 \cup t_2) & \text{if } s_1 = s_2 \\
\text{empty} & \text{otherwise}
\end{cases}
$$

- The intersection operation defines a kind of ‘view’ onto a traversal strategy. Given two traversal strategies $S_1$ and $S_2$, $S_2$ can be used to place further constraints on $S_1$ without extending $S_1$.

$$
\text{intersect}(S_1, S_2) = S_1 \cap S_2
$$

Note that the merge operation extends traversal strategies in that merged traversal strategies may have multiple sources or targets. Multiple sources enable traversals to start in different source nodes, and multiple target are particularly useful if we want to reach targets that do not correspond to one class only.

An updated example of Figure 2.9, which now corresponds to the original TSG from Figure 2.5, can be found in Figure 2.10.
{ 
    A \rightarrow E \text{ bypassing } C \\
    A \rightarrow C \\
    C \rightarrow E \text{ only-through } \rightarrow C,e,E 
}

Figure 2.8: The traversal strategy from Figure 2.5 in strategy-graph notation.

\textbf{from A bypassing C to E} \\
\textbf{from A via C only-through } \rightarrow C,e,E \text{ to E}

Figure 2.9: The traversal strategy from Figure 2.5 in strategy-line notation. Due to the inadequacy of the strategy-line notation to describe more complex TSGs, like the one in Figure 2.5, we must actually come up with two traversal strategies.

\textbf{merge (} \\
\textbf{from A bypassing C to E ,} \\
\textbf{from A via C only-through } \rightarrow C,e,E \text{ to E} 
) 

Figure 2.10: Figure 2.5 as compound traversal strategy.
2.4 Adaptive Visitors

An adaptive visitor is a class that contains a set of visitor methods. Each visitor method represents a code fragment, also known as wrapper, of an operation. Visitor methods are executed in the order given by the traversal that leads the operation. The adaptive visitor is a mean to implement an operation whose behavior is spread across a set of collaborating classes. Here the behavior specification is decoupled from the collaborating classes.

2.4.1 The Visitor Design Pattern

The idea of adaptive visitors is rooted in Gamma’s et al. Visitor design pattern [22]. A visitor can perform an operation on a set of objects without changing their underlying classes. It works as follows.

1. The operation traverses a certain object subgraph containing all collaborating objects. The traversal itself is led by traversal methods.

2. Whenever an object is visited, the traversal method calls the visitor method that corresponds to the object’s class, supplying the object self as argument.

3. Now, as control is transferred to the visitor, the visitor steps into action and executes its code on the visited object.

4. After the execution of the visitor method is finished, control is transferred back to the traversal method, and the traversal proceeds to the next object (Step 2).

Although, the Visitor design pattern already isolates the task of traversal, it neither provides a mechanism to automate that task, nor does it make the traversal adaptive with respect to changes of the class graph. Here comes the adaptive visitor into play which, in combination with traversal strategies, addresses the latter two points. In AP, the traversal methods are automatically generated by traversal strategies, and the visitor methods are expressed by adaptive visitors.

2.4.2 Classification of Wrappers

We already stated that the visitor methods of an adaptive-visitor class are also called wrappers. The term ‘wrapper’ usually refers to something that encloses something else. With regard to the adaptive visitor pattern, a wrapper virtually encloses a vertex or edge of a traversal graph.
CHAPTER 2. ADAPTIVE PROGRAMMING

2.4.2.1 Vertex and Edge Wrappers

A wrapper can basically be attached to a vertex or to an edge of a traversal graph. Consequently, wrappers are classified as vertex or edge wrappers.\(^{20}\)

The difference between both types is that a vertex wrapper relates to a class and an edge wrapper corresponds to a class relationship. Strictly speaking, the code of a vertex wrapper is executed when an object of the wrapper’s assigned class is visited. An edge wrapper is invoked when a traversal uses the assigned relationship to move from one object to another, related object.

2.4.2.2 Prefix, Postfix and Around Wrappers

Furthermore, the traversal direction plays a role when classifying wrappers. Accordingly, a wrapper can be distinguished as prefix, postfix, and around wrapper. Let us imagine an object traversal as a tree where the root is the source object and the leaves are the target objects.\(^{21}\) A traversal descends the tree if it moves from the root to a leaf; otherwise it ascends the tree.

Adaptive visitors allow wrappers to be specified that take action depending on the direction of a traversal. We can define whether a wrapper shall be executed on a descending or an ascending traversal. That naturally corresponds to preorder or postorder traversals, respectively. A preorder traversal invokes some operation (wrapper) on a vertex before it proceeds to child vertices. Such wrappers are denoted prefix wrapper. In contrast, postorder traversals can be expressed by postfix wrappers. Prefix and postfix wrappers may also be applied to edges, as discussed in the previous section. Doing so, a prefix wrapper is processed when the assigned edge is traversed in descending order, while a postfix wrapper is executed as soon as the visitor comes back.

A special case are around-wrappers. Such a wrapper encloses the assigned vertex or edge completely, i.e. it basically is a combination of pre- and postfix wrapper. The difference is that an around-wrapper terminates the traversal branch by default. If we want to continue the traversal from within an around-wrapper, the subtraversal must be invoked explicitly.

What is the motivation behind around-wrappers? Sometimes it can be useful if a visitor method decides whether a traversal should be pruned, e.g. depending on the internal state of the visited object or the visitor itself. Around-wrappers accomplish such behavior. In addition, the combination of preorder and postorder code into one wrapper can be of advantage in that is allows local variables to be shared between both code fragments.

Figure 2.11 sketches the sample object traversal from Figure 2.7 with four attached wrappers. Assuming a traversal, which descends to the left-most child-vertex first, the execution order of the wrappers would be $W_2 \prec W_4 \prec W_3 \prec W_1$.\(^{22}\)

---

\(^{20}\)The original Visitor design pattern only knows visitor methods that are similar to the vertex wrappers of adaptive visitors.

\(^{21}\)Compare to the object traversal from Figure 2.7.
Figure 2.11: The object traversal from Figure 2.7 with attached wrappers. The single wrappers are defined as follows: \( W_1 = \text{after}(A \rightarrow C) \); \( W_2 = \text{before}(B) \); \( W_3 = \text{after}(B) \); and \( W_4 = \text{around}(D) \).

### 2.4.3 Notation

An adaptive visitor is a class enriched by some special methods. It may declare attributes and ordinary methods like every other class. In the context of an operation, the visitor attributes hold the state of that operation. That means, the attributes of a visitor are actually nothing but the local variables of an operation. Beside attributes and ordinary methods, the following special methods can be specified in an adaptive visitor.

- The **before** (prefix), *after* (postfix), and *around* wrappers attach behavior to certain vertices or edges in the object graph.
- The **start** method initializes the visitor's internal state before a traversal begins.
- The **finish** method cleans up the visitor's internal state after a traversal is finished.
- The return-value method, denoted by the **returns** keyword, returns the result of the whole operation. After a traversal is finished, the visitor should hold the result of the operation in its internal state.\(^{22}\) The return-value method makes this result accessible to the environment by providing a standardized interface.

The use of the special methods in adaptive visitors is optional. However, it makes no sense to specify a visitor without any wrapper, except the visitor inherits from another visitor. Last but not least, we give the syntax of the special methods mentioned before.

\[
\text{WrapperMethod ::= <WrapperKind> <HostSpec> <Code>}
\]

\[
\text{WrapperKind ::= "before" | "after" | "around"}
\]

\(^{22}\) Some operations do not compute any result; they only modify the visited objects. In that case a return-value method is not necessary.
HostSpec ::= <ClassSpec> | <EdgeSpec>
ClassSpec ::= <ClassName>
EdgeSpec ::= <PartEdge>
    | <SubclassEdge>
    | <SuperclassEdge>
PartEdge ::= "->" <SourceClassName>
        "", <PartName>
        "", <DestClassName>
SubclassEdge ::= "=>" <SourceClassName> "," <DestClassName>
SuperclassEdge ::= "=>" <SourceClassName> "," <DestClassName>
StartMethod ::= "start" <Code>
FinishMethod ::= "finish" <Code>
ReturnValueMethod ::= "returns" ( <ReturnType> )? <Code>

The non-terminal symbol Code refers to statements in the underlying object-oriented programming language, which have to be enclosed in double curly-brackets. Every code fragment is seen as a building block in AP, i.e. it is a sort of indivisible black box.
Chapter 3

Formalization of Legal Texts

Law is typically written down in natural language. We may understand a legal text as a kind of specification that is written by humans for humans. Computers play a little to no role in that context. However, as the complexity of legal systems grows further, intelligent computer applications may soon become necessary to support humans in legal matters.

Unfortunately, if we want computers to understand what a legal text means, we must first model its concepts and processes into some computer-readable representation. Computers are stupid by nature. Even with the latest techniques of Artificial Intelligence, it is impossible – or, at least, should be questioned – that a computer interprets the quite complex, natural language of legal texts in a satisfying manner. Consequently, human analysts\(^1\) have to transform the legal contents into formal, computer-readable models somehow.

In this chapter an approach to the formalization of law is outlined. We represent legal texts, e.g. statutes or regulations, by an object-oriented model using the Unified Modeling Language (UML) [23] with a few slight modifications.

3.1 Used Terms

This thesis includes computer-related vocabulary among law-related terms. Before going on, we should agree on some common terminology. The used terms are briefly defined in the following.

\(^1\)We may call them knowledge engineers.
**Legal Norm** A legal norm (or rule) constitutes what one is allowed to do and what not. Complete norms include a situation and a consequence. The consequence will be put into action if the situation occurs.

**Legal Text** The term ‘legal text’ refers to what lawyers call ‘primary sources of law’, i.e. statutes, regulations, and codes, as well as judgments on court cases and others. In the scope of this thesis we abstract that term a bit in that we understand a legal text as a document declaring a number of legal norms, which are usually structured into articles and paragraphs.

**Regulations** Regulations are composed of legal norms issued by an official authority. Regulations conduct the execution of law. For that reason, they must always reference the implemented law. Regulations often include penalties for violations.

Additionally we introduce some other terms that may easily lead to misunderstandings between computer scientists and lawyers.

**Concept** A concept is a real or abstract entity of a legal domain or text. In our object-oriented model a concept can be understood as an object or a class of similar objects.

**Process** A process is an activity aiming at the completion a certain task. Let us imagine a student who wants to pass an exam. A corresponding process could start with an enrollment procedure and finish with the exam being assessed.

**Procedure** A procedure is an atomic part of a process. Every procedure alters the state of the associated process. A process is successively completed by a chain of procedures.

**Operation** An operation is the computer representation of the rather law-related term ‘procedure’. An operation often involves the collaboration of related concepts.

**Method** A method is the smallest functional unit of an operation. It is always assigned to a concept.\footnote{These functional units are denoted ‘operations’ in UML. Although our formalism is based on UML, we prefer to term it differently because the term ‘operation’ is already used in AP’s terminology.}

\footnote{For example, the examination and study regulations [3, 4], this thesis deals with, implement the Saxon University and College Act (Sächsisches Hochschulgesetz, SHG) [24] and (SächsHG) [25].}
3.2 Artifacts of a Formal Representation

An informal specification, such as a legal text, is written in a natural language, such as English or German. A formal specification, on the other hand, consists of a few elements with precisely specified syntax and semantics. In general, a formalism might not be as expressive and powerful as a natural language. But formalisms avoid some drawbacks present in informal specifications, as there are: noise, silence, over-specification, contradictions, ambiguity, forward references, and wishful thinking [26] (as cited by [27]).

UML has become a de facto standard for the modeling of object-oriented systems. Accordingly, the majority of software and knowledge engineers are familiar with UML; and many CASE tools for object-oriented analysis and design have adopted UML nowadays. It was shown by Spit et al. [28] that the principles of AP can be integrated into nearly all object-oriented analysis and design methods. We have chosen an UML-based method. All in all, we employ five artifacts of UML to formalize legal texts. In the subsequent sections we will introduce each of them.

3.2.1 Class Diagrams

Class diagrams model the static structure of object-oriented systems. They visualize classes and their relationships in a graph-like manner. Therein, a node, i.e. a box with compartments for the class name, attributes, and methods, represents a class. Edges refer to relationships between classes. The most important relationships are generalizations, associations, and aggregations.

**Generalization** Generalization relationships express abstraction hierarchies. In such hierarchies the common properties of concepts are ‘outsourced’ to a more general concept. Figure 3.1 illustrates the two generalizations WrittenExam⇒Exam and OralExam⇒Exam. It means that written and oral exams are just two special cases of the concept Exam.

**Association** Associations are used to model communication paths between concepts. Such relationships are necessary for objects to interact with each other. Association edges in a class diagram may be labeled to express the semantics of a relationship. Furthermore, specific class roles and multiplicity constraints can be added to the association ends.

**Aggregation** Aggregations are a special case of associations with the semantics of one concept being part of another. Hence they are also referred to as part-of relationships. Figure 3.1 contains the aggregation Course⇒Exam indicating that exams are part of a course.
3.2.2 Object Diagrams

Object diagrams are closely related to class diagrams. In contrast to class diagrams, the nodes of an object diagram show objects and their attribute values. Accordingly, an object diagram may be seen as a detailed snapshot of the system’s state.

In the example from Figure 3.1 an object diagram could be used to list which courses are offered and which exams in particular are required by a course.

3.2.3 State Diagrams

State diagrams are a common mean to model finite state machines (FSMs). They describe discrete behavior, such as processes for instance.

A state diagram consists of states and transitions. It is a directed graph with nodes representing states and edges standing for transitions. At any time an FSM is in exactly one active state and waits for events to trigger state changes. Such changes are called transitions. A transition explicitly defines

- an event triggering the transition,
- a guard enabling or disabling the transition, and
- an action to be performed when the transition takes place.
Moreover, a transition implicitly specifies a source and target state. Having fired a transition, the FSM leaves the active state, which must equal to the source state, performs the action, and enters the target state. A state diagram may also have characteristic nodes corresponding to start or final states. A start state is the state an FSM is initialized with, and a final state marks the end of a process.

Transition specifications follow the format \texttt{event[guard]/action}, where guard and action are optional. Figure 3.2 presents some examples. In the following we take a closer look at the different parts of a transition.

### 3.2.3.1 Events

There are found kinds of events known in UML.

- \textit{Call events} will be sent if a request for a (synchronous) method invocation is received. They generally convey information about the method to be called, i.e. the method name and its arguments.

- \textit{Signal events} will be generated if a particular (asynchronous) signal is perceived. They include the signal that has been caught. A signal is typically represented by a symbolic name.
• Change events will be raised if attribute properties, i.e. attribute values and associations, of the model change. A change event explicitly specifies a boolean condition for the properties to be monitored. The change event triggers when this condition becomes true.

• Time events will be thrown if a specific deadline expires. A deadline can be specified by an absolute or relative time expression. In the latter case, the resulting deadline is scheduled by the time the source state became active plus the duration.

Additionally to these four event kinds we introduce the notion of instantaneous events.

• Instantaneous events are a special case of time events with a relative duration of zero. That means, an instantaneous transition will be triggered as soon as its source state is entered. We describe instantaneous transitions in the shortened form ‘[guard]/action’. The guard expression is mandatory for instantaneous transitions.

3.2.3.2 Guards

A guard is a boolean expression that enables or disables a transition. With the aid of guards we can restrict the initiation of a transition in a fine-grained manner. Whenever an event triggers, the guard decides whether the transition will be fired or rejected. A transition is blocked if the guard expression evaluates to false, otherwise it is opened.

3.2.3.3 Actions

Actions describe what happens when a transition is fired. For our purpose it is sufficient to assume that an action invokes a method or changes model attributes directly.

3.2.4 Object Constraint Language

We employ the Object Constraint Language (OCL)\textsuperscript{4} to express additional constraints in a model. OCL is a pure expression language, i.e. it cannot be used to program any logic or control flow. Unlike formal languages with a strong mathematical background, OCL was originally targeted at the modeling of business systems and, hence, can be easily understood by non-mathematicians, too. In OCL constraints are modeled by boolean expressions that are evaluated without any side effects to the model. A constraint holds if the corresponding expression returns true. Constraints can be invariant conditions, preconditions, and postconditions.

\textsuperscript{4}OCL is a part of UML.
3.2.4.1 Invariant Conditions

Invariant conditions (or invariants for short) represent model constraints that must be satisfied at any time. They are attached to a certain context, which is a class. We provide a simple example:

```
context Grade
inv: 1.0 <= self.value and self.value <= 5.0
```

This invariant states that the attribute `value` of the class `Grade` can be any real number ranging from one to five.

3.2.4.2 Pre- and Postconditions

Pre- and postconditions are connected with methods. They must only hold immediately before (preconditions) and after (postconditions) a method is invoked. Pre- and postconditions may not only include properties of the context class but also the arguments and return value of the context method.

```
context Exam::mark(aGrade:Grade)
pre: self.grade <> null
post: self.grade = aGrade and
      self.passed = (aGrade.value <= 4.0)
```

Here, the method `mark` of class `Exam` is constrained. The operation declares a single argument `aGrade` of type `Grade`. The precondition first checks whether this exam (`self`) has already been marked. The postcondition requires the `aGrade` argument to be assigned to the `grade` attribute after the method is completed. Moreover, the exam is `passed` if the grade's value is less than or equal to four. Note that the previously mentioned invariant on the class `Grade` already restricts grade values to be greater than or equal to one.

3.2.4.3 Constraints on Collections

OCL also offers support for collections such as sets, sequences, and bags. For that reason we can apply a condition to a whole collection of objects at once. For instance, the constraint ‘an academic course is passed if all of its exams are passed’ could be written by the following invariant.

```
context Course
inv: self.passed = self.exams->forAll(e|e.passed)
```

In this case `self.exams` refers to a collection containing the exams of a course. The `forAll` operation on that collection would evaluate to `true` if all exams `e` are passed.
3.2.4.4 Adaptive OCL

We use OCL with an adaptive extension proposed by Lieberherr [29]. He states that the navigation abilities of OCL are not structure-shy with regard to the Law of Demeter. Indeed, OCL allows to create expressions that violate the law, as shown in this example:

```plaintext
context Course::calculateFinalGrade():Grade
pre: self.passed
post: result.oclIsNew and
    result.value = self.exams.grade.value->sum / self.exams.grade->size
```

Instead traversal strategies can be applied to circumvent that structural knowledge is hardcoded into OCL expressions. After applying traversal strategies the given example looks as follows.

```plaintext
context Course::calculateFinalGrade():Grade
pre: self.passed
post: result.oclIsNew and
    result.value = self.{bypassing->*,finalGrade,* to Grade}.value->sum / 
                   self.{bypassing->*,finalGrade,* to Grade}->size
```

This approach neatly integrates into our project aim. If we were hardcoding structural knowledge paths into the model during analysis already, how could we make it adaptive in later development phases?

3.2.5 Sequence Diagrams

Sequence diagrams illustrate the interactions of operations in detail. An operation usually involves a number of collaborating objects. Those objects collaborate by exchanging messages. A sequence diagram displays all messages sent during an operation’s runtime in chronological order.

In a nutshell, a sequence diagram has a horizontal and vertical dimension. Usually the collaborating objects are arranged along the horizontal axis whereas the timeline follows the vertical axis. Arrows are drawn between object lifelines. A lifeline denotes the role of an object in a collaboration. An arrow is labeled with the message being sent. A message represents either a method call (synchronous communication) or a signal (asynchronous communication). The vertical order of arrows determines the chronological sequence of interactions in an operation. Additionally, the message sequence of an operation may be stressed by optionally numbered arrows. The activation period of an object, i.e. the duration in which an object performs some direct or indirect action, is represented by a tall, thin rectangle.
3.2.5.1 Sequence Diagrams for Adaptive Operations

Sequence diagrams model operations in a rigid manner. They enumerate all messages that are sent during an operation’s runtime. If we take into account that each message also specifies structural constraints, such as the sender and receiver object, we can conclude that sequence diagrams on their own require deep knowledge of the class graph although they are primarily meant for behavioral specification.

In this section we propose an extension of sequence diagrams in the light of AP’s traversal strategies from Section 2.3. Interestingly, both sequence diagrams and traversal strategies define navigation paths through class graphs. An important difference is that traversal strategies make less assumptions about the underlying class graph. Why should not we take advantage of the high profile of sequence diagrams and the adaptiveness of traversal strategies?

Our solution does not restrict the expressiveness of sequence diagrams in any way. Messages to ‘closely related’ classes like the ones described in the Law of Demeter are illustrated as usual. However, for adaptive operations we superimpose the corresponding traversal strategy graph (TSG) onto the sequence diagram. Strictly speaking, we replace communication paths in the original sequence diagram by TSG edges.

Figures 3.3 and 3.4 provide two views onto the same operation. The adaptive version in Figure 3.4 specifies a navigation path (Arrow 2) from Course to Grade according to a TSG with one edge Course $\rightarrow$ Grade and the path constraint `$bypassing \rightarrow \ast,finalGrade,\ast$'. That navigation path can safely substitute the messages displayed by Arrow 2 and 3 in Figure 3.3. As a result, objects that do not essentially contribute to the collaboration, such as the Exam objects, can be left out.

Our approach is also applicable in conjunction with multiple adaptive operations per sequence diagram. Nonetheless, it has some drawbacks that should not be hidden here.

- Traversal strategies do not specify a precedence among alternative traversal paths. If a traversal has two alternative traversal paths to follow from the currently visited object, it solely depends on the traversal algorithm which path is gone first. Sequence diagrams always give the order of message sends precisely. In other words, sequence diagrams predefine the traversal order more exactly.

- Traversal strategies that contain more than one TSG path from the source to the target, i.e. that contain TSG nodes $n$ with $\text{indegree}(n) > 1$ or $\text{outdegree}(n) > 1$, will cause problems if mapped onto sequence diagrams. In that case the incoming or outgoing edges of $n$ have to be drawn as

\[\text{Non-essentially contributing objects are objects that only propagate messages but do not affect the computations of an operation.}\]
CHAPTER 3. FORMALIZATION OF LEGAL TEXTS

Figure 3.3: Sequence diagram of the calculateFinalGrade operation.

Figure 3.4: Sequence diagram with superimposed TSG of the (adaptive) calculateFinalGrade operation. Arrows corresponding to TSG edges are marked by a cross. They are labeled with the signature of the adaptive operation. Path constraints on TSG edges can be specified in curly brackets.
equally numbered arrows in the sequence diagram. The semantics of equally numbered arrows are straightforward: messages may be exchanged by each of the equally referenced sender/receiver objects.\(^6\)

- Adaptive visitors may impose further structural constraints that are not considered in our extended sequence diagrams. Let us assume the `calculateFinalGrade` operation wants to weight the different grades depending on the exam's importance. For that matter the adaptive visitor would perform some computation around exams. However, `Exam` objects do not appear in Figure 3.4. Thus, we should be aware of the fact that extended sequence diagrams may hide some details, e.g. objects which are essentially contributing to a collaboration.

### 3.3 From Legal Text to Formal Model

In Section 3.2 we clarified how our formal representation of legal texts looks like. Now we are prepared to investigate the corresponding formalization method.

According to Uschold *et al.* [30] a specification is denoted *semi-informal* if it is ‘expressed in a restricted and structured form of natural language, greatly increasing clarity by reducing ambiguity.’ We believe that this definition applies to legal texts, too. Firstly, a legal text is structured by legal norms that are arranged in numbered articles and paragraphs. Secondly, the juristic language is a very powerful and complex instrument. But, legal texts are formulated in a very precise way trying to avoid ambiguities and contradictions in the interest of a well functioning legal system. These properties should ease the formalization process.

#### 3.3.1 Capturing Static Concepts

Before the semantics of a legal text can be modeled, we have to identify a common vocabulary. For that purpose we must find out the concepts of a legal text and their relationships. Concepts are abstract or real entities from the universe of discourse (or domain). We conceive objects and classes as concepts. Objects represent unique entities that ‘live’ in a domain. Objects define individual attributes and methods. An object interacts with other related objects. Theoretically, we could describe a complete domain by using objects only. Nevertheless, the notion of a class is useful. A class is an abstraction of a set of objects that share common properties. With classes we are able to condense those common properties into one concept instead of modeling the same properties for every object again and again.

\(^6\)It does not mean, though, that multiple interactions take place concurrently.
3.3.1.1 Identifying Concepts

We identify concepts by employing the technique of grammatical inspection. This technique basically assumes nouns in the specification to be potential candidates for concepts, whereas adjectives might point to attributes, and verbs correspond to possible methods. This approach is definitely not a remedy for all difficulties associated with conceptual modeling. In our eyes it works particularly well in conjunction with specifications written in relatively simple sentences, what is usually not the case for legal texts. However, grammatical inspection may serve as a start for capturing the static structure of a domain.

3.3.1.2 Setting up Relationships

Next to the concepts we also have to capture the relationships between concepts. We distinguish three kinds of relationships.

Generalization In general the abstraction process, i.e. identifying more general concepts, is the brainwork of knowledge engineers. There are some indicators on text level, though, that are likely to address generalizations. Forms of the English verb ‘be’ belong to these indicators. For example, the sentence ‘Oral exams are exams that ...’ strongly suggests the concept Exam to be a generalization of the concept OralExam. Thus, generalizations are also known as is-a relationships.

Association We could not find a general rule of thumb for the identification of associations. Often associations are not given explicitly but in the context of some procedure or constraint. For example, the examination regulations [3] do not explicitly state that a grade is associated to an exam. Nonetheless, several procedures and constraints implicitly require a grade to be connected to an exam. For instance § 13(1) Sentence 1 says:

Exams are passed if they have been marked ‘sufficient’ (4.0) at least.
(translated from German)

This invariant expects a grade to be associated with an exam. Consequently, associations, which are not explicitly mentioned, should be modeled as required by the context.

Aggregation During textual analysis aggregations are likely to be found in sentences with aggregating verbs like ‘contain’, ‘comprise’, ‘is composed of’, ‘is part of’, etc. Figure 3.1 shows the concept Exam as a part of the concept Course. 

7This phrase reads ‘Paragraph 1 of Article 13’.
3.3.2 Capturing the Dynamics of Processes

To a large extent, legal texts and especially regulations describe processes. This results from the fact that most legal norms are causal clauses given as if-then patterns. Processes are obtained by combining related legal norms. For example, the examination process from Figure 3.2 may have been described by the following, invented paragraphs:

1. Students wishing to sit an exam must first enroll into the exam.
2. If a student misses to sit an exam he is enrolled in, then the exam counts as if it has never taken place.
3. The examiner must mark an exam within four weeks.
4. An exam is passed if it is marked ‘sufficient’ (4.0) at least. A certificate stating the exam and the achieved grade will be issued if the student passes the exam.
5. If the student fails an exam, he may repeat it once. If a repeated exam cannot be passed either, then the exam is finally not passed.

3.3.2.1 Identifying and Assigning Processes

As a first step we must get clear about which processes are described by a legal text and how they relate to the concepts identified in Section 3.3.1.

The identification of processes sounds more difficult than it actually is. We know that processes are described by legal norms. So, if we can cluster all related norms, we obtain a process description. The semi-informal nature of legal texts helps us here.

- Legal norms are structured in a hierarchy of articles, paragraphs, and sentences. Articles and paragraphs make up organizational units, which represent complete or partial clusters of related norms. Beside that, articles also contain a title among the mandatory reference number. The title or reference number may be used to identify a process. For instance, §§ 8,9 PO are solely dedicated to oral and written exams.

- Legal texts also embody phrases with referencing character. We can utilize these references to find other norms of a cluster.

Having identified a process we must assign it to a concept of the domain. This step needs special care since it will greatly determine the attributes and operations of the assigned concept.

8This phrase refers to the Articles 8 and 9 of the examination regulations [3].
9Remind that a process is conducted by an FSM. In object-oriented modeling one FSM is typically assigned to one concept.
CHAPTER 3. FORMALIZATION OF LEGAL TEXTS

<table>
<thead>
<tr>
<th>Event</th>
<th>Operation Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>call event</td>
<td>actor-initiated operation</td>
</tr>
<tr>
<td>signal event</td>
<td>asynchronous operation</td>
</tr>
<tr>
<td>change event</td>
<td>listening operation</td>
</tr>
<tr>
<td>time or instantaneous event</td>
<td>timed operation</td>
</tr>
</tbody>
</table>

Table 3.1: Classification of operations according to their initiating events.

Possible candidates for assigning processes include key concepts and cluster concepts. By key concepts we mean very important concepts that occur several times in a legal text or that already appear in the title of a legal text. For example, in examination regulations all kinds of exams are likely to be key concepts. By cluster concepts we refer to concepts that are locally defined in the cluster of a process.

3.3.2.2 Deriving Transitions

Legal norms describe operations among some other things. This is particularly the case for regulations because regulations are intended to control the execution of laws. Out of our exemplary paragraphs on page 44 only the first sentence of Paragraph 4 and the last sentence of Paragraph 5 do not specify operations.

We distinguish two groups of operations: operations that just query the model and operations that change the model’s state. When deriving transitions we are mainly interested in the latter group. Queries have no side-effects to the model. That is why they cannot cause state changes. Consequently, it makes little sense to derive transitions for queries.

We further classify operations with respect to their triggering events as summarized in Table 3.1. In the following we examine the properties of each operation class and give hints on how we can recognize operations and their corresponding transitions from legal texts.

**Actor-Initiated Operations** This is probably the most important operation class. Actor-initiated operations are executed on behalf of an external actor. An example shall clarify what we mean: ‘Students wishing to sit an exam must first enroll into the exam’ (Paragraph 1 on page 44). This norm contains two

---

10Remind that procedures and operations view the same thing from different positions (Section 3.1). Considering that, we could use the terms ‘procedure’ and ‘operation’ interchangeably here. In conjunction with transitions we prefer ‘operation’ as better fitting.

11See Section 3.4.4.

12Instead they refer to model constraints which are dealt with in Section 3.3.3.
actor-initiated operations – sitting an exam and enrolling into an exam. Both operations are performed by an external actor\textsuperscript{13}, which is a student in this case.

In our formalism actor-initiated operations represent the external interface of the model. Particularly regulations are known to precisely describe which operations are performed when and by whom. We express that circumstance by the following definition.

**Definition 3.1** Let $SD_c = (S, T, s_0)$ be a state diagram assigned to a concept $c$, where $S$ is a finite set of states, $T$ is a finite set of transitions, and $s_0 \in S$ is the start state. Then an actor-initiated operation $O$ is defined by a triple $O_{SD} = (m, s, a)$ with ‘$m$’ representing a method of $c$, $s \in S$ being a state of the SD indicating when $m$ can be initiated, and ‘$a$’ corresponding to an actor who is permitted to invoke $m$.

That definition introduces a method $m$. At first sight, it is rather confusing why the functionality of an entire operation is represented by one method only. We defined a method as the ‘smallest functional unit of an operation.’ However, methods send messages to other collaborating objects forming an operation. The method $m$ is just a characteristic method that receives the initial invocation message of an operation.

In Definition 3.1 the condition of when an operation can be initialized is modeled by a state $s$. We argue that this does not contradict with legal norms. The consequence of a norm applies whenever the corresponding situation is present. A situation, i.e. a set of facts, can be modeled by the attributes of a concept [7]. We can apply states to describe the situations of legal norms because a state represents a specific combination over the attributes of a concept. One state thus also models exactly one situation.

Furthermore, we express the condition of who is permitted to invoke an operation by an actor $a$. The term *actor* is not new to UML. It is used in use-case diagrams to denote persons or external systems that interface with the model from outside the system boundary.

Having formally defined actor-initiated operations, we are able to derive the corresponding transitions. We model actor-initiated operations by transitions that are triggered by call events and that may be guarded. Actions are not necessary. Given an actor-initiated operation $O = (m, s, a)$ we are able to derive the following transition:

$s \rightarrow t : m(\text{arguments}) [\text{role} = a \text{ and guard-expression}]$

\textsuperscript{13}Basically an actor may also reside in the model. However, since legal texts usually rule the life of persons – whether real or juristic persons does not play a role – we assume the actors of the model to be external.
The target state \( t \), the optional method arguments, and the optional guard have not been mentioned yet. How we obtain the missing details is explained in the following.

The target state \( t \) primarily depends on the operations of a process that logically follow this transition. In our exemplary sentence this task is fairly easy because it explicitly says that the operation enroll directly precedes the operation sit. How we could formalize these two operations is illustrated in Figure 3.2.

The question of what arguments are needed by an operation cannot be answered generally. It usually depends on the context. For example, the mark operation requires at least a grade argument.

Guards are only necessary if the source state \( s \) is not sufficient to represent the preconditions of an operation alone. In that case we have two options. Either we add another, more specific state that satisfies the preconditions, or we control the initiation of the transition by a guard. The former solution is well suited if there are more than one transitions sharing the same precondition. Otherwise guarded transitions are preferred.

All in all, we employ transitions for actor-initiated operations to describe operation protocols. An FSM-controlled process allows only those operations to be invoked which are opened. An opened operation is represented by an enabled transition that is outgoing from the currently active state.

**Asynchronous Operations** Asynchronous operations become valuable if we want concurrent threads of execution to communicate efficiently. In the legal domain we encounter two types of concurrency: interprocess and intraprocess concurrency.

Processes take indefinite time to complete; some processes may not be finished at all. Legal texts usually describe a set of concurrent processes. The notion of concurrency is important here because in a sequential environment we could not carry out two or more processes at once. For instance, if a student has to pass two examinations in one term, the corresponding legal model will have to support two examination processes at the same time. We term that kind interprocess concurrency.

Since interprocess concurrency is relatively coarse grained, we would like to extend concurrency to subtasks of processes, too. Our investigation of the departmental regulations [3,4] has shown that there are many concurrent subtasks. A prominent example are decisions upon request. In that case, first an actor, e.g. a student, applies for something. After a period of time another actor, e.g. the examination board, grants or rejects the application. Without concurrency a process containing such a subtask would be blocked until a decision is made. However, with intraprocess concurrency the process could proceed while the application is being handled.
Unfortunately, not all concurrent tasks are free from interdependencies. We must therefore ensure that interdependent tasks can communicate with each other. Asynchronous operations accomplish that communication using signals.

**Listening Operations** A listening operation responds to changes of concept attributes. It is triggered by change events. Listening operations are not mandatory for the formalization of legal norms. If all state changes of a concept were completely under the control of an FSM, then there would actually be no need for listening operations because uncontrolled changes of the concept’s state would never occur.

But, listening operations can be useful. An example is § 1 Sentence 2 PO:

The academic degree ‘Diplom Informatiker’ or ‘Diplom Informatikerin’ (abbr. ‘Dipl.-Inf’) will be conferred as result of the passed diploma examination. *(translated from German)*

Here we have two possibilities. First, the concept ‘diploma examination’ notifies the concept, which confers academic degrees, that it is passed. Second, the degree conferring concept contains an operation listening for the diploma examination to be passed. Each approach works well, but the latter one seems more elegant.

**Timed Operations** Timed operations are operations that will be initiated when a deadline is not met due to inactivity. A deadline may be given in relative time (duration) or in absolute time (date). Examples are:

The candidate can demand the examination board to check the decisions from (3), Sentence 1 and 2, within two weeks. § 12(4) PO

If the candidate exceeds the enrollment deadline [...], then the examination board will offer late enrollment upon request. § 3(9) PO

We can model such conditions by time events. The corresponding transitions could look like the following.

```
// § 12(4) PO
?->?:after (2 weeks)/complain()
// § 3(9) PO
not_enrolled->deadline_exceeded:after (enrollmentDeadline)/
```

14 This could be realized by signals (asynchronous operations).
Instantaneous events are specially important. Instantaneous transitions are applied if a state represents a situation imprecisely. In that case we may consider to branch the imprecise state into several distinct and more specific states.

The examination process from Figure 3.2 stops at the state marked after the exam was marked. However, the subsequent operations are not interested whether an exam is marked. Instead they want to know if an exam is passed or not. As a consequence, two instantaneous transitions marked→passed and marked→failed are added branching the state marked into the more specific states passed and failed.

Instantaneous transitions should be guarded in a way that prohibits any two outgoing, instantaneous transitions of a state to be enabled at the same time. Otherwise, we get conflicting transitions. State diagrams that violate this rule are not well-formed with regard to the UML specification.

### 3.3.3 Constraining Properties

Until now we have only constrained the control flow of entire processes using state diagrams. What is still missing is a way to place restrictions on the properties of concepts. Constraints in OCL serve that purpose.

#### 3.3.3.1 Modeling Universal Conditions

Legal texts often contain universal conditions that are valid at any time. In our formalism such conditions are expressed by invariants. Altogether we classify three groups of invariants.

**Value Constraints** Sometimes attributes may only permit a limited set of valid values. In that case we speak of value constraints. Value constraints allow every combination of discrete values or ranges. For instance, we may want to restrict the value attribute of grades to any number between 1.0 and 4.0, and 5.0. The corresponding invariant is:

```ocl
class Grade {
  invariant: (1.0 <= self.value and self.value <= 4.0) or self.value = 5.0
}
```

**Multiplicity Constraints** Multiplicity constraints determine the cardinality of an association. Similarly to value constraints, we can describe multiplicity by ranges or discrete integer values. Most often the multiplicities 0..1 (optional concept), 1 (mandatory concept), 0..n (possibly empty set of concepts), and 1..n (non-empty set of concepts) are sufficient.
Preferably we define multiplicity constraints in class diagrams. Only, if we need
finer-grained control over the cardinality of associations, we may express them
in OCL. This may happen if multiplicity constraints are changing over run-time,
e.g. as the result of state changes.

Constraints between Attributes and/or Associations  Legal texts may
also draw interdependencies between specific attributes or associated concepts.
Paragraph 4 Sentence 1 from page 44 provides an example. Here the attribute
passed of the concept Exam depends on the attribute value of the concept Grade.

\[
\text{context Exam}
\text{inv: self.passed = (self.grade.value <= 4.0)}
\]

Additionally, this sentence contains another dependency between the value of a
grade and its description. So we could write:

\[
\text{context Grade}
\text{inv: self.value = 4.0 implies self.desc = 'sufficient'}
\]

The last example demonstrates that invariants are a practical instrument to
describe derived attributes as well.

3.3.3.2 Modeling Legal Norms

We have already explained that legal norms typically follow an if-situation-
then-consequence pattern. Regarding that, legal norms can be transformed to
corresponding operations with the following characteristics.

- A situation is expressed by preconditions. If there exists a transition
corresponding to the legal norm, the situation is additionally expressed by
the source state and guard of the transition.

- A consequence is expressed by postconditions. If there exists a transition
corresponding to the legal norm, the consequence may be represented by
the transition’s action directly.

Situations  Situations can be modeled by both OCL preconditions and state-
guard pairs. Each approach has its benefits and drawbacks.\(^\text{15}\) A state diagram
provides a general overview of the situational constraints of all operations of a
process. However, if the situations of legal norms are very complex, the resulting

\(^{15}\text{In [31] Baar warns of a phenomenon that he calls 'exploitation of only one specification}
\text{technique'. He found out that developers 'tended to use OCL also for purposes where a
StateTransition or Activity diagram is more suitable.'}
CHAPTER 3. FORMALIZATION OF LEGAL TEXTS

state diagram will become cluttered by too many details. In that case, it is worth to partially write situations as OCL preconditions.

As a practical guideline we recommend to describe simple conditions of concept properties by state diagrams and to express more complex conditions or conditions of operation arguments as OCL preconditions.

**Consequences**  Consequences provide information about the functionality of operations. We demonstrate that with an example. First we rewrite Paragraph 5 Sentence 1 from page 44 in rule-based form.

If exam is failed and exam is not repeated then student may repeat exam.

This rule belongs to an operation concerning the repetition of exams. We may name this operation repeat and assign it to the concept Exam. This rule explicitly states a situation that must be met before the operation can be performed. Moreover, it makes an implicit assumption about the consequence, i.e. exams may only be repeated once. So, the consequence must alter the exam's state in a way that does not allow exams to be repeated twice. A corresponding OCL construct could look as follows.

```plaintext
context Exam::repeat()
pre: not self.passed and not self.repeated
post: self.repeated
```

### 3.4 Open Problems

Not everything in legal texts can be formalized. In this section we mention some problems associated with the formalization of law. To a large extent the experiences described here are based on the work of Navratil [7].

#### 3.4.1 Scoping

Law is a highly complex system. It does not only consist of a few interconnected legal documents from the legislative, but also of a vast amount of judgments on those documents. One common task of the judiciary is to find out whether specific paragraphs clash with other paragraphs located in more general laws. Clashing paragraphs are invalid by law. Some countries also permit ‘unwritten law’.

---

16For this example we assume that the repeat operation is not constrained by a state diagram. If it were already constrained by a state diagram like the one in Figure 3.2, we would have to provide the postcondition only.
If we were formalizing all aspects of a legal domain, we would probably end up trying to incorporate loads of legal documents and human expertise into the model. From our point of view such an approach is too ambitious even for smaller domains. Therefore, we need to scope the domain; we have to set feasible boundaries of our model.

However, scoping is problematic and always a compromise. On the one hand, we do not want to exclude any key concepts from our formal model. But, on the other hand, it is practically impossible to reach a closed world assumption.

### 3.4.2 Fuzzy Definitions

Legal texts often contain fuzzy definitions that cannot be fitted into formal concepts easily. The intend behind such fuzzy definition is that legal documents shall be valid over a longer period of time. A legal text would certainly have to be adapted to changing requirements very frequently if it contained too explicit definitions.

Another reason is the scope of discretion. A legal text cannot hold exact consequences for each possible situation. In cases where a situation is unclear or vague, the judiciary’s scope of discretion is asked to make a decision about the legal consequence.

Our formalism is not suited to model that kind of expertise—we term it discretionary knowledge. A solution is described by Stranieri et al. [6]. They apply argument trees based on the Toulmin structure [32] in order to express discretionary knowledge.

### 3.4.3 Incomplete or Ambiguous Clauses

All sorts of informal specifications suffer from incomplete and ambiguous clauses. For instance, Paragraph 3 on page 44 is incomplete since it does not specify what happens if an exam has not been marked within four weeks. Obviously, incomplete and ambiguous legal texts are unavoidable. A formal model should never be ambiguous by definition, although, it may be incomplete\(^{17}\).

### 3.4.4 Legal Norms with Organizational Character

Not every norm in a legal text is rule-based. Generally a legal text may also include:

- A clause stating when the legal text comes into force (e.g. in § 28(1) PO).

\(^{17}\)Compare to Section 3.4.1.
• Clauses stating which paragraphs of former legal texts become invalid or are replaced by the new legal norms (e.g. in § 28(2,3) PO).

• Phrases referencing other legal texts, articles, paragraphs, or sentences (e.g. in § 12(1) SO [4]). Possibly these references point to concepts outside the model’s scope.

• Comments or annotated information (e.g. in § 1 PO).

Of course, attention should be paid to everything written in a legal text. However, we believe that – from the four listed items – only the referencing phrases essentially contribute to the formalization process.
Chapter 4

System Architecture

A formal model is just one side of a coin. The formalism from the previous chapter is nothing but a collection of specifications. To obtain a fully functional information system, we must lay down its architectural foundations, too. This chapter deals with the design of ADONIS. We will present the system architecture of ADONIS from both the user’s and the developer’s perspective.

4.1 The Idea

The premise of a legal advisory system is to give users sensible answers on their legal enquiries. Given a number of facts about a case and a query, the system shall ideally infer the same response that would have been given by a human expert.

Regarding that, we could classify ADONIS as an expert system. Expert systems have been a long-term research topic of Artificial Intelligence. They usually comprise a knowledge base and some inference capability that reasons by using the acquired knowledge. Common to these approaches is that their knowledge representation, i.e. the basic modeling unit, is relatively simple in comparison to the high complexity of the knowledge itself. Often the knowledge is represented in some form of logic or as production rules. The reason is hidden in the inferencing possibility: the simpler a knowledge representation the less complex the inference algorithm.

With ADONIS we follow a completely different approach. We argue that the meaning of many words in legal texts is too complex, vague, and ambiguous to let a single inference algorithm derive possible impacts. Instead we favor a more Software Engineering oriented approach. We let an analyst transform legal texts into the formal model from Chapter 3. That model may be understood as a combination of knowledge base and inference engine. Our ‘knowledge base’
holds rules in form of operations and facts in form of object attributes. Our ‘inference engine’ is assembled by the entirety of operations, i.e. it is implicitly specified in the formal model. In that our approach fundamentally differs from expert systems.

4.1.1 Query and Result Notation

In Chapter 3 we found out that regulations basically describe the operations of processes. Many users of ADONIS will consequently ask for advice about those operations. Therefore, we center our query and result notation around operations. The users may ask questions of the basic form:

Given a certain situation, how can I proceed in the process?

The response to this kind of query contains a possibly empty set of executable, actor-initiated operations\(^1\). In this set every operation defines an alternative way to proceed within a process.

4.1.2 Query Processing

Our query-processing technique is based on the Mealy automaton, which is a kind of finite state machine (FSM).

**Definition 4.1** A Mealy automaton is a tuple \( M = (S, X, Y, \delta, \lambda, s_0) \), where 

- \( S \) is the set of states, 
- \( X \) is the input alphabet, 
- \( Y \) is the output alphabet, 
- \( \delta : S \times X \rightarrow S \) is the transition function, 
- \( \lambda : S \times X \rightarrow Y \) is the output function, 
- and \( s_0 \in S \) is the start state. At any time the automaton is in a state \( s \in S \).

If the automaton receives an input symbol \( a \in X \), it will enter the state \( \delta(s, a) \) and emit the output symbol \( \lambda(s, a) \).

In the following we analyze how a Mealy automaton \( M \) can be applied for query processing. For that matter, we first define the structure of the input and output symbols of \( M \).

Each query specifies a certain situation (or case). This situation has to be input into the legal model somehow. The probably best way to accomplish this would be to recognize the natural language of the user. Human experts are able to understand natural language but computers cannot handle it so easily because natural languages are very complex and ambiguous. Therefore, we rather prefer a more formal input language to describe situations.

Let us assume that each situation within a process \( p \) can be expressed by a state \( s \in S \) in \( M \). A state is equivalent to the Cartesian product over all object

\(^1\) See also Definition 3.1 on page 46.
attributes dedicated to $p$. This assumption implies that our model is expressive enough to distinguish between all possible situations. If we further assume that each state $s$ can be reached from an initial state $s_i$ by an input string $\mathcal{X}$, i.e. a sequence of input symbols $a$, we are also able to express every situation by $s_i$ and $\mathcal{X}$. The basis for that is laid by Definition 4.2.

**Definition 4.2** Let $M = (S, X, Y, \delta, \lambda, s_0)$ be a Mealy automaton. For any input string $\mathcal{X} = X^*$ (including the empty string $\varepsilon$), input symbol $a \in X$, and initial state $s_i \in S$, the transition function on strings $\delta^*$ is defined as

\[
\begin{align*}
\delta^*(s_i, \varepsilon) &= s_i \\
\delta^*(s_i, a) &= \delta(s_i, a) \\
\delta^*(s_i, Xa) &= \delta(\delta^*(s_i, X), a).
\end{align*}
\]

By using the transition function on strings $\delta^*$, we are able to input situations into the legal model. Given an initial state $s_i$ and an input string $\mathcal{X} = \langle a_0a_1 \ldots a_n \rangle$, we obtain the desired query situation $s_q \in S$ by applying

\[
\begin{align*}
s_q &= \delta^*(s_i, \mathcal{X}) \\
&= \delta(\ldots \delta(s_i, a_0), a_1) \ldots, a_n).
\end{align*}
\]

The input symbols $a$ in $\mathcal{X}$ carry information about actor-initiated operations. More precisely, each input symbol is a tuple $a = (o, A)$ containing an operation reference $o$ plus a set of argument values $A$ (including the calling actor).

The result of a query step, i.e. the output of $M$, primarily consists of a set of executable, actor-initiated operations. This set only depends on the current state $s$ of $M$. Strictly speaking, it is derived from those outgoing transitions of $s$ that are currently enabled.\footnote{Enabled transitions can be triggered while disabled transitions cannot.} Regarding that alone, we could switch to the simpler Moore automaton. But, normally a user would also like to know what happens in the system during the query process. That means each query step should be documented and reported appropriately. Since query steps are strongly related to transitions in $M$, we utilize the output function $\lambda$ to report information about query steps. The output symbols delivered by $\lambda$ are tuples $(O, M)$, where $O$ is a set of executable, actor-initiated operations, and $M$ is a set of query messages.

### 4.1.3 Concept of Usage

Before the usage of ADONIS is introduced, we shall look at the way conventional expert systems interface with the user. Such systems commonly interact with users by so called shells. A shell mainly serves two purposes: it allows the knowledge engineer to construct and maintain a knowledge base, and it enables
the end-user to query the system’s expertise. In legal advisory systems a query procedure typically consists of a consulting phase and a reasoning phase. In the consulting phase the system acquires the facts of the case by asking the user query-relevant questions, and in the reasoning phase it tries to answer the query based on the previously acquired knowledge.

ADONIS’ concept of usage varies a bit from the one of conventional expert systems. Broadly speaking, we also divide user queries into a consultation and a reasoning phase. However, a whole query procedure typically involves a sequence of interwoven consultation and reasoning steps, i.e. our query procedures acquire and process knowledge incrementally. We explain the usage with an example.

Let us assume the legal model from Section 3.2 and the following query:

I failed to pass exam \(xyz\) for the second time. What can I do now?

That query is related to the examination process for exam \(xyz\), as illustrated in Figure 3.2. We see that this process contains a state \(\text{failed}\) which represents our query situation. In order to input this situation, the user would have to provide all facts that are necessary to bring the examination process into the state \(\text{failed}\). Since the query procedure is incremental, the user must interact with the system in steps – so called query steps. At each query step the system offers a set of alternatives, i.e. executable, actor-initiated operations in result notation. The user selects the alternative that best approximates his case. Doing so, the user tells the system a ‘story’\(^3\) step by step, which eventually leads to the desired query situation.

Table 4.1 lists an user-system dialog for our example. That dialog actually touches the situation \(\text{failed}\) twice. In Step 4 the student fails exam \(xyz\) for the first time. In that step, the system offers one alternative: to repeat the exam. The student does so (Step 5-8), and fails to pass the exam another time. The underlying legal model only allows exams to be repeated once. So, although the input symbols of the repetition cycle are exactly as the ones of the first trial, the system infers that the exam is \(\text{finally not passed}\) (Step 8). Thus, the query result is the empty set, i.e. no other alternatives are available to proceed.

4.2 Architectural Overview

We decided to design ADONIS as a web-based application. Despite being a hot topic at the moment, web-based applications have three primary advantages from our point of view.

- End-users are familiar with the World Wide Web. They will certainly have little trouble adopting the look and feel of ADONIS.

\(^3\text{Compare with the input strings } X \text{ from Section 4.1.2. Each ‘story’ is basically an input string } X \text{ written in terms of input symbols } a.\)
#### Table 4.1: A progressive user-system conversation.

<table>
<thead>
<tr>
<th>Query</th>
<th>By System</th>
<th>By User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>State</td>
<td>Available Operations</td>
</tr>
<tr>
<td>1</td>
<td>not_enrolled</td>
<td>{enroll}</td>
</tr>
<tr>
<td>2</td>
<td>enrolled</td>
<td>{miss,sit}</td>
</tr>
<tr>
<td>3</td>
<td>participated</td>
<td>{mark}</td>
</tr>
<tr>
<td>4</td>
<td>failed</td>
<td>{repeat}</td>
</tr>
<tr>
<td>5</td>
<td>not_enrolled (repeated is set)</td>
<td>{enroll}</td>
</tr>
<tr>
<td>6</td>
<td>enrolled</td>
<td>{miss,sit}</td>
</tr>
<tr>
<td>7</td>
<td>participated</td>
<td>{mark}</td>
</tr>
<tr>
<td>8</td>
<td>finally_not_passed</td>
<td></td>
</tr>
</tbody>
</table>
End-users do not have to worry about installing and updating another application because ADONIS is maintained on server side completely.

Web-based applications naturally achieve a high level of platform independence on client side. Thus, we can reach a high percentage of potential users with ADONIS.

Figure 4.1 displays the system architecture at a glance. All in all there are three main components in ADONIS.

- The presentation layer (or user interface) is a collection of web pages. The presentation layer is distributed over client and server side. The web server (or, more precisely, the servlet container) dynamically creates web pages and the web browser renders them onto the user’s screen.

- The legal model holds knowledge about a legal domain. It is initialized by an interpreter that understands model descriptions written in our Object Graph Description Language (OGDL).

- The application controller is the backbone of ADONIS. It routes information from the presentation layer to the legal model and vice versa.

ADONIS strictly separates presentation from model. The design pattern behind that paradigm is known as Model-View-Controller (MVC) [34]. In ADONIS we employ a version of MVC that takes the special needs of web-based applications into account. That extension is called MVC/Model 2 [35] and is illustrated in Figure 4.2. The MVC/Model 2 architecture is fairly complex at first sight. It introduces a noticeable overhead in comparison with closer coupling. However, in favor of a cleanly designed and manageable system we recommend using MVC/Model 2.

4.3 A Look at the User Interface

For rather visually driven readers we take a short look at the user interface of ADONIS in this section. Figure 4.3 shows a snapshot of a query procedure. Moreover it identifies five subcomponents. Each of these components fulfills a specific task as described in the following.

**Process Selector** A legal model may consist of more than just one process. In this case we need a tool to pick up a certain process for querying. The process selector enables the user to navigate through a set of supported processes and to select one process. How this navigation can be implemented is left open. At the moment we arrange processes into a tree structure.

---

*OGDL is extensively covered in Appendix A.*
CHAPTER 4. SYSTEM ARCHITECTURE

Figure 4.1: The system architecture of ADONIS.

Figure 4.2: Client-server interaction in MVC/Model 2. The shown interaction is taken from Jakarta Struts [33] – a framework for web-based applications using MVC/Model 2.
CHAPTER 4. SYSTEM ARCHITECTURE

Role Selector Regulations clearly define which person is permitted to carry out which operation. In our legal model such persons are termed actors. An actor always plays one role in ADONIS, e.g. a student is an actor and an examiner is another actor. The role selector enables users to take up different roles. That means, a user is able to look at a situation from different perspectives, e.g. from the position of a student or an examiner. We do not require any authentication services for role switching because ADONIS was designed to give anonymous advice. As a result, security matters, such as sensitive user-specific data, are rarely existing.\footnote{Indeed, a user profile will be thrown away after the user session is finished.}

Inspector We have already mentioned that a query procedure incrementally consults the user and processes the acquired facts. The inspector is responsible for the presentation of the current process state, i.e. it visualizes the current situation. The intention behind that is to prevent users from misunderstandings about the system’s knowledge of a case.

Messenger A query procedure can last a long time depending on the complexity of the query situation and the already input facts. Hence, it becomes necessary to know what actually happens during the processing of query steps. The Mealy automaton conducting a query procedure therefore emits messages informing about its activities or decisions. These messages are collected and displayed by the messenger.

Operator The operator serves two tasks. On the one hand, it shows the result of the last query step. As noted earlier, this result comprises a set of alternative operations, where each operation corresponds to an option for proceeding the query procedure. On the other hand, the operator allows...
the user to initiate such an alternative operation. If an operation requires further facts (in form of arguments), the operator will ask the user to input these. Finally, the operator sends an input symbol, i.e. the initiated operation and its arguments, to the Mealy automaton. All in all, the operator can be understood as a frontend to the Mealy automaton.

From a user’s perspective, a query procedure involves the following activities.\(^6\)

1. Select a process. (Process Selector)
2. Select a role. (Role Selector)
3. Select the operation that approximates the query situation best. (Operator)
4. If the operation requires further information, input the operation’s arguments. (Input form generated by Operator)
5. Look at the result. (Messenger, Inspector)
6. Iterate through 1–5 until the desired query situation has been input. The Operator will finally show all open alternatives, i.e. the query result.

## 4.4 From Formal Model to Source Code

In Chapter 3 we explained how formal models can be extracted from legal texts. As a reminder, our legal models consist of a number of textual and graphical specifications in UML. In this section we proceed with how these specifications can be embedded into the system architecture of ADONIS.

### 4.4.1 Class Diagrams

Class graphs are expressed by class dictionaries in AP. Class dictionaries were introduced long before the advent of UML. However, the Demeter Research Group knows about the importance of UML and supplies a mapping of UML class diagrams to AP class dictionaries that is well suited for our purpose. Since that mapping was already introduced in Section 2.2.2.3, we focus attention on the limitations caused by transforming UML class diagrams into AP’s class dictionaries here. The following comments shall provide some helpful feedback for the formalization process from Chapter 3.\(^7\)

---

\(^6\)The participating components are given in parentheses.

\(^7\)That feedback is irrelevant if designing for other target platforms than AP.
• AP strictly bears the abstract superclass rule [36] in mind, which says that ‘all superclasses must be abstract.’ This rule promotes a separation of classes into concrete classes creating instances and abstract classes organizing other classes through generalizations. According to Hürsch the abstract superclass rule shall be beneficial in many situations. If we nevertheless want to instantiate a superclass, we must reorganize that superclass into an abstract superclass (alternation class) and a concrete subclass (empty construction class).

• The cardinalities of associations are not very well supported in class dictionaries. Only the multiplicities 0..1, 1, 0..n, and 1..n can be expressed in class dictionaries directly. Other than these would have to be constrained individually.

• There is only a limited set of attribute modifiers in class dictionaries. At the moment DemeterJ supports final, static, and derived attributes. In addition, the visibility of attributes can be set ‘protected’ (default), read-only, or private.\footnote{The semantics of read-only and private attributes are a bit misleading in AP. Read-only attributes cannot be set from outside the package, and private attributes cannot be accessed from outside the package. However, both kinds can be queried and updated from classes of the same package.} Moreover, initial attribute values can be defined.

4.4.2 Object Diagrams

Object diagrams can be translated into the application specific languages (ASL) from Section 2.2.3 or our Object Graph Description Language (OGDL), which will be dealt with in Appendix A. Object graphs in ASLs or OGDL are directly instantiable by the target platform.

As a recommendation, object trees or composite objects, i.e. objects that are composed of other objects, are better described in ASLs. But, ASLs on their own are not applicable for the description of cyclic object graphs. OGDL addresses that insufficiency.

4.4.3 State Diagrams

We describe state diagrams in a special notation used by our finite state machine implementation (Appendix B). We are aware of the rudimentary functionality of this FSM implementation. But, we think it works reasonably if we take the shortcomings, as described in Section B.1, into consideration when formalizing legal texts.

As already mentioned, we employ FSMs to specify operation protocols for processes. That means, depending on the current state of a process only a few selected operations are executable. All other operations will be rejected. The
public void operationName(Actor role, ...)  
    throws GuardedTransitionException {
        sm.open("operationName",role,this);
        // here goes the FSM protected code performing some
        // computation
        sm.close();
    }

Figure 4.4: Design pattern of an actor-initiated operation.

external interface of our legal model should only export processes (in form of
classes) and actor-initiated operations (assigned to classes). Operations which
are not executable should throw a suitable exception. So, among the transfor-
mation of UML state diagrams into our FSM notation, we further have to draw
a connection between the operations and the operation protocol of a process.
Figure 4.4 shows how we accomplish this.

In Figure 4.4 the variable $sm$ refers to the FSM that conducts the operation’s
process. Before any computation is done, the FSM checks whether the corre-
sponding call transition is enabled ($open$). This check evaluates the guard of the
transition with the aid of the given actor ($role$) and the process itself ($this$). If
the check fails, the operation will be canceled and a $GuardedTransitionException$
will be thrown. The actual transition is performed on $close$.

4.4.4 Object Constraint Language

From our point of view, model constraints in OCL are used for two reasons:

1. constraints guide the development of algorithms, and
2. constraints are used to (more or less formally) prove the correctness of
   algorithms.

In the scope of this thesis we are mainly interested in the former. We believe
that there is no universal rule of how a developer can turn an OCL specification
into code that is correct with respect to the imposed model constraints. That
is mainly a question of complexity as well as the experience and skills of the
developer. For the moment we assume our algorithms to be correct, i.e. all
model constraints are always satisfied.

9The calling actor must be given explicitly because the target platform does not allow us
to find out which object is the sender of an invocation message.
public void operationName(Actor role, arg1 ... argn)
    throws GuardedTransitionException, PreconditionException {
    sm.open("operationName", role, this);
    // check preconditions over arguments arg1 ... argn
    // do some computation
    sm.close();
}

Figure 4.5: Design pattern of an input-safe, actor-initiated operation.

Concerning the latter item, we think that it is beneficial to incorporate constraint checking into the target code as long as it does not cause too much extra costs.\textsuperscript{10} A steady checking of all model constraints may slow down the system considerably. Thus, OCL expressions should be evaluated throughout the development phases only.

Whatever thorough constraint checking is finally implemented, there is one group of model constraints that should have precedence over the others. We refer to constraints on user inputs. Even if a model works 100 percent fine on valid input data, it may fail due to invalid inputs. Therefore, we should take special care of OCL preconditions concerning the arguments of operations of the external interface. In case of our legal model these are the preconditions of actor-initiated operations. Figure 4.5 demonstrates how input-safe, actor-initiated operations can be achieved. Our approach simply rejects all invalid inputs.

### 4.4.5 Sequence Diagrams

Sequence diagrams are mapped onto methods and method invocations. For each arrow in a sequence diagram we create a corresponding method in the receiver class. Method signatures are provided by the arrow labels. Moreover we add code for the method invocation to the sender class. The sender method must be able to send messages to the receiver object, i.e. at activation time it must hold a reference to the receiver object. The receiver’s reference should be obtained and used according to the Law of Demeter.

Sequence diagrams for adaptive operations\textsuperscript{11} are handled slightly different. Here the arrows corresponding to TSG edges are mapped to a traversal strategy

\textsuperscript{10}Ideally such constraint checks would automatically be generated by CASE tools. However, we doubt that AP is supported by mainstream CASE tools anytime soon. Another solution is presented by Knublauch and Rose [8]. They evaluate OCL constraints on runtime using an OCL interpreter. Their interpreter reads OCL specifications in XMI format [37] and utilizes reflection to introspect the model.

\textsuperscript{11}See Section 3.2.5.1.
and at least one adaptive visitor. Adaptive visitors provide visitor methods, and traversal strategies are compiled into traversal code invoking those visitor methods. The example from Figure 3.4 shall clarify what we mean.

In that figure we have a special arrow (Arrow 2) pointing from Course to Grade. This arrow corresponds to a TSG edge, which is additionally constrained by a path constraint. Since there is no other TSG edge defined in the sequence diagram, we can derive the following traversal strategy easily:

- \textbf{from Course bypassing }\rightarrow \ast ,\text{finalGrade},\ast \text{ to Grade} \text{(in strategy-line notation, Section 2.3.5.2), or}

- \{\text{Course }\rightarrow \text{ Grade bypassing }\rightarrow \ast ,\text{finalGrade},\ast \} \text{(in strategy-graph notation, Section 2.3.5.1).}

We recommend to use the strategy-graph notation because it is more expressive and its grammar is closer to the way our adaptive sequence diagrams model TSGs. However, for simple TSGs, like the one for the calculateFinalGrade operation, there are no objections against using the more intuitive strategy-line notation.

Having derived a traversal strategy, we can create a suitable adaptive visitor.\textsuperscript{12} TSGs do not specify what visitor methods an adaptive visitor has to declare, but they can give us hints about what visitor methods are likely to be needed. Our experience has shown that the target classes of a traversal strategy, i.e. the to part, contribute to an operation essentially. This is especially true for query operations, where the searched information are usually held by target classes. The classes or relations referenced by positive constraints, i.e. the via or through parts, are certain candidates for visitor methods, too. Figure 4.6 shows how the adaptive version of the calculateFinalGrade operation could be implemented.

\textsuperscript{12}In all but the rarest cases one adaptive visitor per traversal is sufficient. In ADONIS we do not use more than one adaptive visitor per traversal.
Course {
    
    public void calculateFinalGrade() 
    
    bypassing -> *, finalGrade, * to Grade {
    
        {{
            private double sum;
            private int totalGrades;
        }}
    
    before Grade {
        sum += host.getValue();
        totalGrades++;
    }
    
    after Course {
        double avgGrade = sum / totalGrades;
        host.setFinalGrade(avgGrade);
    }

}
Chapter 5

Implementation

An experimental prototype of ADONIS has been implemented. Altogether, we could model and implement about 80 percent of the examination regulations [3] and 50 percent of the study regulations [4].\footnote{These figures are solely based on the paragraphs in the stated regulations. Enclosed appendices were left out.} The missing paragraphs or sections were not implemented because the majority of them have organizational character\footnote{See also Section 3.4.4.}, are not relevant to the system, or are simply outdated. The percentage of implemented paragraphs in the study regulations is not as high as in the examination regulations because the former document contains much more explanatory and referencing paragraphs, which hardly contribute to the specification of ADONIS.

We employ different technologies for the system components of ADONIS.

User Interface Since ADONIS is a web-based application, the user interface consists of a set of web pages in the Hypertext Markup Language (HTML) [38]. We explicitly avoid interactive client techniques like Javascript, VBScript, Java Applets, Dynamic HTML, etc. We argue that all computation can be performed on server side. So, we avoid dealing with web browser incompatibilities. In addition, interactive scripting has proven to be a security issue for many users and is sometimes switched off by default. Hence, the application of such techniques could lock out potential users.

For the user interface layout we use the Tiles library [39]. Tiles is a powerful templating technique that allows pieces of markup, e.g. JSP or HTML code, to be assembled in a consistent manner.

Legal Model The primary goal of this thesis is to use AP for implementing legal domains in information systems. We have chosen DemeterJ [5] as target platform for the legal model because of two reasons: DemeterJ best
supports the principles and concepts of AP at the moment, and DemeterJ is based on the popular Java programming language [40], which has widely gained acceptance as a server platform in recent years.

**Middleware** We design and implement the middleware on top of the Jakarta Struts framework [33], which considerably eases the development of web-based applications using MVC/Model 2. Jakarta Struts implies the application of the Java Servlet and JavaServer Pages (JSP) technologies [42,43]. A controller servlet receives user requests and dispatches them to actions, each of which is dealing with one specific request. The query result is eventually forwarded to a JSP that automatically generates a response in HTML.

We have developed and tested ADONIS on Apache Tomcat [44], which is the official reference implementation for servlet containers based on the Java Servlet and JSP specifications. However, ADONIS can be deployed on any application server conforming to the Java Servlet 2.2 and JSP 1.1 standards.

Table 5.1 gives a summary of the used technologies. All listed technologies are available free of charge for academic purposes. Moreover, all required libraries, the DemeterJ tools, and Apache Tomcat are open source.

---

3 This thesis does not cover issues regarding web-based applications in general and Jakarta Struts in particular. Regarding that, we refer to a preliminary paper of ADONIS [41].

4 We have patched the DemeterJ 0.8.4 compiler to solve the ‘optional parts of builtin type’ issue (see BUGS file). ADONIS will not compile otherwise. The patch has been submitted to the maintainer of DemeterJ. The upcoming release of DemeterJ should compile ADONIS without difficulties.
Chapter 6

Evaluation of Adaptive Programming

For virtually any kind of research, critical reviews from independent parties are needed in order to evaluate the importance and potential of a certain work. In this chapter we will have a look at AP in the light of our hypothesis. Although almost all presented findings have been won from our work with DemeterJ 0.8.4, we rather try to address the theoretical concepts of AP than implementation-specific issues of DemeterJ.\footnote{A quite comprehensive list of DemeterJ-related problems can be found in the BUGS file that belongs to the source distribution of DemeterJ [5].}

6.1 Impact on Software Evolution and Adaptive Maintenance

To prevent a single-sided view, we base this evaluation on subjective opinions and experiences as well as a number of objective measures. In this section we will investigate the impact of AP on software evolution and adaptive maintenance from an empirical approach. With aid of six evolution scenarios we recorded data to support our hypothesis. We are aware of the limited scope of this case study. A full-blown empirical study would certainly deliver more accurate results. However, our approach shall be sufficient to fulfill the aim of this thesis.

6.1.1 Evolution Scenarios

We restrict our case study to the structural evolution of class graphs because that is the major goal of AP. Changing use cases are not considered here although they play an important role in development and maintenance, too. In
the following we present the scenarios which this study is based on. For better understanding we group the scenarios into categories.

**Category A** In this category we rename classes in the class graph. In the scenario $A_1$ we rename an abstract superclass, and in $A_2$ the name of a concrete class is changed.

**Category B** This category examines how adaptive programs handle changing roles of classes, i.e. we rename association ends. The scenarios $B_1$ and $B_2$ are illustrated in Figure 6.1.

**Category C** Here we insert additional classes and/or associations into the class graph. In the scenario $C_1$ a new class is placed in between three classes so that any communication between these classes is routed through the new middle class (Figure 6.2). Scenario $C_2$ adds a redundant association into the class graph to improve efficiency (Figure 6.3).

We do not examine how adaptive programs behave if classes or relationships are removed from the class graph. Such changes usually influence use cases, which would lead us to the problems of behavioral evolution. Especially, the target classes of queries should not be removed since they carry mandatory information for use cases. In most cases adaptive maintenance will rather add new pieces of information or re-organize information than remove it.

The scenarios were constructed such that they affect a great share of adaptive operations or traversal strategies. Each scenario could also occur in a similar configuration under real conditions.

### 6.1.2 Methods and Measures

For our experiments we used a recent version of the ADONIS code base. Since not all packages of ADONIS are implemented in AP, we focused on the legal model. That subsystem is still large enough to provide representative measures. It consists of approximately 3,700 lines of DemeterJ source code. Moreover, the legal model specifies 41 traversal graphs and declares 174 classes, which are itemized in Table 6.1. Starting from that code base, we applied the scenarios one by one. For every scenario we recorded a number of measures as given in the following.

2 Scenario $B_2$ can be seen as worst case because a lot of our traversal strategies depend on the presence of up-references. In ADONIS these references are extensively used to navigate up- and downwards a double-linked object ‘tree’. We specify traversal path constraints of the form ‘only-through $\{ \rightarrow \\ast,\uparrow,\ast, \Rightarrow \rightarrow \ast,\ast \}$’ to move upwards the tree and ‘bypassing $\rightarrow \ast,\uparrow,\ast$’ to go downwards. Using this pattern we are able to circumvent a great deal of problems associated with recursion.
Figure 6.1: Scenarios $B_1$ and $B_2$. The initial class graph contains several cycles so that corresponding object graphs look similar to double-linked trees. The scenarios on the right only show those subgraphs of the initial class graph that are modified.

Figure 6.2: Scenario $C_1$. 
Primary classes (domain concepts) 63
Secondary classes (utility classes, exceptions, ...) 35
Visitor classes 63
Interfaces 2
DemeterJ-generated classes 11

\[ \sum 174 \]

Table 6.1: Total number of classes in the legal model.
1. First we determined how many traversal strategies are affected by a scenario (affected strategies). Affected traversal strategies can be recognized by comparing their traversal graphs before and after a scenario was applied. The traversal strategy is affected if the corresponding traversal graph differs. DemeterJ writes out approximated traversal graphs in textual form. So, the comparison can be performed by standard tools, e.g. diff.

2. After applying a scenario, we let DemeterJ generate Java code. During this step we were looking at error messages reported by the DemeterJ compiler. Here the number of non-compiling strategies was particularly interesting for us. Non-compiling strategies are traversal strategies where no corresponding traversal graph can be constructed.

3. Another important measure is the number of surprising strategies. Such traversal strategies compile without any complaints, but their constructed traversal graph does not mirror the original intend. That means, the traversal order is disturbed, ‘surprise paths’ are added, or traversal paths are missing. Adaptive operations based on surprising strategies are likely to produce wrong results. The recognition of surprising strategies is not trivial though. We therefore inspected the traversal graphs of all affected strategies manually.

4. Having corrected all reported errors and surprising strategies, we compiled the code base completely, i.e. the generated Java classes were compiled into bytecode. The errors reported by the Java compiler were counted as compilation errors.

5. In this step we were correcting all compilation errors until no new errors occurred.

6. Finally, we counted all modifications that had been entered into the source code since the scenario was applied. We recorded the number of added lines, deleted lines, and changed lines. Additionally, we counted the lines of the DemeterJ source code and generated Java code in order to determine variations in the code size (Δ lines of DemeterJ code, Δ lines of Java code).

After each experiment the code base was rolled back to its original state. Every experiment proceeded until the legal model compiled without any compilation errors. We did neither consider runtime errors, nor did we carry out any form of verification after.

6.1.3 Results and Discussion

The results of our experiments are summarized in Table 6.2. In the following
Table 6.2: The results of each scenario.

<table>
<thead>
<tr>
<th>Category</th>
<th>A</th>
<th>A₂</th>
<th>B</th>
<th>B₂</th>
<th>C</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aected strategies</td>
<td>15/41 (37%)</td>
<td>18/41 (44%)</td>
<td>6/41 (15%)</td>
<td>18/41 (44%)</td>
<td>6/41 (15%)</td>
<td>8/41 (20%)</td>
</tr>
<tr>
<td>Non-compiling strategies</td>
<td>4/15 (27%)</td>
<td>3/18 (17%)</td>
<td>0/6 (0%)</td>
<td>0/18 (0%)</td>
<td>0/6 (0%)</td>
<td>0/8 (0%)</td>
</tr>
<tr>
<td>Surprising strategies</td>
<td>0/15 (0%)</td>
<td>0/18 (0%)</td>
<td>0/6 (0%)</td>
<td>18/18 (100%)</td>
<td>0/6 (0%)</td>
<td>7/8 (88%)</td>
</tr>
<tr>
<td>Compilation errors</td>
<td>0</td>
<td>15</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Added lines</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Deleted lines</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Changed lines</td>
<td>7</td>
<td>22</td>
<td>3</td>
<td>20</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>∑ added lines</td>
<td>7</td>
<td>22</td>
<td>3</td>
<td>21</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>Δ lines of DemeterJ code</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+1</td>
<td>+9</td>
<td>0</td>
</tr>
<tr>
<td>Δ lines of Java code</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+220</td>
<td>-19</td>
</tr>
</tbody>
</table>
we state our expectations and interpret the results.

The number of affected strategies shall give some evidence about how representative a scenario is. This number also gives a rough indication on how many adaptive operations are affected by a scenario. If we consider reuse of traversal strategies among adaptive operations, the number of affected operations is greater than or equal to the number of affected strategies. Since our scenarios were constructed to cover a high percentage of traversal strategies, the resulting numbers are expectedly high.

The measures of non-compiling strategies and surprising strategies shall suggest how uncomplicated the evolution under a given scenario progresses. The lower these numbers the less source code has to be adapted. Surprising traversal strategies also point out how ‘risky’ the application of a certain scenario may be. By ‘risky’ we mainly refer to broken functionality which cannot be verified easily.

Interestingly, we did not notice any non-compiling strategies for the categories B and C. In case of category C, an explanation could be that additional classes or associations increase the set of possible traversal paths of a traversal strategy. Thus, the corresponding traversal graphs should even become larger. We actually expected that category A and B would contain non-compiling strategies since names of classes and class roles appear both in class graphs and traversal strategies.\textsuperscript{3} Obviously, the succinct way of specifying traversals in AP reduces the work needed to adapt traversal strategies under categories A and B.

The results look two-fold in case of surprising strategies. Either we counted no surprising strategies, or we observed that almost all affected strategies were disturbed ($B_2$ and $C_2$). Regarding the latter, we put the blame on the cyclic nature of our class graph. In that environment surprising strategies will arise, for example, if cycles are added ($C_2$) or constrained class subgraphs, i.e. traversal graphs, are modified such that some of the path constraints lose their effect ($B_2$). In our opinion, it should be much simpler to design traversal strategies, which are robust against surprising traversals, for acyclic class graphs than for cyclic ones.

The number of compilation errors – from compiling the generated code – gives some weak answers on how well the evolution of behavioral code (adaptive operations and ordinary methods) goes along with structural evolution (classes and traversal graphs). The errors that occur during final compilation are usually located in the wrappers of adaptive visitors or in method bodies. AP treats such ‘building blocks’ as black boxes. It does not inspect the content of these blocks and, therefore, cannot assist the evolution process here. The number of compilation errors throughout all scenarios except $A_2$ is next to nothing. We think this results from the consequent application of the Law of Demeter.

\textsuperscript{3}We did not experiment with different name mappings between class graphs and traversal strategies. As far as we know the used version of DemeterJ does not fully support other name mappings than the identity mapping.
However, as $A_2$ signals, compilation errors are not avoidable. Class names, for instance, appear frequently in building blocks as well as in some other places of the source code. Therefore, we must locate and correct the code at the affected places manually, which can take considerable time. In this regard, AP’s condensed form of declaring entire class graphs in one single class dictionary is immensely useful.

We now come to the total number of modified lines of code. That measure is calculated from the numbers of added, deleted, and changed lines. It shall indicate how much work was actually invested in order to adapt the code base for a certain scenario. Unfortunately, the results vary too strongly to draw sound conclusions. For the categories A and B, the adaptive work mainly consists of substituting old names by new ones – a process that may be automated. Category C does not require too much work either. If we leave out surprising strategies, which should rarely occur in conjunction with cycle-free class graphs, most of the time is spent for adding the code of new classes or associations. All in all, the obtained numbers of modifications were surprisingly low compared to our a priori estimates.

We also examined how the size of the generated, object-oriented code ($\Delta$ lines of Java code) evolves in relation to the size of the adaptive source code ($\Delta$ lines of DemeterJ code). Unfortunately, our method of counting the lines of code before and after a scenario was applied does not return adequate results. The main critique is that if source code lines are just changed but not added or deleted, then the difference of the total number of code lines is zero. This does not reflect the real number of changed lines in both source and generated code.\footnote{We applied scenario B1 again to determine how many lines of Java code were actually modified. Using the \texttt{diff} utility we found that 50 lines had been changed.}

The probably best method to compare AP and OOP regarding software evolution and maintenance would involve the evaluation of an adaptive and a purely object-oriented version of the same system. Our evolution scenarios would then be applied to both versions. However, the development and evaluation of a second system prototype comes along with significant costs. Such a comparative study is done by Roskosch [45]. We therefore do not pursue this idea further.

\section*{6.2 Experiences with Traversal Strategies}

Although traversal strategies seem very mature and well defined to us, their application opens some pitfalls. In this section we will mention a couple of problems, or say oddities, that should be considered when designing traversal strategies. Some of the stated problems are rather awkward than serious. However, we find them worth to discuss here. Beside the problems themselves, we will also provide examples and solutions.
CHAPTER 6. EVALUATION OF ADAPTIVE PROGRAMMING

6.2.1 Recursive Traversal Strategies with Negative Path Constraints

Traversal strategies do not handle recursion well. In [1] Lieberherr states that cyclic objects are ‘harder to reason about’ and ‘harder to manipulate because of the danger of infinite loops.’ The problem covered in this section shall illustrate how the semantics of traversal strategies are disturbed by recursion. In the following we demonstrate how negative path constraints are misinterpreted in conjunction with cyclic class graphs.

6.2.1.1 Example

Let us assume the concrete CDG from Figure 6.4a for this example. This class graph contains two cycles $\gamma_1 = A \xrightarrow{b_1} B \xleftarrow{a_1} C \xrightarrow{a_2} A$ and $\gamma_2 = A \xrightarrow{b_2} B \xleftarrow{a_2} C \xrightarrow{a_1} A$. We want to traverse the $\gamma_1$ cycle only. Since both cycles just differ in one place, i.e. the $b_1$ and $b_2$ associations, we would intuitively come up with the traversal strategy from Figure 6.4b.

Paradoxically, if we apply the given class graph to the given traversal strategy, we will not obtain a traversal graph corresponding to $\gamma_1$ but a traversal graph with $\gamma_1$ and $\gamma_2$ as shown in Figure 6.5. How could this happen?

The TSG edge $e_1$ constrains traversal paths correctly. It excludes all edges other than $A \xrightarrow{b_1} B$. The problem must hence be caused by $e_2$. In an acyclic class graph $e_2$ would expand to traversal paths of the form $B \rightarrow v_1 \rightarrow \ldots \rightarrow v_{n-1} \rightarrow A$ where $v_1, \ldots, v_{n-1} \in V \setminus \{A, B\}$, i.e. the path vertices $v$ between the source class $B$ and target class $A$ include all classes $V$ except the source and target classes.

In cyclic class graphs the same class may appear several times in a traversal path because it is visited once per recursion step. In our example $e_2$ expands to traversal paths of the form $B \rightarrow C \rightarrow A \rightarrow B \rightarrow \ldots$ and not into $B \rightarrow C \rightarrow A$. If a traversal is at $A$, it can freely choose the $b_1$ and $b_2$ association to proceed with since $e_2$ does not specify any negative path constraints. That is why our traversal strategy fails to produce correct $\gamma_1$ traversals.

6.2.1.2 Solution

There is no universal solution for the recursion problems of traversal strategies. We found that traversal strategies on cyclic class graphs are more likely to be bloated with structural details than traversal strategies on acyclic class graphs. That makes them harder to evolve adaptively.

However, a specific solution for our example is given in Figure 6.6. The idea is to break up the recursion for $e_2$ by removing all outgoing edges of $A$. The resulting traversal graph only contains the $\gamma_1$ cycle. Note that this solution does not prevent traversals from getting caught in infinite loops.
Figure 6.4: Cyclic CDG (a) and TSG (b). The traversal strategy represented by the TSG in (b) shall traverse to all classes \( B \) of the CDG in (a) that are reachable from class \( A \) through the association \( A \rel b_1 \rightarrow B \).

Figure 6.5: Traversal graph of the CDG and TSG from Figure 6.4.

Figure 6.6: Correct TSG.
6.2.2 Role-Dependent Traversal Strategies

In OOP a class can play different roles based on the context which it appears in. The original Visitor design pattern is not suitable to attach role-dependent behavior to classes because control is dispatched to a visitor method in accordance with the runtime type of the visited class. The runtime type of a class does not necessarily express the role of a class, though. The adaptive visitors presented in this thesis are not different to the original Visitor pattern in that regard.

In addition to the specification of role-dependent behavior, we also want to define role-dependent traversal strategies. Such traversal strategies infer possible traversal paths considering the role of classes instead of their type. The traversal strategies from Section 2.3 are not able to accomplish that.

6.2.2.1 The Inlaws Example

The inlaws problem formulated by Werner [46] is a prominent example showing a weakness of visitor-style programming. Figure 6.7 illustrates the problem. A person can be married to a spouse and can have a number of siblings. An inlaw is a sibling of the spouse or the spouse of a sibling. We want to find all inlaws of a certain person.

The class Person takes four roles in the inlaws example:

1. the start person who we want to find all inlaws from,
2. the spouse of the start person,
3. a sibling of the start person, and
4. an inlaw of the start person.

To find all inlaws of a person, a traversal would have to traverse the siblings of the spouse of the start person and the spouse of each sibling of the start person. Obviously, every traversal path to an inlaw must have a length \( l = 2 \). After the first step \( (l = 1) \), the traversal would stop at the spouse or a sibling of the start person. The difficulty lies in the fact that the traversal must decide upon the role how to proceed further. If it stops at a spouse, then the traversal must...
proceed to the siblings, and vice versa. After the final step ($l = 2$), the traversal would reach an inlaw.

Ordinary traversal strategies cannot infer traversal paths based on different roles. Therefore, they are inadequate for the given problem.

### 6.2.2.2 Solution

According to Lieberherr $et$ $al.$ [47], the inlaws problem for traversal strategies can be solved by extending traversal strategies in order to express roles of classes. Figure 6.8 displays Lieberherr’s approach in detail. Instead of having just one TSG node `Person`, the extended TSG accommodates a separate node for each role. Thus, combined with a few changes to the traversal algorithm, traversals are theoretically able to consider additional role information.

### 6.2.3 Traversals through Collections

Class dictionaries do not support the definition of cardinalities by default. Instead a fixed number of cardinalities is predefined by class dictionaries. Altogether we are able to declare $1 : 0..1$, $1 : 1$, $1 : n$ and $1 : 1..n$ associations. We see this limitation as secondary since those four cardinalities already cover the majority of uni-directional associations.

A stronger burden is the realization of one-to-many associations ($1 : n$ and $1 : 1..n$). The expressiveness of one-to-many associations is added by repetition

---

5 Basically, a traversal has to map roles to objects as described by the extended TSGs. In the inlaws example the first occurrence of a `Person` object would be considered a `start` person, the next occurrence of Person would be interpreted as `spouse` or `sibling` depending on which TSG edge was expanded, etc.

6 Role-dependent traversal strategies are not implemented yet.
classes and edges in AP. This approach does not seem natural to us since additional repetition classes place abstract, technical entities, which are actually not related to the domain in any way, into the class graph. So, under circumstances we may have to worry about the repetition classes when designing traversal strategies.

6.2.3.1 Example

The difference between class graphs with true cardinalities and class dictionary graphs (CDGs) is shown in Figure 6.9. In the CDG we notice a repetition class \( R \) in between the classes \( A \) and \( B \) that – in connection with the dashed edge (optional repetition relation) – expresses a \( 0..n \) multiplicity for \( B \). In other words, \( A \) references a collection of \( B \)s. Contrary to the UML version, the edge \( A \xrightarrow{b} B \) is not existent in the CDG. If we nonetheless have to constrain the \( b \) edge in a traversal strategy, then one of the variants \( A \xrightarrow{b} R \) or \( A \xrightarrow{b} * \) should be used instead.\(^7\)

6.2.3.2 Solution

We propose to abolish the concept of repetition classes and to introduce cardinality constraints which are directly attached to relation edges. This should improve AP’s flexibility and intuitiveness with regard to class graphs. As a consequence, true cardinality constraints would also make the optionality constraints of AP, as expressed by optional construction edges and optional repetition edges, obsolete. An optional edge could then be expressed by a \( 1 : 0..1 \) cardinality.

\(^7\)The asterisk notation should be preferred. It does not fix a specific target class and thus improves adaptiveness.
6.2.4 Traversals through Inherited Associations

In the object-oriented world public or protected associations declared in a superclass are also visible in its subclasses. In traversal strategies the common parts of alternation classes, i.e. the associations of abstract superclasses, cannot be accessed by subclasses directly. That circumstance slightly reduces the intuitiveness of traversal strategies.

6.2.4.1 Example

The class graph in Figure 6.10a models an abstract superclass $A$ which is related to a class $C$ via the uni-directional association $A \xrightarrow{ci} C$. Additionally, there is a subclass $B$ of $A$ that inherits the $ci$ association from $A$ and declares another association $B \xrightarrow{c} C$.

The curiosity is that $B$ is able to send messages to $C$ via the $ci$ association without problems. A method call would look like $ci.M$ where $M$ is method of $C$. However, in a traversal strategy it is not possible to traverse from $B$ to $C$ via the inherited association $B \xrightarrow{ci} C$. Given the concrete class graph from Figure 6.10a, the traversal strategy

$$\text{from B only-through } \rightarrow \text{ \rightarrow B,ci,C to C}$$

would not find a traversal path to $C$.

6.2.4.2 Solution

The reason is hidden in the class graph model behind traversal strategies. For better understanding we introduce the following two class graph models.
A semi-CDG is a CDG extended by inheritance edges. Inheritance edges represent the counterpart to alternation edges, i.e. inheritance edges always point to the reverse direction of alternation edges.

A flattened CDG is a CDG without alternation classes. It can be derived from any CDG by the CDG flattening algorithm [1, p 435]. This algorithm moves the common parts of alternation classes down the inheritance hierarchy to construction (sub)classes and removes all alternation classes and edges afterwards.

The traversal computation algorithm of traversal strategies is based on semi-CDGs. As we see, the semi-CDG in Figure 6.10a does not contain the inherited association $B \xrightarrow{c_i} C$ while the flattened CDG in Figure 6.10b does. Therefore, a traversal strategy will not recognize $B \xrightarrow{c_i} C$ as valid association.

To circumvent our traversal problem, we have to add instructions to traversal strategies that ask them to look for inherited associations. The previous traversal strategy could be rewritten as follows.

1. from B via A only-though \rightarrow A,ci,C to C
2. from B only-through \{ \rightarrow A,ci,C \} to C

Solution 1 works reasonably well for the given example. However, it would fail if $C$ were a subclass of $A$. Therefore we recommend Solution 2, which restricts traversal paths from $B$ to $C$ to include inheritance edges and the $c_i$ association only.

We tend to argue that an association should be directly traversable from a class as long as it is visible to that class. The approach would prefer flattened CDGs. Otherwise, semi-CDGs preserve the originally intended class structure much better than their flat counterparts. We have shown that traversals over inherited associations in semi-CDGs can be constrained by Solution 2 in a simple manner.

### 6.3 A Comparison

In this section we draw a comparison between the benefits and drawbacks of AP. Here our main focus is on technical matters. General arguments, such as ‘cutting-edge programming paradigms require expensive training for developers and are thus not worthy for business projects’ may be decisive for IT managers but have no particular value for Software Engineering. Hence, they shall not flow into this comparison.

Moreover, too specific or implementation-dependent arguments are not discussed either. The DemeterJ tools showed several issues that make developing
Table 6.3: The pros and cons of AP at a glance.

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separation of structural and behavioral concerns improves the</td>
<td>Recursive programming is not well supported by traversal</td>
</tr>
<tr>
<td>readability and clarity of code.</td>
<td>strategies.</td>
</tr>
<tr>
<td>Visitor-style programming with explicit traversal specifications</td>
<td>Evolution of too general traversal strategies often leads to</td>
</tr>
<tr>
<td>opens an intuitive way to implement and maintain complex operations.</td>
<td>surprising results. The ridge between adaptiveness and</td>
</tr>
<tr>
<td>Higher abstraction level leads to smaller source code.</td>
<td>robustness is narrow.</td>
</tr>
<tr>
<td></td>
<td>Error tracing is somewhat complicated.</td>
</tr>
</tbody>
</table>

in AP experimental. One could probably write several pages about the deficiencies and bugs present in current AP technology. But, that is not the point. AP is a research topic and not a product.

Table 6.3 gives a brief overview of our findings. We do not claim completeness of the arguments presented here.

6.3.1 Benefits

6.3.1.1 Separation of Concerns

AP is an advocate of the multi-dimensional separation of concerns principle because it decomposes the two concerns class graph and behavior in a loose manner. As a close relative of aspect-oriented programming [48], AP is also open for other aspects. For example, DemeterJ can factor out the synchronization and remote invocation aspects [49,50]. The advantages of a decomposition into aspects lie in the localization of related code. Class dictionaries exclusively declare classes and their relationships while behavior files deal with the algorithms on those classes only. If some piece of code has to be changed, then the impact of that modification will likely be bound to a single decomposition unit.

The decomposition units of AP are not scattered throughout the code base unlike in conventional OOP. Thus, the localization also improves the readability and clarity of code. Class dictionaries provide a compact overview of the class graph, traversal strategies succinctly define the collaborating classes of complex operations, and adaptive visitors specify the functionality of complex operations in one place.
6.3.1.2 Visitor-Style Programming

Visitor-style programming is an intuitive way to implement complex operations. Traversal strategies in conjunction with adaptive visitors help to keep the administrative overhead of this programming style low. So, AP users may fully concentrate on the collaborating classes and their role in a complex operation. Traversal strategies and visitor classes can be specified explicitly or implicitly. The former allows traversals to be better combined with different visitors. It supports the re-usability of both traversal strategies and visitors. The latter neatly unites a traversal strategy and a visitor within an inlined, adaptive operation. This enables even trivial operations to be implemented in visitor-style.

6.3.1.3 Abstraction Level

AP’s higher abstraction level leads to smaller code in comparison with conventional OOP. Large parts of the underlying object-oriented code are automatically generated. Table 6.4 gives a short overview of what code pieces are derived from which language construct. All in all, a high percentage of source code in AP is only concerned with the actual problem and not with tedious, frequently repeating tasks.

The legal model of ADONIS has reached a size of about 3.700 lines of DemeterJ code. The generated Java code adds up to nearly 31.000 lines. Even if we take a noticeable overhead in the generated code into account, this relation clearly indicates an improved understandability of the system in favor of AP.

6.3.2 Drawbacks

6.3.2.1 Recursive Programming

Traversal strategies do not pay particular attention to cyclic class graphs. It is possible to design traversal strategies that let traversals get stuck in infinite
looms. Lieberherr denotes non-terminating recursions as ‘bad recursions’. We suggest some solutions that prevent from unwanted recursion in the following.

1. A class cycle in a traversal graph can be broken by applying negative path constraints. Such constraints remove nodes (classes) or edges (relations) from the cycle. Consequently, it breaks into a non-cyclic graph.

2. Traversals in AP are performed depth-first. We could modify the traversal algorithm\(^8\) such that no object is visited twice. The approach is based on node coloring. Only those objects (nodes) of the object graph are visited that have not been colored yet.

3. Around-wrappers in adaptive visitors can prune traversal paths independently of the computed traversal graph. This feature may be utilized to implement stop-criteria for recursions.

Unfortunately, each solution is just a compromise. Solution 1 causes traversal strategies to become more complex because for each cycle there must be added one path constraint at least. But, sometimes there is no simple way to specify a traversal strategy in a clean, adaptive manner.\(^9\) Solutions 2 and 3 do not touch any traversal strategy, but the computed traversal code might not be as space-efficient as the code from Solution 1. The difference is that Solution 1 prunes static traversal paths at compilation time while the other approaches work at runtime. Solution 3 requires explicit handcoding to stop recursions, but in contrast to Solution 2 it does not need modifications in the AP runtime environment.

The difficulties of traversal strategies to handle cyclic class graphs have side-effects on the design of class graphs itself. A smart rearrangement of classes and relations in a CDG can simplify the construction of traversal strategies considerably. By smart we chiefly mean to reduce the number of circular classes according to the Law of Demeter for Classes [1].\(^{10}\) However, not all object-oriented models can be reorganized in this way. Some domains inherently contain cyclic structures that should not be crippled by CDG style rules.

### 6.3.2.2 Over-Adaptivity

Another problem of AP is identified by Spit [51]. The evolution of a class graph can break an adaptive operation in two ways. On the one hand, the new class graph becomes incompatible with the adaptive method. In that case the traversal graph computation would fail. On the other hand, the new class graph is

---

\(^8\)See the Traverse algorithm from Section 2.3.4.

\(^9\)See also Section 6.2.1.

\(^{10}\)Note that the Law of Demeter for Classes is different from the Law of Demeter (for functions) discussed in Section 2.1.2.
still structurally compatible, but the adaptive operation produces invalid results. This is what Spit terms ‘over-adaptivity’.

Over-adaptivity is caused by too general traversal strategies. When designing a traversal strategy we must reach a compromise between generalization and specialization. The traversal should be specific enough to visit all collaborating classes in right order, but it should also be general enough to evolve with the class graph. A good compromise fulfills both criteria. A too general traversal strategy compiles very well with different class graphs. But, on the other hand, chances are high that it introduces so called ‘surprise paths’, i.e. paths that were not intended. Unfortunately, the recognition of misbehaving traversals with surprise paths is not always trivial.

In general, AP aims at the principle of least modification. After changing a class graph, an adaptive program should require as little modification as possible in order to run on the new class graph. This contradicts somehow with the principle of least surprise which says that broken traversal semantics induced by changing class graphs should be localized easily.

6.3.2.3 Error Tracing

Error tracing in adaptive programs is complicated. This problem is common to all preprocessing or generative approaches. The reason lies in the fact that the source code is transformed into some form of intermediate code before it is compiled into machine-executable code. Compilation errors of the latter stage as well as runtime errors are both linked against the intermediate code. They cannot be traced back to the source code by conventional means. So, locating such errors in the source code must be done manually.

---

11 The compilation process in DemeterJ is actually three-fold. First, the class dictionary and behavioral specifications are transformed into weaver files containing instructions about how to arrange certain Java code fragments. These code fragments are then woven into valid Java source files by an aspect weaver. In the third stage, a Java compiler produces executable byte code.
Chapter 7

Conclusions

In this thesis we have introduced a legal advisory system that is based on AP. We have demonstrated how the principles of AP can be used in the modeling and implementation of legal texts in order to obtain object-oriented legal models that are fairly robust under changing environments. The development process of ADONIS supplied us with valuable insights into AP that have flowed into an evaluation of this novel programming paradigm. Finally, we have identified some benefits as well as weaknesses of AP.

In general we can recommend AP for developing any kind of object-oriented system except apart from commercially critical projects. We find this paradigm especially suitable for projects related to compiler construction. In this special field AP can utilize features like automatic parser generation and visitor-style programming best. Actually, this is not astonishing if we consider that the Demeter compilers are being implemented with AP itself and that the development on these tools has influenced research on AP to some extent. Although AP shows its advantages mainly on large-scale systems comprising hundreds of classes, we find that it is also applicable in much smaller projects.\(^1\) Here, the higher abstraction level and the generative features can save development time and costs considerably.

Regarding our initially stated hypothesis, we take a mixed position. Generally speaking, we agree that AP positively affects evolvability and maintainability. The results of the case study from Section 6.1 partially support this argument. However, we still have to bear two interrelated problems, recursive traversals and over-adaptivity, in mind that may account for a reduced evolvability as well as maintainability.

The class hierarchy of ADONIS is inherently cyclic. Not only that this circumstance causes the design of traversal strategies to be fairly complicated, it

\(^1\)We have successfully used AP for the OGDL and FSM implementations introduced in Appendix A and B respectively.
also bears the risk of an unpredictable evolution of traversals guided by these strategies. In ADONIS, for example, we have reached a point where the fear to obtain misbehaving traversals (due to surprising strategies) slowly outweighs the benefits of the principle of least modification. We think that this unfortunate situation was mainly caused by three factors:

- human mistakes in the design of our traversal strategies,
- harder reasoning about traversals on cyclic class hierarchies, and
- the experimental development status of the Demeter tools itself.

A lot of effort has to be put into these issues. Thus, we should not generalize the problems yet. As a matter of fact, AP has proven very valuable in connection with some other minor projects carried out by us and a couple of success stories reported by the Demeter Research Group. Perhaps, we should conceive traversal strategies and adaptive visitors primarily as an elegant way to implement complex operations involving several collaborating classes. The fact that these operations are quite robust under evolving class hierarchies is welcome but secondary.

In conclusion we can state that AP is definitely a suitable technology for implementing evolutionary systems. We have indicated that there are some profound problems in conjunction with traversal strategies on cyclic class hierarchies. However, AP offers other benefits that go beyond traversal strategies, e.g. a broad range of generative features. From our personal point of view, AP’s approach to spread complex operations across collaborating classes is worthy not only for intentionally adaptive applications but also for ordinary object-oriented programs, which just want to take advantage of a very efficient and intuitive form of visitor-style programming.

7.1 Outlook

We started the development on ADONIS from scratch, and the system has not entered the maintenance phase yet. Therefore, we could only give some presumable answers on maintenance matters until now. Surely, our ‘invented’ evolution scenarios may approximate reality more or less but they cannot substitute further studies under real maintenance conditions. As a consequence, we have to carry out more in-depth investigations into AP’s influence on adaptive maintenance. The work on ADONIS as an evolutionary system is not going to be finished in the near future.

Additional research is needed for the transformation of our UML-based formalism into AP source code. Here, we find that a few transformation processes could be automated; especially the mappings for OCL constructs and state diagrams seem to be straightforward. In our opinion, model constraints in OCL
and operation protocols as state diagrams are just two other aspects that might be integrated into DemeterJ as it was done with the synchronization aspect (COOL) \cite{49} and the remote invocation aspect (RIDL) \cite{50}. DemeterJ’s built-in aspect weaver would then be responsible to place the code for checking model constraints or controlling operation protocols in the right places. This approach\textsuperscript{2} would certainly decrease development and maintenance time further. It would also make the development and maintenance of adaptive legal advisory systems less error-prone.

Another topic to follow is how AP integrates into component architectures such as CORBA, EJB (Enterprise JavaBeans), or COM (Component Object Model). The trend goes to software that will be assembled from entire components of objects instead of single classes. From our perspective it will be interesting how generative approaches, such as AP, can be used to build and assemble such components. In this context more recent developments of the Demeter Research Group like adaptive plug-and-play components (APPCs) \cite{52} or aspectual components \cite{53} might be worth to look at.

\textsuperscript{2} Aspect-oriented programming is based on the separation of concerns principle.
Appendix A

Object Graph Description Language

In this section we present a solution for the description and initialization of complex object graphs. Our approach is based on DemeterJ’s application specific languages (ASL) from Section 2.2.3. In comparison to ASLs, our solution is not solely restricted to the initialization of tree-like object graphs, but it can also be applied to cyclic object graphs. We name this solution Object Graph Description Language (OGDL).

OGDL may be used in any Java 2 application, even though it shows particular strength in conjunction with DemeterJ-enabled applications. The OGDL interpreter is a separate package written in DemeterJ. The main features include:

- instantiation of classes by invoking the default constructor,
- instantiation of classes (composites) by parsing ASLs (DemeterJ only),
- invocation of public instance methods, and
- assignment of associations (1 : n associations for DemeterJ only).

OGDL takes advantage of the fact that each object graph can be clustered into subgraphs that are completely tree-like. Such subgraphs can be initialized using DemeterJ’s built-in ASLs. Moreover, OGDL is able to label each object in the object graph with an unique identifier. Identified objects can be linked together. As a consequence, we are able to successively instantiate and link objects. A welcomed side-effect of this solution is that at least the object subgraphs keep their adaptiveness property.

OGDL is very simple; it has not built in any control structures. So, the only flow of control is sequential. The connection between the OGDL interpreter and the resulting object graph relies completely on the reflection capabilities of the Java programming language.
A.1 Syntax

ObjectGraphDescription ::= ( Statement )* "end"
Statement ::= Assignment | Invocation | Comment
Comment ::= "//" Line
Assignment ::= SingleAssignment | MultipleAssignment
SingleAssignment ::= ReferencedObject "." FieldName "=" SimpleObject
MultipleAssignment ::= ReferencedObject "." FieldName "+" 
"{" SimpleObjectList "}"
ReferencedObject ::= Ident
Invocation ::= CreatedObject
SimpleObject ::= CreatedObject
CreatedBy ::= ( ConstructedObject
| ParsedObject
| MethodInvocation
| ( "as" ReferencedObject )? )
ConstructedObject ::= "new" ClassName
ParsedObject ::= "parse" ClassName "(" ObjectDescription ")"
ObjectDescriptor ::= StringDescriptor | FileDescriptor
StringDescriptor ::= String
FileDescriptor ::= "file:" String
MethodInvocation ::= ReferencedObject "." MethodName
"(" ( ArgList )? ")"
ArgList ::= ReferencedObject ( "," ReferencedObject )*
SimpleObjectList ::= SimpleObject ( "," SimpleObject )* 
ClassName ::= Ident ( "." Ident )* 
FieldName ::= Ident ( "." Ident )* 
MethodName ::= Ident

A.2 Semantics

An ObjectGraphDescription contains a list of Statements followed by the end key-
word. The OGDl interpreter processes that list one by one starting from the begin-
ing. There are basically two groups of statements, Assignments and Invo-
cations. Let us omit Comments whose only purpose is to improve the readability
of object graph descriptions.

An invocation may either call the public default constructor (ConstructedObject),
a special class method (ParsedObject), or any public method of an object in-
stance (MethodInvocation). Common to each invocation is that it returns an ob-
ject as return value. Special attention should be drawn to parsed objects. De-
meteJ allows to define grammars for the object representation (ObjectDescriptors)
of classes in class dictionaries. These grammars are used to parse and initialize objects from their object descriptors. The classes of parsed objects must declare a static class method `parse(String)` which returns the initialized object instance according to the supplied object descriptor.¹

Before we examine assignments, let us have some words about how object instances are addressed in OGDL. We know that an object may be referenced several times in an object graph. Thus, a mechanism to uniquely name objects is needed. In OGDL, every object that is returned by an invocation instruction can be named by an unique identifier. This is done by attaching the directive ‘as <identifier>’ (where <identifier> must be replaced by some identifier) to the invocation instruction. Once an object is announced to the interpreter in this way, we may reference the object by providing its identifier.

Last but not least, assignments certainly form the most important part of OGDL. Without assignments we could create single objects, perhaps composite objects, but not cyclic object graphs. In OGDL we distinguish between two assignment types. **SingleAssignments**, on the one hand, assign exactly one object to an object attribute. **MultipleAssignments**, on the other hand, assign a list of objects to an object attribute. The latter, though, may only be used in conjunction with 1:n associations as expressed by AP’s repetition relations.

### A.3 Interpreter

The OGDL interpreter is started by invoking the static method `process(InputStream)` of the class `Initializer`, which takes an object graph description as argument. During runtime there may occur one of the three exceptions: `IOException` on problems reading the input stream, `ParseException` in case of a syntactically wrong object graph description, or `InitException` on errors regarding object graph initialization. A hash-map containing identifier-to-object mappings will be returned on successful completion.

¹DemeterJ generates the `parse` method automatically.
Appendix B

A Finite State Machine Implementation

When the development on ADONIS began, we were looking for a way to implement a Mealy automaton as introduced in Section 4.1.2. Preferably that automaton should be configurable by the UML state diagrams from Section 3.2.3.

The transformation of state diagrams into executable code is a solved problem. Good CASE tools offer convenient code generation for state diagrams that is targeted at various object-oriented programming languages. But none of them supports the DemeterJ language at the moment.\footnote{The only CASE tool supporting DemeterJ is AP Studio, which is distributed with the DemeterJ development tools [5]. AP Studio looks promising for the construction of class dictionaries and traversal strategies. However, it is not meant to cover the full range of UML specification techniques.} Moreover there are a few specialized compilers for state diagrams, such as iState [54] for instance. They are not applicable for our purpose either because of their lacking AP support.

Eventually, we decided to develop a solution of our own. The basic motivation behind that step was to find a transparent way that enables us to coordinate the actor-initiated operations of processes without having to hardcode too much control logic into the legal model. We wanted to separate the control logic defined by Mealy automata from the actual operation code as good as possible. The reason for that is simple. In our legal models state diagrams are primarily used to define operation protocols. They exactly specify under which circumstances (or situations) a certain operation is open for invocation requests. We strongly believe that operation protocols and operation behavior are two distinct concerns of ADONIS. Thus, they should be separated.

The solution described here consists of a textual notation and a corresponding interpreter. The interpreter is written in DemeterJ and comes as a separate
package. It includes the following main features.

- **Conditional execution of protected code fragments**. Such code fragments are enclosed by open and close statements. A protected code fragment widely corresponds to the action of a transition. The execution of protected code fragments will be rejected if the related transition is disabled.

- **Notification of currently enabled transitions**. Such ‘open’ transitions refer to executable, protected code fragments.

- **Emission of signals on state change**. That means, the environment of an FSM is notified about state changes.

- **Emission of messages about internal activities or decisions**. Custom messages are fired when an FSM enters/leaves a particular state or begins/ends a certain transition. All messages can be directly specified in our FSM notation.

## B.1 Shortcomings

Our FSM implementation is tailored around the special needs of ADONIS. We neither intended to make it feature complete with respect to UML state diagrams, nor do we see our FSM implementation as a general-purpose tool. Hence, our notation only covers a small subset of the expressiveness of state diagrams. The most important restrictions are itemized below.

- **No state activities**. All activities taking place during execution of an FSM shall reside in transition actions.

- **No intraprocess concurrency**.

- **No compound states**. Instead we use an extension mechanism for refining diagrams. So, we are able to add more specific states and transitions for ‘compound’ states.

- **Only two event kinds: call events and instantaneous events**.

- **Call transitions should define an actor who is permitted to call the specified operation**. Otherwise, such a transition is considered as system transition and used for interprocess concurrency.

- **Two instantaneous transitions with the same source state shall never be enabled at the same time**. Two or more enabled instantaneous transitions would result in a non-deterministic FSM.

\(^2\)All actor-initiated operations contain protected code fragments.
Guards shall not access argument values. That means, a guard should only evaluate the properties of the assigned class or the classes reachable from the assigned class. If it is necessary to check arguments, then OCL preconditions should be used.

The operations of call transitions and actions of instantaneous transitions shall refer to public operations of the assigned class. The operations of call transitions must include at least one argument for the invoking actor, and instantaneous actions must reference operations without any arguments.

B.2 Notation Syntax

```plaintext
StateMachine ::= "statemachine" QualifiedClassName
   ( "extends" <StateMachineRef> )?
   "{" StateMachineBody "}"
StateMachineRef ::= QualifiedClassName
StateMachineBody ::= ( State )+
State ::= "#" Ident
   ( "onEntry" Message )?
   ( "onExit" Message )?
   ( Transition )*  
StateRef ::= "#" Ident  
Transition ::= "->" StateRef
   ( "onStart" Message )?
   ( "onEnd" Message )?
   ":" TransitionBody  
TransitionBody ::= CallTransitionBody | InstantaneousTransitionBody  
CallTransitionBody ::= MethodInvocation
   ( "[" Guard "]" )?
   ( "by" ActorRef )?  
InstantaneousTransitionBody ::= "[" Guard "]"
  ="/" MethodInvocation  
MethodInvocation ::= VoidMethod ( "(" Message ")" )?
Guard ::= ( "not" )? BooleanMethod  
ActorRef ::= QualifiedClassName  
VoidMethod ::= Ident  
BooleanMethod ::= Ident  
QualifiedClassName ::= IdentSegment ( "." IdentSegment )*  
IdentSegment ::= Ident  
Message ::= String ( "[" ArticleList "]" )?
ArticleList ::= Article ( "," Article )*  
Article ::= NumberedArticle | UnnumberedArticle  
NumberedArticle ::= "§" Integer
   ( "(" ParagraphItemList ")" )?
```
APPENDIX B. A FINITE STATE MACHINE IMPLEMENTATION

Act

UnnumberedArticle ::= Ident Act
ParagraphItemList ::= ParagraphItem ( "," ParagraphItem )* 
ParagraphItem ::= Paragraph | ParagraphRange
Paragraph ::= Integer
ParagraphRange ::= Integer "-" Integer
Act ::= Ident
Appendix C

Specification and Source Code

The UML specification of the legal model and the source code of ADONIS are available separately [55].
Bibliography


BIBLIOGRAPHY


[46] M. Werner, 1996. Personal communication to the Demeter seminar at Northeastern University, Boston, MA.


