MASTER

Towards a generic structure editor for annotated documents

van den Boom, S.R.

Award date:
2009

Link to publication
Towards a
Generic Structure Editor
for Annotated Documents

by
Serge van den Boom

Supervisor:
dr. ir. Tom Verhoeff

Eindhoven, August 2009
Contents

1 Introduction ......................................................... 4
  1.1 Context ....................................................... 4
  1.2 Outline of the work ........................................... 4
  1.3 Guide to this report .......................................... 5

2 Concepts ......................................................... 6
  2.1 Editor ......................................................... 6
  2.2 Structure ..................................................... 7
  2.3 Annotation .................................................... 8

3 Goal ............................................................. 9
  3.1 Properties .................................................... 9
     3.1.1 Flexible and widely applicable (genericity) ............ 9
     3.1.2 Maintainable and extensible (reusability) ............. 9
     3.1.3 Suitability as a teaching tool .......................... 10
     3.1.4 Industry usability ...................................... 10
     3.1.5 Tie in with other SET projects ......................... 10
  3.2 Steps ......................................................... 11

4 Related Work .................................................... 12
  4.1 Eclipse ....................................................... 12
  4.2 Eclipse IMP .................................................. 12
  4.3 The ASF+SDF Meta-Environment ............................. 13

5 Conceptual Design .............................................. 14
  5.1 Introduction ................................................ 14
  5.2 The traditional approach .................................... 14
  5.3 An alternative approach ..................................... 15
  5.4 The Sead approach ......................................... 15
     5.4.1 Handlers ................................................. 17
     5.4.2 Traits ................................................... 21
     5.4.3 Annotations ............................................ 22
     5.4.4 Related designs ....................................... 22
List of Figures

2.1 The general overview of an IDE and related concepts . . . . . . 7
5.1 A traditional representation of an abstract syntax tree . . . . 15
5.2 An overview of document type support in Sead . . . . . . . 17
5.3 Representation of an abstract syntax tree in Sead . . . . . . 18
5.4 A simple hierarchy of node types in Sead, with handlers . . 19
5.5 The process through which handlers are invoked for operations 19
6.1 The relationship between Document, Nodes, Annotations, and AnnotationRoles. . . . . . . . . . . . . . . . . . . . . . . . 24
6.2 The relationship between Nodes and NodeTypes . . . . . . 25
6.3 The relationship between traits and handlers . . . . . . . . . 26
6.4 A partial hierarchy of handlers and HandlerData . . . . . . 28
9.1 The abstract syntax tree for the example scenario . . . . . . . 45
9.2 Editing an AST in the proof of concept editing component . . 46
Chapter 1

Introduction

1.1 Context

This report describes the work done as part of my graduation project for the Computer Science (IN5) program at Eindhoven University of Technology. It was carried out within the Software Engineering & Technology (SET) group of the Department of Mathematics & Computer Science, under the supervision of dr. ir. Tom Verhoeff.

Within the SET group there is an ongoing project VIDE which aims to create a verifying IDE, i.e. an integrated development environment which can help create provably correct software. I have carried out my work with a possible application within that project in mind.

1.2 Outline of the work

My graduation project concerns the design of the basis for a structure editor for annotated documents, where the software helps the user to create valid and correct (substructures of) documents. In this report, I present my design for a generic representation of structured documents, which incorporates several novel ideas. I have implemented the core of this design, which can serve as a framework for future work. I have also built a simple editor plugin for the Eclipse framework based on this core.

The distinguishing features of this design are the representation of the document and document type by generic objects which are decorated with objects orthogonally providing additional behaviour.

I use the name Sead (for Structure Editor for Annotated Documents) for my work towards creating this structure editor for annotated documents.
1.3 Guide to this report

In chapter 2, I describe the concepts which are relevant for the idea of what I mean by a Structure Editor for Annotated Documents. These concepts will be used throughout this report, and as such it is important that this chapter is not skipped.

In chapter 3, I describe the original goal of my project. While this goal proved to be overly ambitious, it does show the context in which I have carried out my work. Apprising oneself of the contents of this chapter is recommended to fully understand the design decisions which I made, and which are documented in later chapters.

The chapters 5, 6 and 8 together form the core of this report.

In chapter 4, I describe related projects which may be interesting to those who want to know more about the subject matter.

In chapter 5, I describe my design in terms of concepts and the rationale behind it, without going into implementation details, which I describe in chapter 6.

In chapter 8, I describe how support for a new document type can be added, building on the concepts described in chapter 5.

In chapter 9, I present a scenario which shows the concepts of my design in action. This chapter may also serve as an additional example to some of the concepts presented in chapters 5 and 6.

Finally, I present the conclusion of my work in chapter 10.
Chapter 2

Concepts

My work revolves around the creation of (parts of) a Structure Editor for Annotated Documents. I will here explain what I mean by this, by first looking at the various parts of the title, starting with “Editor”.

2.1 Editor

The purpose of an editor is to provide its user with the means to interactively create and modify documents. These documents can be stored on and retrieved from permanent storage. Most editors are designed to edit a single type of document. In the case of a text or hex editor, the document is treated as a sequence of characters or bytes, respectively. If the editor is aware of the underlying structure, we speak of a structure editor. I will elaborate on structure editors a bit more in the next section.

In my work, I am working towards the creation of an editor which can operate on many different types of documents. A key concept here is genericity. The generic applicability of the structure editor is reflected in the genericity of the design.

Related to the editor is the integrated development environment (IDE). An integrated development environment is software which consists of several components to help in the development of software. The central component, from the perspective of the user, is an editor. Other common components are build tools (including a compiler and linker), a debugger, source refactoring facilities, and revision control system clients. The VIDE project (see section 1.1) aims to add components to (help) verify the correctness of program texts, and to (help) derive program texts which are correct by construction (with respect to some specification).

For my work, this is relevant in that the proof of concept implementation of my ideas is an editor component for the Eclipse IDE, and that the facilities provided by my design can be used by other components to inspect and manipulate the structure of a document.
Figure 2.1 gives an overview of a modern IDE and related concepts, built on the Model-View-Controller architectural pattern [12]. My work concerns the editor component, in particular the model part of it, though I have also made some code for the user interface in the proof of concept editing component for Eclipse.

![Integrated Development Environment (IDE)](image)

Figure 2.1: The general overview of an IDE and related concepts. The core of my work is in the grey part, but I have also made some code for the user interface in the proof of concept editing component for Eclipse.

### 2.2 Structure

The concept of *structure* comes into play in the following senses:

- structured documents, and
- structured editing

A **structured document** is a document with some specific underlying structure limiting what elements may occur in which places. This may be purely a syntactic structure (i.e. context free syntax), or include semantic constraints (i.e. context-dependent properties). While in a sense any document can be considered a structured document — sequences of bytes, sentences of words of letters — we will reserve the term for documents with
a more complex structure. A structured document can be defined by its structural elements, and the rules (invariants) which govern where each element may occur. Some of those rules may be purely syntactic, while others — depending on the type of document — may involve semantic information. In the rest of this report, unless otherwise specified, the term document will refer to a structured document. We consider a structured document as an instance of a specific document type. A document can be a valid instance of more than one document type. For instance, XHTML is XML, most C code will be valid C++ code, and an instance of some propositional logic may be considered as an instance of a first-order logic. A structure editor will however typically regard only one document type of a document at a time.

When we talk about structured editing we are referring to the editing of structured documents. The two aspects which come in here are safeguarding the integrity of the structure of the document, through input validation, or rewriting or quarantining of incorrect input, and structure-aware visualisation. While my proof of concept contains very simple input and visualisation capabilities, this is not a central point of my work. However, what I have provided are basic facilities which can be — and in my proof of concept, are — used in the implementation of structure-aware input and visualisation.

2.3 Annotation

Associated with a structured document can be annotations. When I use this term, I am referring to meta-information associated with (elements of) a structured document. Examples are a formal proof for the correctness of a computer program, a refinement history in a computer program, or plain unstructured inline documentation.

An annotation may be structured itself. This could be the same structure as of the document proper, but this will usually not be the case. An annotation sharing parts of the structure of the document is also possible; for instance, the same expressions may be used in both a computer program and a proof of its correctness.

Annotations can also contain derived information. Examples are the context in a programming language, type information, or references to elsewhere in the document. These annotations may only exist as information attached to objects in the abstract syntax tree of a document in the memory of a structure editor, and do not have to be present in the document when stored on a permanent medium.
Chapter 3

Goal

The goal of the project is the creation of the basis for a structure editor for annotated documents.

3.1 Properties

The desired properties of the structure editor are:

- flexible and widely applicable (genericity)
- easily maintainable and extensible (reusability)
- suitable as a teaching tool
- usable in industry
- tie in with other SET projects

3.1.1 Flexible and widely applicable (genericity)

The applicability of the editor should not be limited to documents of a few built-in document types. The particular properties of the type of the document being edited should as much as possible be runtime parameters to the editor.

3.1.2 Maintainable and extensible (reusability)

The software is intended to be built upon by further extensions. For this purpose, it is important that the software is designed with extensibility in mind, and that the software is easy to maintain.
3.1.3 **Suitability as a teaching tool**

The software should be built with the possible use as a teaching aid in mind, specifically for teaching programming at Eindhoven University of Technology.

General properties to make the editor well-suited as a didactic tool include:

- having an intuitive interface, so that the overhead of learning to use the software is minimised. Specifically, the functionality and interface should be consistent, and as simple as possible.

- requiring little prior knowledge; it should not count on the user having specific knowledge about notation, functionality of IDEs, programming languages, etc. in advance.

- detailed feedback (if desired); if something goes wrong, try to explain why this happened.

Specifically to the Eindhoven school of program design, the editor should enable the use of stepwise refinement techniques [7, 8, 17].

3.1.4 **Industry usability**

The software which we aim to develop is not intended to be just another research toy. It should be a useful tool for the development of documents, specifically software. While a full-fledged IDE is beyond the scope of this project, the editor should at least provide a suitable, extensible, basis. In addition, by choice of a suitable framework to build upon, existing functionality may be used.

3.1.5 **Tie in with other SET projects**

Within the Software Engineering & Technology group of the Department of Mathematics & Computer Science of Eindhoven University of Technology, there is an ongoing effort to create a “Verifying Integrated Development Environment” (VIDE). My work could be used in the editor and infrastructure for this project. Even if this editor is only intended to work on a single document type, that document type has not yet been defined. And especially when the document type is still under development, the generic applicability of the structure editor would be advantageous.

Another related project within the SET group is Cocktail [11], a tool for program derivation. Providing the user interface for a program derivation tool is one application of the structure editor. For this purpose, the backend of Cocktail could be used with the front-end of the structure editor. As Cocktail is written in Java, writing the structure editor in Java would be an advantage to enable this interfacing.
Another research topic within the Software Engineering & Technology group is that of *model-driven engineering*, where software is developed using models of a high level of abstraction, which are then transformed into models of a lower level of abstraction, until a result is reached which can be executed or interpreted by a computer. These models are described in structured documents, which need to be edited in some way. A structure editor able to recognise these types of documents would be very useful. And as above, when the document type is still under development, a structure editor which is generic in applicability would be particularly advantageous.

3.2 Steps

To achieve this goal, my original plan involved performing the following steps:

- compiling an inventory of the exact requirements and wishes for the functionality of the software;
- examining of similar projects;
- evaluating the usability of existing frameworks;
- designing and implementing the framework;
- performing experiments with the result.
Chapter 4

Related Work

Other works exist which share some of the ideas with my design, solve the same problems in another way, or are otherwise related to my work. In this chapter, I compare Sead to a few of these related works.

4.1 Eclipse

Eclipse [9] is an extensible IDE written in Java. Support for multiple document types, mostly programming languages, is present via plugins. However, unlike Sead, this is not done in a generic way. While some shared library packages exist, which can be reused, the code supporting one language will have to do a lot of the same work which is done by the code supporting other languages.

The proof of concept for Sead is implemented as a plugin for Eclipse.

4.2 Eclipse IMP

Eclipse IMP [10] is an “IDE Meta-Tooling Platform”. It aims to make it easier to develop support for new programming languages in Eclipse. This is done through tools which generate (parts of) the code for the components which are used by Eclipse. It is different in this respect from Sead, where generic code supports a wide variety of document types. Both IMP and Sead use a plain-text description of parts of the document type, but for IMP this is used to generate code, while in Sead it is used as a parameter to the generic code.

Another difference is that Sead’s design does not restrict itself to programming languages; it could be applied to any type of structured document.
4.3 The ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [13, 4] is a “framework for language development, source code analysis and source code transformation”. It uses SDF [15, 16] to describe the syntax of a document type, and ASF [2, 6] to define transformations on documents. A component named ApiGen [1] can be used to generate C or Java code from SDF files to create the data structures describing the document. When applied to a document type, ApiGen would provide the API for use in, among other things, a structure editor. The ASF+SDF Meta-Environment uses the ATerm library to represent the abstract syntax tree of documents via generic nodes, in a similar way as I do in Sead (see section 5.4.4).

The ASF+SDF Meta-Environment is very generic in design and can work on a wide range of document types. In the ASF+SDF Meta-Environment, semantics are described as transformations on documents. This is a useful way to extract facts and to rewrite documents. For use in a structure editor however, a way is needed to associate behaviour with the elements of a document — for instance to specify how invariants can be maintained when modifications are made to a document. Code would have to be written especially for this, and if the code generated by ApiGen is used for this purpose, it will be restricted to one type of document (though libraries could be used to share code as much as possible).

The idea of extending elements of a document with behaviour, as is done in Sead, does not conflict with the approach of the ASF+SDF Meta-Environment, but could work supplementary to it.
Chapter 5

Conceptual Design

5.1 Introduction

Central to the design of Sead — as I have named the Structure Editor for Annotated Documents which is the eventual aspiration of this work — is the goal of genericity. Most importantly, this means that the editor should be able to work on a wide range of (types of) documents. Each type of document comes with its own internal relations, and as a structure editor, Sead needs to be aware of these. Before we look at how we can implement this in a generic (document type independent) way, we first examine how structure editors commonly deal with these kinds of semantic properties of a document type.

5.2 The traditional approach

A common way in which support for specific types of documents is implemented in structure editors, is by creating classes for each of the structural elements. Each class contains code specific to that element. The instances of these classes form the nodes of the abstract syntax tree representing a document. Figure 5.1 shows as an example a graphical representation of the abstract syntax tree for the lambda expression \( \lambda x.(\lambda y.(x \ y)) \). Duplication of code may be avoided through the use of inheritance, or by building on a library of common functionality. This approach however is not generic. If support for a new document type is to be added, classes for the new structural elements of a document need to be defined and implemented. For each new document type, new code will need to be used, which will have to be trusted by the user. This is how language support in IDEs like Eclipse works.
5.3 An alternative approach

An alternative approach is to describe the type of the document in a separate description file, and then to automatically generate the classes for the various structural elements of the type of document from this description, supplemented by manually written code to effect the behaviour specific to the document type. This eliminates some of the work of the implementor of support for a document type, and reduces the risk of introducing errors, but it still requires new code to be trusted for each document type. IMP [10], the IDE Meta-Tooling Platform for Eclipse, uses this approach. Also Kees Hemerik’s FoolProof [14], which represents the document in three layers of abstraction (context free syntax, added bindings, and added document-type specific properties), uses this approach.

5.4 The Sead approach

In Sead, I take a different approach. As in the previous approach, there is a separate description file describing the syntactic properties of the document type. What distinguishes my approach from the previous one is the way in which the semantic properties are handled.

What I am doing in Sead, is to divide the semantic properties of a document type into parts which are as small as possible. Each of these parts pertains to an invariant of (part of) a document type. I then create code which guards or reestablishes these invariants for various operations on
(part of) a document. These pieces of code I call handlers. As an example, when the document is a Java file, there may exist one handler to guard the invariant “an identifier is defined in its context” when a new identifier is inserted into the document. Apart from guarding invariants, handlers may also be used to extract information from the document. For example, in the case of a program, a handler may be written to generate the proof obligations for a node.

The document type description file then describes declaratively which handlers are in effect for each type of node in the abstract syntax tree. Since many types of documents share the same conceptual properties, it is possible to reuse these handlers for many different types of documents. In fact, if a sufficiently extensive library of handlers is available, it may be possible to add support for a new document type without adding any code at all.¹

Figure 5.2 shows an overview of document type support in Sead. As Java code, there are the document type independent core of Sead, and document type dependent handlers. In non-executable plain-text files, we have the reusable fragments of document type descriptions, and document type descriptions for various languages. This consists of a syntactic (context-free) part, described directly in these document type descriptions, and a semantic part, described by (indirect — see section 5.4.2) references to handlers.

Unlike in traditional approaches, the abstract syntax tree which represents a document does not consist of different types of objects for each type of node. As the behaviour of nodes is provided by handlers which decorate them (via the node type objects), a single type of object suffices. This allows the representation of different types of nodes without the introduction of new Java classes (hand-written or generated), which would be the case in traditional approaches.

Figure 5.3 shows the abstract syntax tree of the same lambda expression as shown in figure 5.1. The same tree is recognisable, but instead of objects which are instances of classes which are specific to the type of the node, in Sead the objects which form the abstract syntax tree are instances of Node. The types of the nodes are represented by separate NodeType objects, shown in grey. The handlers, which add behaviour to the nodes, are not shown in this diagram.

To allow for reuse of code, node types can inherit behaviour (i.e. handlers) from other node types. A hierarchical structure of node types is formed, similar to the hierarchical structure of classes in an object-oriented programming language. Sead also supports multiple inheritance. The relations between the NodeType objects used in the lambda expression example

¹Even without the need to add any code, an implementor of this design may still decide to use generated code in their design to implement parts of it. For instance, to parse a document, the software may generate a dedicated parser, based on the grammar from the document type description, and then load and use the resulting object code.
are shown in figure 5.4. The NodeType objects are shown in grey. At the top we have a node for the root type; every NodeType has this NodeType as its topmost ancestor. Derived from this NodeType is a NodeType representing an expression. Derived from this are three more NodeTypes, representing a function definition, function application, and an identifier. In white, we see some handlers attached to the various NodeTypes. Some NodeTypes have more than one handler, one has none. Handler 5 is used by more than one NodeType.

5.4.1 Handlers

Example

Handlers are invoked as a result of a variety of triggers. We describe the process by which this is effected in the following section. This is done with the help of an example, which we introduce in this section.

When adding content to a document, there will be restrictions on the new content, based on the document type and the existing content. To
be able to provide the user with possible suggestions for new content, the structure editor will need to know what can be inserted at a specific place in a document. For this, it needs to know which types a new node may have for it to be allowed to be inserted at a specific place in the abstract syntax tree representing the document. In the proof of concept, this information is used directly — the user can select the type of a new node to be inserted in a hole in the abstract syntax tree representing a document.

Node type suggestion — as I will call the process of determining which types of nodes may be used as a new node at a specific location in a document — is document type specific, which makes it a task for handlers. Some node types may be suggested by handlers, while some may be forbidden entirely. Instead of requiring the handlers to generate an exhaustive list of forbidden node types, we split the node type suggestion in two parts: 1) generating suggestions, and 2) removing suggestions. Because all suggestions need to be generated before suggestions can be removed, these parts need to be performed in two successive stages. Each stage has its own type of handlers.
Both handlers get an object as parameter containing a set of suggestions. This set is initially empty. As part of the contract of these functions, we require that the handlers of the generating kind — called in the first stage — only add suggestions to this set, while the handlers of the removing kind — called in the second stage — only remove suggestions from the set.\footnote{This is not currently enforced in the code.}

The process of invoking handlers

The process through which handlers are invoked is illustrated in figure 5.5.

The invocation is organised in the following way:

1. **Initiator** — The process begins when an operation is initiated. This is usually an action performed by the user in the user interface of the
structure editor. But other means of activation are also conceivable, such as a timer running out, a packet arriving on the network, or a message arriving from an external tool. In our example, as implemented in the proof of concept, the operation is initiated when the user right clicks on a hole node, and selects “add Node”.

2. **Controller layer** — This initiating action will cause code to be activated through the usual means (an event handler, message pump, etc.). This code is part of the controller in the Model-View-Controller [12] architectural pattern. It will gather the objects relevant to the operation, and pass these along to an *activation function*. In our example, this code may figure out where in the abstract syntax tree the editing takes place, use this information as an argument to the specific *activation function* for generating node type suggestions, and then finally present the suggestions to the user.

3. **Activation layer** — In the next step, the *activation function* — which can be considered part of the model in the Model-View-Controller pattern — will actually start performing the operation on the specified parameters. The parts of this operation which are independent of the type of document, it will carry out itself; for those which are not, it will call *distribution functions* in the node type objects to which the operation applies. It will pass along as parameters the types of the handlers to call, and the data to those handlers. In our example, the *activation function* will consecutively call a *distribution function* to generate suggested completions, then a *distribution function* to filter out disallowed suggestions, and finally sort and return the result to the caller.

4. **Distribution layer** — The *distribution function* — which is a member of the object representing the types of nodes, and which is hence also part of the model — will then invoke all the handlers of the specified type, passing along the supplied data. It does this by first invoking the handlers associated with the parent node type object, and then invoking the handlers directly associated with the node type object itself, as specified in the document type description for the type of document being edited (see chapter 8). This code is entirely independent of the document type and the semantics of the handlers to be called, but which handlers will be called — determined by the parameter supplying the type of those handlers — *is* specific to the document type.

---

3Other calling orders are conceivable, and may have their use; further experiments should be performed to determine what the best order is, or whether a multi-phased approach may be more suitable.
5. **Handler layer** — It is then up to the handlers to perform the document type specific part of the operation.

**Types of handlers**

Handlers are conceivable for many kinds of document type specific operations. I list here a few possible uses of handlers. Note that only a few of these are implemented at this time.

Some operations modifying the structure of the document are:

- creating a single new node
- destroying a single node
- inserting a node in the abstract syntax tree
- removing (detaching) a node from the abstract syntax tree

Some operations providing information to the structure editor are:

- determine syntax highlighting
- suggest node types for a location in the abstract syntax tree
- validate allowed node types at a location in the abstract syntax tree
- suggest values for nodes (e.g. identifier names)
- parsing of a string as a value of a node
- generate the proof obligations for a node in the abstract syntax tree of a program text
- generate template or boilerplate document fragments
- generate a simplified version of a sub-tree of the abstract syntax tree

5.4.2 **Traits**

While the handlers are associated directly with the node types in Sead’s internal representation, there is an intermediate layer in the document type description files. In these files, a concept of traits is used. Traits group together handlers which take part in realising a common abstraction. For instance, a Trait associated with the node type for a procedure body in a programming language may represent the concept “this Node establishes a new context”. This trait may then group together handlers for updating or inspecting the context on operations like inserting a node, removing it, code completion, etc.
In the document type description files, traits are assigned to node types. These files (or fragments included by them) define each trait as a group of handlers. While this is currently not implemented, traits should also be able to include other traits in this group.

5.4.3 Annotations

One key concept of Sead is that of “annotations” associated with nodes. In section 2.3, I mentioned that these annotations may contain derived information, which is not included in the representation of a document when in permanent storage. Typically, this information will be generated by a handler which needs to store additional information with a node. This information in principle takes the form of a tree of the same generic nodes which make up the structure of the document proper. However, situations are conceivable where a structure of nodes is not sufficient to capture the information which is to be stored in an annotation. For these cases, there is also the option to attach an arbitrary object to an annotation. This object will then only have a meaning to specific handlers. Using this option should however be avoided where possible, so that the infrastructure which is in place for nodes can be reused as much as possible.

Because there may be multiple annotations associated with a node, I need some way to distinguish them. Because the same types of nodes can be used in different annotations, the node types of the nodes in the annotation can not serve this role. Specifically for this reason of distinguishing annotations, I have added the concept of an “annotation role”. Handlers and other parts of Sead can find and distinguish annotations by these annotation roles.

5.4.4 Related designs

The ASD+SDF Meta-Environment [4], a framework for language development, source code analysis and source code transformation, uses the ATerm library to represent abstract syntax trees. This is a library for “annotated terms”, where, like in Sead, all “Nodes” are generic objects, and can have associated annotations. These annotations are trees of ATerms themselves. Similarly, in Sead, the annotations proper are trees of “Node” objects. While in my prototype I used my own classes for the nodes in the abstract syntax tree representing a document, I could imagine an implementation of the ideas behind Sead which uses ATerms for this purpose.
Chapter 6

Code Design

In this chapter, I describe the design of the code, by outlining the components which make up the software, and how these relate to each other. I also expand on the design decisions I made while developing this design.

6.1 Description of classes

We can distinguish the classes which make up Sead into the function which they have in the Model-View-Controller [12] architectural pattern.

In the model we have the classes which represent documents, document type descriptions, their parts, as well as some general purpose classes. The design of the model forms the core of my work. These classes can form the basis of future work.

The view and controller are not as clearly separable as the model, because the visual components which are used to visualise (parts of the) document, are also used to manipulate it. I created these classes purely for the purpose of demonstrating the core of my work. These classes should not be used beyond this prototype.

Furthermore, there are some classes which are used as placeholder for other necessary components of a practical product. These build up some data structures which can then be used in other parts of the software. These classes are not needed beyond this prototype. However, they form a good illustration of the interface of the core (i.e. the model part) of the software.

In the following sections, I will describe the classes in more detail.

6.1.1 The model

The model is made up of:

- the classes describing a document being edited, i.e. `Document`, `Node`, and `Annotation`,
• the classes describing the type of a document, i.e. `DocumentType`, `NodeType`, `AnnotationRole`, `Trait`, `IHandlerData`, and

• their manager classes `NodeTypeManager`, `AnnotationRoleManager`, `TraitsManager`, and `HandlersManager`, as well as

• a few classes which are not specific to either document or document type.

We now describe the notable classes of each of these groups.

The Document

The following classes are used to represent a document and all of its constituent parts. See also figure 6.1.

**Document**

A Document object contains all the information pertaining to an (annotated) document being edited in Sead. A Document object contains the abstract syntax tree for the document, through a reference to the root Node. It references a DocumentType describing the type of the document being edited.

![Figure 6.1: The relationship between Document, Nodes, Annotations, and AnnotationRoles.](image_url)

**Document** A Document object contains all the information pertaining to an (annotated) document being edited in Sead. A Document object contains the abstract syntax tree for the document, through a reference to the root Node. It references a DocumentType describing the type of the document being edited.
Node  The Node class is used to represent nodes in the abstract syntax tree of a document. Associated with each Node object is the NodeType object for the type of the node. Each Node has an ordered collection of zero or more Nodes as children. Also, each Node, apart from the root node, has one Node as a parent. A Node has a reference to the Document of which it is a part. A Node may have Annotation objects associated with it. No ordering of these objects is specified.

Annotation  An Annotation object represents an annotation with a Node. To represent the information contained in the annotation, it can itself have an associated Node as the root of another abstract syntax tree. An Annotation has a specific role, which identifies the purpose of the annotation. This role is represented by an AnnotationRole object associated with the Annotation. For the situation where information needs to be stored which cannot be captured by a structure of Nodes, an annotation object may additionally contain a reference to an arbitrary object, the meaning of which is dependent on the AnnotationRole.

The Document Type Description

The following classes are used to represent the type of a document. See also figure 6.2.

Figure 6.2: The relationship between Nodes and NodeTypes.
**DocumentType**  The document type describing the document being edited is represented by an object of the **DocumentType** class. It is the owner of a **NodeTypeManager**, **HandlersManager**, and **TraitsManager**. A **DocumentType** object can be shared by multiple **Documents**.

**NodeType**  The **NodeType** class represents the type of a **Node**. Associated with each **NodeType** are the handlers to maintain the invariants corresponding to (part of) a **Trait**. These handlers are objects of classes implementing the **IHandler** interface. There is only one **NodeType** object for each type of node; all nodes of the same type refer to the same **NodeType** object. **NodeType** objects for a document type are created and tracked by a **NodeTypeManager**, which is owned by one **DocumentType**. Each **NodeType** has a reference to this **NodeTypeManager**. **NodeTypes** may share handlers with other **NodeTypes** via an inheritance relation. A single **NodeType** may have multiple **NodeTypes** as base types.

**AnnotationRole**  An **AnnotationRole** describes the role (purpose) of an **Annotation**. There is only one **AnnotationRole** object for each role which an annotation can have; all annotations which have the same role share the same **AnnotationRole** object. **AnnotationRole** objects are created and tracked by an **AnnotationRoleManager**.

**Trait**  A **Trait** object groups together a number of **IHandler** objects which belong together conceptually. **Traits** are used only during the construction of the **DocumentType**. After this, only the **IHandler** objects are referenced, by the **NodeType** objects. See figure 6.3.

**IHandler**  The **IHandler** is a generic interface for classes implementing the concept of a “handler” (see section 5.4), adding behaviour to **Nodes** (via **NodeTypes**).

There are two interfaces which directly extend **IHandler**, viz. **IHandlerTest** and **IHandlerAction**. **IHandlerTest** is used to check whether an action is allowed in the current state. If the action is allowed, **IHandlerAction** is used to update the state when the action is actually performed.

These two interfaces each have a single method, **test()** and **action()** respectively. When I talk about “invoking” a handler, this is in the Java
code implemented as a call to this method. Note that the use of objects for handlers is necessitated by the fact that you cannot create references to methods in Java. In a language like C++, a handler could be implemented as a pointer to the function doing what our test() and action() methods do.

The test() and action() methods only have a single argument, which is an object of a class implementing IHandlerData.

For each action on the abstract syntax tree which can be performed in the editor, there are interfaces specific to this action extending the IHandlerTest and IHandlerAction interfaces, to be implemented by handlers intended to participate in this action.

Figure 6.4 shows a partial hierarchy of handlers and HandlerData. Above the dotted line are the generic interfaces IHandler, IHandlerTest, and IHandlerAction. These interfaces are parameterised by a data argument implementing IHandlerData. Shown directly below the line are two example interfaces for specific types of handlers, viz. INodeMoveOk and INodePreMove. For these interfaces, the parameter is instantiated to NodeMoveData. At the lowest level are then the two classes IsVariableInContext and RemoveVariableReference, implementing INodeMoveOk and INodePreMove respectively.

IHandlerData IHandlerData is a generic interface for the arguments to the test and action methods of IHandlerTest and IHandlerAction handlers. The specific classes implementing these interfaces use an object of a class implementing IHandlerData as parameter for their test() or action() method.

Managers

The document type description contains various NodeType, AnnotationRole, Trait, and IHandler objects. To keep track of these, I have defined several Manager classes. These managers make it possible to look up the various objects by some identifying value. They also serve as a factory [12] for each of said objects. Each NodeType, AnnotationRole, Trait, and IHandler object is registered with exactly one NodeTypeManager, AnnotationTypeManager, TraitsManager or HandlersManager, respectively. Exactly one instance of each of those these manager classes is associated with a DocumentType.

Manager The Manager class is the base class for all other manager classes.

NodeTypeManager A NodeTypeManager object is responsible for creating and tracking all the NodeType objects in a DocumentType.

---

1This has only been partially implemented in the prototype.
Figure 6.4: A partial hierarchy of handlers and HandlerData. Above the dotted line are the generic interfaces, below are interfaces and implementations for specific types of handlers.

**AnnotationTypeManager**  A AnnotationTypeManager object is responsible for creating and tracking all the AnnotationType objects in a DocumentType.

**TraitsManager**  A TraitsManager object is responsible for creating and tracking all the Trait objects in a DocumentType.

**HandlersManager**  A HandlersManager object is responsible for creating and tracking all the Handler objects in a DocumentType.

**Various**

There is one more notable class in the model which does not fall in the categories above, the ASTException class.
**ASTException**  

ASTException is the base class of exceptions that may be thrown by the various classes dealing with the operations on the abstract syntax tree of a document. In particular, the `test()` method of handlers (indirectly) implementing `IHandlerTest` may throw exceptions of (a subtype of) `ASTException`.

### 6.1.2 The View

In the view we find the classes `ASTTreeContentsProvider`, `ASTTreeLabelProvider`, `MultiPageEditor`, and `MultiPageContributor`. These form the basis of the user interface of the proof of concept editing component, building upon the Eclipse, SWT and JFace libraries. As visualisation and interacting are done using the same user interface elements, these classes also perform some controller functionality.

**ASTTreeContentsProvider**  
The `ASTTreeContentsProvider` class implements the `org.eclipse.jface.viewers.ITreeContentProvider` interface defined by the JFace framework. It contains the functions which translate the Nodes of an abstract syntax tree into a form which the JFace `TreeViewer` uses to generate the structure of the graphical tree component.

**ASTTreeLabelProvider**  
The `ASTTreeLabelProvider` class implements the `org.eclipse.jface.viewers.ColumnLabelProvider` interface defined by the JFace framework. It contains the functions which generate the labels (and optional images) for the nodes of the graphical tree component, from the Nodes in an abstract syntax tree.

**MultiPageEditor**  
The `MultiPageEditor` class extends the Eclipse `org.eclipse.ui.part.MultiPageEditorPart` class, and defines the editing component of the user interface. In it, one user interface tab is defined, containing the tree displaying the abstract syntax tree of the document. In it, various event listeners are defined which interface with the controller, as well as code to update the user interface.

**MultiPageContributor**  
The `MultiPageContributor` class extends the Eclipse `org.eclipse.ui.part.MultiPageEditorActionBarContributor` class, and adds some menu items in Eclipse’s menu. Right now, it still consists almost entirely of Eclipse boilerplate code. In a fully functional product, more useful user interface functionality would be added here.

### 6.1.3 The controller

As mentioned above, the classes in the view also perform some controller functionality. The only other notable controller classes are the `ASTActions`,
LambdaTest and SeadPlugin classes.

ASTActions  The ASTActions class is a static class defining a number of generic methods operating on Nodes.

LambdaTest  The LambdaTest class contains a few static methods which build a DocumentType and a Document for the lambda expression example.

SeadPlugin  SeadPlugin is the class which defines the actual plugin. It extends the Eclipse class org.eclipse.ui.plugin.AbstractUIPlugin, and performs some plugin-wide initialisation.

6.2 Invoking Handlers

In section 5.4.1 we showed the process through which handlers are invoked. We will now explain the code which makes this happen.

We start by describing the interface IHandler:

```java
public interface IHandler<Data extends IHandlerData> {
}
```

As you can see, this is an entirely empty interface. It is only used as a supertype of all handlers. It is a generic interface, having one parameter which specifies the data associated with the handlers. For this most generic interface IHandler, the data parameter is only required to implement the most generic interface for data for handlers, IHandlerData. This interface is nothing more than:

```java
public interface IHandlerData {
}
```

There are two interfaces which extend IHandler, viz. IHandlerAction, and IHandlerTest, as described in section 6.1.1. These handlers serve a very similar function, but I still decided to use two different interfaces. The reason for this decision is discussed in section 6.3.3. These two interfaces look like this:

```java
public interface IHandlerAction<Data extends IHandlerData> extends IHandler<Data> {
    void action(Data data);
}
```

```java
public interface IHandlerTest<Data extends IHandlerData> extends IHandler<Data> {

```

30
void test(Data data) throws ASTException;
}

Here we declare the single method of each of these interfaces, action() for IHandlerAction, and test() for IHandlerTest. These take a parameter of the type which is the parameter of the interface itself. We see that the signature for the action() method is almost the same as for test(). The only difference is that the latter may throw an error. The reason for this is discussed in section 6.3.3.

Each interface extending the IHandlerAction or IHandlerTest interface still only has the action() and test() methods, with a single argument extending IHandlerData to contain the data for the specific type of handler. Why I have not chosen to give each handler its own method with its own arguments, I discuss in section 6.3.4.

We now show an interface for a specific type of handler, extending IHandlerAction. There will be many such interfaces, but we only show one, as an example. This is INodeTypeSuggestionFilter:

```java
public interface INoteTypeSuggestionFilter
extends IHandlerAction<NodeTypeSuggestionData> {
  void action(NodeTypeSuggestionData data);
}
```

This interface extends IHandlerAction, filling in the parameter to that generic interface with the type NodeTypeSuggestionData. The class NodeTypeSuggestionData is defined as follows:

```java
public class NodeTypeSuggestionData implements IHandlerData {
  final public Node parent;
  // Parent of the node for which suggestions
  // are pursued.

  final public int index;
  // Index in the parent for the child node
  // for which suggestions are pursued.

  final public Set<NodeType> suggestions;
  // Will contain the suggestions generated.

  public NodeTypeSuggestionData(Node parent, int index) {
    this.parent = parent;
    this.index = index;
    suggestions = new HashSet<NodeType>();
  }
}
```

This is the data object used in each INoteTypeSuggestionFilter handler (as well as in INoteTypeSuggestionGenerate). The purpose of handlers implementing INoteTypeSuggestionGenerate is to generate a list of suggestions,
and the purpose of handlers implementing `INodeTypeSuggestionFilter` is to filter out disallowed suggestions. These handlers use an object of type `NodeTypeSuggestionData` to store this list, in the `suggestions` attribute. The `parent` and `index` attributes furthermore tell these handlers for which child of which `Node` the suggestions are to be generated.

Next, we show an implementation of the interface `INodeTypeSuggestionFilter`, viz. `SuggestTypeFromSyntaxFilter`:

```java
public class SuggestTypeFromSyntaxFilter implements INodeTypeSuggestionFilter {
    public void action(NodeTypeSuggestionData data) {
        NodeTypeWithChildren parentType = (NodeTypeWithChildren) data.parent.getType();
        NodeType childType = parentType.getChild(data.index);
        Iterator<NodeType> iterator = data.suggestions.iterator();
        while (iterator.hasNext()) {
            NodeType suggestion = iterator.next();
            if (!suggestion.isSubTypeOf(childType)) iterator.remove();
        }
    }
}
```

This class implements the `action()` method, which goes through all of the types which have been suggested by a preceding call to `SuggestTypeFromSyntaxGenerate`, and removes those which are not a subtype of the required type of the node for which the suggestions are to be generated. It uses the data stored in its `NodeTypeSuggestionData` argument for this.

We now show how these handlers are invoked. First, we show the activation function described in section 5.4.1:

```java
public static Collection<NodeType> generateNodeTypeSuggestions(Node parent, int index) {
    // Get the type to invoke the handlers of:
    NodeType parentType = parent.getType();

    // Create an object with the data for the handlers:
    NodeTypeSuggestionData suggestionData = new NodeTypeSuggestionData(parent, index);

    // Invoke the handlers which wish to contribute
    parentType.callHandlerAction(
        INodeTypeSuggestionGenerate.class,
        suggestionData);
}
```
// Invoke the handlers which wish to veto suggestions:
parentType.callHandlerAction(
    INodeTypeSuggestionFilter.class,
    suggestionData);

// Sort the results:
List<NodeType> result = new Vector<NodeType>(
    suggestionData.suggestions);
Collections.sort(result, nodeNameComparator);

return result;
}

This function first creates a NodeTypeSuggestionData object, to be passed along to the handlers. Then it calls the distribution function (as described in section 5.4.1) callHandlerAction of the parent node of the type for which suggestions are pursued, to invoke all the relevant handlers. It does this twice: once with as argument the type “INodeTypeSuggestionGenerate”, and once with “INodeTypeSuggestionFilter”, so that first all of the handlers of the former class type are called, and then all of those of the latter class type. We make use of Java’s runtime type information capabilities for this. Finally, the results are sorted and returned.

Finally, we show the implementation of the distribution function NodeType.callHandlerAction:

public void
callHandlerAction(Class<? extends IHandlerAction
    <? extends IHandlerData>> handlerType,
    IHandlerData data) {
    // Call the handlers for all the base types.
    for (NodeType baseType : getBaseTypesRef())
        baseType.callHandlerAction(handlerType, data);

    // Call the handlers associated directly with this
    // type:
    for (IHandler<IHandlerData> handler : handlers) {
        if (!((handlerType.isAssignableFrom(
            handler.getClass()))))
            continue;
        ((IHandlerAction<IHandlerData>) handler).action(
            data);
    }
}

This method of NodeType first recursively calls itself for all of the base types of the NodeType, and then calls the handlers of the requested type which are associated with the NodeType itself.
6.3 Rationale

In this section, I will describe my rationale for various design decisions. I will do this by addressing various questions which may arise when one considers the design.

6.3.1 Why split actions into stages with their own handlers?

Some actions can be divided into multiple stages. For instance, when a Node is moved from one place in the abstract syntax tree to another, we may distinguish the removal of the node from one location, and the insertion in another location. By allowing separate handlers for insertion and removal, we make it possible to reuse these handlers. In this example, the handler for insertion of a Node during moving may also be used when a new Node is inserted, and when a Node is removed altogether, the handler for the removing step of a move may be reused. In general, dividing actions into successive stages allows for maximum reuse of handlers throughout multiple actions.

Also, by splitting the actions, we are able to invoke all the handlers for one stage of the action, before invoking any of the handlers for the next stage, separating concerns, which decreases the chance for interference between handlers.

Furthermore, for some actions, it is necessary to perform it in multiple stages. For instance, to generate the types of possible nodes which can be inserted in an abstract syntax tree (with holes), individual handlers may suggest certain types, while other handlers may forbid certain types. If we would only have one type of handler, and these handlers are invoked only once, each handler needs to generate both a collection of suggested node types, as well as an exhaustive collection of all of the types which it forbids, because each handler needs to be able to forbid types which are only suggested by a handler invoked later. If on the other hand, we split the action of generating node suggestions into a stage to generate a collection of suggestions, and a stage to filter out forbidden node types from this collection, all the suggestions are known before the choice has to be made to forbid a type, and handlers only need to examine the suggested types for forbidden ones.

For most actions, we even went so far as to allow handlers to be invoked before, and after the action proper is taken (e.g. pre-insertion and post-insertion).

In an earlier version of the software, I only had a single handler for an entire action. Its action() method would then be called multiple times, with a stage field set in the data object to specify at which stage of the action it is called. This had the advantage that fewer handlers were needed, but it would result in a lot of code duplication in the implementation of each
handler to check for which stage it was called. For genericity, I decided to abandon that approach and go with a separate handler for each stage, as is done now.

6.3.2 Why distinguish IHandlerTest and IHandlerAction?

For most actions we will want to allow handlers to veto an action, if the possibility exists that an action would violate an invariant which the handler is guarding. If a handler can both modify the state and veto the action, we could find ourselves in the situation where one handler vetoes the action after another handler has already modified the state. In this case the state modification would have to be reversed. To avoid this situation, we first give handlers the opportunity to veto the execution of an action, without modifying the state. Only if no handler vetoes the action, we allow handlers to modify the state according to the action taking place, at which time it is too late to veto the action. The semantics of handlers which test whether an action is allowed and those which actually modify the state, are significantly different. The former handlers will never change the state, and once one vetoes the action, the others do not need to be queried any more. The latter actions on the other hand should never fail, and should all be called. Because these categories of handlers are so different, and because this distinction is so often relevant, I have decided to create separate classes for each category. This allows for the introduction of extra sanity checks, some of which can be performed at run time. Specifically, when I am using the Java exception mechanism to report the vetoing of actions (see below), only the handlers allowed to issue that veto need to be able to throw exceptions.

In an earlier version of the software, I had a combined handler class, with both a method to test whether an action is allowed, and a method to actually perform the action. Since not every invocation of the action method needs to be preceded by an invocation of a testing method, I decided to abandon this approach. For instance, when moving a node, there are opportunities for handlers pre-removal, post-removal, pre-insertion, and post-insertion of the node, while the check whether the move is allowed, needs to be determined only once.

6.3.3 Why use Exceptions in IHandlerTest.test()?

To determine whether actions may be executed, the IHandlerTest class defines a method test(). Instead of using a boolean return value to signal acceptance or vetoing of the action, this method has a void return type, but may throw an ASTException to veto the action. The reason for this decision is that it may be desirable to supply extra information regarding the reason for the refusal. So what I need is to provide an object with an error status and a reason, which may cause the invocation of further testing methods to
be stopped. This is exactly what is offered by the exception mechanism.

6.3.4 Why use handlers with a single function with a single parameter?

In the final version of the software, the handlers have a method with a signature of the form `void test(Data data) throws ASTException;` (for `IHandlerTest` classes) or `void action(Data data);` (for `IHandlerAction` classes), where `Data` is a class implementing `IHandlerData`. In an earlier version of the software, the `test` and `action` methods had differing arguments depending on the specific purpose of the handler. The problem with this approach is that to call these methods for each handler, there needs to be separate code for each one of them. In the final approach, the data object is first created, after which a generic function is called to call all the `test` or `action` methods of the relevant `IHandlers`, with the same data object as its single parameter (see section 6.1.1). Note that this does not make the handlers less safe to use; each interface extending `IHandlerData` has its own constructor, with its own arguments which are checked at compile-time, as would the `test()` or `action()` method in the earlier design. There is a slight disadvantage though, in that in the implementation of `test()` and `action()`, the names of supplied “parameters” (which are in the data object) are not immediately apparent as they would be as real parameters. In modern IDEs this disadvantage is largely offset by features such as code completion.

6.3.5 Why use inheritance of NodeTypes?

I have chosen to introduce a form of inheritance for `NodeTypes`. Each `NodeType` may have multiple parent `NodeTypes`, whose handlers are inherited. Whenever handlers to respond on some action are to be executed for some `NodeType`, first the relevant handlers of the base `NodeTypes` of the `NodeTypes` will be invoked, and then the relevant handlers of the `NodeType` itself are invoked.

The reason for this inheritance is the same as with inheritance in object oriented programming, namely to avoid duplication; a handler can be attached to a base `NodeType`, instead of to each of the types derived from it.

6.3.6 Why use multiple inheritance of NodeTypes?

We allow multiple inheritance of `NodeTypes`, to enable maximum sharing of the behaviour of `NodeTypes`. We consider this worth the (slight) extra difficulties in the implementation of inheritance.

We remark that by supporting multiple inheritance, the so called diamond problem, where a single `NodeType` is inherited multiple times, via different ancestor `NodeTypes`, is an issue. As the inheritance relation is an
“is a” relation, it is desirable to consider duplicate ancestor \textit{NodeType}s only once. However, in the prototype, duplicate ancestors are currently not detected; if one \textit{NodeType} is used as an ancestor of another \textit{NodeType} via multiple ‘routes’, multiple copies of its handlers will be invoked when the action to which they respond is being executed. The implementation of the detection of duplicate ancestors and the ascertainment that each \textit{NodeType} is considered only once, is a future work.

6.4 Other remarks

6.4.1 Management of handlers in \textit{NodeType}s

In this prototype, all handlers associated with a \textit{NodeType} are stored in a single list, with no distinction to the type of the handler. Actions are conceivable which work on each node in the abstract syntax tree, and for large languages, in a fully featured editor, the number of handlers associated with a \textit{NodeType} may be large. In such a future editor, it may be worth implementing a more efficient management of handlers.
Chapter 7

Implementation

In this chapter, we give some more information about the implementation of the design presented in this report, as well as the proof of concept editor component.

7.1 Overview

The implementation at the time of writing consists of 62 Java files, for a total of 3638 lines, or 101051 characters. Each class is contained in its own .java file. The code has been written in Eclipse 3.4.2, and runs on Java 1.5.0.

7.2 Code organisation

We give here an overview of the source code produced in this project. The file counts of each package do not include the files in sub-packages mentioned elsewhere in this list, but the descriptions given relate to each entire package, sub-packages and all.

sead.controller (1 file) package containing the interfaces and classes relating to the controller in the Model-View-Controller [12] architectural pattern

sead.controller.editors (4 files) package containing the interfaces and classes responsible for the proof-of-concept editor component for Eclipse

sead.model (6 files) package containing the interfaces and classes relating to the model in the Model-View-Controller architectural pattern

sead.model.annotation (2 files) package containing the interfaces and classes relating to annotations
sead.model.handlers (0 files) package containing the interfaces and classes relating to handlers, both the generic interfaces, and the actual handlers

sead.model.<operation> (23 files in total) package containing the interfaces and classes for handlers relating to a specific operation

sead.model.nodes (12 files) package containing the interfaces and classes relating to nodes

sead.model.traits (2 files) package containing the interfaces and classes relating to traits

sead.utils (4 files) package containing universal utility classes

Apart from these packages, there are a few other classes worth mentioning:

sead.LambdaTest demonstration code to create the document type and document for the lambda expression example

sead.Resources string manager

sead.SeadPlugin class which defines the actual plugin

The following classes were used in earlier experiments, but are not currently used by any code, and are not included in the Eclipse project file:

sead.ATermLoader class for loading ATerms [3]

sead.Cocktail class for loading cocktail files
Chapter 8

Adding support for a new document type

8.1 Introduction

This chapter describes the steps which an implementor of a new type of document would have to go through to add support for a new type of document built on my design.

We can distinguish two parts which make up the support for a type of document:

- the textual description of the document type
- the code which brings about the functionality specified in the description

A textual description of the document type will have to be written for each document type. However, these description files can import (fragments of) other document type descriptions. Hence similar functionality can be reused, and a library of reusable fragments can be created. These document type descriptions specify the syntactic and semantic structure of the document type, declaratively.

The code consists of the handlers which effect behaviour on node types. As to which handlers are applied to nodes of which types is specified in the textual description. The intent is that these handlers are reused by multiple document types.

8.2 The document type description file

We explain the syntax and semantics of document type description files through an example. We will specify the document type for basic lambda
expressions. The complete document type definition file can be found in appendix A. In this section we will go through this file in parts.

We should note that currently no code exists to actually parse these document type description files. However, the methods which such a parser would call to construct the objects representing the document type description have been implemented, as well as proof of concept code which calls these methods in place of the parser, building the document type for the lambda expressions example as described in this section.

A document type description file consists of four parts:

1. definitions of traits
2. definitions of the types of the nodes which will make up the abstract syntax tree describing the document
3. a specification of which traits are in effect for which node types
4. a specification of how the input file is parsed into nodes, in the form of a grammar with constructor statements

In addition to this, a document type description file may import definitions from other files. This is what we start with, to import the definitions from context.dtd and nodetypesuggest.dtd. These files define the traits which we use further on in this description.

# Make the types and traits relevant to contexts available. %import "context.dtd";
# Make the traits relevant to node type suggestion available. %import "nodetypesuggest.dtd";

The # character is used to start a comment. The %import keyword is used to import the definitions from the specified file.

In the nodetypesuggest.dtd file, we can find statements like this:

%deftrait sead.traits.editing.suggestTypesFromSyntax
  sead.model.handlers.nodeSuggestion.SuggestTypeFromSyntaxGenerate,
  sead.model.handlers.nodeSuggestion.SuggestTypeFromSyntaxFilter;

This defines the trait sead.traits.editing.suggestTypesFromSyntax to consist of the Java classes sead.model.handlers.nodeSuggestion.SuggestTypeFromSyntaxGenerate and sead.model.handlers.nodeSuggestion.SuggestTypeFromSyntaxFilter.

A %deftrait statement may be split into multiple lines, so that handlers can be added to traits from different files:

%deftrait sead.traits.editing.suggestTypesFromSyntax
  sead.model.handlers.nodeSuggestion.SuggestTypeFromSyntaxGenerate;
%deftrait sead.traits.editing.suggestTypesFromSyntax
  sead.model.handlers.nodeSuggestion.SuggestTypeFromSyntaxFilter;
We continue the main definition file, where we define the types of the nodes which will make up the abstract syntax tree describing the documents of this type:

```plaintext
# Define the node types of the AST:
%type Expression() : ROOT();
%type Function(Identifier, Expression) : Expression();
%type Apply(Expression, Expression) : Expression();
%type Identifier(IDENT) : Expression();
```

The `%type` keyword defines a node type. It is followed by the name which identifies the node type, with in parentheses a comma-separated list of the types of the child nodes. Then a colon followed by a comma-separated list of the types from which this type inherits, and terminated with a semicolon. Not shown here, and not currently implemented is the possibility to supply an optional name with each type of the child nodes. These names can then be used as parameters to the types from which the type being defined inherits, for when the base type is defined with children of its own.

Next we define which traits are in effect for the node types:

```plaintext
# Enable the suggestion of the types of nodes based on the
# syntax described in this document, for nodes of all types.
%trait ROOT(expr)
    sead.traits.editing.suggestTypesFromSyntax();

# The context is used from the parent node.
%trait Expression()
    sead.traits.context.propagateContextFromParent();

# A new context starts when a Function is encountered:
%trait Function(ident, expr)
    sead.traits.context.enterContextBefore();

# The context ends after a Function definition:
%trait Function(ident, expr)
    sead.traits.context.leaveContextAfter();

# Define an identifier in a context.
%trait Function(ident, expr)
    sead.traits.context.addToContext(ident);
    # The order is significant. Introduce context
    # before adding identifiers to it.

# Stipulate that an identifier must be defined.
%trait Identifier(ident)
    sead.traits.context.useInContext(ident);
```

The `%trait` keyword applies a trait to a node type. It is followed by the node type, with names in parentheses, followed by a comma-separated list of
names of traits, also with names in parentheses. The names in parentheses after the node type identify the child types of the node type. These names may then be used in between the parentheses after the name of the trait, where they serve as parameters to the trait specifying a specific child type of the node type. The name of the trait is a string with periods used to organise them with related traits.

Next is the grammar describing how to translate the input document to an abstract syntax tree of nodes.

```plaintext
# Define the grammar

%start expression

expression: function | application | identifier : $1;

function: "\"λ\"" identifier "." expression : Function($2, $4);

application: expression expression : Apply($1, $2);

identifier: IDENT : Identifier($1);
```

The grammar is in a style similar to that used by parser generators such as Yacc. It is given as production rules with an added specification of the type of the Node produced by each one. Specifically, each rule consists of a non-terminal, a colon, a list of terminals (optionally grouped by parentheses, and with alternatives specified with |), the resulting NodeType and a semicolon. The $ symbol followed by a number $N$, refers to the $N$th symbol on the right hand side of the production rule (starting with 1, as Yacc does).

The starting symbol is specified with %start.

We remark that definitions may be split over multiple files. For instance, one file may define traits, another may apply them to node types, and yet another one applies more traits to the same node types. This way, functionality can be grouped within a single file, which may be used by multiple document type descriptions.

### 8.3 Code for document type dependent behaviour

The code part of the support for a type of document consists of the handlers. The handlers are intended to be orthogonal, so that they can be reused and combined in different contexts. If a document type requires new functionality which cannot be provided by existing handlers, then adding support for
this document type would include writing new handlers. These handlers can then be packaged up into a .jar file which can be used without modification of the rest of the Java code. As long as it is placed in some location which is in the Java class path, Sead should be able to find it.

8.4 Summary

We can summarise the addition of support for a new document type as follows:

- write the document type description
  - define the types of nodes
  - define any needed traits which do not exist yet
  - define which traits apply to the node types
  - define the translation from input file to nodes (grammar with constructor statements)

- create the code for required handlers which do not exist yet
  - write the Java code for the new handlers
  - package the Java code in a .jar file
We illustrate the working of handlers through a scenario. Not all of the functionality described in this chapter has been implemented in the prototype. Parts which are not or only partially implemented are marked with an asterisk (*).

9.1 Scenario: inserting a node in the AST

A user has a document open, containing the lambda expression λx.(λy.(□ y)). The symbol □ is a hole in the abstract syntax tree, which has yet to be filled in by the user to make the document comply to the document type. The user wants to insert the variable x into the hole.

![Abstract Syntax Tree](image.png)

Figure 9.1: The abstract syntax tree for the example scenario

In this scenario, we consider an editing view which works directly on
a tree representation of the abstract syntax tree, as is done in the proof of
concept editing component I created. First, the user selects the hole; he then
calls up the “Set Node” menu (either via the mouse or the keyboard). The
menu shows the possible choices in this location. The user picks “Identifier”,
and a node of type “Identifier” is inserted in the tree. The next step is to
name the variable. The user selects the newly inserted node, and calls up
the “Set Name” menu. The declared variables are listed. The user picks
“x”.

Figure 9.2: Editing the abstract syntax tree of the example scenario in the
proof of concept editing component

9.1.1 Behind the scenes

Generating the node type suggestions

When the user calls up the “Set Node” menu, a list of possible node types is
presented. This list is generated in two steps. In both steps, handlers asso-
ciated with the NodeType of the prospective parent Node — either directly
or via one of its base types — are invoked. First, the handlers implementing
the Java interface INodeTypeSuggestionGenerate are invoked with a NodeTypeSuggestionData argument. Each of these handlers may add suggestions
to the set of NodeTypes which are stored in this argument. In the second
step, the same NodeTypeSuggestionData argument is passed to all of the
handlers implementing \textit{INodeTypeSuggestionFilter}\textsuperscript{1}. These handlers may then remove suggestions from the same set.

In this example, handlers \textit{SuggestTypeFromSyntaxGenerate} and \textit{SuggestTypeFromSyntaxFilter} are used to generate the node types based on the semantic structure specified in the document type description. These handlers were attached to the \textit{InheritanceRoot NodeType}, which is at the base of the inheritance hierarchy of each \textit{NodeType}, and are hence in effect for all nodes.

\subsection*{9.1.2 Creating a new node}

When a selection has been made, a new \textit{Node} of the specified type is created. The \textit{INodeCreate} handlers associated with the type of this node are then invoked\textsuperscript{*}.

Then the following steps are taken to test whether the node may be inserted\textsuperscript{*}.

- The \textit{moveOk} handlers associated with the \textit{NodeType} of the new node are called to test whether insertion in the specified location is allowed.
- The \textit{childMoveOk} handlers associated with the \textit{NodeType} of the \textit{Node} which is to be the parent are called to test whether it accepts the new \textit{Node} as its \textit{N}th child.

If one of these handlers vetoes the insertion of the node, the operation is cancelled, and a message specifying the reason is reported to the user. When a handler issues a veto, the handlers which have not been called yet, will not be called at all. If none of these handlers vetoes the operation, then the next steps are executed\textsuperscript{*}.

- The \textit{preMove} handlers associated with the \textit{NodeType} of the new \textit{Node} are called to allow for preparatory actions to be taken.
- The \textit{preChildInsert} handlers associated with the \textit{NodeType} of the new parent \textit{Node} are called to allow for preparatory actions to be taken.
- The new \textit{Node} is registered as a child with the parent \textit{Node} and conversely, the new parent \textit{Node} is registered as such with the new \textit{Node}. As this step is independent of the type of document being edited, this is not done via handlers (also see \textit{activation layer} in section 5.4.1).
- The \textit{postChildInsert} handlers associated with the \textit{NodeType} of the new parent \textit{Node} are called to allow the parent to respond to the new \textit{Node}.
- The \textit{postMove} handlers associated with the \textit{NodeType} of the new \textit{Node} are called to allow them to respond to the new parent.

\textsuperscript{1}There is only one \textit{INodeTypeSuggestionGenerate} handler and one \textit{INodeTypeSuggestionFilter} handler implemented in the prototype.
Generating the identifier name suggestions

The handler infrastructure for generating the list of suggested identifier names works analogously to the generation of the node type suggestions described in section 9.1.1*. A handler implementing identifier name suggestion would however base its result on the context of the Node. In such a case, an implementation would store a representation of that context in an Annotation associated with the Node establishing that context.

9.1.3 Setting the identifier name

When a name has been selected, the following steps are taken*:

- The setValueOk handlers associated with the NodeType of the node are called to test whether changing the name of the identifier is allowed. If it is not, the action is aborted.

- The preSetValue handlers associated with the NodeType of the Node are called to allow for preparatory actions to be taken.

- The name of the node is changed. As this step is independent of the type of document being edited, this is not done via handlers (also see activation layer in section 5.4.1).

- The postSetValue handlers associated with the NodeType of the Node are called to allow them to respond to the new name.
Chapter 10

Conclusion

In this report, I have presented my design for an internal representation of documents in a generic structure editor for annotated documents. This design incorporates several novel ideas. I have implemented the core of this design, which can serve as a framework for future work. I have also built a simple editor plugin for the Eclipse framework based on this core.

10.1 Summary of the design

Central to my design is the representation of the structure of documents, in a generic way. This can be summarised as follows:

- the document is represented as an abstract syntax tree;
- the abstract syntax tree uses generic objects as nodes;
- attached to the objects for the nodes are objects representing the type of the nodes;
- the node type objects are decorated with handlers, code fragments which add document-specific behaviour to nodes of this type;
- handlers are intended to be small, orthogonal, and reusable within a variety of document types;
- annotations may be attached to nodes to associate meta-information with these nodes;
- annotations may themselves contain a tree of generic nodes with attached node types;
- annotations may represent derived information not present in the stored version of the document, generated and used by handlers;
- the document type is described declaratively in a description file;
• within a document type description file, traits are used to group handlers together;

• the document type description includes definitions of traits and node types, assignments of traits to node types, and a grammar describing the syntactic structure;

• fragments of document type description files may be included from other document type description files, further enabling the grouping together of concepts and the reuse of partial document type descriptions.

10.2 Assessment of the results

The original goals of my project were much more ambitious, encompassing a much larger part of a structure editor. Still, the design for the representation of the structure of documents in a generic way, as described in this document, is a novel idea; to my knowledge it has not been used in a structure editor before. The proof of concept is too limited to be able to conclude that this is a good approach for a full structure editor, but in my opinion it is a promising design, and worth investigating further.

In chapter 3, I laid out the goals of a structure editor for annotated documents. I will now consider to what degree these goals have been achieved in my work.

Flexible and widely applicable (genericity) Genericity has been a guiding principle throughout my work, and is visible throughout my results. Document types are described in document type description files, separate from the editor, which serve as parameters to generic code. Specific functionality is contained in small orthogonal handlers, which can be mixed and matched so that they can be applied to a wide range of different document types. Also, fragments of the document type description files can be split off in “include” files, so that specific functionality in them can be reused without duplication. As shown in chapter 8, a large part of adding support for a new document type consists of indicating which existing components are to be reused.

Maintainable and extensible (reusability) The handler architecture is inherently extensible. Also the rest of the software is well structured, which should help in keeping it maintainable and extensible. It will however be important for future developers to be acquainted with the concepts involved, something with which this report should help.
Suitability as a teaching tool  As the core of my work has been in internal structures of the editor which will be mostly invisible to the end user, it does not in any major way affect the suitability as a teaching tool of a full-scale editor built on this work. One point in which Sead does contribute to the suitability as a teaching tool is the way in which handlers report that an action cannot be performed. For this, exceptions are used, which can carry with them additional information regarding the cause of the “failure”. This allows for verbose error messages.

Industry usability  By implementing my design in Java, one barrier for industry acceptance has been avoided. If, like the proof of concept editing component, a full implementation of my work is built on top of the Eclipse framework, another such barrier will be avoided.

The suitability of my design for larger scale application has not yet been evaluated in practice, and this remains an open question for now.

Tie in with other SET projects  Both the VIDE project and the model driven engineering projects can benefit from a generic structure editor, which could be built upon my designs. Beyond this, the connection between my work and these projects is small.

Additionally, I used Java as the programming language to implement (the core of) my design in, which will be an advantage when interfacing it with Cocktail.

10.3 Future work

To further develop the ideas behind Sead and the implementation thereof, suitable next steps would be:

- to test the design on a larger scale, in a full editing component;
- to allow node types to be defined with parameters, so that a base type can have as children a selection of the children of the derived type;
- to allow traits and handlers to be defined with parameters specifying that they are applied to a specific child of a node type;
- to add more sanity checks to the code, specifically the detection of cycles in a hierarchy of NodeType;
- to add detection and resolution of diamond inheritance in NodeType.
10.4 Acknowledgements

I would like to thank

- My mother, Mariëtte van de Winkel, for supporting me throughout my studies;
- Tom Verhoeff, for going beyond the call of duty to help me bring this project to a successful end;
- Mark van den Brand, for giving me every opportunity to complete my graduating project, and hence my studies;
- Marc Voorhoeve, together with Tom Verhoeff and Mark van den Brand, to serve as my graduating committee, with great flexibility;
- My employers at Madison Gurkha B.V., for being flexible and accommodating, enabling me to complete my graduation project.
Bibliography


Appendix A

Document type description for the lambda calculus example

# Document type description of untyped Lambda calculus.

# Make the types and traits relevant to contexts available.
%import "context.dtd";

# Make the traits relevant to node type suggestion available.
%import "nodetypesuggest.dtd";

# Define the node types of the AST:
%type Expression() : ROOT();
%type Function(Identifier, Expression) : Expression();
%type Apply(Expression, Expression) : Expression();
%type Identifier(IDENT) : Expression();

# Define the traits in effect for the node types:

# Enable the suggestion of the types of nodes based on the syntax described
# in this document, for nodes of all types.
%trait ROOT(expr) sead.traits.editing.suggestTypesFromSyntax();

# The context is used from the parent node.
%trait Expression() sead.traits.context.propagateContextFromParent();

# A new context starts when a Function is encountered:
%trait Function(ident, expr) sead.traits.context.enterContextBefore();

# The context ends after a Function definition:
%trait Function(ident, expr) sead.traits.context.leaveContextAfter();
# Define an identifier in a context.
%trait Function(ident, expr) sead.traits.context.addToContext(ident);
  # The order is significant. Introduce context before adding
  # identifiers to it.

# Stipulate that an identifier must be defined.
%trait Identifier(ident) sead.traits.context.useInContext(ident);

# Define the grammar

%start expression

eexpression: function | application | identifier :
  $1;
function: "lambda" identifier "." expression :
  Function($2, $4);
application: expression expression :
  Apply($1, $2);
identifier: IDENT :
  Identifier($1);