Andreas Bartho

Creating and Maintaining Consistent Documents with Elucidative Development

Beiträge aus der Informatik

Andreas Bartho

Creating and Maintaining Consistent Documents with Elucidative Development

VOGT

Dresden 2014

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Bibliographic Information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.dnb.de.

Zugl.: Dresden, Techn. Univ., Diss., 2014

Die vorliegende Arbeit stimmt mit dem Original der Dissertation „Creating and Maintaining Consistent Documents with Elucidative Development“ von Andreas Bartho überein.

Gesetzt vom Autor

ISBN 978-3-938860-76-2

Jörg Vogt Verlag
Niederwaldstr. 36
01277 Dresden
Germany

Phone: +49-(0)351-31403921
Telefax: +49-(0)351-31403918
e-mail: info@vogtverlag.de
Internet : www.vogtverlag.de

Creating and Maintaining Consistent Documents with Elucidative Development

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Andreas Bartho
geboren am 07.12.1980 in Dresden

Gutachter:
Prof. Dr. rer. nat. habil. Uwe Aßmann (Technische Universität Dresden)
Associate Professor Kurt Nørmark (Aalborg University)

Tag der Verteidigung: 27.05.2014

Dresden im August 2014

Abstract

Software systems are usually not defined in one big, all-encompassing model, but they consist of a multitude of different views from multiple technological spaces, such as requirements, class diagrams, or source code. These views contain redundancy, i.e., they share some of their information. If redundant information in different views contradicts with each other, the views become inconsistent. Inconsistency is a source of errors, and much effort is spent on research and tool development to avoid it.

Documents are also views on a software system. They are usually written by human authors. In practice, documents and other views often change at different paces. Document updates are frequently omitted because they are expensive and do not pay off immediately. Consequently, documents are often outdated. Outdated documents communicate wrong information about the software. The severity of outdated information can range from a minor inconvenience for the reader to complete uselessness.

Sometimes documents are generated. If generated documents are outdated, they can easily be regenerated. However, in many cases it is not possible to generate the desired document content.

In this thesis, we introduce *Elucidative Development (ED)*, an approach to create documents from other views by partial generation. Partial generation means, that some document content is generated, and the remaining document content is added manually, afterwards. Unlike naive generation approaches, ED retains manually written content when the generated content is regenerated. A guidance system informs the author about changes in the generated content and helps him update the manually written content.

In an evaluation we present our findings regarding the applicability and versatility of ED. First, we analyse two model specifications, one of them being the Unified Modeling Language (UML) specification, for inconsistencies and show that the use of ED would have prevented these inconsistencies. We also show how ED helps with the update of the specifications. Then, we present several examples where we successfully wrote documents using ED.

Acknowledgement

During the writing of this thesis, I was supported by many people, who deserve my gratitude. First of all, I would like to thank my former colleagues at the university, who supported me with new ideas, collaborations, joint publications and suggestions for case studies, especially Birgit Demuth, Sven Karol, Claas Wilke, Julia Schroeter and Katja Siegemund. I am particularly grateful for the support of Sebastian Richly, whose valuable criticism and recommendations had a great impact on this thesis. I would also like to thank my former assistants Frank Herrlich and Sebastian Patschorke. Their excellent theoretical and practical work on the DEFT prototype provided valuable inspirations for the thesis.

Another big thank you goes to my friends, who proofread the thesis and helped me improve it, in particular René Pönitz, Frank Herrlich and, most importantly, my dear girlfriend Claudia Geitner.

Finally, I would like to thank Prof. Dr. Uwe Aßmann for the possibility to join his group, take part in a research project that met my personal interest and turn it into a PhD thesis. Prof. Dr. Aßmann's visionary ideas and his extensive knowledge of the work of fellow researchers helped me to broaden my thinking, see new connections, and relate them to my own work.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API	Application Programming Interface.
AR	Artefact Removal.
AU	Artefact Update.
BPMN	Business Process Modelling Notation.
CASE	Computer-Aided Software Engineering.
CCM	Cool Component Model.
CDF	Computed Document Fragment.
CIM	Computation Independent Model.
COM	Component Object Model.
CORBA	Common Object Request Broker Architecture.
DEFT	Development Environment For Tutorials.
DSL	Domain Specific Language.
DTD	Document Type Definition.
EAT	Energy Auto-Tuning.
ED	Elucidative Development.
EDE	Elucidative Development Environment.
EP	Elucidative Programming.
GBM	Grammar-based Modularisation.

GUI	Graphical User Interface.
HTML	Hypertext Markup Language.
IBM	International Business Machines Corporation.
JSP	Java Server Pages.
LiMonE	Literate Modelling Editor.
LM	Literate Modelling.
LP	Literate Programming.
MDA	Model-Driven Architecture.
MDSD	Model-Driven Software Development.
MOF	Meta Object Facility.
MOST	Marrying Ontology and Software Technology.
OCL	Object Constraint Language.
ODF	Open Document Format.
ODRE	Ontology-Driven Requirements Engineering.
OLE	Object Linking and Embedding.
OMG	Object Management Group.
PDF	Portable Document Format.
PIM	Platform Independent Model.
PREP	Propagate Replay Evaluate Pick.
PSM	Platform Specific Model.
QVT	Query/View/Transformation.
RO	Requirements Ontology.
RTE	Round-Trip Engineering.

RTF Rich Text Format.

SMIL Synchronized Multimedia Integration Language.

SOM System Object Model.

SVG Scalable Vector Graphics.

SVN Subversion.

TGG Triple Graph Grammar.

UML Unified Modeling Language.

UNO Universal Network Objects.

URL Uniform Resource Locator.

W3C World Wide Web Consortium.

WWW World Wide Web.

WYSIWYG What You See Is What You Get.

XHTML Extensible Hypertext Markup Language.

XML Extensible Markup Language.

XSLT Extensible Stylesheet Language Transformations.

Chapter 1

Introduction

Software is subject to changes during its entire lifetime. In the early phases of the software life-cycle, documents are written and models, such as Unified Modeling Language (UML) use-case diagrams, are created. Based on those documents and models, program code is written. Software development is usually an iterative process. Many of the documents and models and much of the code are revised multiple times, both during the initial creation and maintenance of the software.

The reasons for document changes during software development are manifold. Requirements specifications are written in multiple iterations. Architecture documentation is written early in the development process, but unforeseen technological or organisational problems might require modifications. Application Programming Interface (API) documentation and example-style tutorials are important for software libraries, frameworks, or software with plug-in interfaces. When the API changes, the documentation must be updated accordingly. End-user documentation explains the usage of the software to the end user. It contains instructions for the achievement of certain goals, often enriched with screenshots of the Graphical User Interface (GUI). When the software changes such that it affects the GUI, the instructions and screenshots must be changed, too.

In practice, documents, models and code often change at different paces. Document updates are frequently omitted because they are expensive and do not pay off immediately. Consequently, documents are often outdated. Outdated documents communicate wrong information about the software. The severity of outdated information can range from a minor inconvenience for the reader to complete uselessness. In any case, outdated documents are generally a problem during software development and maintenance.

1.1 Contributions

This thesis makes several contributions to the field of document management and consistency enforcement. The main contribution of this thesis is the introduction of *Elucidative Development (ED)*, a novel approach, which simplifies the creation and maintenance of documents which describe software artefacts. ED is based on partial generation, i.e., some content of a document is generated and some content is handwritten. In contrast to naive approaches, ED takes care that handwritten content is not destroyed when documents are regenerated. Another noteworthy characteristic of ED is the tight integration of a guidance system, which helps the author keep the documents in a consistent state.

There are many kinds of documents in the software development life-cycle, which differ by their level of formality. Based on the definition of document formality we present ED as an extensible approach for document creation and maintenance.

- First, we introduce the basic concepts of ED and show how they can be applied to semi-formal documents.
- Then, we extend ED for the application to formal documents.

We also present two other extensions, which do not depend on the formality of documents.

- Documents written with ED contain special directives, which control the content generation. These directives prevent standard validation approaches, such as checking an Extensible Markup Language (XML) document against a schema. We show possibilities to validate these documents nonetheless.
- Sometimes it is desirable to modify generated document content and propagate the change to the original data, from which the content has been generated. We show how this can be achieved by employing backpropagation-based round-trip engineering.

Our second big contribution is the evaluation of ED to show its applicability and versatility. In two case studies, we show how ED improves the document quality and eases the document maintenance. Then, we present many examples of the successful application of ED.

Our third big contribution is the comparison with related work because ED is based on many existing ideas. This includes the discussion of related documentation approaches from the literature, such as *Literate Programming*

(LP) or *Elucidative Programming (EP)*, as well as consistency management concepts, such as *transclusion* and *transconsistency*.

1.2 Scope of the Thesis

In this thesis, we concentrate on the theoretic and technical aspects of ED. This includes the problems of computing document content from arbitrary views of a software system, embedding it into documents, and keeping it up to date. We neither discuss how the other views can be kept consistent, nor organisational rules that help the author decide when to write or update documents. We presume that the consistency of views can be achieved to a satisfactory degree and that the author knows when documents must be written or updated.

Furthermore, we do not explicitly consider the collaboration of multiple authors on the same document. We do not forbid multiple authors working on one document, but we also do not promote it. In this thesis we refer to both a single author and a group of authors as “the author”.

1.3 Organisation

In Chap. 2, we explain the necessity of redundancy in a software system in general and how it can lead to inconsistency. Then, we motivate full and partial generation as possibilities to keep formal and semi-formal documents, such as requirements analyses, specifications and documentation, consistent with the rest of the software system.

In Chap. 3, we present some background information on concepts and technologies, which are used in the thesis. The aim of this chapter is to introduce the reader to these concepts and technologies.

In Chap. 4, we propose the novel approach ED for the semi-automatic creation and maintenance of semi-formal documents. We present a number of challenges that must be solved and derive requirements that ED must fulfil. This includes the partial generation of content, ensuring that generated content is always up to date, and a guidance mechanism that informs the author about outdated or recently updated content. Afterwards, we show the realisation of these requirements.

In Chap. 5, we show how ED can be used in a Model-Driven Software Development (MDSD) setting. We explain, why the basic ED approach from Chap. 4 is not sufficient for uniformly structured, formal documents, such as the UML specification. Based on this, we present extensions, which overcome these limitations.

In Chap. 6, we present further extensions of ED, namely:

- support for structural validity checking, such as checking an XML document against a Document Type Definition (DTD)
- support for round-trip engineering, which allows the synchronisation of changes in the document with the described software artefacts.

In Chap. 7, we give recommendations regarding the implementation of an Elucidative Development Environment (EDE). This includes technical optimisations of the concepts presented in Chap. 4 and 6, but also thoughts about reusing existing tools and editors for ED.

Afterwards, in Chap. 8, we review the concepts and techniques that have influenced ED. Unlike in Chap. 3, we also compare these concepts and techniques to ED. Additionally, we show concepts and technologies that did not explicitly influence ED, but which are related.

In Chap. 9, we present several evaluations that show the applicability and usefulness of ED. The evaluations have been performed with our EDE called Development Environment For Tutorials (DEFT).

Finally, we summarise the results of this work in the conclusion in Chap. 10.

Chapter 2

Problem Analysis and Solution Outline

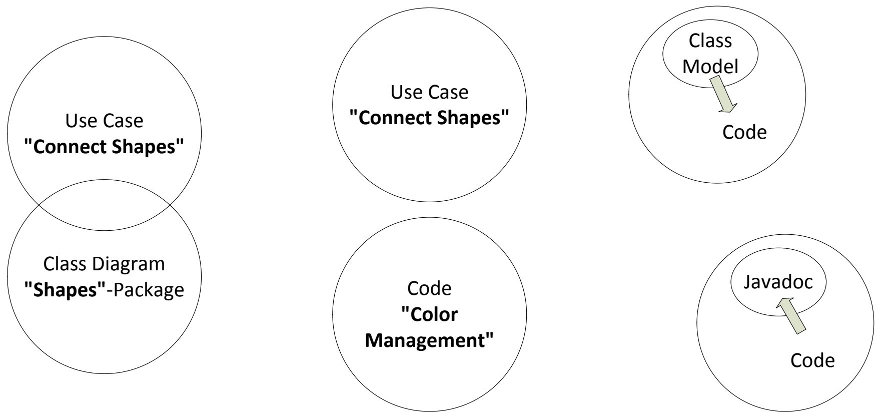
In this chapter, we explain the connection between redundancy and inconsistency, and show how generation (i.e., automatic document creation from existing data) improves consistency between documents and other parts of a software system. Since complete generation is not always possible, we present the advantages and disadvantages of partial document generation and show which kinds of documents are suited for partial generation.

2.1 Redundancy and Inconsistency

Software systems are usually not defined in one big, all-encompassing model, but they consist of a multitude of different parts from multiple technological spaces, such as requirements, class diagrams, or source code. These parts describe a software system "from different angles and in different levels of abstraction, granularity and formality" [52]. We call these parts *views*, in accordance with [20].

All views of a software system share information with one or more other views. This overlap of information is called *redundancy*. Redundancy is necessary to connect several views to one coherent description of the software system. Redundancy can be very technical, for example a Java class that corresponds to a Unified Modeling Language (UML) class. But redundancy can also be very abstract, such as the existence of a certain concept, which appears in several views.

Two different views can overlap partially, fully, or not at all. Figure 2.1 shows examples, inspired by a figure from [20].

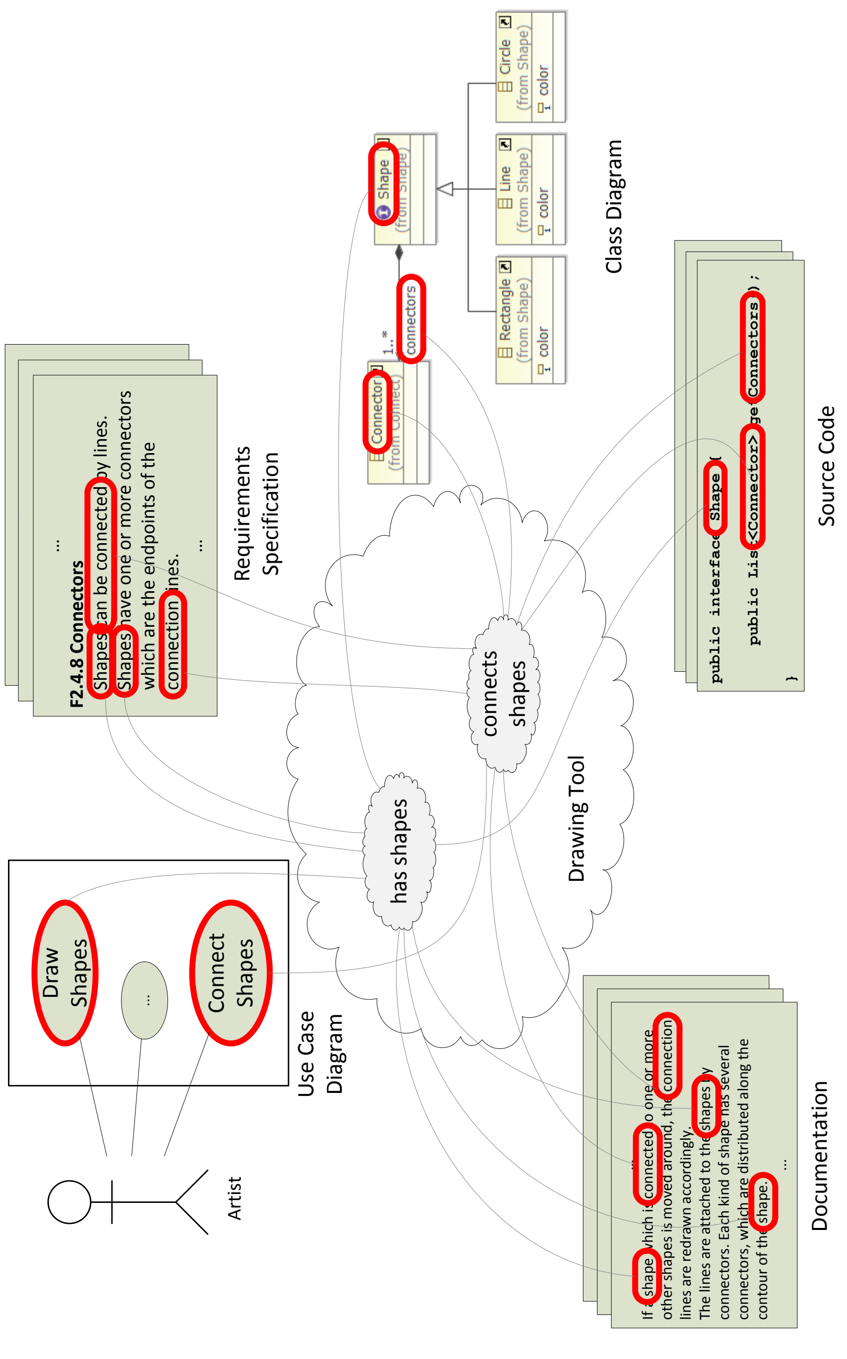


(a) Partial overlap. (b) No overlap. (c) Full overlap.

Figure 2.1: Types of view overlaps.

In Fig. 2.2, there is an example of 5 views referring to the ability of a drawing tool to draw shapes and connections. In other words, the information that the tool allows for drawing shapes and connecting them is contained redundantly in different views.

The use-case diagram and the class diagram in Fig. 2.2 are partially redundant, as represented by Fig. 2.1a. Both contain information regarding shapes and their connections. The overlap in this case is rather small. It comprises the fact that the user can connect shapes with the drawing tool. The views represented by Fig. 2.1b do not share any information and thus contain no redundancy. The use-case of connecting shapes, represented by the use-case diagram, has nothing to do with colour management and its implementation. Figure 2.1c shows that there are cases where one view is completely contained within another view. This means that the *inner view* (the view represented by the inner circle in the figure) is completely redundant and does not provide any additional information. It is not useless, though. Two completely redundant views usually have a different level of detail and stem from different phases of the software development process. For example, the inner view could be a UML diagram and the *outer view* (correspondingly, the view represented by the outer circle in the figure) could be the source code whose skeleton has been generated from the diagram, i.e., the outer view is a refinement of the inner view from a later development phase. Another possibility is that the inner view has been transformed from the outer view to contain only a subset of the information, but in a clearer and more concise fashion. This would be the case with Javadoc, which can be generated from Java source code.


F2.4.8 Connectors …

Shapes can be connected by lines.
Shapes have one or more connectors
which are the endpoints of the connection lines.

…

```
public interface Shape {
    public List<Connector> getConnectors();
}
```

…

If a shape which is connected to one or more
other shapes is moved around, the connection
lines are redrawn accordingly.
The lines are attached to the shapes by
connectors. Each kind of shape has several
connectors, which are distributed along the
contour of the shape.

…

Figure 2.2: Overlapping views of a software system.

If views contain redundant information, they can become inconsistent. This is because the views "overlap – that is, they incorporate elements which refer to common aspects of the system under development – and make assertions about these aspects which are not jointly satisfiable" [52]. We call this kind of inconsistency *global inconsistency* [19]. Correspondingly, we speak of *global consistency* if all assertions are satisfiable.

Besides global inconsistency, there can also be *local inconsistency*. Local inconsistency arises from contradictions within the same view. Correspondingly, we speak of *local consistency* if the view contains no contradictions.

Since redundancy cannot be avoided, redundant descriptions of the software system must somehow be kept consistent. This can be achieved manually or with tool support. Keeping multiple views of a software system consistent manually is usually very difficult and time-consuming due to the sheer amount of information that overlaps. Tools can aid to some degree. For example, a model-to-text transformation tool can create a source code skeleton from a UML class diagram, with all defined classes, attributes, methods and relationships. Unfortunately, tool support is not available for all kinds of overlapping views. Therefore, inconsistency is still a problem in today's software development.

2.2 Improving Consistency with Partial Generation

As shown in Fig. 2.2, documents, such as specifications and documentation, are also views on a software system.

Definition 1 (Document)**.** *Documents are special views on a software system. Their content is primarily meant to be read by humans. Usually, they consist mostly of text, but they can also include structured information, such as tables or listings, and media, such as images.*

Since documents are views on a software system, they contain redundancy. They are mostly *handwritten*, i.e., written by humans, which makes them susceptible to inconsistencies. Global inconsistencies occur if the documents describe other views incorrectly. During the initial creation of a document, the author might accidentally omit important information or include wrong information due to a lack of understanding. Another source of inconsistencies are incorrect references to other views, such as mentioning a UML class which does not exist in the UML class diagram. Local inconsistencies occur if different parts of one document contradict with each other. An example of a local inconsistency is a code listing in a documentation which contains a method `connectShapes()`, and explanatory text which calls the same method `connect()`.

If the documented views of the software system are changed after a document has been written, the document can also become inconsistent. We say that the document is *outdated*. Outdated documents must be *updated* to make them consistent again. The first step of a document update is to find all outdated parts. If even one outdated part is not found, the document cannot become consistent. But even if all outdated parts are identified, it is still necessary that they are updated correctly. Failing to do so can result in a globally and locally inconsistent document. Global inconsistency arises if some parts of the document do not correspond to the rest of the software system. Local inconsistency arises if the document contains contradictory outdated and updated information at the same time.

In some cases, document *generation* can be used to prevent inconsistencies. Documents are generated by a transformation whose input are views with formal content, such as source code or UML models. Assuming that the transformation works correctly, generated documents are both locally consistent and globally consistent to the view(s) from which they are generated. Whenever generated documents are outdated, the transformation can be executed again and new, updated documents are created.

Generated documents can only contain information which already exists in other views. However, they can present the information in a way that is easier to understand by humans. An example is Javadoc documentation, as indicated in Fig. 2.1c. The Javadoc tool can create a navigable set of Hypertext Markup Language (HTML) documents, which contain for each Java class and interface all fields and methods, together with documentation text that has been annotated to the source code. Javadoc also makes implicit information explicit. Among others, the generated documentation shows the inheritance hierarchy for all classes and interfaces, and it shows for all methods which superclass methods they override.

If documents contain new information, i.e., information that does not already exist in other views, they cannot be fully generated. However, they can be *partially* generated. After the generation, the author can add the missing information manually. The update of outdated partially generated documents is difficult. A regeneration is not easily possible, because the handwritten content would be overwritten. A manual update by the author has the same disadvantages as the update of completely handwritten documents. Therefore, partial generation is rarely used in contrast to manual document writing. The main goal of this thesis is to present a partial generation approach, which overcomes these problems.

The degree to which a document can be generated depends on its level of formality. We distinguish between informal, semi-formal and formal documents, but the boundary is blurred. Usually, the latter are used to describe formal views. A formal view is a view which has a well-defined syntax and possibly static or dynamic semantics. Examples are source code or UML models, such as class diagrams.

Definition 2 (Informal Document)**.** *Informal documents describe the software system in an abstract way. They give a high-level overview and omit details. The structure and the content of informal documents are primarily determined by the author. It is the author's choice, which topics to emphasise, which topics to omit, and how to arrange the content.*

An example of an informal document is an introductory documentation of the software, which sketches its overall goal and outlines the most important features. Informal documents are usually the first documents read by someone who wants to learn about the system. They are not suited for generation, so they are manually created and maintained by the author.

Definition 3 (Semi-formal Document)**.** *Semi-formal documents describe a formal view or related formal and informal views of the software system in more detail. They do not necessarily describe the view(s) exhaustively. A substantial part of a semi-formal document contains information on a very technical, formal level. Apart from that, there is additional, more abstract, background information.*

An example of semi-formal documents are so-called "How To" cookbooks [5]. They are task-oriented framework documentations for software developers. They contain code examples of framework instantiation, optionally together with explanatory text. The targeted audience of semi-formal documents are usually persons who are starting to get involved with the details of the system. Semi-formal documents can be partially generated.

Definition 4 (Formal Document)**.** *Formal documents describe formal views or some parts of a formal view. They are comprehensive and the author does not emphasise or omit any parts. Formal documents usually contain a number of uniformly structured chapters and sections, and the document structure follows the structure of the view.*

An example of a formal document is generated Javadoc documentation. Another example, which is slightly less formal, is the UML 2.3 superstructure specification[1]. It contains, among other information, descriptions of all UML

[1] `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/`

metaclasses. Each class description consists of an enumeration of supertypes (“generalizations”), a textual description, a listing of attributes and associations together with their descriptions, and more. The targeted audience of formal documents are usually persons who already have some knowledge of the system (or at least the view) and want to look up more details. Formal documents can be partially generated. If all of their content can be computed from other views, they can even be fully generated.

The need for partial generation in document writing and the corresponding tool support has also been mentioned in the literature. One example is a survey, which identifies several attributes that influence the quality of documentation [21]. The most important ones are documentation content, actuality (i.e., global consistency), availability and the use of examples. A lot of participants of the survey thought that much information can be extracted from source code. They generally saw the automation of documentation positively. However, they also admitted that full automation is not possible, because the automated documentation tools “don’t collect the right information”. Consequently, most of the participants find a tool useful that can track changes in a software system for the purpose of documentation maintenance. For example, such a system would identify all parts of the documentation that refer to changed source code. The authors suggest that “the technology support traceability among documents as well as between source code”. Thus, the survey showed that documentation systems with partial document generation and guidance support for the documentation author are needed.

An improved approach for the creation and maintenance of consistent documents and the corresponding tool support is not only needed for documentation. This is evident from many papers which point out inconsistencies in the UML specification, for example [9, 61]. The UML specification is written and maintained manually, and many of the identified inconsistencies could have been avoided with tool support.

2.3 Conclusion

In this chapter, we discussed the necessity of redundancy in software systems and the resulting consistency problems. We presented generation and partial generation as possibilities to ensure consistency, depending on the document’s level of formality. Partial generation has been identified as promising approach for the creation and maintenance of semi-formal and formal documents, but the problem is that manually written content is overwritten when the document is regenerated. Finding a solution to this problem is the main goal of this thesis.

Chapter 3

Background

In this chapter, we present an overview of various technologies, on which the thesis relies. It is meant as a short introduction, where the basic principles of the technologies are explained. Additionally, this chapter contains citations, which can serve as a starting point for further, more detailed, investigations.

First, we present Grammar-based Modularisation (GBM), an approach for software composition. Then, we introduce Model-Driven Software Development (MDSD), a software development methodology which relies on models and transformations. Finally, we cover Round-Trip Engineering (RTE), whose purpose is to keep multiple views of a software system consistent.

3.1 Grammar-Based Modularisation

GBM is a modularisation approach first proposed by [6]. It allows the definition of programs with "holes", which can later be filled with fragments in a type-safe manner. The basic building block in GBM is a *form*. A form is a sentential form of the language. For example, an A-form is a sentential form which has been derived from the nonterminal A. "A" is called the *syntactic category*. If a form additionally has a name, it is called a *fragment form*.

The following example is taken from [25]. It shows a datalog *rule*-form, which contains two nonterminals: ⟨num⟩ and ⟨atom⟩:

bonus(X, ⟨num⟩) :- employee(X), ⟨atom⟩.

In GBM, nonterminals, which are meant to be replaced by fragments, are called *slots*. Thus, slots are an area of variability. They have a name and a syntactic category. In the following example, the nonterminals have been replaced by slots:

bonus(X, «SLOT value:num») :- employee(X), «SLOT condition:atom».

Slots can be replaced by fragment forms with the same name and the same syntactic category. Complete programs are assembled by binding fragment forms to declared slots recursively.

3.2 Model-Driven Software Development

MDSD is a software development method, which uses transformations to generate software from formal models. Formal models[1] have an abstract syntax and static semantics, which can be defined by a metamodel [53]. The abstract syntax can be expressed by one or multiple concrete syntaxes, or Domain Specific Languages (DSLs). For example, the well known graphical notation with boxes and arrows is one possible concrete syntax for Unified Modeling Language (UML) class diagrams. Additionally, there are several textual DSLs[2] [24].

A metamodel itself is also a formal model and has a structure. This structure is described by its metametamodel. Well-known examples of metametamodels are Meta Object Facility (MOF)[3] and Ecore[4]. Ecore is an implementation of Essential MOF, a MOF subset.

Models are often only created for documentation purposes. In MDSD, however, models are first-class development artefacts. Multiple models are used to describe different structural and behavioural aspects of the software system. Thus, models are views, or parts of views, of the software system.

Models have different levels of abstraction. The distinction is blurred, as we will see from slightly inconsistent naming schemes in the literature. At the most abstract level, there are domain-specific models, i.e., models, which describe the domain of the software system, for example aviation or logistics. A domain-specific model is abstracted from programming languages and platforms. In Model-Driven Architecture (MDA), which can be seen as a specialisation of MDSD, a domain-specific model is called *Computation Independent Model (CIM)*[5]. The term *Platform Independent Model (PIM)* is used for a model which describes the software "system, but does not show details of its use of its platform". In [53], on the other hand, a domain-specific model is called PIM.

[1]We are aware that the notion of formality is also sometimes used to include dynamic semantics. This is not the case in this thesis. Here, we use the definition from [53].

[2]`http://modeling-languages.com/uml-tools/#textual`

[3]`http://www.omg.org/mof/`

[4]`http://www.eclipse.org/modeling/emf/`

[5]`http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`

A CIM or PIM can be transformed into a more technical model via a model-to-model transformation. Models which contain platform-specific information are called *Platform Specific Models (PSMs).* The final transformation step usually generates code from a PSM and is therefore called model-to-code transformation. A PIM can be transformed into a PSM in one single transformation, or it can be transformed stepwisely, using multiple transformations. The decision depends on the requirements of the software project and is made by the developers. Figure 3.1, which is based on a figure from [53], shows an example transformation chain from a PIM via multiple PSMs to code.

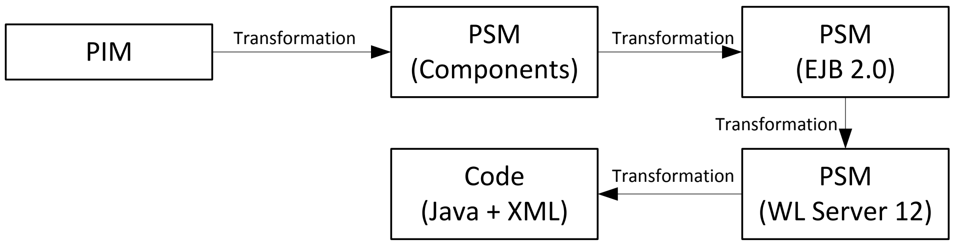


Figure 3.1: Model transformations.

Model-to-model transformations are defined as mapping between the source and the target metamodel. The source and the target metamodel can be the same or different. If the target model is more platform-specific than the source model, the metamodels usually differ. The source and the target metamodel have the same metametamodel.

Model-to-code transformations, on the other hand, usually have no target metamodel. The target code is created by outputting text, possibly using a template engine. However, the input of a model-to-code transformation must still be based on a metamodel.

3.3 Round-Trip Engineering

According to [1], "the goal of round-trip engineering is keeping a number of artifacts, such as models and code, consistent by propagating changes among the artifacts. Making artifacts consistent by propagating changes is also referred to as synchronization. Round-trip engineering is a special case of synchronization that can propagate changes in multiple directions, such as from models to code and vice versa".

The idea of *RTE* is closely related to the *view update problem* of relational databases. Here, the term *view* has a different meaning than in the rest of the thesis. We understand a view as a simplified representation of database content. Queries against the view can be easily evaluated because they can be mapped to queries against the original database. However, this is not the case with update operations. A mapping of view update operations to database operations may not be unique or it may not exist at all. This is called the view update problem and has been extensively studied by the database community starting in the late 1970's.

The automatic translation of view updates into corresponding database updates was analysed in [18]. The work also takes integrity constraints on the schemata into account. Another approach was followed in [28]. It was proposed that at the time of the view definition an update translator must be chosen, which translates view updates into database updates in a certain manner. The underlying assumption was that the database administrator who defines the view knows best which database update strategy to employ for certain view updates.

In general, the view update problem is the problem of modifying generated data and keeping it consistent with the original data. This problem also occurs in software development, as we have outlined above. RTE systems usually synchronise artefacts by applying inverse transformations, as illustrated in Fig. 3.2. Unfortunately, this does not work for many real world scenarios, because there are transformations which do not have an inverse, or whose inverse is difficult or impossible to compute.

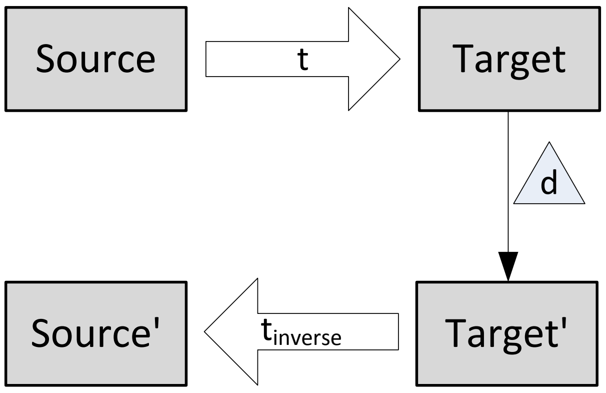


Figure 3.2: Round-trip via inverse transformation.

Pierce et al. have introduced the notion of *lenses* for the synchronisation of trees [22]. Lenses comprise two functions, which are used to formulate bidirectional transformations. The first function is called *get*. It is a forward transformation, which transforms a tree (called concrete tree) into an abstract tree, i.e., a tree which contains less information than the concrete tree. The second function is called *putback*. It is a transformation which transforms a modified abstract tree into a modified concrete tree. The putback function computes the modified concrete tree from the original concrete tree, the abstract tree, and the modified abstract tree. In [11] the concept of lenses has been applied to relational data. Lenses were defined whose get and putback functions could synchronise databases and views.

Another means to formulate bidirectional transformations are *Triple Graph Grammars (TGGs)* [47]. TGGs establish relationships between two or more graphs, which conform to different graph grammars, by means of a correspondence graph. The correspondence graph also conforms to a grammar, the so-called correspondence grammar, hence the name Triple Graph Grammar. Even though TGG rules are purely declarative, they can be used as input for unidirectional or bidirectional transformation tools. Since models are graphs, too, TGGs can also be used to specify the relationship and transformations between models. For example, Fujaba[6], a Computer-Aided Software Engineering (CASE) tool for model-based software engineering and re-engineering, uses TGGs for model synchronisation [29, 30].

The *Query/View/Transformation (QVT)* model transformation language, which has been standardised by the Object Management Group (OMG), has been developed to specify relations between MOF-based models. QVT is in many respects very similar to TGGs. A comprehensive comparison is presented in [23].

3.4 Conclusion

In this chapter, we shortly introduced three technologies and approaches, which are the foundations of some of the contributions presented in the following chapters.

GBM is a modularisation and composition approach for source code. It allows the specification of typed slots in a program. A running program can be composed by filling the slots in a type-safe manner with typed fragments.

MDSD is a software development methodology which uses models as first-class development artefacts. Models can describe the domain of the software system, but also concrete technical realisations. Transformations are used to transform domain-specific models to more technical models and source code.

[6]`http://www.fujaba.de`

RTE is used to keep derived software artefacts, such as a generated model or a database view, consistent with their sources, such as the source model or a database table. When changes are made to the derived software artefact, the source artefact is modified correspondingly. However, it is not always possible to modify the source artefact automatically. There are many different RTE approaches, each with their own strengths and weaknesses.

Chapter 4

Elucidative Development

In this chapter, we will present *Elucidative Development*, our approach for the creation and maintenance of semi-formal documents with partial generation. First, we give a definition of Elucidative Development and sketch the basic idea. Then, we list the challenges of partial document generation and maintenance, and derive requirements for their realisation. The remainder of the chapter is centred around these requirements and how Elucidative Development fulfils them.

4.1 General Idea and Running Example

Partial generation of documents is a possibility to reduce the risk of inconsistencies due to human mistakes. However, as outlined in Chap. 2, the maintenance, i.e., the update of such documents, is not easy, because the regeneration of the documents would discard the manually written content.

We propose a novel method for the creation and maintenance of partially generated documents. This approach uses generation directives, which transform views or parts of views of the software system, such as models or code, to document content, such as text or images. The generation directives exist side by side with manually written content. If the document must be updated, it is not completely regenerated. Instead, only the content created by the generation directives is regenerated, leaving the manually written content intact. Additionally, changes in the views that occur after the writing of the document are automatically identified and reported with the help of the generation directives. We call this new approach *Elucidative Development*. The name is inspired by *Elucidative Programming*, a documentation approach for source code (see Sect. 8.1.3). The term *"elucidative"* means, according to the Merriam-Webster online dictionary[1], “to make (something that is hard to understand) clear or easy to understand”. The term *“development”* has been chosen to express that this approach is feasible throughout the whole software development process.

Definition 5 (Elucidative Development)**.** *Elucidative Development (ED) is an approach for the creation and maintenance of consistent documents by partial generation. Partial generation means that some parts of the documents are automatically created and updated, while other parts are handwritten. The generation of document content is controlled by generation directives, which are embedded in the handwritten document parts. Document content can be generated from views of the software system or from existing content of the document itself. Elucidative Development comprises tool support for:*

- *the evaluation of the generation directives*
- *the reporting of changes in the referenced views*
- *the reporting of changes in the generated content*

Documents which have been created by means of ED are called *elucidative documents.* We will refer to the non-generated parts of elucidative documents as “handwritten”, like in Definition 5, or as “manually written”. Unless noted otherwise, we consider an elucidative document a logical unit, even if it might physically be spread across multiple files. Therefore, we always talk of “the elucidative document” in the singular.

Definition 6 (Computed Document Fragment)**.** *Computed Document Fragments (CDFs) are the content fragments of elucidative documents which are automatically created.*

In some cases, it is necessary to reference an entire view to generate a CDF. In other cases, it is sufficient to reference only a part of a view, for example, a single source code file from the source code view. We call the generation directives *active references* from now on. A full definition of active references is given in Sect. 4.3.

Figure 4.1 shows an example of documentation that has been created using ED. It contains both handwritten content and CDFs. The CDFs are the figure, which displays a class diagram and a caption, and the class and attribute names in the running text, which are framed by a black box.

[1]`http://www.merriam-webster.com/dictionary/elucidative`

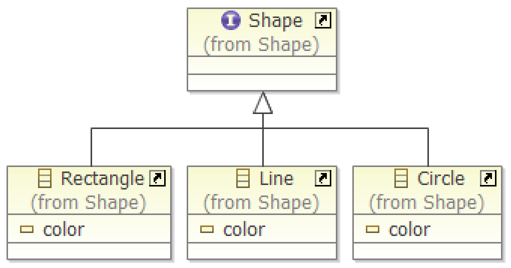


Figure II: Shapes

This diagram shows that the different shapes all implement the `Shape` interface. This makes it possible to use them all in a uniform manner. One method which takes a shape as parameter can work on rectangles, lines and circles.

...

The colour of the circles can be changed. The `Circle` class has a `color` attribute (see Fig. II), which can be set. After the colour has been set, the circle can be drawn.

Figure 4.1: Document with computed text fragments (framed) and a figure.

4.2 Requirements of Elucidative Development

ED can be used for the creation and maintenance of semi-formal and formal documents. In the following, we will discuss ED for semi-formal documents. ED for formal documents will be covered in Chap. 5.

The creation and maintenance of semi-formal documents presents several challenges. We explain these challenges and derive requirements, which ED must fulfil.

A major property of ED is, that every time an active reference is evaluated, it produces a consistent CDF. When a document is to be published, all active references must be evaluated and replaced by their resulting CDFs.

Challenge 1. *Dealing with active references during document creation is potentially troublesome for the document author because the active references are very technical and interrupt the flow of reading and writing. Additionally, active references do not give an impression what the final CDF is going to look like.*

Requirement 1. *The author must have the possibility to hide the technical details of active references while editing the elucidative document. He shall be able to see the CDFs instead of the active references.*

This can be achieved by the introduction of a presentation layer. Such a presentation layer can display active references as CDFs. When the views or the parts of the elucidative document, which are referenced by the active reference, are changed, the presentation layer updates the CDFs accordingly.

Challenge 2. *According to the discussions from above, the presentation layer would update the CDFs as soon as the referenced data is changed. This behaviour is desired if the elucidative document itself is referenced, i.e., if the CDF is computed from the content of the elucidative document. In this case, document changes are immediately reflected in the CDFs and local consistency within the document is enforced. However, the behaviour is not desired if another view of the software system is referenced, i.e., if the CDF is computed from the content of another view. The reason is that the software system might be globally inconsistent when views are changed. These inconsistencies must not be automatically reflected in the elucidative document.*

Requirement 2. *When a part of an elucidative document is changed and there is an active reference which points to this part, the corresponding CDF must immediately be updated. In contrast, if a referenced view is changed, the corresponding CDF must not be updated without explicit permission of the author.*

Challenge 3. *If the document is big, the author might be unaware that referenced views have changed. In that case, he cannot trigger the update of the affected CDFs and the document becomes inconsistent. Additionally, even if all CDFs have been updated, they might not look like expected. Imagine, for example, that several classes have been added to a Unified Modeling Language (UML) class diagram, which destroyed the layout of the corresponding graphical class diagram CDF.*

Requirement 3. *The author must be supported by a guidance system. He must be informed when referenced views have been changed in order to allow or deny an update of the corresponding CDFs. Additionally, he must be informed when CDFs have been updated, so he can proofread them.*

ED focuses on the specialties of partly generated documents. Standard document management tools and techniques are not explicitly considered. Nevertheless, ED tools must provide common features, for example a revision history or change tracking management. Otherwise they will probably not be accepted by users.

Challenge 4. *Building word processors or other document editing tools specific for ED requires tremendous effort. Additionally, people tend to avoid using new editors if they do not have to.*

Requirement 4. *The implementation of tool support for ED should focus on the problems with the generation and maintenance of partly generated documents. Existing standard components, such as editors, should be reused if possible.*

4.3 Structure and Basic Concepts of Elucidative Documents

When the author writes an elucidative document, he can write text and insert images, like in a normal document. Additionally, he can insert *active references* with the help of the *Elucidative Development Environment (EDE)*, a special editing environment for the creation and maintenance of elucidative documents. Active references comprise the necessary information for the computation of CDFs. The structure of an elucidative document and its active references is shown by the metamodel in Fig. 4.2. In this section, we will describe the classes of the metamodel and their relationships.

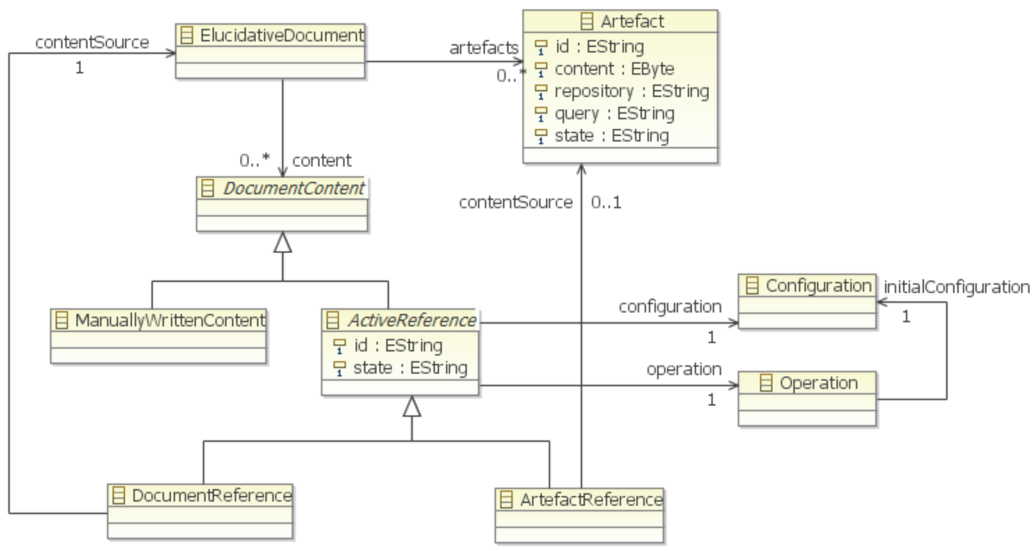


Figure 4.2: Elucidative document metamodel.

Figure 4.3 shows an overview of the main constituents of ED and the relationships between them in a more informal way. The general composition of the figure matches that of the figures from Sect. 8.1 in the Related Work chapter. This eases the comparison to related documentation approaches.

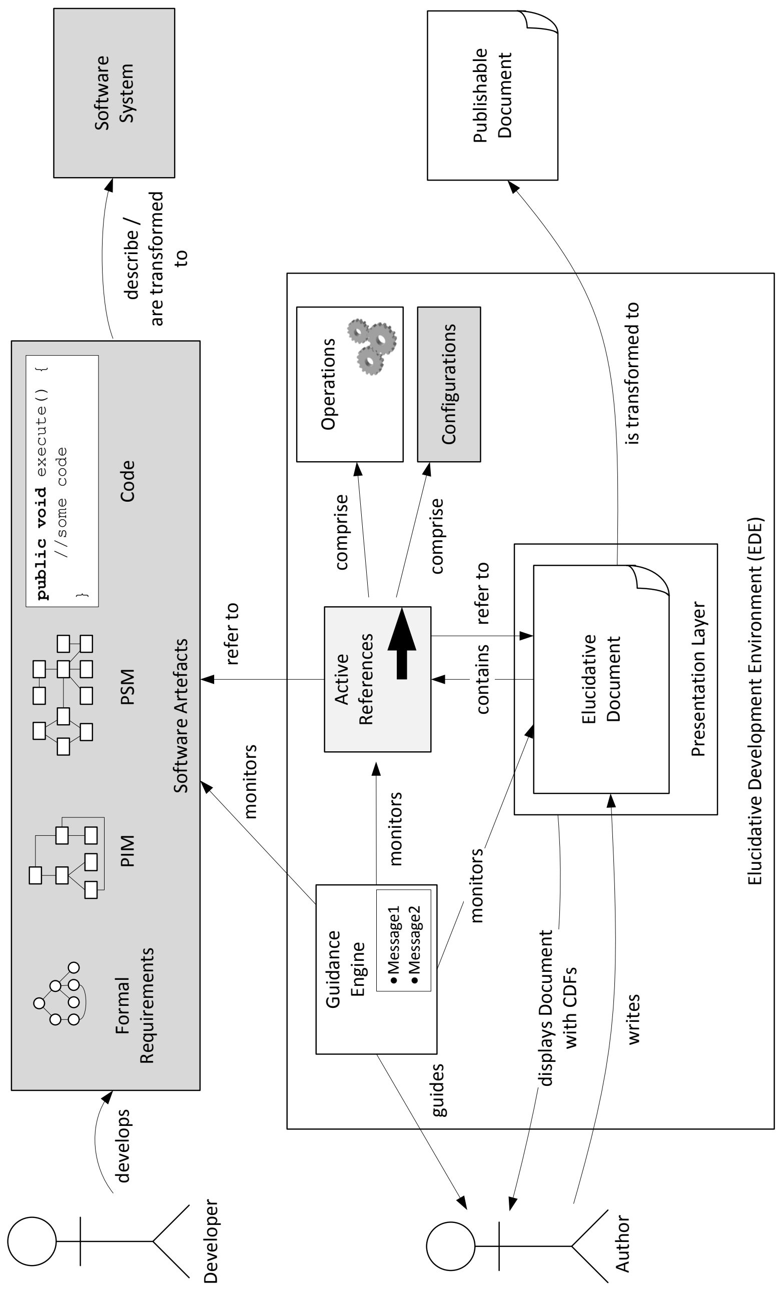


Figure 4.3: Elucidative Development overview.

The author can choose how the embedded active references are displayed. They can either be shown directly or they can be hidden and the corresponding CDFs are displayed instead (see Requirement 1). The author can select the desired display mode in the EDE. Figure 4.4 shows a schematic representation of an elucidative document where the active references are visible.

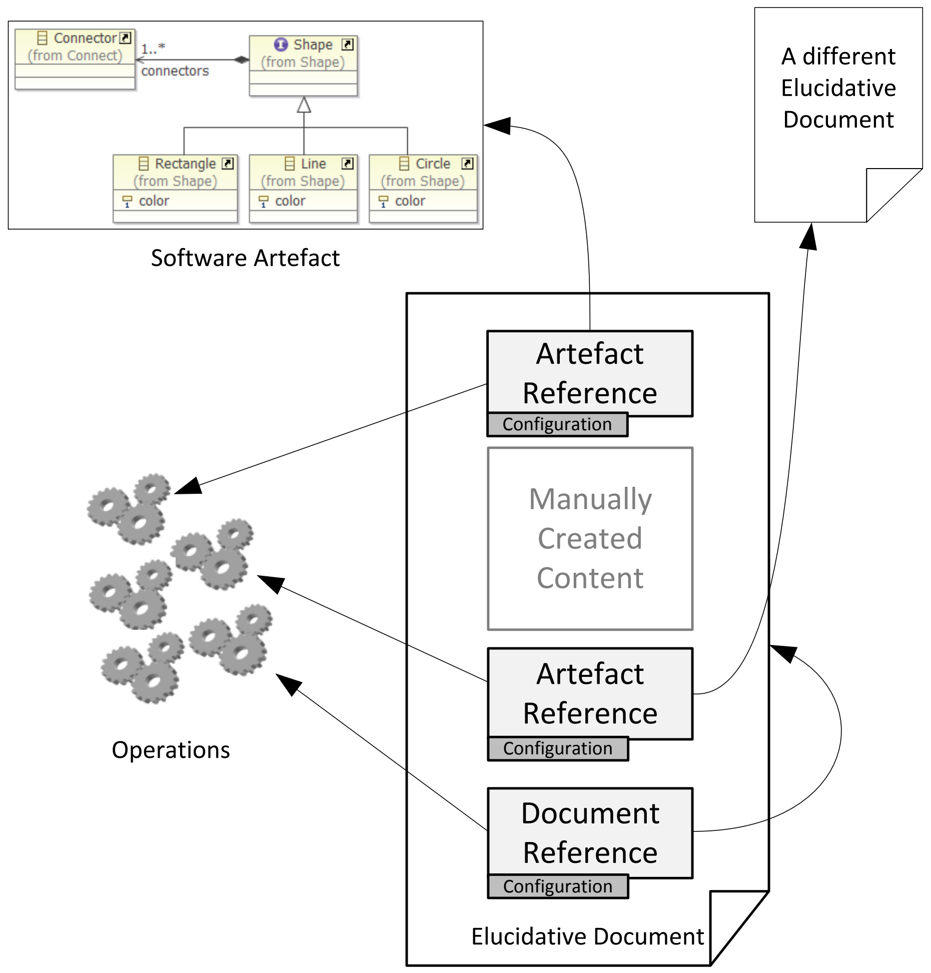


Figure 4.4: Different kinds of active references.

4.3.1 Artefact

ED has been designed to ease the writing about one or several views of a software system. Therefore, it must be possible to insert information from the views into an elucidative document. *Artefacts* represent information from the views and can be used as source for the computation of CDFs.

Definition 7 (Artefact)**.** *An artefact is a copy of a view of the documented software system or a part thereof.*

In the following, we say that an artefact represents a view when the artefact is a copy of that view. This includes the case that the artefact is only a copy of a part of the view (e.g., a single file).

An artefact comprises the following data:

- `id`: The unique identifier of the artefact.
- `content`: The copied data of the view, or view excerpt, which this artefact represents.
- `repository`: The data store which contains the original view of the artefact, e.g., the local file system, a source control repository, or the internal repository of a modelling tool.
- `query`: The location of the artefact's original view within the repository, e.g., a file path, a path in the source control repository, or a query for the modelling tool repository.
- `state`: A flag indicating the update state of the artefact.

Together, the `repository` and the `query` describe the origin of the view. Thus, it is possible to inspect the original view and compare its content to the artefact. If the view and the artefact `content` are identical, the artefact represents the current state of the view. If the view and the artefact differ, the view has been changed since the artefact has been created, and the artefact is outdated. The information whether an artefact is outdated is important for the guidance system, which will be covered in more detail in Sect. 4.5. The `state` flag carries the update state of the artefact and is also explained in Sect. 4.5.

An artefact is initially created by the document author. With the help of the EDE, he can interactively select the desired repository and formulate a query. In the simplest case, this would comprise selecting a file from the file system. If the content of the view is different from the content of the artefact, the author can choose to update the artefact. Updating the artefact means that its content is replaced by the current content of the view. When an artefact is updated, all CDFs which depend on this artefact are recomputed (see Sect. 4.4.1).

4.3.2 Active Reference

Definition 8 (Active Reference). *An active reference is a collection of data, which allows the computation of CDFs.*

The term *active reference* has been chosen because an active reference *references* the information which is necessary to compute a CDF. The term *active* has been chosen because the term *reference* is rather general and used very frequently in computer science. Furthermore, it indicates the relationship to *active documents* (see Sect. 8.2.2).

An active reference comprises the following data:

- `id`: The unique identifier of the active reference.
- `state`: A flag indicating the update state of the corresponding CDF.
- `configuration`: The configuration, which influences the content and the formatting of the CDF.
- `operation`: The operation which creates the CDF.

Additionally, an active reference, or rather its concrete subclasses, have a content source. The content source of an active reference is the source of information from which the CDF will be computed. This can either be the elucidative document itself or an artefact. Depending on the kind of content source, the active reference is either called *document reference* or *artefact reference* (see below).

Usually, an active reference is added to the document content by the document author via the EDE. Exceptions are active references in formal documents, which are automatically created. They are discussed in Chap. 5.

Definition 9 (Document Reference). *A document reference is an active reference whose content source is the elucidative document itself.*

Only the current elucidative document can be referenced with a document reference. For other elucidative documents it is necessary to use artefact references instead (see Fig. 4.4).

Definition 10 (Document CDF)**.** *A document CDF is a CDF which has been computed from a document reference.*

Definition 11 (Artefact Reference)**.** *An artefact reference is an active reference whose content source is an artefact.*

There can be arbitrarily many artefact references for each artefact. We say that an artefact reference whose content source is a certain artefact *depends* on that artefact.

Definition 12 (Artefact CDF)**.** *An artefact CDF is a CDF which has been computed from an artefact reference.*

4.3.3 Configuration

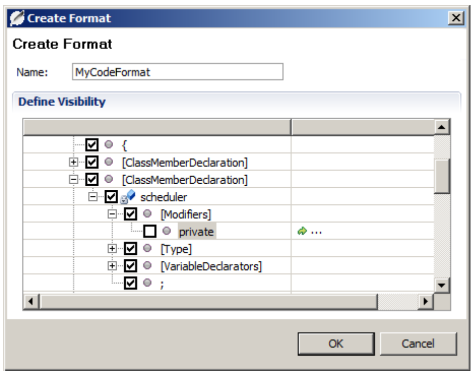
A CDF is not only computed from the content source. The author can also provide additional values that influence the CDF. These values are called *configuration.* They are basically parameters of the operation.

Definition 13 (Configuration)**.** *A configuration is a set of key-value-pairs, which influence the generation of a CDF. The possible keys and value ranges are defined by the operation. The concrete configuration values are provided by the author.*

The values of a configuration are used during the execution of an operation. It is necessary that the configuration contains all values expected by the operation. Since the operation knows which values it needs, it is the operation's responsibility to provide an *initial configuration*, which contains all the keys for which the author must specify a value.

Configuration values can be used to influence both the content and the visual appearance of the CDF. An example of the former is a configuration value used as a filter, which references the desired part(s) of the content source. For configurations of document references, the filter could consist of XPath expressions, IDs of document content, or even line numbers. For configurations of artefact references, the filter could match a method from a class in source code or some classes from a UML class diagram. Examples of the latter are the setting of a background colour, the setting of a scaling factor if the CDF is an image, or the specification of an image caption.

A configuration is filled by the author who inserts the active reference. According to Requirement 1, the author should not be forced to deal with the technical details of a configuration. Therefore, the EDE should present the author a dialogue where some options can be set, such as in Fig. 4.5. The selected options are then translated by the EDE into a configuration.



Create Format

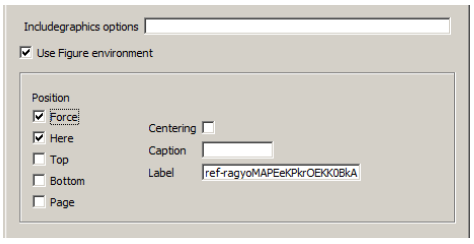
Create Format

Name: MyCodeFormat

Define Visibility

```
☑ ● {
⊞ ☑ ● [ClassMemberDeclaration]
⊟ ☑ ● [ClassMemberDeclaration]
  ⊟ ☑ scheduler
    ⊟ ☑ ● [Modifiers]
        ☐ ● private      ...
    ⊞ ☑ ● [Type]
    ⊞ ☑ ● [VariableDeclarators]
      ☑ ● ;
```

OK Cancel



Includegraphics options

☑ Use Figure environment

Position

☑ Force

☑ Here

☐ Top

☐ Bottom

☐ Page

Centering ☐

Caption

Label ref-ragyoMAPEeKPkrOEKK0BkA

(a) Dialogue for filtering CDFs which represent a code listing.

(b) Dialogue for the formatting of figure CDFs in a LaTeX-based elucidative document.

Figure 4.5: Configuration dialogue examples.

4.3.4 Operation

Definition 14 (Operation)**.** *An operation o is a function which transforms several inputs into a CDF.*

$CDF = o(s, c_1, \ldots, c_n); n \geq 0$, *where s is the content source and c_1 to c_n are values from the configuration (it is also possible that there are no configuration values at all).*

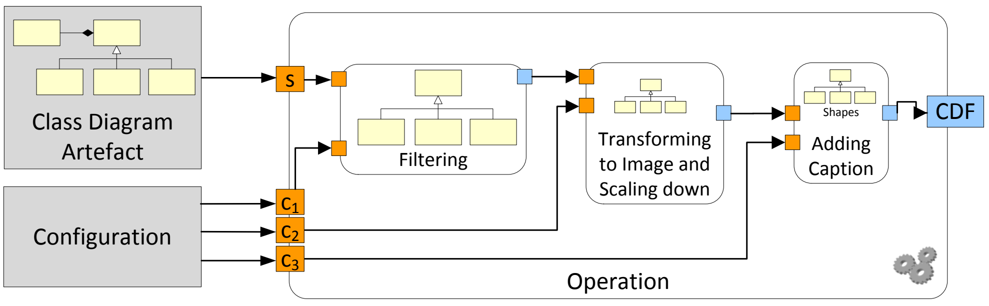


Figure 4.6: Operation as data-flow diagram.

Figure 4.6 shows an operation, which transforms a class diagram and a configuration with 3 values into an image with a caption. The operation is divided into 3 stages. The first stage is a filter, which removes some unwanted classes from the diagram. The classes to be removed are specified in the configuration. The second stage converts the diagram into an image and scales it down. The scaling factor is provided by the configuration. The third stage adds a caption to the figure, i.e., it creates a compound CDF, which consists of an image and the caption text. The caption to be set is also provided by the configuration. Finally, the complete CDF is returned by the operation.

Operations are developed and deployed to the EDE by a developer. The author only uses the built-in operations.

4.3.5 Elucidative Development, Grammar-Based Modularisation and Slots

For some discussions in the upcoming sections and chapters, the distinction between active references and CDFs is not important. Both are two sides of the same coin. Therefore, we introduce the concept of *slots*. This term has been chosen in accordance with the term *slot* from Grammar-based Modularisation (GBM), which we introduced in Sect. 3.1.

Definition 15 (Slot)**.** *A slot is an area of variability in a document, which is described by an active reference and can be filled with a CDF. Thus, a slot is a part of an elucidative document that represents generable content.*

If the format of an elucidative document is based on a tree grammar, it is considered typed, as discussed in Sect. 6.1. In this case, ED is an application of GBM. The "language" in GBM corresponds to the document format in ED, which is based on a tree grammar. A slot in an elucidative document contains both a name and a syntactic category, like a slot in GBM.

If we look at the slot as an active reference, the slot name is the id of the active reference. The syntactic category is either explicitly stated in the active reference or it can be queried using the EDE. If we look at the slot as a CDF, again, the slot name is the id of the active reference. The syntactic category is the type of the CDF root node.

In general, GBM allows the nesting of slots and fragments. A fragment form can contain slots, which are bound to other fragment forms, which in turn can contain other slots, and so on. While it would theoretically be possible to extend ED with nested slots, we have not found indications that this would add value. Therefore, slots in ED are defined only at the root level, i.e., in the elucidative document. CDFs, which correspond to fragment forms, do not contain slots.

4.4 Presentation Layer

According to Requirement 1, the author should be able to hide the active references during the writing of the elucidative document. A presentation layer is responsible for displaying the CDFs instead of the active references and for updating CDFs when the content source, i.e., the artefacts or the elucidative document, has been changed. This gives rise to a number of issues that must be considered.

4.4.1 Updating Computed Document Fragments

Artefact CDFs and document CDFs have different update strategies (see Requirement 2). Document CDFs must be updated *immediately* when the referenced document content has changed. This ensures that the document is locally consistent.

In contrast, artefact CDFs must not be updated immediately when the corresponding view of the software system has changed. The update of the artefact CDFs in the elucidative document must be *delayed* until the software system is globally consistent. In general, it is not possible for a tool to decide when all views of a software system are globally consistent. Therefore, the document author must explicitly update the artefacts, when he believes the corresponding views of the software system are globally consistent. After the artefact update, the EDE recomputes all artefact CDFs.

4.4.2 Displaying Incomputable References

There are cases in which it is not possible to compute a CDF. For example, the artefact might have been deleted, or the configuration contains values which cause the operation to fail. These errors must be reported to the author. One way is to employ guidance, as described in Sect. 4.5. Additionally, the presentation layer can be used to display special CDFs which contain error messages. Once the error has been resolved, the error message CDFs can be replaced by the "real" CDFs again.

4.4.3 Chaining Slots

Document CDFs can be computed from arbitrary document content, including other CDFs. We say, that a slot can depend on other slots. Thus, it is possible to create chains of slots. In the following, we will explain how chained CDFs are computed. Afterwards, we will show the advantages of chained CDFs.

If a slot depends on one or more other slots, the CDF cannot be computed before the other CDFs have been computed. Such a *dataflow graph* can theoretically be arbitrarily long and must not contain cycles. The first CDF in a dataflow graph is either computed from an artefact or from the handwritten content of the elucidative document. Figure 4.1 on page 21 shows an excerpt of an elucidative document with dependent CDFs. The example shows an elucidative document with a CDF that represents a figure and some CDFs that represent identifiers from the figure (framed). Since the CDFs of the identifiers describe the content of the figure, it makes sense to use the figure as content source.

The approach of updating document content which depends on other document content or external data is known as *transconsistency* [3]. Transconsistency is discussed in Sect. 8.2.2. The dataflow graph is called *transconsistent dataflow graph* in [3].

A document is transconsistent if its computable components are immediately updated along the dataflow graph when an input value changes. Thus, the automatic update of document CDFs is an application of transconsistency. In contrast, the automatic update of artefact CDFs is not an example of transconsistency, because the update does not immediately take place when the corresponding view is changed. Instead, the update of artefact CDFs is delayed until the author triggers the update of the dependent artefact. We call this modified concept *weak transconsistency.*

Definition 16 (Weak Transconsistency)**.** *Weak transconsistency is a strategy of delayed updates for documents with a transconsistent dataflow graph. In contrast to transconsistency, dependent values along the dataflow graph are not immediately recomputed when an input value has changed. The recomputation requires an explicit permission, usually by a human.*

Figure 4.7 shows the dataflow graph of the example from Fig. 4.1. Solid lines indicate that CDFs are immediately updated when their input CDF (the figure) is changed. The dashed line indicates that the figure is an artefact CDF which is only updated after an explicit artefact update.

As shown in Fig. 4.7, it is sometimes desired to compute CDFs from CDFs which contain a figure. But raw images are not suited as input for other CDFs, because it is difficult to automatically extract useful information from them. Therefore, we propose a method to make intermediate results of the CDF operation accessible, which are better suited as input for the computation of CDFs than images.

As we have seen in Fig. 4.6, operations do not necessarily have to be performed in one atomic step. They can be divided into multiple stages. The result of each operation stage is the input for the next operation stage.

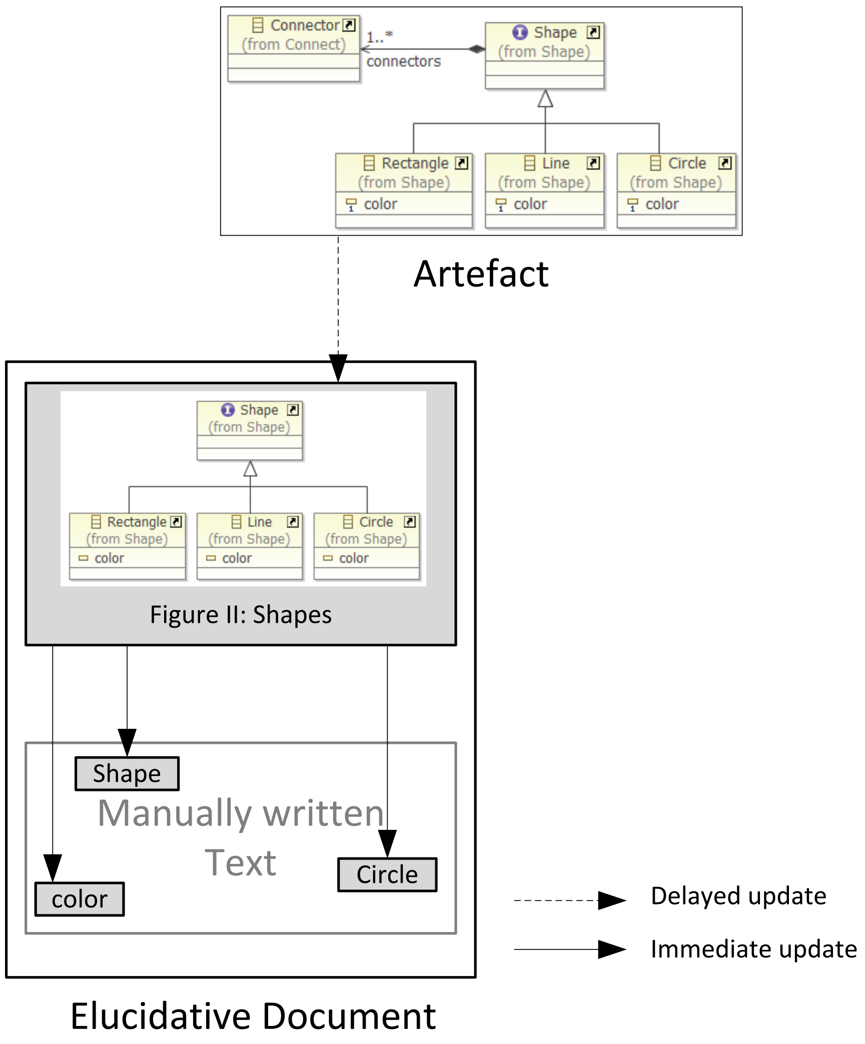


Figure 4.7: Dataflow graph for weak transconsistency.

The last stage produces the CDF. If the results of the intermediate operation stages are made accessible from outside the operation, they can be used as input for other operations. Figure 4.8 shows the operation from Fig. 4.6, but now the result of the filtering stage is made available via output r_1. Thus, the text CDFs from Fig. 4.1 can be computed from the results of the operation's intermediate filtering stage. Figure 4.9 displays this approach graphically.

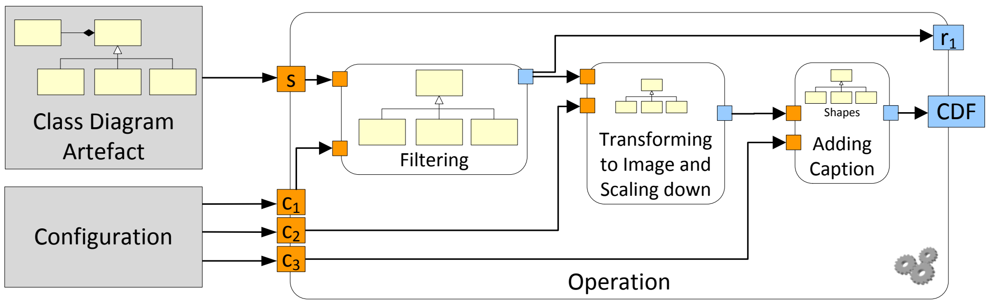


Figure 4.8: Operation with multiple outputs.

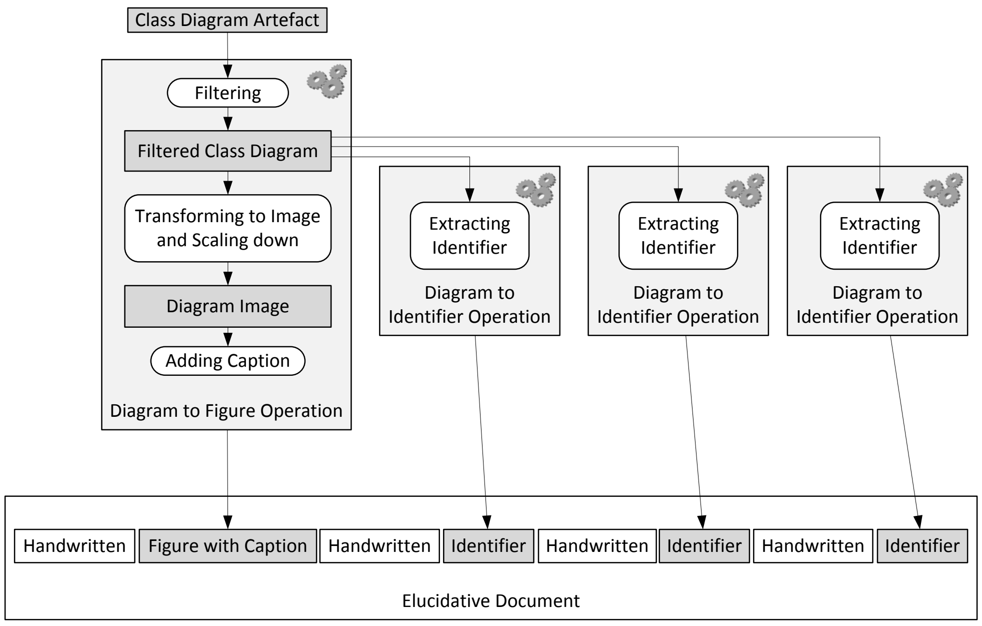


Figure 4.9: Detailed dataflow graph with intermediate operation results.

Operations with multiple outputs are not covered by Definition 14, though. Therefore, we present an extended definition.

Definition 17 (Operation (extended))**.** *An operation o is a function which transforms several inputs into a tuple of outputs.*

$(CDF, r_1, \ldots, r_m) = o(s, c_1, \ldots, c_n); m \geq 0, n \geq 0$ *, where r_1 to r_m are intermediate results, s is the content source and c_1 to c_n are values from the configuration.*

The advantages of chained CDFs in comparison to “normal” CDFs are better consistency enforcement and easier maintenance.

Better Consistency

Imagine an operation that transforms a class diagram artefact into a CDF with an image, like in the example from Fig. 4.1. The image shows various classes and the running text explains them. The textual explanation explicitly refers to the image, using CDFs with the class and attribute names.

At some time in the future, the author decides that the diagram image is too big and should be made smaller. He changes the configuration and filters out some classes which he considers unimportant for the documentation. However, the text still explains these classes and states that they are shown in the figure. Therefore, the text and the figure are locally inconsistent.

If the CDFs are chained, a semantic relationship is established between them. Local inconsistencies as the one described above can automatically be found and reported. The CDFs with the class and attribute names (identifiers) are computed from the CDF with the figure instead of the original diagram artefact. If a class is removed from the figure, the CDFs with identifiers of the removed class cannot be computed anymore. The guidance system (see Sect. 4.5) can give the author an appropriate error message. This would not have been possible, had the CDFs with the identifiers been computed directly from the diagram artefact.

Easier Maintenance

Now imagine that the referenced class diagram has been moved to a different location by a developer. All CDFs in the elucidative document whose active references point to the old diagram location cannot be computed anymore. However, if the CDFs with the identifiers depend on the CDF with the figure, the author only has to repair or recreate the active reference of the CDF with the figure, so that it points to the new location of the class diagram. Once the CDF with the figure can be computed again, the CDFs with the identifiers can be automatically computed, too. Had all CDFs with identifiers pointed to the class diagram directly, it would have been necessary to repair each active reference individually.

4.5 Guidance

An important property of ED is guidance for the author, as stated by Requirement 3. ED does not enforce the use of a certain technology for the guidance system. Furthermore, the scope of the guidance system is variable, depending on a concrete implementation of the EDE. In this section, we will present a minimal, state-based guidance system, that covers basic consistency scenarios. After the formal description, we will present an example, how the guidance system behaves during the update of an elucidative document.

For the guidance system presented in this section, we make the following assumptions. These assumptions may or may not apply to alternative realisations of the guidance system.

- An artefact reference is immutable. It is not possible to change the content source of an artefact reference in order to point to another artefact. If another artefact should be referenced, the old artefact reference must be deleted and a new one must be created.
- The document format of the elucidative document supports cross references. The EDE is aware of the cross references.
- A large part of the guidance system is meant to inform the author about CDF changes. Thus, the EDE must be configured to display the CDFs instead of the plain references.

4.5.1 Formal Definition of the Guidance State

ED guidance provides the author with information about inconsistencies in the document, together with recommendations how to resolve them. For example, the author must be notified

- when document CDFs have been automatically modified, so that the author can proofread them,
- when views of the software system have been changed, so that the artefacts are outdated and the elucidative document is not globally consistent anymore,
- when artefact CDFs have been modified after the author has updated the artefacts,
- or when CDFs cannot be computed, because the referenced artefacts or the referenced document content have been deleted.

The guidance system presented in this section is state-based. All notifications for the author and the recommended follow-up actions depend solely on the update states, also called just *states* in the following, of the active references and the artefacts. State changes of artefacts can influence the state of artefact references. Therefore, we are going to present the guidance system based on UML state charts, which can fire and react on events.

Definition 18 (Guidance System)**.** *A guidance system GS is a state system $GS = (artefacts, references_A, references_D, SC, state_{Artefacts}, state_A, state_D, cr, crmodified)$, where*

(i) $artefacts$ *is the set of existing artefacts.*

(ii) $references_A$ *is the set of artefact references.*

(iii) $references_D$ *is the set of document references.*

(iv) SC *is a triple of state charts* $(SC_{Artefacts}, SC_A, SC_D)$*, which define the states and transitions of the artefacts, the artefact references and the document references, respectively.*

(v) $state_{Artefacts} : artefacts \rightarrow SC_{Artefacts,states}$ *is a function that returns the current state of an artefact.* $SC_{Artefacts,states}$ *is the set of possible states defined by the state chart* $SC_{Artefacts}$*.*

(vi) $state_A : references_A \rightarrow SC_{A,states}$ *is a function that returns the current state of an artefact reference.* $SC_{A,states}$ *is the set of possible states defined by the state chart* SC_A*.*

(vii) $state_D : references_D \rightarrow SC_{D,states}$ *is a function that returns the current state of a document reference.* $SC_{D,states}$ *is the set of possible states defined by the state chart* SC_D*.*

(viii) cr *is the set of cross references inside the elucidative document.*

(ix) $crmodified \subseteq cr$ *is the set of cross references that point to document references whose state indicates a modification.*

We will present the actual state charts for artefacts, artefact references and document references below. Each state chart will first be presented graphically. The transitions between the states are labelled with numbered abbreviations. These abbreviations denote events that trigger the transitions and events that are fired by the transitions. Detailed textual descriptions of all events will be given below each state chart, grouped by states.

4.5.2 Artefact State Chart

Changes in the views of the software system might make the elucidative document globally inconsistent. If this is the case, the affected CDFs must be recomputed. But before the CDFs can be recomputed, the author must update the corresponding artefacts (see Sect. 4.4.3 and Requirement 2). Therefore, it is important that the author is notified about changes in the views.

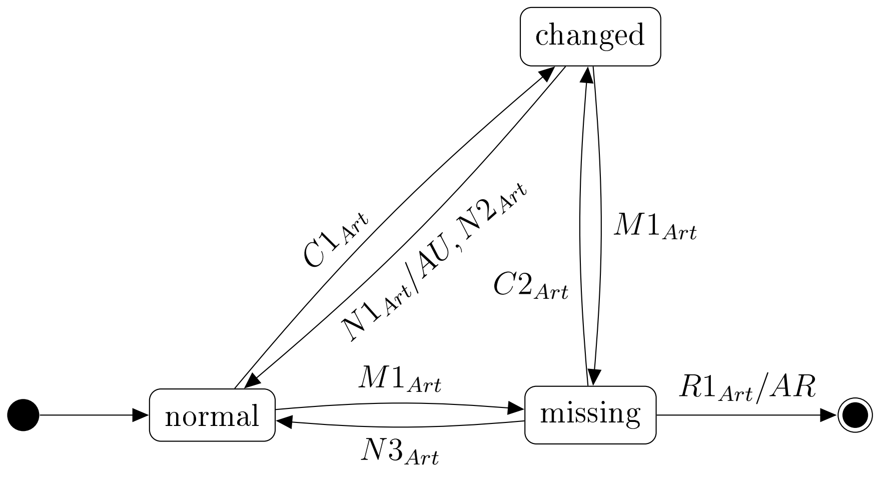


Figure 4.10: Artefact state chart.

The artefact state chart $SC_{Artefacts}$, shown in Fig. 4.10, is the foundation of this notification mechanism. When a new artefact is created, a new artefact state chart for that artefact is initialised. The state chart comprises the states *normal*, *changed*, *missing* and a final state. When the state chart enters the final state, the artefact is deleted.

Normal State

An artefact is in the *normal* state when it has been created. If an artefact is in the *normal* state, the content of the artefact is the same as the content of the view it represents. Therefore, no guidance messages are necessary.

The following events can occur in the *normal* state:

$C1_{Art}$ The view has been changed. Its content is different from the artefact content. The resulting state is *changed.*

$M1_{Art}$ The view has been deleted or moved. The resulting state is *missing.*

Changed State

An artefact is in the *changed* state when its content is different from the content of the view it represents. This can happen, for example, if a developer changed code for which an artefact has been created. If an artefact is in the *changed* state, a corresponding message must be presented to the author.

The following events can occur in the *changed* state:

$N1_{Art}$ The author has accepted the view changes and considers them a new revision that must be documented. He uses the EDE to overwrite the artefact content with the new view content. We say that the artefact is being *updated.* The resulting state is *normal.* Furthermore, an *Artefact Update (AU)* event is fired. This event triggers a transition in the state charts of the dependent artefact references and causes a recomputation of the corresponding CDFs. See the artefact reference state chart below for details.

$N2_{Art}$ Before the author has updated the artefact, the view has been changed again. The view content now happens to be the same as before, identical to the artefact content. We say that the view has been *restored.* The resulting state is *normal.*

$M1_{Art}$ The view has been deleted or moved. The resulting state is *missing.*

Missing State

An artefact is in the *missing* state when its view has been deleted or moved to a different location. If an artefact is in the *missing* state, a corresponding message must be presented to the author. The author can then decide to accept that the view has disappeared and invalidate the artefact and all dependent artefact references. Alternatively, he can wait for the view to be recreated or moved back.

The following events can occur in the *missing* state:

$N3_{Art}$ The view has been restored, i.e., the view has been recreated or moved back to its former location. Furthermore, its content is the same as the artefact content. The resulting state is *normal.*

$C2_{Art}$ The view has been recreated or moved back to its former location. However, its content is different from the artefact content. The resulting state is *changed.*

$R1_{Art}$ The author has accepted that the view has been removed. This causes the artefact to be deleted. The resulting state is the final state. Furthermore, an *Artefact Removal (AR)* event is fired. Like the artefact update event, the artefact removal event triggers a transition in the state charts of the dependent artefact references. It causes the invalidation of all artefact references that depend on the removed artefact. See the artefact reference state chart below for details. The artefact removal is permanent.

Final State

An artefact is in the final state when the author has accepted that the corresponding view is missing. In contrast to the *missing* state, the final state cannot be left. If an artefact enters this state, it cannot be used anymore and is deleted. The deletion of the artefact fires the artefact removal event. This event invalidates all artefact references, which depend on the artefact, so they must eventually also be deleted (see the artefact reference state chart below for details).

4.5.3 Artefact Reference State Chart

When an artefact state chart fires an artefact update event, the dependent CDFs must be recomputed. A recomputed CDF might not look like the author has intended, though. Therefore, he should check it, possibly supported by a visualisation of the differences, and adjust it if necessary. Adjusting the CDF includes changing the configuration, or completely replacing the artefact reference with a new one.

If a recomputed CDF looks different from before, we call it *modified*. For each modified CDF, the author should check the surrounding document content, such as running text, for consistency. Following the spirit of Literate Programming, we believe that document content which refers to CDFs is usually close to these CDFs. This enables the author to deal with many local inconsistencies between CDFs and manually written content[2]. After the author has approved or revised the content surrounding a CDF, he can explicitly accept the CDF modification.

Similarly, when an artefact state chart fires an artefact removal event, the dependent artefact references must be invalidated. As a result, the cor-

[2]This is only a heuristic. It is not defined how big the "surrounding document content" is. It could be the whole chapter or only a few lines of text before and after the CDF. There is no guarantee that the author will find all parts of the content that are inconsistent with the modified CDF, but chances are increased that he does.

responding CDFs must be removed or replaced by error message CDFs (see Sect. 4.4.2). The author must be informed about all CDFs that have been modified or removed.

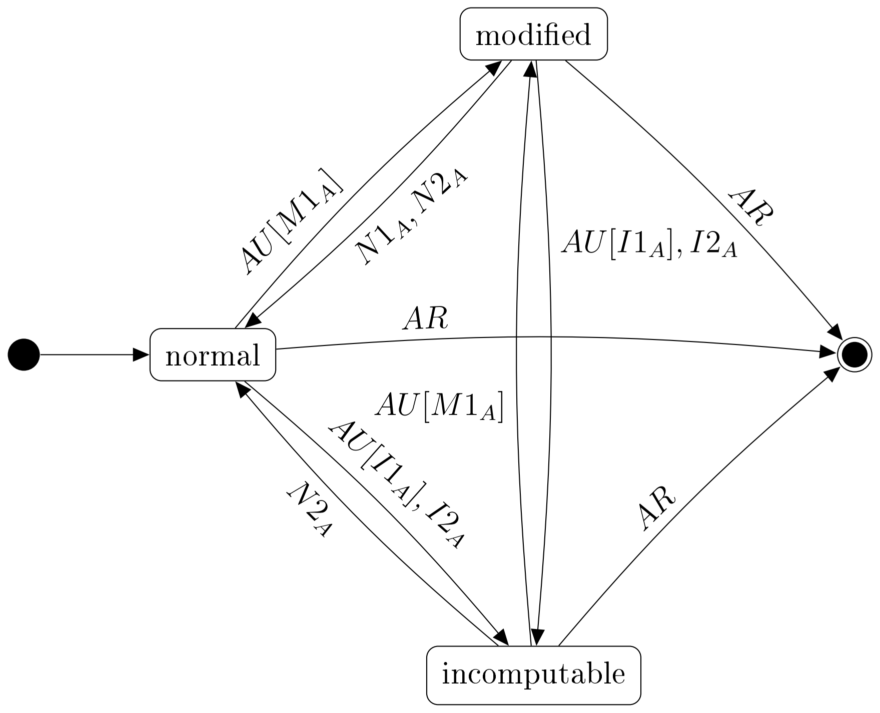


Figure 4.11: Artefact reference state chart.

The artefact reference state chart SC_A, shown in Fig. 4.11, is the basis for this notification mechanism. It comprises the states *normal*, *modified*, *incomputable* and a final state.

Normal State

An artefact reference is in the *normal* state when it has been newly created. An artefact reference being in the *normal* state means that there are no inconsistencies in the elucidative document w.r.t. the corresponding artefact. Therefore, no guidance messages are necessary.

The following events can occur in the *normal* state:

$AU[M1_A]$ The author has applied an update of the corresponding artefact, i.e., an artefact update event has been fired by the artefact state chart. The artefact update causes a modification of the CDF. The resulting state is *modified*.

$AU[I1_A]$ The author has applied an artefact update, and the CDF cannot be computed anymore. The resulting state is *incomputable.*

$I2_A$ The author has changed the configuration, and the CDF cannot be computed anymore. The resulting state is *incomputable.*

AR The author has applied the removal of the corresponding artefact, i.e., an artefact removal event has been fired by the artefact state chart. The resulting state is the final state.

Modified State

An artefact reference is in the *modified* state when a CDF has been modified due to an artefact update. If an artefact reference is in the *modified* state, a corresponding message must be presented to the author. When the author sees this message, he must check the modified CDF and proofread the surrounding document content. Afterwards, he must explicitly confirm the correctness and consistency of the CDF and the surrounding document content. Furthermore, the document might contain cross references to the modified CDF, which might have become inconsistent. Thus, a corresponding message must be presented to the author for each cross reference that points to the modified CDF. Again, the author must proofread the content surrounding the cross references and edit it if necessary. Then, he must confirm that it is consistent.

The following events can occur in the *modified* state:

$N1_A$ The author has examined and explicitly accepted the modified CDF. The resulting state is *normal.*

$N2_A$ The author has changed the configuration, which results in a new CDF. The resulting state is *normal.*

$AU[I1_A]$ The author has applied an artefact update, and the CDF cannot be computed anymore. The resulting state is *incomputable.*

$I2_A$ The author has changed the configuration, and the CDF cannot be computed anymore. The resulting state is *incomputable.*

AR The author has applied the removal of the corresponding artefact. The resulting state is the final state.

Incomputable State

An artefact reference is in the *incomputable* state when a CDF cannot be computed due to a configuration change or an updated artefact. If an artefact reference is in the *incomputable* state, the author must be informed that the CDF cannot be computed. The author can try to resolve the issue by changing the view, if he is allowed to, and updating the artefact again or by changing the configuration.

The following events can occur in the *incomputable* state:

$N2_A$ The author has changed the configuration, which results in a new CDF. The resulting state is *normal*.

$AU[M1_A]$ The author has applied an update of the corresponding artefact. The artefact update causes a modification of the CDF. The resulting state is *modified*.

AR The author has applied the removal of the corresponding artefact. The resulting state is the final state.

Final State

An artefact reference is in the final state when the corresponding artefact has been deleted. An artefact reference cannot leave the final state. The artefact reference has become useless and must eventually be deleted. A message must be presented to the author.

4.5.4 Document Reference State Chart

When the part of the elucidative document to which a document reference points is modified, the corresponding CDF might change. In contrast to artefact CDFs, the recomputation of document CDFs is not delayed, because this would make the document locally inconsistent (see Requirement 2). A recomputed CDF might not look like the author has intended, though. Therefore, he must check it, possibly supported by a visualisation of the differences, and adjust it if necessary. Adjusting the CDF includes changing the configuration, modifying the referenced document content, or completely replacing the document reference with a new one. Additionally, the author must check surrounding document content for consistency with the updated CDF. After the author has approved or revised the content surrounding a CDF, he can explicitly accept the CDF modification. The author must be informed about all document CDF changes.

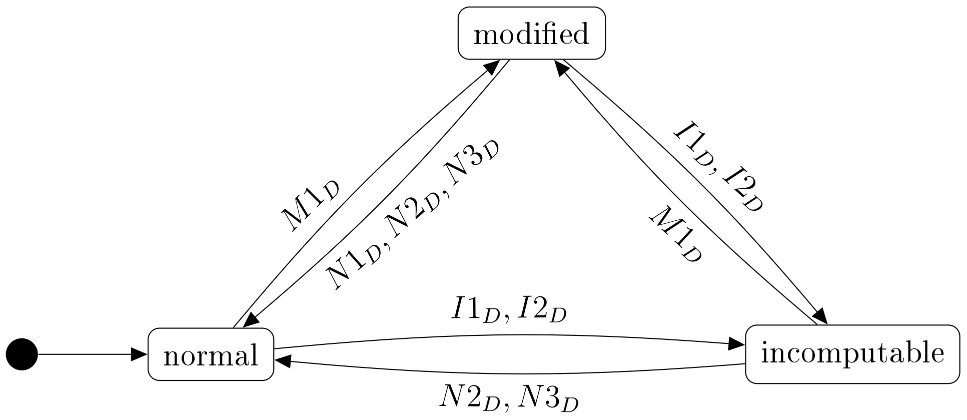


Figure 4.12: Document reference state chart.

The document reference state chart SC_D, shown in Fig. 4.12, is the basis for this notification mechanism. It comprises the states *normal*, *modified* and *incomputable*. In contrast to the artefact reference state chart there is no final state.

Normal State

A document reference is in the *normal* state when it has been newly created. A document reference being in the normal state means that there are no inconsistencies in the elucidative document w.r.t. the document reference. Therefore, no guidance messages are necessary.

The following events can occur in the *normal* state:

$M1_D$ The author has edited the referenced document content, which results in a different CDF. The resulting state is *modified*.

$I1_D$ The author has edited the referenced document content, and the CDF cannot be computed anymore. The resulting state is *incomputable*.

$I2_D$ The author has modified the configuration, and the CDF cannot be computed anymore. The resulting state is *incomputable*.

Modified State

A document reference is in the *modified* state when a CDF has changed due to changed document content. If a document reference is in the *modified* state, a corresponding message must be presented to the author. When the author sees this message, he must check the changed CDF and proofread the surrounding document content. Afterwards, he must explicitly confirm that the CDF and the surrounding document content are correct and consistent. Furthermore, the document might contain cross references to the updated CDF, which might have become inconsistent. Thus, there must be a corresponding message for each cross reference that points to the updated CDF. Again, the author must proofread the content surrounding the cross references and edit it if necessary. Then, he must confirm that it is consistent.

The following events can occur in the *modified* state:

$N1_D$ The author has examined and explicitly accepted the modified CDF. The resulting state is *normal.*

$N2_D$ The author has changed the configuration, which results in a new CDF. The resulting state is *normal.*

$N3_D$ The author has edited the referenced document content, which causes the CDF to change back to its original form. The configuration has not been changed. The resulting state is *normal.*

$I1_D$ The author has edited the referenced document content, and the CDF cannot be computed anymore. The resulting state is *incomputable.*

$I2_D$ The author has changed the configuration, and the CDF cannot be computed anymore. The resulting state is *incomputable.*

Incomputable State

A document reference is in the *incomputable* state when a CDF cannot be computed due to a configuration change or edited document content. If a document reference is in the *incomputable* state, the author must be informed that the CDF cannot be computed. The author can try to resolve the issue by changing the referenced document content or changing the configuration.

The following events can occur in the *incomputable* state:

$N2_D$ The author has changed the configuration, which results in a new CDF. The resulting state is *normal.*

$N3_D$ The author has edited the referenced document content, which causes the CDF to change back to its original form. The configuration has not been changed. The resulting state is *normal.*

$M1_D$ The author has edited the referenced document content, which results in a different CDF. The resulting state is *modified.*

4.5.5 Example

We will now illustrate these rather abstract descriptions with an example. The example is based on the elucidative document from Fig. 4.1. We will show how the elucidative document is updated and how the update affects its guidance state.

In order to guide the author, the guidance state must be transformed into messages. If possible, the guidance system should also show tasks, i.e., items of work that the author must perform in order to make the elucidative document consistent again. Some tasks can be executed automatically. It is only necessary for the author to trigger their execution, such as "Remove artefact". Other tasks can only be completed manually, such as "Proofread section". The example will contain both guidance messages and tasks.

After the document has originally been written, it has the guidance state presented below. We use symbolic names for the artefact and reference names. They appear in the same order as in Fig. 4.1. We do not explicitly list the state charts in the example, because we will always use the state charts that we introduced above.

$$\begin{aligned}
artefacts =& \{(\texttt{Shape.ecorediag})\}\\
references_A =& \{ref\_figure\}\\
references_D =& \{ref\_Shape, ref\_Circle, ref\_color\}\\
state_{Artefacts} =& \{(\texttt{Shape.ecorediag}, \mathit{normal})\}\\
state_A =& \{(ref\_figure, \mathit{normal})\}\\
state_D =& \{(ref\_Shape, \mathit{normal}), (ref\_Circle, \mathit{normal}), (ref\_color, \mathit{normal})\}\\
cr =& \{cr\_figure\}\\
crmodified =& \emptyset
\end{aligned}$$

Imagine, that the documented software has evolved after the creation of the elucidative document. The class diagram with the shapes has also changed. A new abstract class called `AbstractShape` has been added. The `color` attribute has been moved from the `Rectangle`, `Line` and `Circle` classes to `AbstractShape`. Consequently, the elucidative document has become inconsistent and must be updated. In the following, we will describe the necessary steps to update the elucidative document and how the guidance system supports the author. Fig. 4.13 shows what the elucidative document is going to look like after the update.

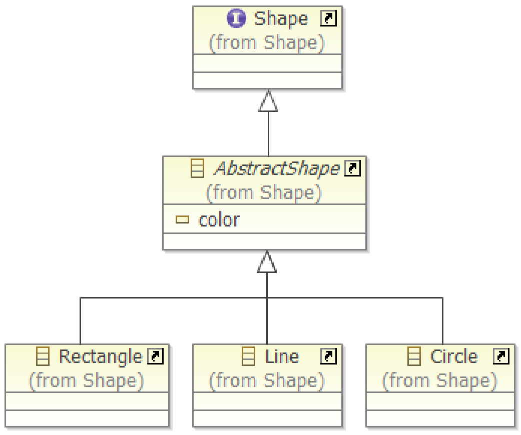


Figure II: Shapes

This diagram shows that the different shapes all extend `AbstractShape`, which in turn implements the `Shape` interface. This makes it possible to use all shapes in a uniform manner. One method which takes a shape as parameter can work on rectangles, lines and circles.

. . .

The colour of the circles can be changed. The `Circle` class has a `color` attribute (inherited from `AbstractShape`, see Fig. II), which can be set. After the colour has been set, the circle can be drawn.

Figure 4.13: Updated and corrected documentation.

In the EDE, there is an artefact which represents the class diagram. Therefore, the guidance system notices the change in the class diagram and the guidance state is changed to:

$$
\begin{aligned}
artefacts =& \{(\texttt{Shape.ecorediag})\} \\
references_A =& \{ref\_figure\} \\
references_D =& \{ref\_Shape, ref\_Circle, ref\_color\} \\
state_{Artefacts} =& \{(\texttt{Shape.ecorediag}, \mathit{changed})\} \\
state_A =& \{(ref\_figure, \mathit{normal})\} \\
state_D =& \{(ref\_Shape, \mathit{normal}), (ref\_Circle, \mathit{normal}), (ref\_color, \mathit{normal})\} \\
cr =& \{cr\_figure\} \\
crmodified =& \emptyset
\end{aligned}
$$

The guidance system reports the view change to the author. The message and the task that the guidance system presents could look like this:

- The view `Shape.ecorediag` has been changed.
  - Update artefact `Shape.ecorediag` in the repository.

When the author executes the task, an artefact update event is fired. This causes the CDF with the figure to be recomputed. The guidance state is changed to:

$$\begin{aligned}
artefacts =&\{(\texttt{Shape.ecorediag})\}\\
references_A =&\{ref\_figure\}\\
references_D =&\{ref\_Shape, ref\_Circle, ref\_color\}\\
state_{Artefacts} =&\{(\texttt{Shape.ecorediag}, normal)\}\\
state_A =&\{(ref\_figure, modified)\}\\
state_D =&\{(ref\_Shape, normal), (ref\_Circle, normal), (ref\_color, normal)\}\\
cr =&\{cr\_figure\}\\
crmodified =&\emptyset
\end{aligned}$$

This is only an intermediate state, though. In this example, all the document references point to the artefact reference ref_figure, which computes the class diagram CDF (Figure II). The class diagram CDF has been modified, and all the depending document CDFs must also be recomputed. It turns out, that the CDFs with the `Shape` and `Circle` identifiers are not affected by the modification of the figure CDF. Thus, they keep their *normal* state. But the referenced `color` attribute of the `Circle` class cannot be found, because it has been moved to the `AbstractShape` class. Therefore, the CDF of the document reference ref_color cannot be computed anymore and the state is changed to *incomputable*. Furthermore, the cross reference cr_figure points to the artefact reference ref_figure, which is marked as modified. Therefore, the cross reference is also marked as modified. The resulting guidance state is:

$$\begin{aligned}
artefacts =& \{(\texttt{Shape.ecorediag})\}\\
references_A =& \{ref\_figure\}\\
references_D =& \{ref\_Shape, ref\_Circle, ref\_color\}\\
state_{Artefacts} =& \{(\texttt{Shape.ecorediag}, \mathit{normal})\}\\
state_A =& \{(ref\_figure, \mathit{modified})\}\\
state_D =& \{(ref\_Shape, \mathit{normal}), (ref\_Circle, \mathit{normal}), (ref\_color, \mathit{incomputable})\}\\
cr =& \{cr\_figure\}\\
crmodified =& \{cr\_figure\}
\end{aligned}$$

The guidance system can use the guidance state to display the following information and tasks:

- Content “Figure II” generated from artefact `Shape.ecorediag` has automatically been modified.
  - Show in editor.
  - Mark as checked.
- There is 1 cross reference which points to the modified content “Figure II”.
  - Show cross reference in editor.
  - Mark cross reference as checked.
- Content “identifier” cannot be generated.
  - Show in editor.
  - Edit configuration.

The author uses the information from the guidance system to update the elucidative document. He changes the text, introduces CDFs for the new `AbstractShape` and edits the configuration of the incomputable document reference ref_color. He also marks the modified artefact reference ref_figure and the cross reference cr_figure as checked. The final guidance state looks like this:

$$\begin{aligned}
artefacts =& \{(\texttt{Shape.ecorediag})\}\\
references_A =& \{ref\_figure\}\\
references_D =& \{ref\_AbstractShape\_1, ref\_Shape, ref\_Circle, ref\_color,\\
& ref\_AbstractShape\_2\}\\
state_{Artefacts} =& \{(\texttt{Shape.ecorediag}, normal)\}\\
state_A =& \{(ref\_figure, normal)\}\\
state_D =& \{(ref\_AbstractShape\_1, normal), (ref\_Shape, normal),\\
& (ref\_Circle, normal), (ref\_color, normal),\\
& (ref\_AbstractShape\_2, normal)\}\\
cr =& \{cr\_figure\}\\
crmodified =& \emptyset
\end{aligned}$$

All artefacts and active references are in the *normal* state. The author sees no more guidance messages, so he knows that the elucidative document is globally consistent again. Thus, the guidance system has successfully guided the author through the process of identifying outdated and (potentially) inconsistent CDFs and making them consistent with the other views of the software system, as well as the handwritten document content.

4.6 Conclusion

In this chapter, we presented Elucidative Development as a means to create and maintain semi-formal, partially generated documents.

- We explained the structure of active references, how operations can compute content (CDFs) from a software artefact or the elucidative document itself, and how the author can influence the CDFs with configurations. Based on this, we showed how active references can be hidden from the author, so that he always sees up-to-date CDFs instead (Requirement 1).

- Afterwards, we introduced different update strategies, transconsistency and weak transconsistency, for document references and artefact references (Requirement 2). The discussion of transconsistency and weak transconsistency also included advantages of chained references and their realisation.

- Finally, we described how update states can be translated into guidance messages, which inform the author about problems or content that has been automatically changed (Requirement 3).

Chapter 5

Model-Driven Elucidative Development

In this chapter, we will extend the concept of Elucidative Development (ED), which has been presented in Chap. 4, so that it can be used for formal documents. While formal documents with manually written content could be created and maintained with ED in its basic form, called *basic ED* in the following, there are obstacles which make its use cumbersome.

In the following, we will present the reasons why basic ED is not suited for formal documents. Based on these, we will derive requirements for the use of ED for formal documents. Afterwards, we present an extension of basic ED that fulfils the new requirements and supports the creation and maintenance of formal documents.

For the motivation and examples we focus on Model-Driven Software Development (MDSD). MDSD is a software development methodology which uses models and metamodels as first-class citizens during development. An introduction to MDSD has been presented in Sect. 3.2. During an MDSD project, many models are created, and the documentation of models is usually very formal. Thus, MDSD is a suitable application area for the extended ED approach, which we call *model-driven elucidative development*[1].

5.1 General Idea and Running Example

The chapters and sections of a formal document are usually uniformly structured. Suppose that there exist models which describe the drawing tool from

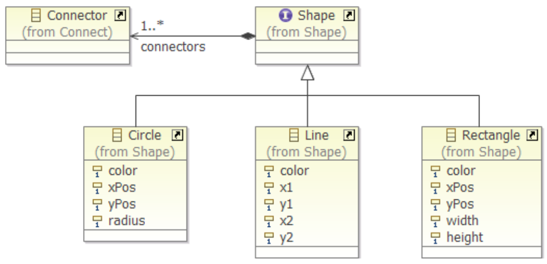
[1]MDSD is the main application area, but model-driven elucidative development is not limited to MDSD. It can also be used to create formal documents describing other structured views.

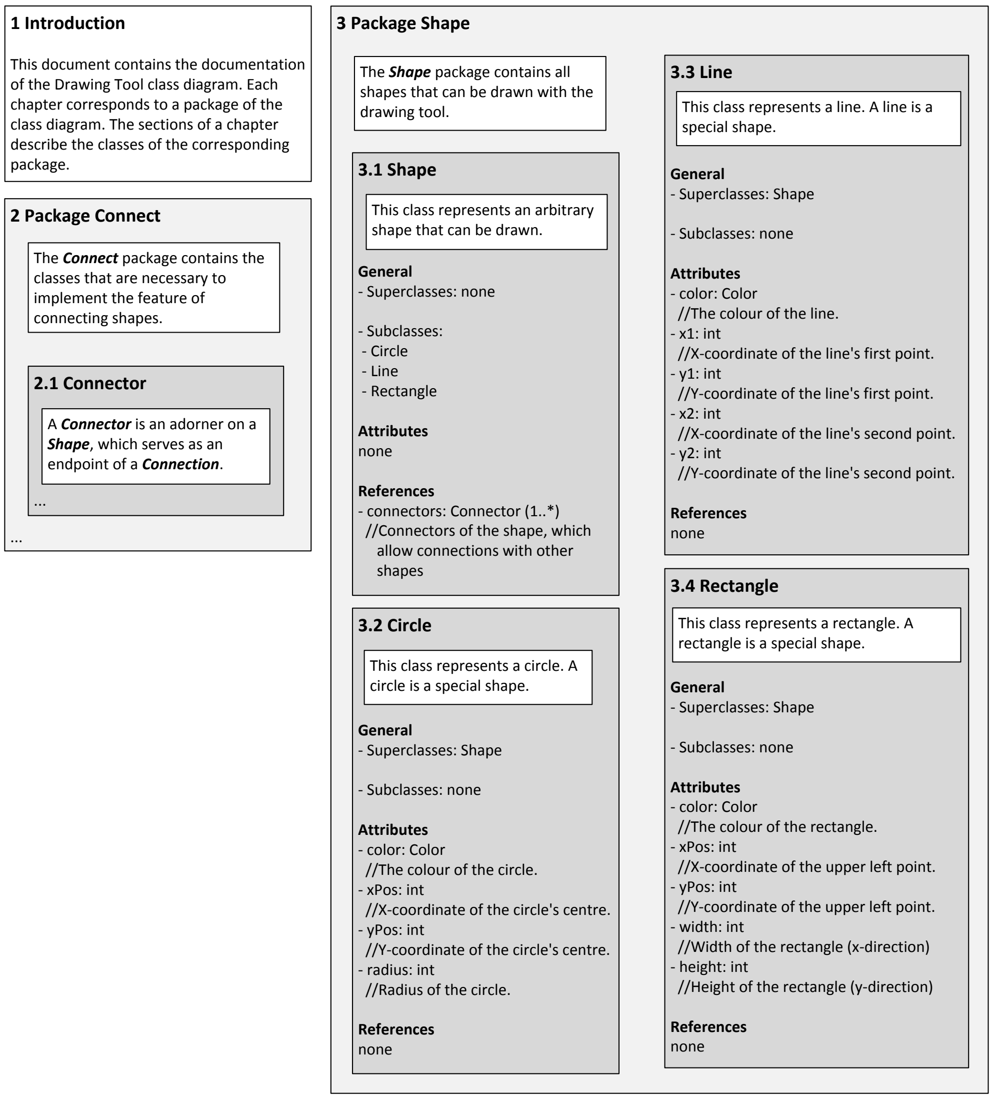
the running example in Chap. 2 and 4, and that these models must be documented. In the examples presented in this chapter, we will focus on a Unified Modeling Language (UML) class diagram, but the explained concepts can be applied to other kinds of models, too. The class diagram comprises multiple packages and each package contains multiple classes. In the following, we use the term *classes* to denote both interfaces and "real" classes, unless stated otherwise. Packages and classes will sometimes also be called *model elements*. The structure of the documentation reflects the structure of the class diagram.

Figure 5.1 shows an example. There is one chapter for each package, and one section for each class of the corresponding package. The first chapter is the main introduction, all other chapters are package descriptions. The chapters and the sections have a uniform structure. Chapters start with a heading, followed by an introductory text and the sections, which describe the individual classes. Sections do also start with a heading and an introductory text. Additionally, they contain technical information such as the superclasses, subclasses, attributes and references.

The colours in the figure indicate the content that could be generated from the model. The content of the dark-grey boxes, i.e., the sections with the class descriptions, could be generated from the classes. Superclasses and subclasses can be derived from the class diagram structure. Attributes and references can be read directly from the classes. The attributes and references of the classes can have comments. Therefore, the descriptions of the attributes and references in the document can also be generated. The introductory text, indicated by a white box, cannot be generated, though, because its content is not present in the class diagram. The content of the light-grey boxes, i.e., the chapters, could be generated from the packages. The chapter content can also be generated, except for the introductory text.

The aim of model-driven ED is to enable easy creation and maintenance of formal documents, taking into account the need for manually written content.





1 Introduction

This document contains the documentation of the Drawing Tool class diagram. Each chapter corresponds to a package of the class diagram. The sections of a chapter describe the classes of the corresponding package.

2 Package Connect

The ***Connect*** package contains the classes that are necessary to implement the feature of connecting shapes.

2.1 Connector

A ***Connector*** is an adorner on a ***Shape***, which serves as an endpoint of a ***Connection***.

...

...

3 Package Shape

The ***Shape*** package contains all shapes that can be drawn with the drawing tool.

3.1 Shape

This class represents an arbitrary shape that can be drawn.

General
- Superclasses: none

- Subclasses:
 - Circle
 - Line
 - Rectangle

Attributes
none

References
- connectors: Connector (1..*)
 //Connectors of the shape, which
 allow connections with other
 shapes

3.2 Circle

This class represents a circle. A circle is a special shape.

General
- Superclasses: Shape

- Subclasses: none

Attributes
- color: Color
 //The colour of the circle.
- xPos: int
 //X-coordinate of the circle's centre.
- yPos: int
 //Y-coordinate of the circle's centre.
- radius: int
 //Radius of the circle.

References
none

3.3 Line

This class represents a line. A line is a special shape.

General
- Superclasses: Shape

- Subclasses: none

Attributes
- color: Color
 //The colour of the line.
- x1: int
 //X-coordinate of the line's first point.
- y1: int
 //Y-coordinate of the line's first point.
- x2: int
 //X-coordinate of the line's second point.
- y2: int
 //Y-coordinate of the line's second point.

References
none

3.4 Rectangle

This class represents a rectangle. A rectangle is a special shape.

General
- Superclasses: Shape

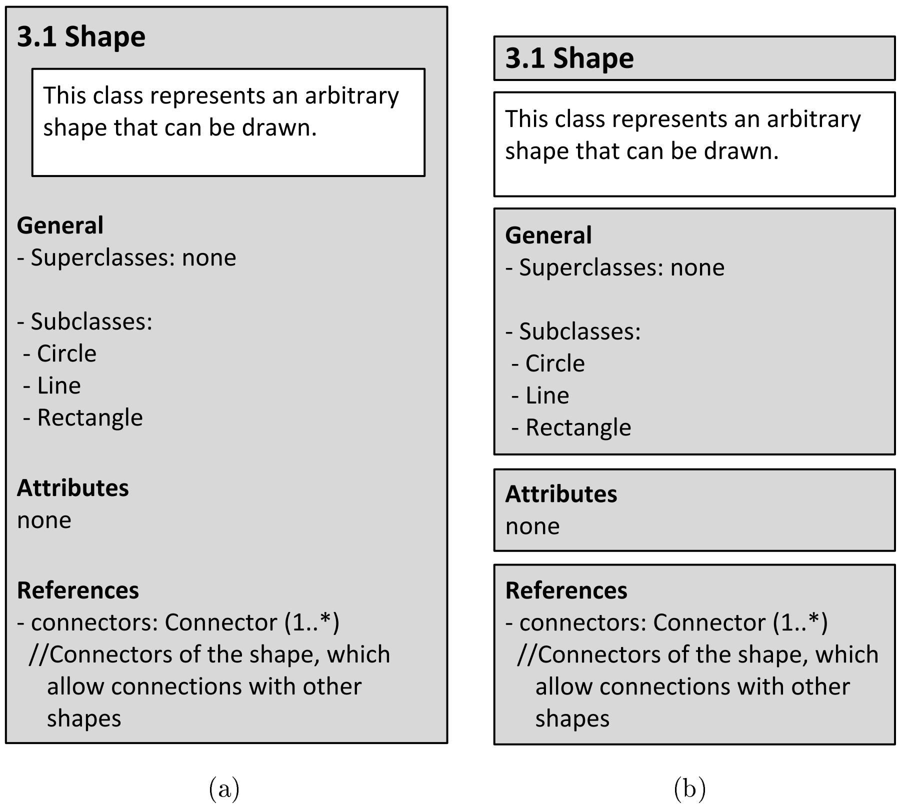
- Subclasses: none

Attributes
- color: Color
 //The colour of the rectangle.
- xPos: int
 //X-coordinate of the upper left point.
- yPos: int
 //Y-coordinate of the upper left point.
- width: int
 //Width of the rectangle (x-direction)
- height: int
 //Height of the rectangle (y-direction)

References
none

Figure 5.1: Documentation of a class diagram.

In terms of basic ED, a section can be viewed as a Computed Document Fragment (CDF). The manually written content inside the CDF could be determined by the configuration of the active reference. However, the practical realisation would be difficult. If the author should be able to format the text, e.g., make it bold or italic, the user interface to configure the active reference would have to duplicate many text formatting features of the text editor. For cases, where this is not feasible or possible, we propose an alternative approach. Generated content with embedded manually written content can be assembled from multiple CDFs. The author can add manually written content between the individual CDFs. We call a group of CDFs which form a logical unit *CDF group*. Figure 5.2 shows an example.


3.1 Shape

This class represents an arbitrary shape that can be drawn.

General
- Superclasses: none

- Subclasses:
 - Circle
 - Line
 - Rectangle

Attributes
none

References
- connectors: Connector (1..*)
 //Connectors of the shape, which allow connections with other shapes

(a)

3.1 Shape

This class represents an arbitrary shape that can be drawn.

General
- Superclasses: none

- Subclasses:
 - Circle
 - Line
 - Rectangle

Attributes
none

References
- connectors: Connector (1..*)
 //Connectors of the shape, which allow connections with other shapes

(b)

Figure 5.2: CDF groups: (a) shows the logical structure of a CDF group, (b) shows multiple physical CDFs (grey), which together form a CDF group.

Definition 19 (CDF Group). *A CDF group is a collection of one or more CDFs, which together form a logical unit. CDF groups can contain manually written content before, between or after the individual CDFs.*

5.2 Requirements of Model-Driven Elucidative Development

The management of CDF groups results in new requirements. In the following, we present the challenges of model-driven ED and the resulting requirements, similar to Sect. 4.2.

Challenge 5. *While it would be possible to let the author build CDF groups from simple CDFs, it would be inconvenient. If there are* n *CDF groups and each CDF group consists of* m *CDFs, the author would have to add* $n * m$ *CDFs in total. Depending on the size of the model, a document can contain hundreds of CDF groups. The author would have to take care that all CDFs are added in the correct order with the correct configuration. Creating the whole document that way is tedious and error-prone.*

Requirement 5. *It must be possible to add a CDF group as a whole to the elucidative document.*

Challenge 6. *In model-driven ED, the structure of the elucidative document reflects the structure of the documented model. If models are structured hierarchically, the documents must also be structured hierarchically.*

Requirement 6. *It must be possible to nest CDF groups to reflect the structure of the model in the elucidative document.*

Challenge 7. *If the structure of the model is changed, e.g., a class is added to, removed from, or moved within a class diagram, the elucidative document must be changed correspondingly. Furthermore, CDF groups must often appear in a certain order, e.g., alphabetically. If a model element, e.g., a class, is renamed, the corresponding CDF group must be moved to a different location in the elucidative document. With basic ED, it is not possible to add, remove or move active references automatically.*

Requirement 7. *It must be possible to automatically add, remove, rename and move CDF groups, reflecting the initial structure of the model and subsequent changes of the model structure. When CDF groups are physically moved inside the document, it is necessary that both the CDFs and the manually written content are moved together.*

Challenge 8. *In the spirit of Requirement 2, CDF groups which are computed from artefacts must not be updated without explicit permission of the author. In model-driven ED, this includes the addition, removal, renaming and moving of CDF groups. Thus, the author must be informed if the model has changed in a way that requires any of these changes in the elucidative document. However, this is beyond the scope of the basic guidance system.*

Requirement 8. *The guidance capabilities of basic ED must be extended to support the addition, removal, renaming and moving of CDF groups.*

5.3 Structure and Basic Concepts of Elucidative Documents in Model-Driven Elucidative Development

Model-driven ED is an extension of basic ED. It contains a number of new concepts, which build upon the concepts that we introduced in Sect. 4.3. The metamodel in Fig. 5.3 shows the new classes and their relation to the old metamodel. In this section, we will describe the new classes of the metamodel and their functionality.

5.3.1 The Unison of Active Reference Groups and CDF Groups

In Sect. 4.3.5, we explained how CDFs and active references are two sides of the same coin. The same relationship holds for CDF groups and *active reference groups*. We will explain active reference groups in more detail soon. Active reference groups are divided into static reference groups and dynamic reference groups. We will call a CDF group computed from a static reference group *static CDF group*, and a CDF group computed from a dynamic reference group a *dynamic CDF group*.

CDF groups are, possibly hierarchically structured, collections of CDFs which form a logical unit. Each CDF of a CDF group is computed from an individual active reference. The individual active references are automatically added to the elucidative document when an active reference group is added. From this follows that active reference groups are groups of active references, as the name suggests. The relationship between an active reference group and its active references is isomorphic to the relationship between the CDF group and its CDFs.

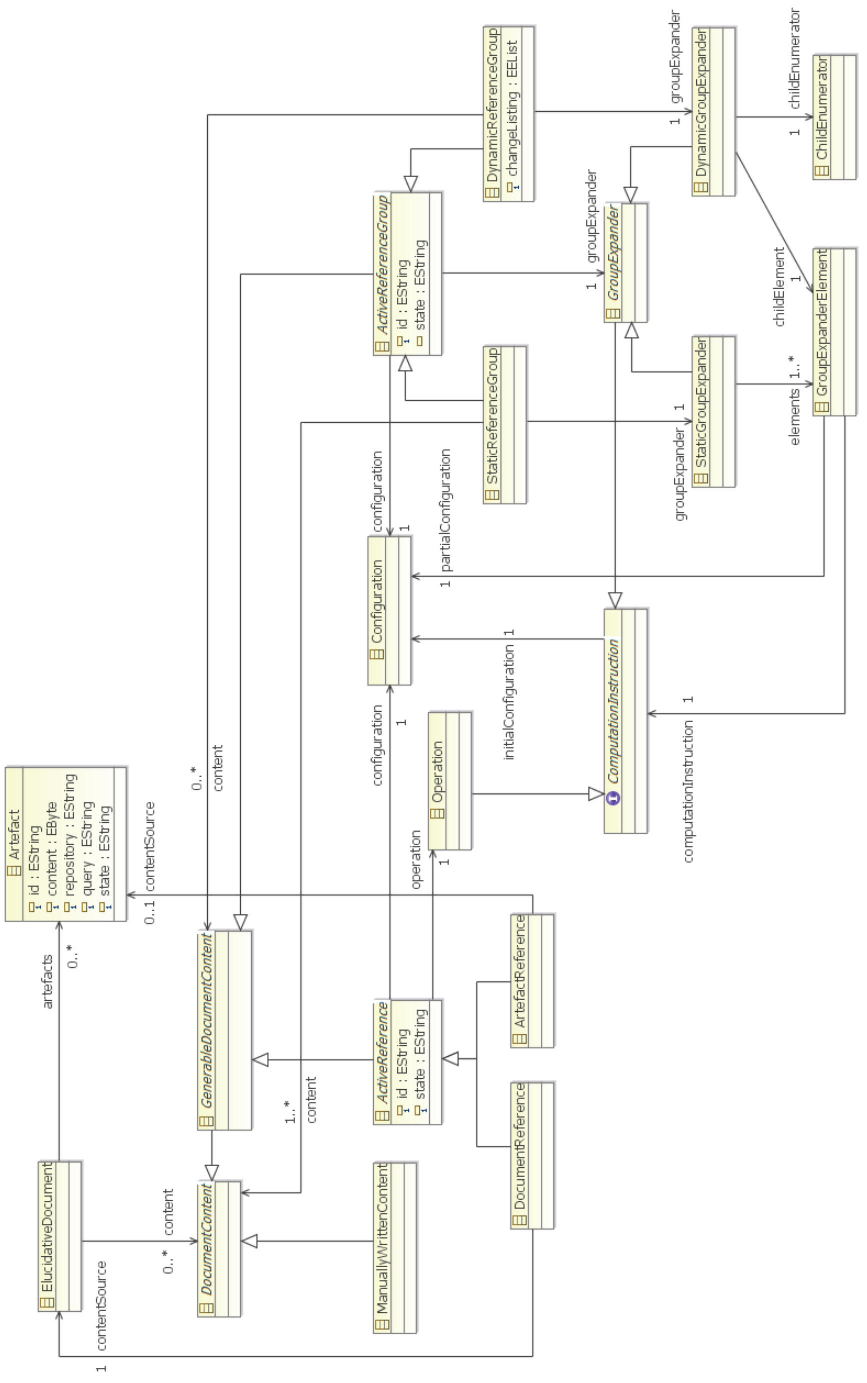


Figure 5.3: Model driven elucidative development metamodel.

5.3.2 Active Reference Group

CDF groups have been introduced as a way to combine CDFs and manually written content. According to Requirement 5, it must be possible to add CDF groups as a whole, instead of adding multiple CDFs individually. This can be achieved by active reference groups.

Definition 20 (Active Reference Group)**.** *An active reference group is a container inside an elucidative document. It contains active references or other active reference groups. This makes it possible to display an active reference group as CDF group. It also comprises the necessary data for the computation of the contained active references or active reference groups.*

Active references created by and contained within an active reference group are called *child active references* of the active reference group. Similarly, active reference groups created by and contained within an active reference group are called *child active reference groups*. If the distinction between child active references and child active reference groups does not matter, we only talk about *children*.

When an active reference group is added to the elucidative document, it is physically represented as a container or a range of text. Inside the container, or range of text, the computed children will be added. This will eventually lead to a collection of, possibly hierarchically structured, active references, which together can be displayed as the CDF group.

An active reference group comprises the following data:

- `id`: The unique identifier of the active reference group.
- `state`: A flag indicating the update state of the corresponding CDF group.
- `configuration`: The configuration, which influences the content and the formatting of the CDF group and its children.
- `groupExpander`: The computation directive for the children.

The children of an active reference group are computed when the active reference group is added to the elucidative document. We call this the *expansion* of the active reference group into a collection of children. The expansion of an active reference group is performed by the so-called *group expander*.

Definition 21 (Group Expander)**.** *A group expander is a computation instruction for the content of active reference groups. The group expander has two inputs, a content source (usually an artefact) and a configuration. With* *these two inputs, the group expander computes the children and recursively adds them to the elucidative document.*

Group expanders are usually parameterised. Similar to operations, which need a configuration to compute a CDF, a group expander needs a configuration to compute child active references or child active reference groups. Group expanders also have an initial configuration, which contains the keys for which a value is required. When the author adds an active reference group to the elucidative document, the Elucidative Development Environment (EDE) queries him for these configuration values. The complete configuration is passed to the group expander as input. If a group expander inserts another active reference group, it is the group expander's responsibility to forward the required configuration values to the child group expander.

There are two different kinds of active reference groups, namely *static reference groups* and *dynamic reference groups.* Both differ in the content they can contain (dynamic reference groups cannot contain manually written content) and the way their group expanders work. The two kinds of active reference groups and their group expanders are explained in the following.

5.3.3 Static Reference Group

The chapters and sections of a formal document have a mostly uniform structure. For example, all sections of the document shown in Fig. 5.1, which describe classes of a class diagram, look similar. They have a heading and consist of three parts: "General", "Attributes" and "References". The repeated use of the same structure is the application area of static reference groups.

Definition 22 (Static Reference Group)**.** *A static reference group is an active reference group which can contain manually written content, child active references, and child active reference groups. Static reference groups have a fixed structure.*

When a static reference group is added to the elucidative document, only its children are automatically added. Manually written content can be added afterwards by the author.

A static reference group comprises the following data:

- `content`: The list of document content that the static reference group contains. This includes other active reference groups, active references and manually written content.
- `groupExpander`: The static group expander, which is responsible for the computation of the content, i.e., the child active reference groups or child active references.

Static reference groups can be nested, i.e., they can contain other active reference groups and active references. Figure 5.4 shows an excerpt of Fig. 5.3, making the underlying composite design pattern clearly visible.

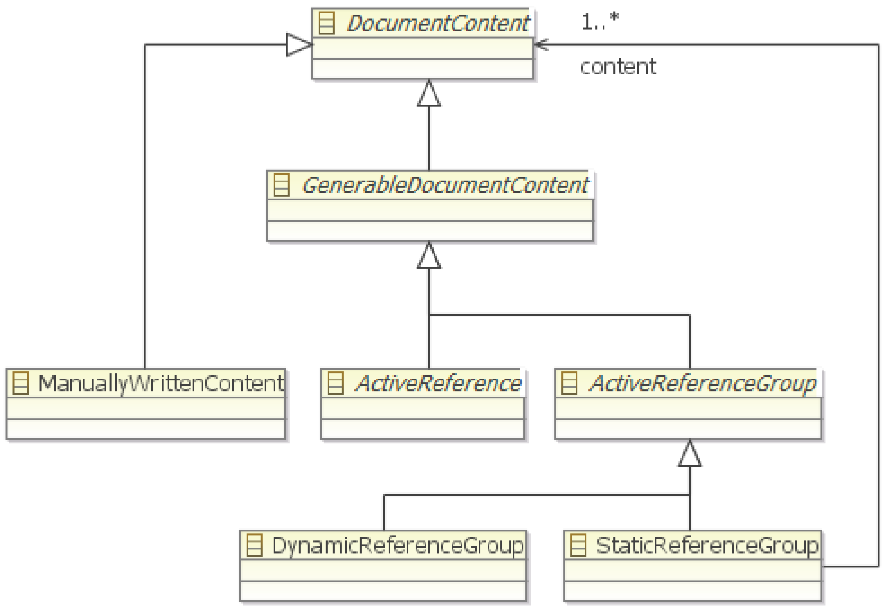


Figure 5.4: Static reference groups can contain arbitrary document content.

Constituents and Structure of Static Reference Groups

The expansion of a static reference group is performed by the *static group expander*. A static group expander contains an initial configuration, whose purpose we already explained, and a list of so-called *group expander elements*.

Definition 23 (Group Expander Element)**.** *A group expander element is a container for information that is used to create and configure a child active reference or child active reference group during the expansion of an active reference group.*

A group expander element consists of a *partial configuration* and a *computation instruction*. A partial configuration is a configuration in which some values are missing. The missing values are added before the static reference group is expanded. They are taken from the configuration which has been set as input of the static group expander. All values required by the group expander elements must be present in the group expander configuration, otherwise the expansion is not possible.

The computation instruction of a group expander element can either be an operation or another group expander. If the computation instruction is an operation, an active reference will be created. The active reference will later use this operation to compute a CDF. Figure 5.5 shows a static group expander, where all computation instructions are operations. This static group expander has been used to create the CDF group from Fig. 5.2b.

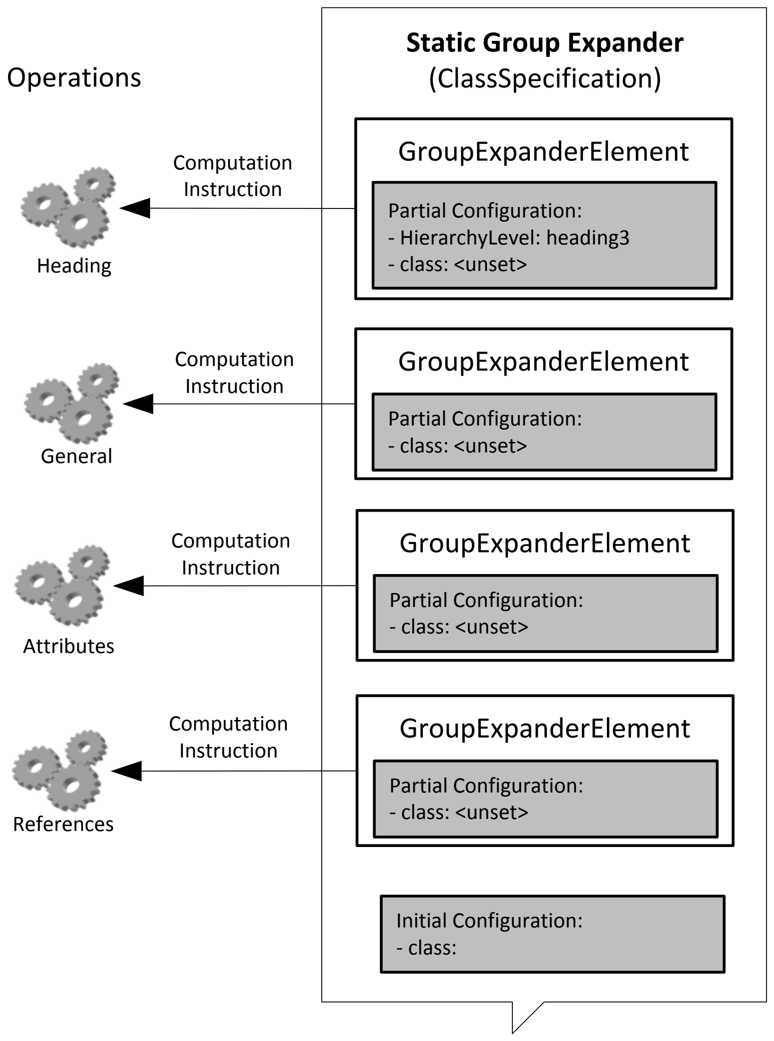


Figure 5.5: Static group expander with operations.

All partial configurations in the example have an unset “class” entry. The “class” entry determines, which class of the model the operation must use to create the CDF. The “Heading” configuration has an additional, preset entry called “HierarchyLevel”, which is set to heading3. The entry means that the CDF generated by the operation should be marked up as level 3 heading, for example, by enclosing it in `<h3>` tags in Extensible Hypertext Markup Language (XHTML). In our example, this entry has been preconfigured by the developer of the group expander because it was known in advance that the resulting static reference groups will always be at the third level in the document hierarchy[2].

If the computation instruction is a group expander, an active reference group will be created instead. The group expander will insert a child active reference group into the elucidative document. Figure 5.6 shows a static group expander with two group expander elements. It has been used to create chapters 2 and 3 (the package descriptions) of the example documentation from Fig. 5.1. The computation instruction of the second group expander element is a dynamic group expander.

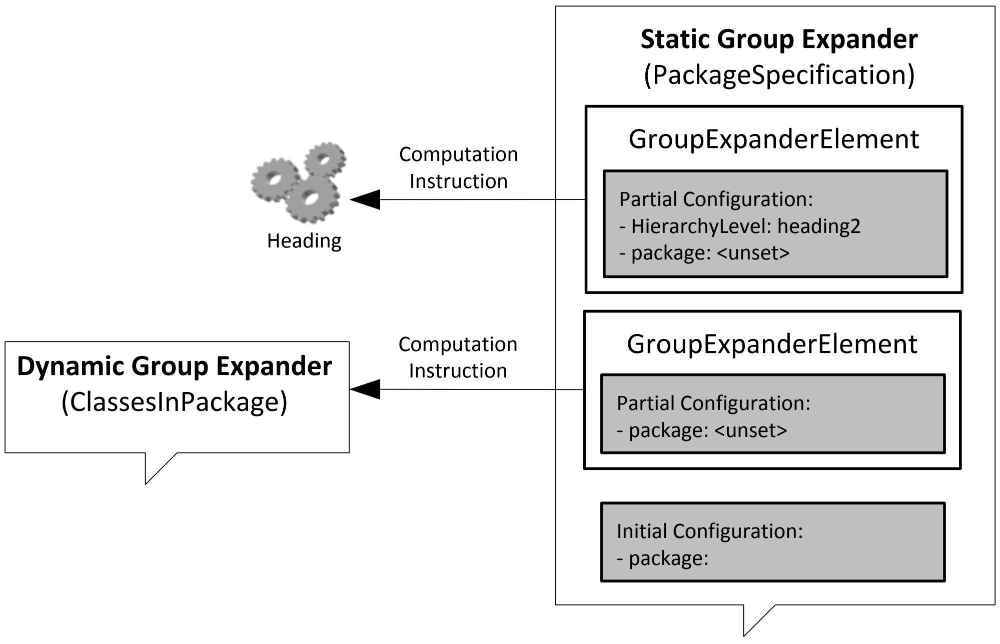


Figure 5.6: Static group expander with a dynamic group expander.

[2]The first level is the root document level and the second level is the package description.

Adding Static Reference Groups to the Elucidative Document

When the author adds a static reference group to the elucidative document, the EDE analyses the initial configuration and queries him for the required configuration values. The values are added to the configuration, which is then set as input of the static group expander. Afterwards, the static reference group is expanded and the child active references or child active reference groups are added to the elucidative document.

We will illustrate the process with the help of our running example. We will explain the insertion of the static reference group from Fig. 5.5, which contains four group expander elements with operations. This static reference group is called “Class Specification” group in the following.

When the author adds the “Class Specification” group to the elucidative document, he must provide the missing configuration values. The initial configuration contains an entry with the key “class”. Thus, the EDE queries the author for a class name. The result is stored in the configuration. Now, the static reference group can be added to the elucidative document, referencing the static group expander and the configuration. Afterwards, the static group expander can perform the expansion using the configuration.

The first step of the expansion is the completion of the group expander elements’ partial configurations. All partial configurations lack the “class” value. Therefore, the static group expander copies the “class” value from its configuration to all partial configurations. Afterwards, the partial configurations have no more unset values.

Then, all of the four group expander elements must be processed. All group expander elements have an operation as computation instruction. This means, that an active reference must be created for each group expander element. All information is available to create the active references and insert them into the elucidative document:

- The content source of the active reference is the content source of the static reference group, i.e., the class diagram.
- The operation of the active reference is the operation stored in the group expander element.
- The configuration of the active reference is the configuration stored in the group expander element.

After the insertion of all child active references, the static reference group is completely expanded. It can now be used to compute a CDF group.

Updating Static Reference Groups

It is not necessary to update static reference groups when the documented model has been modified, because the structure of a static reference group is independent from the model. However, it might be necessary to recompute the corresponding CDF groups. The content of static CDF groups is only determined by their child CDFs. Therefore, the recomputation of a static CDF group requires the recomputation of all child CDFs.

Summary

The use of static reference groups has several advantages over individual active references:

- The insertion of static reference groups into the elucidative document is easier, faster and less error-prone than the manual insertion of multiple active references.
- The generated content always has the same structure and layout. It is not possible that the author accidentally changes the order of the CDFs, or sets a wrong configuration value for the layout.
- The generated content of the multiple CDFs always stems from the same part of the content source. For example, it is not possible that the author accidentally selects two different classes from the class diagram as content source for the active references.

Static reference groups allow the insertion of whole CDF groups into the elucidative document. Requirement 5 is fulfilled. Furthermore, static reference groups can contain other static reference groups as children. This makes it possible to nest CDF groups. Requirement 6 is also fulfilled.

With static reference groups, the author could add the CDF groups for all classes of the referenced model. However, static reference groups cannot be used to automatically reflect structural model changes (adding, removing, renaming, moving classes) in the elucidative document. This is where dynamic reference groups come into play.

5.3.4 Dynamic Reference Group

The structure of a formal document corresponds to the structure of the documented model, as illustrated in Fig. 5.1. Usually, if the document structure differs from the model structure, the document is regarded inconsistent. For example, if a chapter of the document describes a package of a class diagram, and the individual sections describe the package's classes, it is necessary that there is one section in the chapter for each class. If the author has to add an active reference group or an active reference for each class by himself, he might miss a class or add some classes multiple times.

Furthermore, maintaining the elucidative document after a change in the class diagram would be difficult. The author would have to check each package for new classes and add the corresponding active reference groups or active references to the document. Similarly, he would have to check for incomputable active references and remove them if they refer to a deleted class. This approach is both time-consuming and error-prone. Therefore, model-driven ED offers the possibility to generate a whole list of active reference groups or active references and add them to the elucidative document automatically. Changes in the model can be automatically applied to the generated document content. Manually written content within the description of the individual classes is retained. This can be achieved by dynamic reference groups.

Definition 24 (Dynamic Reference Group). *A dynamic reference group is an active reference group whose children are determined dynamically from the model structure. If the model structure changes, the children of the dynamic reference group are automatically adapted. All children have an identical structure because they are computed from the same computation instruction. Dynamic reference groups cannot contain manually written content.*

Our running example contains a model package named `Shape`, which contains the the four classes `Shape`, `Circle`, `Line` and `Rectangle`. A dynamic reference group for the `Shape` package would generate four static reference groups, one for each class of the package.

A dynamic reference group comprises the following data:

- `changeListing`: A list of changes to the model artefact since it was last updated. The change listing is used for the computation of guidance hints.

- `content`: The list of generable document content that the dynamic reference contains. This includes other active reference groups and active references.

- `groupExpander`: The dynamic group expander, which is responsible for the computation of the content, i.e., the child active reference groups or child active references.

Constituents and Structure of Dynamic Reference Groups

The expansion of a dynamic reference group is performed by the *dynamic group expander*. In addition to the initial configuration, a dynamic group expander contains a group expander element and a *child enumerator*.

Definition 25 (Child Enumerator)**.** *A child enumerator is a function that returns a list of model element names. For each element of the list, a child will be added to the dynamic reference group during its expansion.*

All children of the dynamic reference group have the same structure. The structure is determined by the computation instruction of the group expander element. This ensures a uniform appearance of the resulting CDFs or CDF groups. Figure 5.7 shows an example of a dynamic group expander.

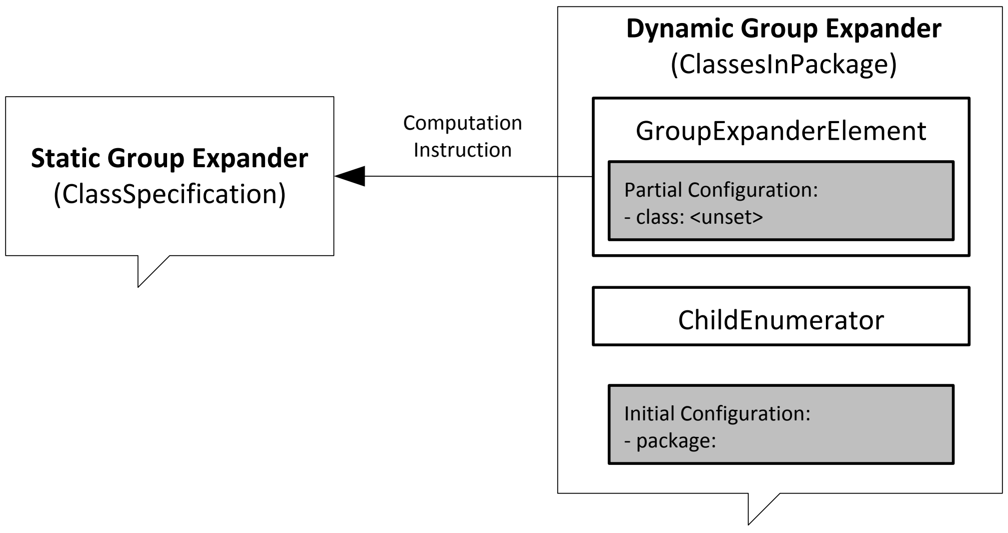


Figure 5.7: Dynamic group expander.

Adding Dynamic Reference Groups to the Elucidative Document

We will illustrate the process of adding a dynamic reference group to the elucidative document with the help of our running example. The following explanation is very comprehensive because it involves the recursive insertions of nested active reference groups.

Expanding the Package In the example, the author adds a static reference group with the static group expander from Fig. 5.6. We will call this static reference group the “Package Specification” group in the following. The resulting CDF group should be a whole chapter, as shown in Fig. 5.1. It should contain a heading with the package name and sections with descriptions of the package’s classes. Between the heading and the first class description, a manually written package description should be added.

In order to add the “Package Specification” group to the elucidative document, the author must provide the missing configuration values. The initial configuration contains an entry with the key “package”. Thus, the EDE queries the author for the name of a package. The selection of the author will be stored in the configuration. Now, the static reference group can be added to the elucidative document, referencing the static group expander and the configuration. Afterwards, the static group expander can perform the expansion using the configuration.

During the expansion, the partial configurations are completed, as explained in the previous example (see Sect. 5.3.3). That is, the “package” values in both partial configurations are set. Then, the first group expander element is processed and the active reference for the heading is added to the elucidative document. This has also been described in the previous example.

Expanding the List of Classes Afterwards, the second group expander element is processed. Its computation instruction is the dynamic group expander from Fig. 5.7. Thus, a dynamic reference group is added to the elucidative document, which is going to use this dynamic group expander for its expansion. We will call the dynamic reference group the “Classes in Package” group.

After the dynamic reference group has been added to the elucidative document, it is expanded. The expansion consists of three steps. First, the child enumerator receives the “package” value from the configuration and computes an alphabetical list of the class and interface names of the corresponding package. Then, copies of the group expander element are created, one for each element in the list. The list values are stored in the only unset entry of the partial configurations (the “class” entry in this example). Finally, for each group expander element copy, a static reference group is added to the elucidative document and expanded.

Expanding a Single Class The computation instruction of the group expander elements is the static group expander from Fig. 5.5. Its expansion has already been explained in the previous example (see Sect. 5.3.3).

The Result All active references and active reference groups have been recursively computed and added to the elucidative document. It is now possible to compute a nested CDF group like in Fig. 5.1.

Updating Dynamic Reference Groups

A dynamic reference group must be updated if the children do not correspond to the values computed by the child enumerator, i.e., there are excess or missing children, or the order of the children is different from the order computed by the child enumerator. This situation can occur for two reasons.

The first reason for an update of a dynamic reference group is a change in the configuration of the dynamic reference group. Imagine, that the dynamic reference group contains child active reference groups for all classes and interfaces of a class diagram package, like in our running example. In contrast to the running example, imagine that the configuration of the dynamic reference group has an additional entry that determines the sort order of the children. Initially, the value of this entry was set to alphabetical sorting of the dynamic reference group's children. Later, the configuration was changed, so that first the interface descriptions appear alphabetically, and then the class descriptions appear alphabetically. This requires a reordering of all child reference groups.

The second reason for an update of a dynamic reference group is that an artefact update event has been fired for the model artefact. The kind of change determines how the dynamic reference group must be updated:

- If a new model element has been added, a new child must be added to the dynamic reference group. The position of the new child must correspond to the position of the model element name in the list computed by the child enumerator.

- If a model element has been deleted, the corresponding child must also be deleted. This includes the manually written content that has been added by the author.

- If a model element has been renamed, and the new name appears at a different position in the list of model element names, the corresponding child must be moved. This comprises removing the child from its dynamic reference group and inserting it again at the new position. The manually written content must also be moved.

- If a model element has been moved (e.g., a class which has been moved from one package to another), the corresponding child must be moved.

This comprises removing the child from its old dynamic reference group and inserting it into the new dynamic reference group (if it exists in the document). The manually written content must also be moved.

Identifying Types of Changes When the model is changed, it is possible that some model elements (classes, packages) are missing, while others have been newly inserted. In general, it is difficult to distinguish whether the changes are "real" removals and additions, or whether some model elements have in fact been renamed or moved (e.g., a class being moved from one package to another package). The distinction matters because when model elements are renamed or moved, the corresponding child active reference groups or child active references must also be renamed or moved. Manually written content inside the active reference groups must be preserved. Only when a model element has really been deleted, it is permitted to remove the corresponding child, including the manually written content.

The changes performed on the model can be identified by *model matching*. A model matching algorithm produces a mapping between two models, stating which model element from one model corresponds to which model element from the other model. If the "original" and the changed model are matched, it is possible to deduce the basic changes that have been performed. Figure 5.8 shows an example. If two matching model elements reside in different places in the model (e.g., classes in different packages), the model element has been moved. If two matching model elements have a different name, the model element has been renamed. If a model element from the first model has no matching model element in the second model, it has been removed. If a model element from the second model has no matching model element in the first model, it has been added.

In advanced model matching scenarios, the splitting or joining of model elements must also be considered. However, for our work it is sufficient to consider only the basic model changes.

In general, model matching is very challenging and beyond the scope of this thesis. An overview and references for further reading can be found in [33]. The approaches presented there can be divided into *identity-based matching* and *similarity-based matching*.

For an identity-based matching approach, the model elements must have automatically assigned, immutable IDs. The IDs never change, even if the model elements are renamed or moved. Thus, model elements are mapped to each other if they have the same ID.

For a similarity-based matching approach, an ID is not necessary. As the name suggests, matches between model elements are computed according to

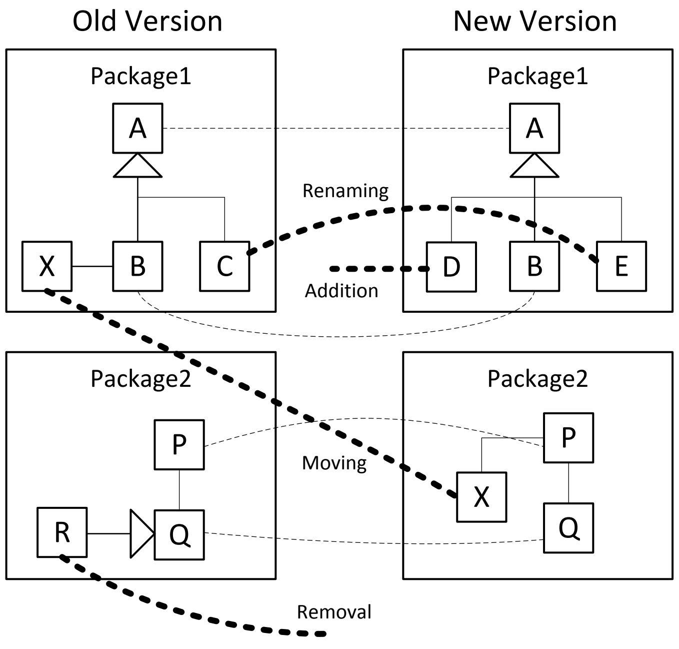


Figure 5.8: Matching model elements from different artefact versions.

their similarity. While in the identity-based matching approach two model elements are either matching or not (binary decision), in the similarity-based matching approach two model elements match to a certain degree. Similarity is computed from a number of properties. For example, similarity between two classes is determined by their position in the model (i.e., their package), their inheritance hierarchy, or their attributes and references (types and names). The individual properties can be weighted according to their importance. If the computed similarity value lies above a certain threshold, the model elements are considered matching and are mapped to each other. One should be aware that similarity-based matching is a heuristic, which can produce false positive and false negative results.

In this chapter, we usually assumed that the model is a class diagram. But as stated in the beginning, the formal document might also be computed from a completely different model. Different kinds of models have different properties, which in turn have a different influence on similarity. Therefore, each kind of model has a different set of optimal similarity weights. Finding optimal similarity weights is often a slow and time-consuming manual process, but it is necessary to achieve good results.

Alternatively, it would be possible to use a general similarity matching algorithm for all kinds of models. Similarity-based matching is based on graph matching, and many kinds of models can be viewed as graphs. While the use of a general similarity matching algorithm reduces the implementation effort, it also reduces the quality of the results. General similarity algorithms perform weaker than specialised algorithms.

Since it is not possible to guarantee fully correct results, the author should be given the possibility to intervene. Before the dynamic reference groups are physically modified (see below), the EDE should present the identified add, remove, rename and move operations to the author and allow him to override them if necessary.

Performing Dynamic Reference Group Updates When the necessary modifications are known, they must be executed. Here we present an algorithm to update the dynamic reference groups. The algorithm is rather simple and leaves room for improvement if performance is critical.

- Input
  - Elucidative Document
  - Modification instructions: descriptions of necessary modifications of the dynamic reference group. There are four kinds of modification instructions:

    Additions A list of addition instructions, each containing the dynamic reference group and the new child.

    Removals A list of removal instructions, each containing the dynamic reference group and the child to remove.

    Renamings A list of renaming instructions, each containing the child to be renamed, and the new name of the child.

    Moves A list of move instructions, each containing the source dynamic reference group, the target dynamic reference group, and the child to be moved. The source and target dynamic reference groups must be compatible, i.e., their group expander element must have the same computation instruction.
- Execution
  - For all move instructions: remove the child from the source dynamic reference group and add it to the target dynamic reference group (including all its content).

  - For all renaming instructions: change the configuration of the affected child, i.e., set the new name.
  - For all removal instructions: remove the child from the dynamic reference group (including all its content).
  - For all addition instructions: add the new child to the dynamic reference group.
  - For all dynamic reference groups where the computed order of the model element names is different from the actual order of the children: remove the wrongly ordered child and insert it in the right order (including all its content).

- Result

  - All dynamic reference groups contain all children and the order is the same as specified by their child enumerators.

In this basic algorithm, the manually written content of a child active reference group is lost if the child active reference group is deleted. In general, this is the desired behaviour. But it is possible that deleted classes of the model are restored and must therefore appear again in a dynamic reference group. Then, the author would also like to get the manually written text back, which has previously been deleted. An advanced algorithm could achieve this by maintaining a history of deleted children.

Summary

Dynamic reference groups are a powerful concept to automatically match the structure of a model in the elucidative document. They can also be automatically updated if structural changes have been applied to the model or if the configurations have been changed. If the model has changed, it is necessary to identify the types of changes that occurred. We proposed to use model matching to derive the changes and presented an algorithm to apply the model changes or the configuration changes to the dynamic reference groups. As a result of this section, Requirement 7 is fulfilled.

5.4 Guidance

When the author applies the update of an artefact, potentially many parts of the elucidative document are changed. For each update, there must be a notification message, so that the author can systematically proofread the changed document. The notifications are provided by the guidance system. Guidance is necessary in one of the following cases:

- A model element has been changed, such that a CDF must be recomputed. Guidance concerning the changes of CDFs has already been explained in Sect. 4.5 and will not be discussed here again.

- The structure of the model has been changed, such that dynamic reference groups have been modified. The rest of this section will discuss this case.

As shown in Sect. 5.3.2, active reference groups have an update state, which is evaluated by the guidance system, similar to active references. We extend the formal definition of the guidance system from Sect. 4.5.1 to include active reference groups. We call the result *Extended Guidance System.*

$EGS = (GS, refgroups_{stat}, refgroups_{dyn}, state_{stat}, state_{dyn})$, where

- GS is the guidance system for basic elucidative development.

- $refgroups_{stat}$ is the set of static reference groups.

- $refgroups_{dyn}$ is the set of dynamic reference groups.

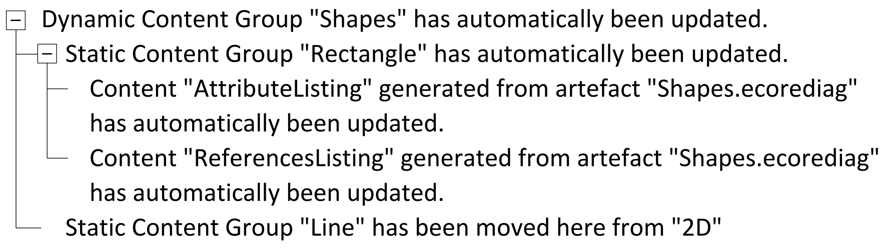
- $state_{stat} : refgroups_{stat} \rightarrow SC_{A,states}$ is a function that returns the state of a static reference group. $SC_{A,states}$ is the set of possible states defined by the state chart SC_A.

- $state_{dyn} : refgroups_{dyn} \rightarrow SC_{A,states}$ is a function that returns the state of a dynamic reference group. $SC_{A,states}$ is the set of possible states defined by the state chart SC_A.

In the context of this thesis, state chart SC_A comprises the states *normal*, *modified*, *incomputable* and the final state (see Sect. 4.5.3).

5.4.1 Hierarchical Guidance Messages

The update state of active reference groups is mostly determined by the update states of their children. Consequently, the EDE should group the guidance messages according to the hierarchy of the active reference groups. Figure 5.9 shows an example.



Dynamic Content Group "Shapes" has automatically been updated.
Static Content Group "Rectangle" has automatically been updated.
Content "AttributeListing" generated from artefact "Shapes.ecorediag" has automatically been updated.
Content "ReferencesListing" generated from artefact "Shapes.ecorediag" has automatically been updated.
Static Content Group "Line" has been moved here from "2D"

Figure 5.9: Hierarchical guidance messages.

5.4.2 Guidance for Static Reference Groups

When the CDFs or CDF groups inside a static reference group are in the *modified*, *incomputable* or the final state, the guidance system should also display a message that something has happened to the static reference group. This can be achieved by setting the state of the static reference group depending on the state of its children.

A static reference group is in the *normal* state if all its children are in the *normal* state. As long as the static reference group is in the *normal* state, no guidance messages are necessary.

A static reference group is in the *modified* state if one or more of its children are in the *modified* state and no children are in the *incomputable* or final state. When a static reference group is in the *modified* state, a corresponding message must be presented to the author. This message must also include all child active references and child active reference groups which are in the *modified* state. The message should display the children in a hierarchical fashion, according to the concept presented in Fig. 5.9.

A static reference group is in the *incomputable* state if one or more of its children are in the *incomputable* state and no children are in the final state. When a static reference group is in the *incomputable* state, a corresponding message must be presented to the author. This message must also include all child active references and child active reference groups which are in the *incomputable* and the *modified* state. The message should display the children in a hierarchical fashion.

A static reference group is in the final state when the artefact has been deleted. Like the final state of an active reference, the final state of a static reference group cannot be left. The static reference group with all its children must eventually be deleted. A corresponding message must be presented to the author.

5.4.3 Guidance for Dynamic Reference Groups

When child active reference groups or child active references inside a dynamic reference group have been added, removed, renamed or moved, the guidance system must display a corresponding message. This can be achieved by recording the executed addition, removal, renaming or move instructions in the so-called *change listing.* It contains an entry for all changed children, together with the type of change (added, removed, renamed, moved here, moved away). The state of the children and the entries in the change listing both influence the state of a dynamic reference group, which in turn determines the guidance messages to be displayed.

A dynamic reference group is in the *normal* state if all children are in the *normal* state and there is no entry in the change listing, i.e., there were no additions, removals, renamings or moves. As long as the dynamic reference group is in the *normal* state, no guidance messages are necessary.

A dynamic reference group is in the *modified* state if one of the following conditions are true:

- One or more of its children are in the *modified* state and no children are in the *incomputable* or final state.
- There is at least one entry in the change listing and no children are in the *incomputable* or final state.

When a dynamic reference group is in the *modified* state, a corresponding message must be presented to the author. For all children which are in the *modified* state, the guidance system must display their standard guidance messages as child of the dynamic reference group message, as shown in Fig. 5.9. Additionally, the guidance system must provide a message for each entry in the change listing and display it as child of the dynamic reference group message, too. If a child has been added or removed, the message should mention that. If a child has been renamed, the message should mention both its old and its new name. If a child has been moved away, the message should mention its name and its new parent. If a child has been moved to the dynamic reference group, the message should mention its name and its old parent.

A dynamic reference group is in the *incomputable* state if one or more of its children are in the *incomputable* state or if the child enumerator could not compute a result. No child must be in the final state. When a dynamic reference group is in the *incomputable* state, a corresponding message must be presented to the author. The message must also include all children which are in the *incomputable* and the *modified* state. The messages for added, removed, renamed and moved children should be the same as described above in the *modified* state.

A dynamic reference group is in the final state when the artefact has been deleted. Like the final state of an active reference, the final state of a dynamic reference group cannot be left. The dynamic reference group with all its children must eventually be deleted. A corresponding message must be presented to the author.

5.5 Conclusion

In this section, we discussed the application of model-driven ED to formal documents. We identified four requirements that extend the basic ED requirements and presented solutions that fulfil the requirements:

- Logical groups of CDFs and manually written text between them should be treated as a unit (Requirement 5). This has been achieved by the introduction of the reference group concept.
- Formal documents are often structured hierarchically (Requirement 6) because they reflect the structure of models. This has been addressed by the possibility to nest active reference groups.
- Changes in the model sometimes require structural changes in the document (Requirement 7). This has been addressed by the possibility to add, remove, rename or move active reference groups.
- Formal documents need additional kinds of guidance (Requirement 8). This has been addressed by the introduction of states and a discussion of guidance messages for active reference groups.

With the solutions presented in this section, model-driven ED is suited for the creation and maintenance of formal documents which describe models.

Chapter 6

Extensions of Elucidative Development

In this chapter, we present two extensions of Elucidative Development (ED) that provide additional value. The extensions address very different problems and are not related to each other.

First, we motivate the need to validate structured elucidative documents. We show how tree grammars can be used for the validation of Extensible Markup Language (XML)-based elucidative documents.

Then, we show how Round-Trip Engineering (RTE) can be used together with ED. We allow the author to make changes in the Computed Document Fragment (CDF) and use backpropagation-based RTE to propagate the changes to the corresponding view or the configuration.

6.1 Validating XML-based Elucidative Documents

Many document formats, such as LaTeX, or XML dialects like Extensible Hypertext Markup Language (XHTML), explicitly describe the structure of documents. These documents are therefore called *structured documents.* If the structure of a document conforms to certain syntactic rules, it is said to be *valid.* It is important to know whether a document is valid because validity is the precondition for correctly displaying the document and for correct automatic processing, such as outline generation or conversion to other document formats. For example, documentation written in the XML-based DocBook format can be transformed to XHTML or Portable Document Format (PDF). But the transformation works only reliably if the document structure conforms to the syntactic rules of the DocBook document format.

Otherwise, the output might be flawed or the transformation might even fail. A document could additionally contain non-structural errors, but these are beyond the scope of this thesis.

Advanced editing environments automatically take care of validity. What You See Is What You Get (WYSIWYG) editors, such as Microsoft Word or OpenOffice, only allow edit operations that result in valid documents, i.e., it is not possible to create invalid documents. XML editors and other text editors cannot prevent the insertion of invalid content. However, they can inform the author when invalid content has been entered.

Validity is also important for structured elucidative documents, but the validation is performed differently because active references require special treatment. In this section, we will show how XML-based elucidative documents can be checked for validity with the help of so-called *tree grammars*. We will explain the difference between XML and LaTeX documents and why the latter cannot be validated with tree grammars. Then we will present the formal concepts of structured documents and validity. Based on those, we will develop the formal concepts of structured elucidative documents and validity. Finally, we give an overview under which conditions existing validation tools can be used for the validation of structured elucidative documents.

6.1.1 Difference between XML and LaTeX documents

We use the term *structured documents* to denote files which contain hierarchically structured textual information, possibly enriched with media, such as images (either embedded or referenced). We focus on hierarchically structured documents because many modern document formats have an explicit hierarchical structure.

Document structure is often represented by XML trees. The trees are finite, ordered and unranked [17].

- A tree is *finite* if the number of its nodes is finite. This means that a document cannot be infinitely long or infinitely deeply nested.
- A tree is *ordered* if the order of tree nodes' children is important. This means, for example, that the order of paragraphs of the document must be retained.
- A tree is *unranked* if the number of child nodes of a node type is not fixed. For example, different section nodes of a document may have a different number of paragraph child-nodes.

Finite, ordered, unranked trees can be described and validated by means of tree grammars, as we will see below.

While LaTeX documents are also hierarchically structured, LaTeX is based on TeX, a macro processor. A LaTeX document is transformed into a printable format, such as PDF, by expanding macros and typesetting text. The macro language is a small programming language and it allows the definition of user-defined macros. Whether a LaTeX document is valid or not can only be discovered after the macros have been expanded. The macro syntax rules are hardcoded in the macro processor and thus not easily accessible. Consequently, LaTeX documents cannot be validated by means of tree grammars and will not be considered further in this section.

6.1.2 Structured Documents and Validity

In the following, we will show how structured documents are validated. Before we can do that, it is necessary to formally define structured documents.

Definition 26 (Structured Document). *A structured document* $D = (N, \Theta, type, children, root)$ *is a tree of document nodes (or nodes, for short), where:*

(i) $N = S \cup C$ *is the set of nodes of* D *such that*

- S *is the set of structural nodes (s-nodes), which define the tree (hierarchy) structure of the document.*
- C *is the set of content nodes (c-nodes), which carry actual document content, such as text or images.*

(ii) Θ *is a set of labels which represent the types of document nodes.*

(iii) $type : N \to \Theta$ *is a function that assigns a type label to a node.*

(iv) $children : S \to N^*$ *is a function that assigns a sequence of (child) nodes to an s-node.*

(v) $root \in S$ *is the root node of the document.*

This definition does not only apply to a structured document D as a whole, but also to any subtree of D. This means, that subtrees of D can also be treated as structured documents.

In terms of XML, we only consider elements and ignore the attributes because we do not need them for the discussion of structured elucidative documents later.

Example 1. *Figure 6.1 shows an example structured document. It is an excerpt of the document from Fig. 4.1. Listing 6.1 shows the XHTML structure of the document. Listing 6.2 shows the formal document definition. The definition uses the terminal "text" to denote arbitrary text content.*

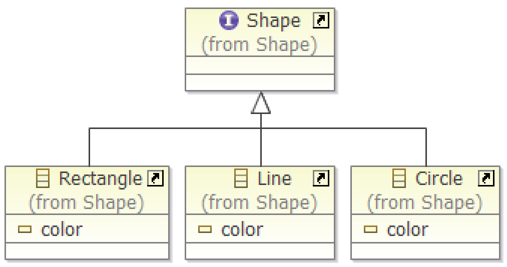


Figure II: Shapes

This diagram shows the `Shape` interface.

The `Circle` class has a `color` attribute (see Fig. II), which can be set.

Figure 6.1: Example document with cross reference.

```
<html>
    <body>
        <div id="fig_id">
            <img src="shapes.gif" />
            <p>Figure II: Shapes</p>
        </div>
        <p>This diagram shows the <span>Shape</span>
        interface.</p>
        <p>The <span>Circle</span> class has a
        <span>color</span> attribute (see
        <a href="#fig_id">Fig. II</a>), which can be set.</p>
    </body>
</html>
```

Listing 6.1: Example document as HTML.

$$\begin{aligned}
S =& \{\mathit{html}_1, \mathit{body}_1, \mathit{div}_2, \mathit{img}_5, p_{24}, p_{25}, p_{42}, \mathit{span}_{33}, \mathit{span}_{34}, \mathit{span}_{35}, a_{12}\}\\
C =& \{\texttt{"Figure II: Shapes"}, \texttt{"This diagram..."}, \texttt{"Shape"},\\
& \texttt{" interface."}, \texttt{"The "}, \texttt{"Circle"},\\
& \texttt{" class has a "}, \texttt{"color"}, \texttt{" attribute (see "},\\
& \texttt{"Fig. II"}, \texttt{"), which can be set."}\}\\
\Theta =& \{\mathit{html}, \mathit{body}, \mathit{div}, p, a, \mathit{img}, \mathit{span}, \mathit{text}\}\\
\mathit{type} =& \{(\mathit{html}_1, \mathit{html}), (\mathit{body}_1, \mathit{body}), (\mathit{div}_2, \mathit{div}), (p_{24}, p), (p_{25}, p), (p_{42}, p),\\
& (a_{12}, a), (\mathit{img}_5, \mathit{img}), (\mathit{span}_{33}, \mathit{span}), (\mathit{span}_{34}, \mathit{span}), (\mathit{span}_{35}, \mathit{span}),\\
& (\texttt{"Figure II: Shapes"}, \mathit{text}), (\texttt{"This diagram..."}, \mathit{text}),\\
& (\texttt{"Shape"}, \mathit{text}), (\texttt{" interface."}, \mathit{text}), (\texttt{"The "}, \mathit{text}),\\
& (\texttt{"Circle"}, \mathit{text}), (\texttt{" class has a "}, \mathit{text}), (\texttt{"color"}, \mathit{text}),\\
& (\texttt{" attribute (see "}, \mathit{text}), (\texttt{"Fig. II"}, \mathit{text}),\\
& (\texttt{"), which can be set."}, \mathit{text})\}\\
\mathit{children} =& \{(\mathit{html}_1, \mathit{body}_1), (\mathit{body}_1, \mathit{div}_2\ p_{25}\ p_{42}),\\
& (\mathit{div}_2, \mathit{img}_5\ p_{24}), (p_{24}, \texttt{"Figure II: Shapes"}),\\
& (p_{25}, \texttt{"This diagram..."}\ \mathit{span}_{33}\ \texttt{" interface."}), (\mathit{span}_{33}, \texttt{"Shape"}),\\
& (p_{42}, \texttt{"The "}\ \mathit{span}_{34}\ \texttt{" class has a "}\ \mathit{span}_{35}\ \texttt{" attribute (see "}\ a_{12}\\
& \texttt{"), which can be set."}),\\
& (\mathit{span}_{34}, \texttt{"Circle"}), (\mathit{span}_{35}, \texttt{"color"}), (a_{12}, \texttt{"Fig. II"})\}\\
\mathit{root} =& \mathit{html}_1
\end{aligned}$$

Listing 6.2: Formal definition of the example document.

The permitted structure of a document tree can be described by a *Regular Tree Grammar*. Regular tree grammars are tree grammars where the nonterminals have arity 0 and the production rules have a certain form [17], as we will see in Definition 27 below. Similar to string grammars, which describe permissible strings [35], tree grammars describe permissible trees. Regular tree grammars are the formal foundation of schema languages, such as Document Type Definition (DTD) and XML Schema.

Definition 27 (Regular Tree Grammar [35])**.** *A regular tree grammar is a 4-tuple $G = (N, T, S, P)$, where:*

(i) *N is a finite set of nonterminals*

(ii) *T is a finite set of terminals*

(iii) *S is a set of start symbols, where $S \subseteq N$*

(iv) *P is a finite set of production rules of the form $X \to a\ r$, where $X \in N$, $a \in T$, and r is a regular expression over N. X is called the left-hand side, $a\ r$ is called the right-hand side, and r is called the content model of the production rule.*

Example 2. *Listing 6.3 shows a regular tree grammar of a small subset of XHTML.*

$$
\begin{aligned}
N =& \{Html, Body, Div, Img, P, Span, A, PCDATA\}\\
T =& \{html, body, div, img, p, span, a, text\}\\
S =& \{Html\}\\
P =& \{Html \rightarrow html\ (Body), Body \rightarrow body\ ((Div \mid P)*)\\
& Div \rightarrow div\ ((Div \mid Img \mid P \mid Span \mid A \mid PCDATA)*),\\
& P \rightarrow p\ ((Img \mid Span \mid A \mid PCDATA)*),\\
& Span \rightarrow span\ ((Img \mid Span \mid A \mid PCDATA)*),\\
& A \rightarrow a\ ((Img \mid Span \mid PCDATA)*),\\
& Img \rightarrow img\ \epsilon, PCDATA \rightarrow text\ \epsilon\}
\end{aligned}
$$

Listing 6.3: Tree grammar of XHTML subset.

If a document can be mapped to a regular tree grammar, it is said to *conform* to the grammar, or to be *valid*. For XML documents, this mapping is called *schema validation*. Schema validation is performed by so-called *schema validators*.

Definition 28 (Document validity (based on [35])). *A document D is valid w.r.t. a regular tree grammar G if there exists a mapping M, such that:*

(i) *$M : D_N \rightarrow G_N$ is a function that assigns a nonterminal of G to a document node.*

(ii) *$M(n_{root})$ is a start symbol, and n_{root} is the root node of D.*

(iii) *for each node n and its children $n_0, n_1, \ldots, n_i$ there exists a production rule $X \rightarrow a\ r$ such that*

- *$M(n)$ is X*
- *the terminal of n is a*
- *$M(n_0)\ M(n_1)\ \ldots\ M(n_i)$ matches r, i.e., it can be specified/recognised by the content model (see Definition 27), which is a regular expression.*

Example 3. *Figure 6.2 shows the mapping of the structured document from Listing 6.1 to the XHTML grammar from Listing 6.3. This means, that the document is valid w.r.t. the grammar.*

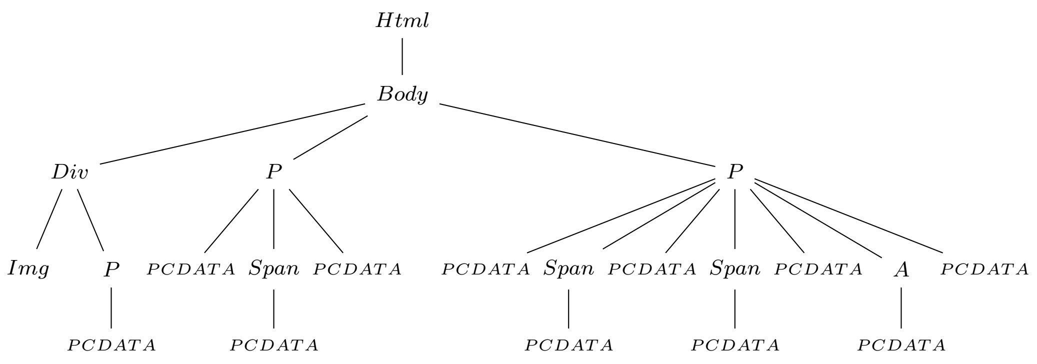


Figure 6.2: Mapping from document to XHTML grammar.

6.1.3 Structured Elucidative Documents and Validity

A structured elucidative document is a structured document with slots (see Sect. 4.3.5), which represent dynamic content. Slots can be inserted at any position in the document.

Slots are physically represented in the document either as active references or as CDFs (see Sect. 7.3). If slots are represented as CDFs, the structured elucidative document can be validated like a normal structured document. However, if slots are physically represented as active references, validation becomes more difficult because tree grammars defined for normal structured documents do not contain rules for active references.

There are two possibilities to enable validity checking of structured elucidative documents with active references:

1. Extend the tree grammar.

2. Use a modified validity checking method.

Extending the Tree Grammar

The first possibility, extending the tree grammar, is straightforward. Existing rules of the tree grammar (i.e., the XML schema language) must be modified and new rules added, so that active references can appear at all desired positions. This approach is possible if the tree grammar is available for modifications. However, sometimes the grammar is hardcoded into editors or subsequent processing tools (e.g., an Extensible Stylesheet Language Transformations (XSLT) processor with a DocBook stylesheet, which expects a standard conformant DocBook document as input).

Modifying the tree grammar, even if possible, might be too expensive, though. This is the case if the tree grammar is very big, or if grammar modifications require further changes, e.g., in related XSLT stylesheets.

Using a Modified Validity Checking Method

For the case where the extension of the tree grammar is not possible or feasible, we present a slightly modified validation approach. In the following, we will present a definition of structured elucidative documents, which is based on the definition of structured documents (Definition 26). Then, we will define validity of structured elucidative documents, similar to the definition of validity of normal structured documents. Finally, we will discuss the feasibility of this alternative validity checking approach.

The difference between structured documents and structured elucidative documents are *slot-nodes*.

Definition 29 (Slot-Node). *A slot-node is a structural node in a structured elucidative document, which represents both an active reference and the corresponding CDF. A slot-node is empty, i.e., it has neither child nodes nor content. A slot-node has a type, which equals the type of the CDF root node.*

Definition 30 (Structured Elucidative Document). *A structured elucidative document is defined as* $ED = (N, \Theta, type, children, root)$*, where*

(i) $N = S \cup C$ *is the set of nodes of* ED *such that*

- S *is the set of structural nodes (s-nodes), which define the tree (hierarchy) structure of the document.*
- C *is the set of content nodes (c-nodes), which carry actual document content, such as text or images.*
- $Slot \subset S$ *is the set of slot-nodes.*

(ii) Θ *is a set of labels which represent the types of document nodes.*

(iii) $type : N \to \Theta$ *is a function that assigns a type label to a node.*

(iv) $children : S \to N^*$ *is a function that assigns a sequence of (child) nodes to an s-node.*

(v) $root \in S$ *is the root node of the document.*

Example 4. *Let us now assume that Fig. 6.1 is a structured elucidative document. The class diagram figure and the identifiers are CDFs, like in the original Fig. 4.1 on page 21. Listing 6.4 shows an excerpt of its formal definition and Figure 6.3 shows the XHTML document tree. Slot-nodes are marked as circles. The corresponding CDFs are marked with blue background.*

$$
\begin{aligned}
S =& \{html_1, body_1, p_{25}, p_{42}, a_{12}, div_{a1}, span_{d1}, span_{d2}, span_{d3}\} \\
C =& \{\texttt{"This diagram..."}, \texttt{" interface."}, \\
& \texttt{"The "}, \texttt{" class has a "}, \\
& \texttt{" attribute (see "}, \texttt{"Fig. II"}, \\
& \texttt{"), which can be set."}\} \\
Slot =& \{div_{a1}, span_{d1}, span_{d2}, span_{d3}\} \\
\Theta =& \{html, body, div, p, a, span, text\} \\
type =& \{(html_1, html), (body_1, body), (div_{a1}, div), (p_{25}, p), (p_{42}, p), \\
& (span_{d1}, span), (span_{d2}, span), (span_{d3}, span), (a_{12}, a), \ldots\} \\
children =& \{\ldots\} \\
root =& html_1
\end{aligned}
$$

Listing 6.4: Excerpt of the formal definition of the elucidative document.

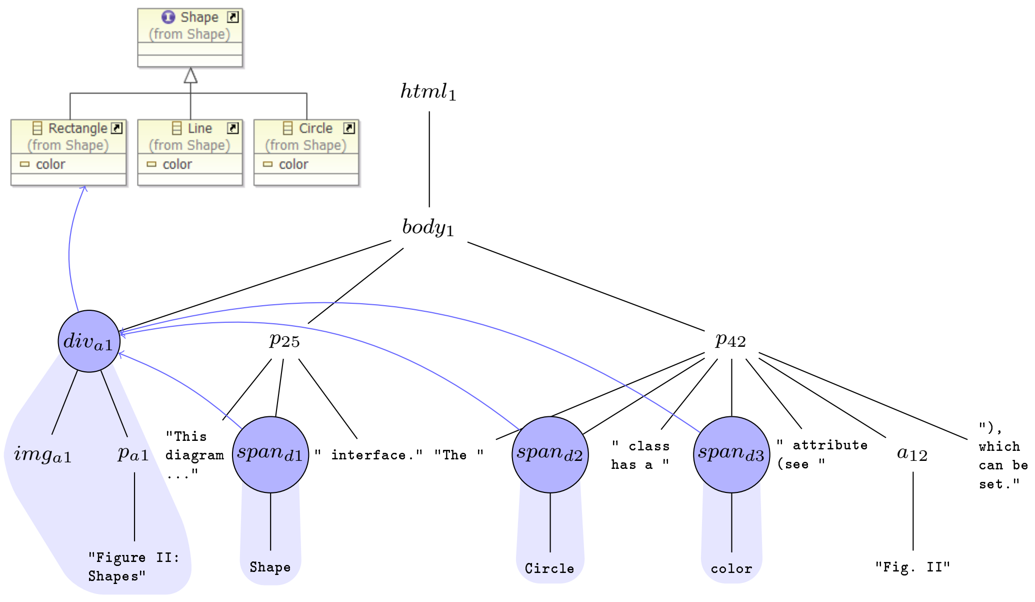


Figure 6.3: Elucidative document as tree.

Like a normal structured document, an elucidative structured document is valid if it can be mapped to a regular tree grammar. But as said before, we cannot use Definition 28 to decide if an elucidative structured document is valid, because the definition does not take the characteristics of slot-nodes into account. Therefore, we extend the definition. It covers two cases:

- The slot-nodes are represented as active references. The CDFs are unknown.
- The slot-nodes are represented as CDFs. The structured elucidative document looks like a normal structured document.

We define that a slot-node can be mapped to the grammar if it appears at a valid position in the tree, i.e., if the grammar permits a node of that type at that position, irrespective of whether the grammar requires child nodes. This covers the case that the slot-nodes are physically represented as active references and the CDFs are not available. Additionally, we define that if CDFs are available, they must be validated against the grammar. The grammar for the CDF validation is almost the same as for the validation of the whole document. The only difference is that the start symbol is the type of the CDF root node.

Definition 31 (Document Validity for Structured Elucidative Documents)**.** *An elucidative document ED is valid w.r.t. a regular tree grammar G if there exists a mapping M, such that:*

(i) $M : ED_N \rightarrow G_N$ *is a function that assigns a nonterminal of G to a document node.*

(ii) $M(n_{root})$ *is a start symbol, and* n_{root} *is the root node of ED.*

(iii) for each slot-node s, there exists a production rule $X \rightarrow a\ r$ *such that*

- $M(s)$ *is X*
- *the terminal of s is a*

(iv) for each slot-node s, for which a CDF can be computed, the CDF is valid w.r.t. G_s*.* G_s *is identical to G, except that* $M(s)$ *is a start symbol.*

(v) for each non-slot-node n and its children $n_0, n_1, \ldots, n_i$ *there exists a production rule* $X \rightarrow a\ r$ *such that*

- $M(n)$ *is X*
- *the terminal of n is a*
- $M(n_0)\ M(n_1)\ \ldots\ M(n_i)$ *matches r*

Example 5. *Figure 6.4 shows the mapping of the elucidative document from Fig. 6.3 to the XHTML grammar from Listing 6.3. This includes the mapping of the `div` and `span` CDFs. The elucidative document is valid w.r.t. the XHTML grammar.*

We have shown how a structured elucidative document can be theoretically checked for validity, but how does it work in practice? One possibility is the implementation of a schema validator that can validate documents

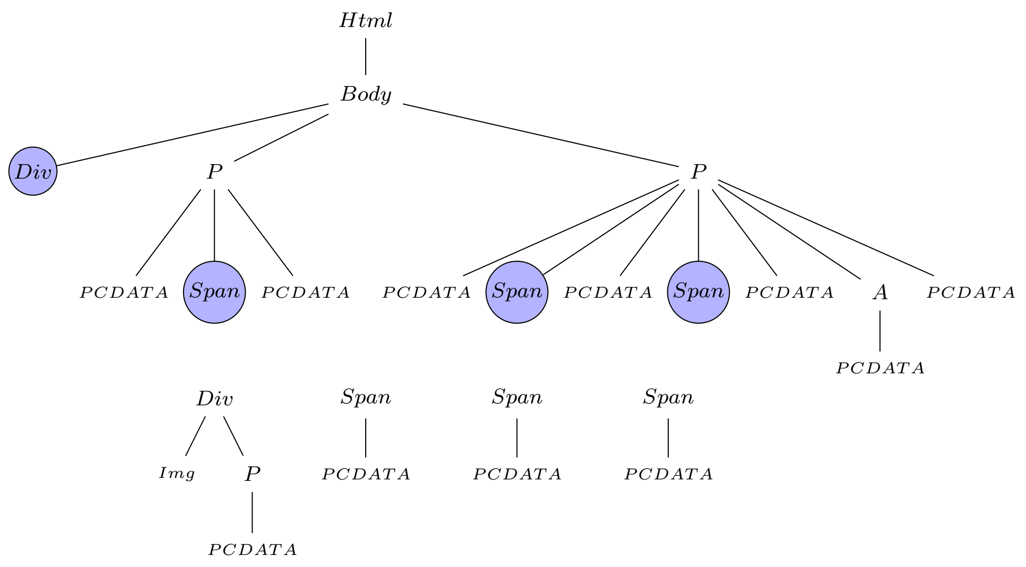


Figure 6.4: Mapping from elucidative document to XHTML grammar.

according to Definition 31. In general, the implementation of a schema validator is viable, but it costs some effort and it should be evaluated whether the benefit outweighs the effort. In [35], DTD validation and some variations are presented, which can serve as a starting point for a realisation.

There are also cases where structured elucidative documents can be validated with existing validators.

- If all slots are physically represented as CDFs, the structured elucidative document can be validated like a normal structured document.

- The concept of the *nillable* property in XML Schema[1] allows us to mimic the concept of typed empty slot-nodes using standard validators. This requires a few minor tweaks to the schema, but the changes do not affect the actual tree grammar defined by the schema. That is, the used schema language must be XML Schema and the schema must be available for minor modifications.

We will revisit both cases in Sect. 7.4, where we present possible technical realisations of validity checking.

[1] `http://www.w3.org/TR/xmlschema-1/#Element_Declaration_details`

6.2 Backpropagation-Based Round-Trip Engineering for Computed Text Document Fragments

Until now we have only considered unidirectional operations. The operations take an input, such as a software artefact, and compute a CDF. Changes in the input are noticed and trigger a recomputation of the CDF.

If the author wants to change a CDF, he can either ask a developer to change the corresponding view, or he can change the configuration. While this approach is usually feasible, there are cases, where it is cumbersome, for example, if the change comprises only the correction of a typing error. It would be useful to allow the author to make simple changes directly in the CDF and apply them to the artefact and/or the configuration automatically. This is an application of *Round-Trip Engineering (RTE)*.

In this section, we will present a possible application of RTE to elucidative development. First, we will give a short introduction to backpropagation-based RTE. Afterwards, we demonstrate the application of backpropagation-based RTE with a specific example.

6.2.1 Introduction to Backpropagation-Based Round-Trip Engineering

In Sect. 3.3, we introduced RTE as a means of synchronisation using inverse transformations. Backpropagation-based RTE [48] follows a different idea. Modifications of generated artefacts are not propagated back by means of inverse transformations. Instead, the *changes* performed on the generated artefacts are transformed and applied to the source artefacts.

In many cases, there are multiple possibilities to propagate target artefact changes to source artefacts. In order to find the best possibility, a procedure called Propagate Replay Evaluate Pick (PREP) has been developed. Backpropagation-based RTE with PREP is depicted in Fig. 6.5. Due to limited space, the figure shows only one source artefact and one target artefact. But the approach also works with multiple source and target artefacts.

First, all possible propagations are individually applied to the source artefacts (*Propagate*). Then, for each updated set of source artefacts the transformation is executed (*Replay*). The result of the replay transformation is called *echo artefact*. All propagations must be examined in order to find the best one (*Evaluate*). The propagations which result in echo artefacts that are different from the modified target artefacts are considered invalid. The remaining propagations are rated according to the "quality" of the modified source artefacts. Finally, the change propagation with the best evaluation result is chosen for the actual execution of the round trip (*Pick*).

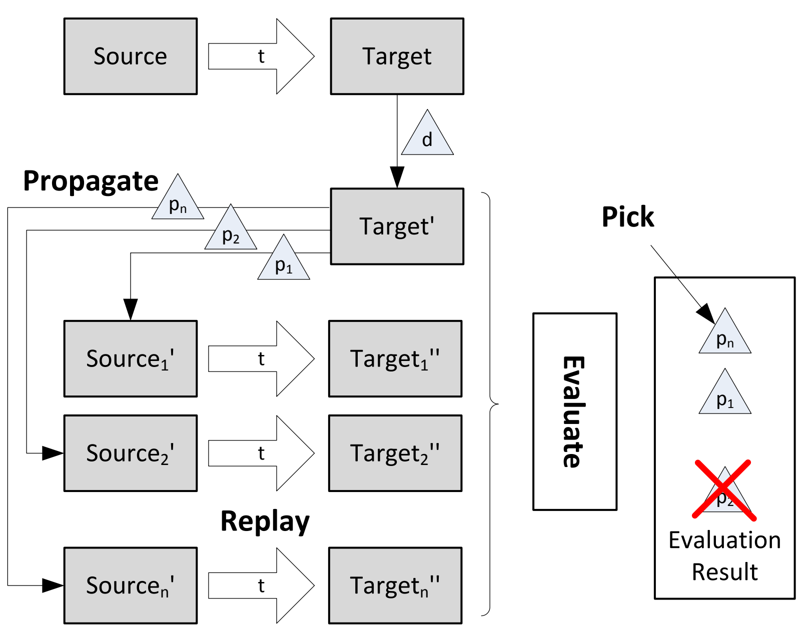


Figure 6.5: Backpropagation-based round-trip engineering.

RTE becomes easier if the artefacts to be synchronised are small. In many synchronisation scenarios, it is possible to concentrate on an excerpt of the artefacts and ignore the rest because only parts that share information, i.e., which are redundant, are relevant for RTE. In [48], the artefact parts which share information are called *skeletons*, the rest is called *clothing*. If the artefacts can be automatically divided into skeleton and clothing before the synchronisation and combined again after the synchronisation, as shown in Fig. 6.6, the synchronisation can be simplified.

6.2.2 Application to Elucidative Development – An Example

In the following, we will present an example of the applicability of backpropagation-based RTE to elucidative development. The CDF in this example is a code listing, i.e., the following explanations focus on the modification of source code. RTE for other kinds of CDFs require different algorithms for the propagation of changes.

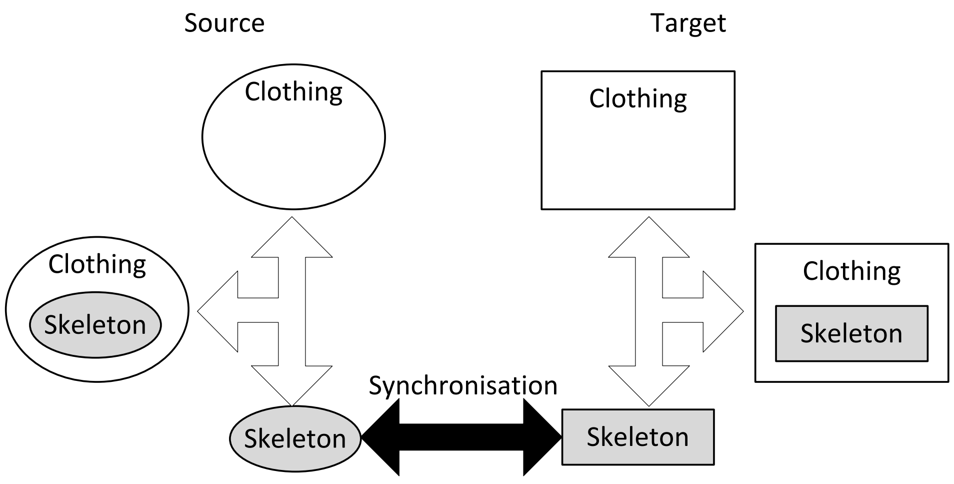


Figure 6.6: Skeletons and clothings of source and target artefacts.

The concepts of backpropagation-based RTE and elucidative development can be mapped to each other.

- The RTE transformation corresponds to the operation of an active reference. In this example, the transformation transforms source code into a code listing.
- The RTE source artefacts correspond to an artefact and a configuration. In this example, the artefact contains source code.
- The RTE target artefact corresponds to a CDF. In this example, it represents a code listing.

The way a code listing CDF can be synchronised with its artefact and the configuration depends on how it is computed, i.e., how the operation is implemented. Therefore, we will first sketch the basic functionality of the operation. An excerpt of the source code that should be transformed into a code listing is shown in Listing 6.5.

The operation starts with parsing the code file and creating a syntax tree. The configuration contains a query expression that denotes a node of the syntax tree. In our example, this is the root of the `createLine` method. This node and its descendants are used for the computation of the code listing CDF. In other words, the query expression is a filter that selects an excerpt of the code file. The selected `createLine` method is a skeleton in this synchronisation scenario, whereas the rest of the code file is clothing.

```
public class ShapeFactory {

    public Line createLine(Point p1, Point pEnd) {
        Logger.log("Creating line");
        Line line = new Line();
        //the order of the points does not matter
        line.setStartPoint(p1);
        line.setEndPoint(pEnd);
        return line;
    }

    public Circle createCircle(Point center, int radius)
    {
        ...
    }

    public Rectangle createRectangle(Point p1, Point p2,
            Point p3, Point p4)
    {
        ...
    }
}
```

Listing 6.5: Source code artefact.

Furthermore, the configuration can contain a mapping from query expressions to strings which describe replacements of syntax tree nodes. The operation removes for each query expression the corresponding node (if a node is found by the query) and replaces it by a token that contains the replacement text. The configuration is shown in Listing 6.6, and the resulting modified syntax tree is shown in Fig. 6.7.

```
[Filter Query]
//MethodDeclaration[@name='createLine']

[Replacement]
Body/ExprStmt[1] → "..."
Body/Comment[1] → ϵ
```

Listing 6.6: Configuration.

Finally, the syntax tree is serialised and leading spaces are removed. The resulting string is the CDF content, as shown in Listing 6.7. It contains content from the code file and replacement strings from the configuration. The meaning of the highlighted code parts will be explained soon.

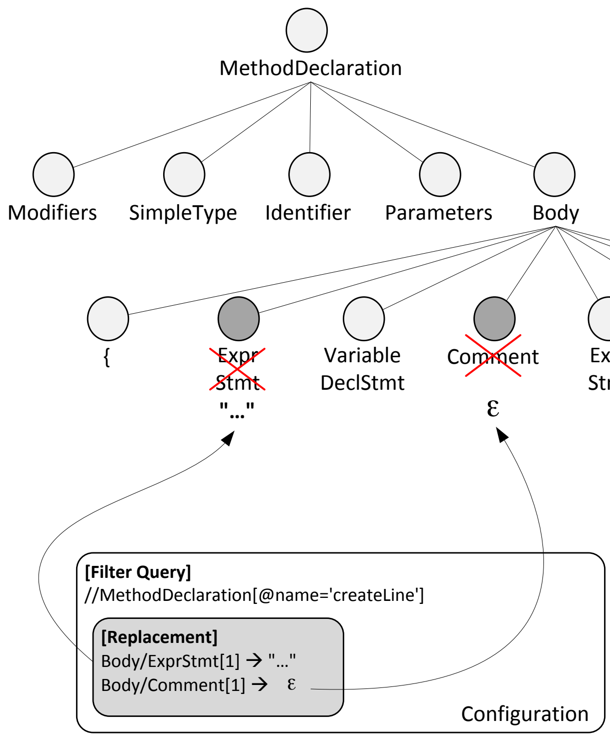


Figure 6.7: Configuration determines replacement of syntax tree nodes.

```
public Line createLine(Point p1, Point pEnd) {
    ...
    Line line = new Line();
    line.setStartPoint(p1);
    line.setEndPoint(pEnd);
    return line;
}
```

Listing 6.7: Resulting code listing.

In our example, the author notices that the parameters in the code listing are not uniformly named. They should either be called `p1` and `p2` or `pStart` and `pEnd`. Furthermore, the author thinks that the replacement of the logging statement with an ellipsis was not sufficient. He wants to replace the parameter `p1` with `pStart` and add the comment `//some logging` after the ellipsis. The corresponding locations in the code listing are highlighted in Listing 6.7.

The author decides to modify the code listing CDF directly. The changes that he performs are saved. We call them *string modification records* in the following. A string modification record contains the type and the location of the change, for example "replaced 2 characters at offset 29 with `pStart`".

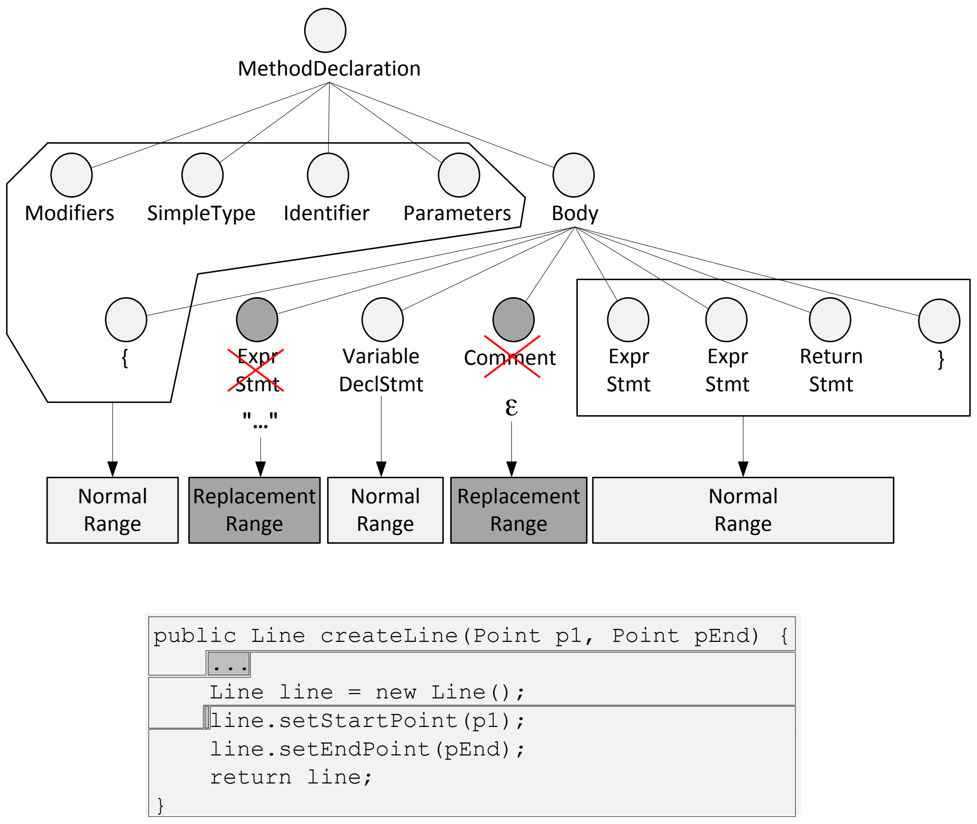
The string modifications must be propagated to the code file or to the configuration. This is similar to the approach presented in [14], where changes in a program with woven aspects must be propagated either to the program core or to the correct aspect. In order to decide whether the code file or the configuration is affected by a certain string modification, the code listing CDF is divided into a set of *ranges*. A range is a consecutive set of characters which either stem from the code file (*normal range*) or the configuration (*replacement range*). The ranges are computed from the syntax tree, as shown in Fig. 6.8. A normal range knows the start and end offset of its origin in the code file. A replacement range knows the query expression that was responsible for its creation.

A modification of the CDF can now be related to a certain range by comparing the offsets of the string modification record to the offsets of the ranges. Changes in an original range must be propagated to the code file, whereas changes in a replacement range must be propagated to the corresponding replacement rule in the configuration.

The actual change application is straightforward. If the configuration is affected, the replacement text of the corresponding query expression must be replaced by the modified content of the range. If the code file is affected, the offset of the modified original range must be translated to the corresponding position in the code file. Then that part of the code file can be replaced by the modified content of the range. A modification of the code file could cause syntactic or semantic errors in the code file, e.g., if a variable declaration is renamed, but the variable uses are not. An advanced implementation could identify this and similar cases and perform a refactoring. If a refactoring is not possible, or not implemented, the syntactic or semantic errors will be identified in the replay and evaluate phase of the PREP approach and the modification will be discarded.

In the example, both changes from `p1` to `pStart` take place inside a normal range. The changes are therefore applied to the code file. The new comment string is meant to be added after the ellipsis, which belongs to the first replacement range. However, the modification takes place at the end of the replacement range, which is the same as the beginning of the subsequent normal range. Therefore, it is not clear whether the change should be propagated to the configuration or to the code file.

The PREP approach described above can handle this ambiguity by performing both possible propagations. For the first propagation, the configura-



```
public Line createLine(Point p1, Point pEnd) {
        ...
        Line line = new Line();
        line.setStartPoint(p1);
        line.setEndPoint(pEnd);
        return line;
}
```

Figure 6.8: Computing ranges from the syntax tree.

tion is changed to produce the replacement `"... //some logging"` instead of `"..."`. The identifiers `p1` in the source code are changed to `pStart`, and the changed source code of the skeleton (the method) is added to the clothing (i.e., the rest of the code file) again. Then, the operation is replayed.

For the second propagation, the string `"//some logging"` is added to the corresponding position in the skeleton (the method). The identifiers `p1` in the source code are changed to `pStart`, like in the first propagation. Then, the changed skeleton is added to the clothing. Depending on how deeply the Elucidative Development Environment (EDE) and the round-trip system are integrated into the development tool landscape of the software project, the compilation of the code file could be triggered in order to verify that it is syntactically and static-semantically valid.

In our example, both propagations are valid. Therefore, the operation is replayed on both updated sets of source artefacts (source code artefact and configuration) to compute the echo artefacts. It turns out that in both cases the echo artefact is equal to the modified CDF. Therefore, the RTE system asks the author which change should be picked[2]. The author decides that the change should be applied to the configuration. Finally, the change is committed and the code listing CDF is recomputed.

6.3 Conclusion

In this chapter, we presented two extensions of ED. The first extension allows for the validation of structured elucidative documents. First, we motivated the need for the validation of structured documents in general. We identified XML documents as a family of documents, which can be easily checked against schemas. Since schemas are based on tree grammars, we presented the definition of tree grammars from the literature, together with a definition of validity. Afterwards, we extended the definition of validity, so that it applies to structured elucidative documents, which contain slots. For a practical realisation of validity checking for structured elucidative documents we refer the reader to Sect. 7.4.

The second extension is the use of RTE in ED. We explained backpropagation-based RTE and how it can be related to ED. We presented a possibility to modify textual CDFs and propagate the modifications to the corresponding artefact or the configuration. With the help of a running example, we showed how backpropagation-based RTE can be used for the modification of source code CDFs.

[2]Had both propagations been invalid, the RTE system would have informed the author that the desired change could not be applied.

Chapter 7

Tool Support for an Elucidative Development Environment

In Chap. 4, we presented Elucidative Development (ED) in a conceptual, implementation independent way. In this chapter, we cover issues related to the implementation of ED tool support, based on our experience with our own Elucidative Development Environment (EDE), called Development Environment For Tutorials (DEFT)[1] [4, 7, 59, 60]. First, we present implementation relevant issues concerning the management of active references. Then, we investigate under which circumstances existing editors can be used in an EDE and how Computed Document Fragments (CDFs) can actually be added to an elucidative document. Based on this, we introduce CDF caching, a way to realise transconsistency efficiently. Finally, we explain a possibility to validate elucidative documents with standard Extensible Markup Language (XML) Schema validation tools.

The problems and solutions presented in this chapter apply to a single user environment. We do not examine multi-user scenarios.

7.1 Managing Active References

Active references are a part of elucidative documents and are the source of CDFs. There are two ways to store active references physically: either directly inside the document file or externally, for example in a database. We will show why the database solution is preferable, what the challenges are, and how they can be addressed. The term *document file* is used to denote an individual file of an elucidative document, in contrast to the concept of an elucidative document as a logical unit as described in Chap. 4.

[1] `http://deftproject.org`

If active references are stored directly in the document file, they can be easily managed. If the author moves a CDF from one document file to another, it is only necessary to move the physical active reference correspondingly. A disadvantage is, that active references are difficult to search for. For example, finding all active references which depend on a certain artefact requires searching in all documents files. An even bigger disadvantage is that CDF caching is not possible (see Sect. 7.3).

Therefore, we recommend to store active references outside the document file, for example, in a database. Each active reference in the database has a unique ID. The document files contain placeholders with those IDs instead of the actual active references. Storing active references inside the database makes it easier to search and manipulate them, but it also requires some implementation effort. First of all, the active references in the database must additionally store their origin, i.e., the document file to which they belong. Secondly, copying, moving and deletion of active references becomes more complicated because the document files and the information in the database must be kept synchronised. Imagine, for example, that the author selects a CDF in the editor, cuts it out, and saves the document file. Even though the active reference is not used in any document file at the time of saving, it must not be deleted from the database, because the author might insert it into the same or a different document file later. In the following, we discuss one possibility to implement this behaviour.

In order to keep the document files and the active reference information in the database synchronised, it is necessary to update the active references in the database when a document file is saved. This requires comparing every active reference ID in the document file with the corresponding active reference in the database. Active references must not be deleted from the database when they are removed from a document file, because the removal might be undone by the author. Instead, an active reference needs a deletion status[2], which can have the values *deleted* and *normal*.

In the following, we will present four basic scenarios that can occur when the author removes and copies CDFs and then saves the document file.

- There is an active reference ID in the database without a corresponding ID in the document file. This can happen if a CDF has been deleted or moved to another document file. The active reference's deletion status must be set to *deleted*.

[2]The deletion status has nothing to do with the update state, which is used for guidance (see Sect. 4.5).

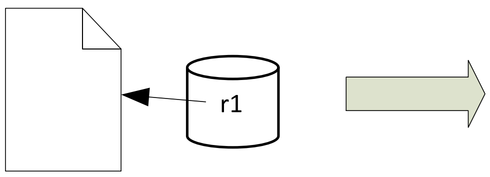


Figure 7.1: Scenario 1 – Active reference ID missing in document.

- There is an active reference ID in the document file whose correspon- ding active reference in the database has the deletion status *deleted*. This can happen if a CDF has been deleted, the document file has been saved, and then the CDF has been added again. The deletion status of the active reference in the database must be set to *normal*.

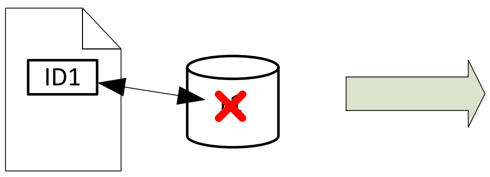


Figure 7.2: Scenario 2 – Inconsistent deletion status.

- There are two or more identical active reference IDs in the document file. This can happen if a CDF has been duplicated (copied and pasted) multiple times inside the same document file. The active reference IDs in the document file must be replaced by new IDs. The active reference information in the database must be duplicated, and each duplicate must get one of the new IDs.

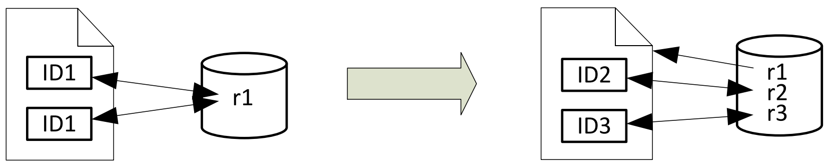


Figure 7.3: Scenario 3 – Identical active reference IDs in document.

- There is an active reference ID in the document file which exists in the database, but the active reference information says it belongs to a different document file. This can happen if a CDF has been copied or moved from one document file to another. The active reference ID in the document file must be replaced by a new ID. The active reference information from the database must be duplicated. The duplicate must get the new ID and the origin of the duplicate must be set to the new document file.

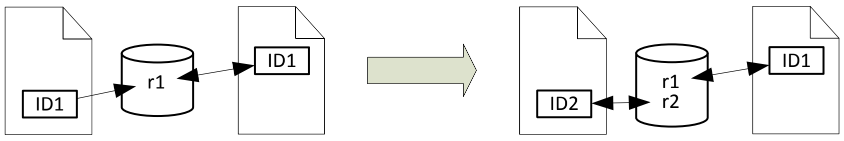


Figure 7.4: Scenario 4 – Inconsistent active reference location.

Special care must be taken when combinations of the above cases are encountered. For example, a document file might contain an ID multiple times. The corresponding active reference information in the database has the deletion status *deleted*, and the origin is a different document file. This can happen if a CDF has been removed from another document file and copied into to the new document file multiple times. Then each ID in the document file must be replaced by a new ID and the active reference information must be duplicated. The duplicates must then be assigned a new origin and their deletion status must be set to *normal*.

7.2 Inserting Computed Document Fragments

Until now, we have only examined the creation and updating of CDFs. The purpose of this section is to discuss the insertion and the removal of the created CDFs into and from the elucidative documents. First, we present two possible ways to insert and remove CDFs: modifying the document file directly or scripting the editor. Then, we discuss the problem that different document formats need different CDFs. Finally, we explain how to handle CDFs which contain media, such as images.

7.2.1 Document File Manipulation vs. Editor API

The editor of the EDE must be able to display CDFs. This requires that the presentation layer of the EDE removes the active references and inserts

CDFs. The removal of CDFs must also be handled by the presentation layer. There are two possibilities to insert a CDF, either by changing the document file directly or by controlling the editor (*scripting*). In both cases, the used editor must have certain capabilities, otherwise it is not suited for integration into the EDE. In the following, we will compare the two possibilities and the required Application Programming Interface (API), based on our experience with our EDE DEFT. The results of this comparison can be used to decide whether a certain editor is suited for integration into an EDE (see Requirement 4).

Common Editor API Requirements

The editor API must provide basic operations. These are needed regardless of whether the document file should be modified directly or via editor scripting. The editor API must allow:

- the retrieval of the current cursor position.
- the retrieval of the currently selected document content range.
- the scrolling of the document to a certain position.

These capabilities allow for the realisation of essential EDE features. Firstly, the cursor position is necessary to insert and edit CDFs.

- New CDFs/active references are usually added at the cursor position. If the document format of the elucidative document can be validated (see Sect. 6.1), it is even possible to check where a certain CDF may be inserted in the document. This enables features such as highlighting valid positions or prohibiting the insertion of the active reference at wrong positions.
- If the cursor is known to be inside a CDF and the editor API supports the creation of context menus, CDF specific context menus can be provided. CDF context menus can contain menu items for the editing of the configuration or the change of the CDF update state.
- If the cursor is inside a CDF and the editor can intercept editing commands (such keystrokes and mouse events), it is possible to notice when the author tries to change a CDF manually. An error message could be presented telling the author that editing CDFs directly in the document is not supported. Alternatively, the changes could be propagated to the artefact (see Sect. 6.2).

Secondly, document selections must be recognised, so the EDE can keep track of copied, moved or deleted CDFs. When the author copies and pastes a CDF, the EDE must internally copy and paste the corresponding active reference or active reference ID in the document file (see Sect. 7.1). When the author has selected only a part of a CDF and tries to copy and paste it, the EDE must either perform a full CDF copy or reject the copy operation altogether. If the copy operation is performed, the active reference or active reference ID must be copied to the new location in the document, so that the CDF can be created there. If the copy operation is rejected, nothing happens. Similar issues must be considered if the author tries to delete a partially selected CDF.

Thirdly, scrolling in the editor is necessary, so the author can navigate to specific CDFs. This is useful in conjunction with guidance, for example, when a CDF has changed and the author is required to proofread it.

Editor API Requirements for Document File Manipulation

If CDFs should be added, removed or modified by manipulating the document file directly, there are only few additional requirements for the editor. Most importantly, the editor must not lock the document file while it is displayed, so it can be modified. Afterwards, the editor must reload the modified document file.

The difficulty of manipulating the document file directly depends on the selected document format. Some document formats are easy to process, for example Extensible Hypertext Markup Language (XHTML). Other document formats are very complex and difficult to work with, for example Open Document Format (ODF). Fortunately, there are libraries for some document formats, which make the processing of complex documents, such as ODF, easier[3].

Editor API Requirements for CDF Processing by Scripting

If CDFs should be added, removed or modified by controlling the editor (scripting), the editor must have an API that allows the creation and removal of document content. We differentiate between text editors and What You See Is What You Get (WYSIWYG) editors.

The API for text editors can be rather simple. It must support:

- the insertion of text at a certain position.
- the removal of text from a certain range.

[3]`http://incubator.apache.org/odftoolkit/`

The required API for WYSIWYG editors is more comprehensive:

- It must allow the direct processing of formatted text, images, tables, and other document content that the CDF consists of.
- It should be possible to modify the undo stack. Unlike text editors, WYSIWYG editors usually do not allow the insertion of a CDF in one atomic step. If the CDF is added in multiple steps (e.g., first an image, then an image caption), the author could partially undo the CDF insertion and end up with an inconsistent document state. If the undo stack can be modified, the intermediate steps could be removed, so that an undo operation would undo the complete CDF insertion.

We have integrated three editors into DEFT, which fulfil the mandatory requirements. *TeXlipse*[4] is an Eclipse[5]-based text editor for LaTeX documents. It is open source, so we could modify it to fit into DEFT, which is also based on Eclipse. We were able to use the standard Eclipse editor APIs for getting and setting the cursor, scrolling, selecting text, getting selected text, and modifying text.

OpenOffice and its successor *LibreOffice*[6] are office suites, which include the WYSIWYG word processor *Writer*. The office tools have an API called Universal Network Objects (UNO), so they can be controlled from external programs, including Java programs. This allowed us the integration of Writer into DEFT. The basic API requirements are fulfilled, it is possible to get and set the cursor position, scroll to a certain document position, select document content and get selected document content. It is possible to automatically create, delete and modify CDFs which contain text, images, tables, or listings. There is no API for the modification of the undo stack, though. Thus, if a CDF is inserted in multiple steps, it is possible for the author to undo a part of the CDF insertion and provoke an inconsistent state.

Vex[7] is an XML editor for Eclipse. It hides the XML tags from the author and displays its content similar to WYSIWYG editors. At the same time, it enforces the compliance with an XML grammar specified by a Document Type Definition (DTD). Like TeXlipse, Vex is open source and allowed us after some modifications full control for adding, removing and modifying CDFs. We used Vex to write elucidative documents in the XHTML and DocBook format.

[4]`http://sourceforge.net/projects/texlipse/`
[5]`http://eclipse.org/`
[6]`http://www.libreoffice.org/`
[7]`http://www.eclipse.org/vex/`

7.2.2 Unifying CDF Insertion with Integrators

The structure of CDFs depends on the document format of the elucidative document. For example, a CDF for an XHTML document must consist of a tree of XHTML nodes. The same CDF for a LaTeX document must consist of text and LaTeX macros. If a CDF should be available for multiple document formats, the CDF operation must be implemented multiple times, once for each document format. Depending on the number of document formats that the EDE should support, this requires much effort.

Therefore, we recommend to separate the CDF computation into a document format-independent and a document format-specific part. As explained in Sect. 4.3.4, operations can consist of several stages. The first stages of the operation should compute an intermediate representation of the CDF, which contains no document format-specific properties. Based on our evaluations, we propose that the intermediate CDFs support at least the following document structures:

- Styled Text, i.e., text with style information, such as font family or size.
- Styled Code, which is a special type of styled text, used for code listings. It has a monospace font.
- (Bullet point) listings, which are usually a collection of (styled) text.
- Definition Listings, which are special listings with words or phrases, which are then described. The listings in the Cool Software specification (see Sect. 9.1) and in the Unified Modeling Language (UML) specification (see Sect. 9.2) are definition listings.
- Images.
- Tables.

The last stage of the operation should compute the final document format-specific CDF from the intermediate representation. Since the operation requires knowledge of the document format, it can additionally contain the code to perform the actual integration into the document. Therefore, we call the last stage of such an operation *integration*. An integration is performed by a so-called *integrator* [41].

Similar to operations, integrators can be configured. Configurations can be used to determine representation-specific attributes, such as the scaling factor of an image or the background colour of a table heading. Figure 7.5

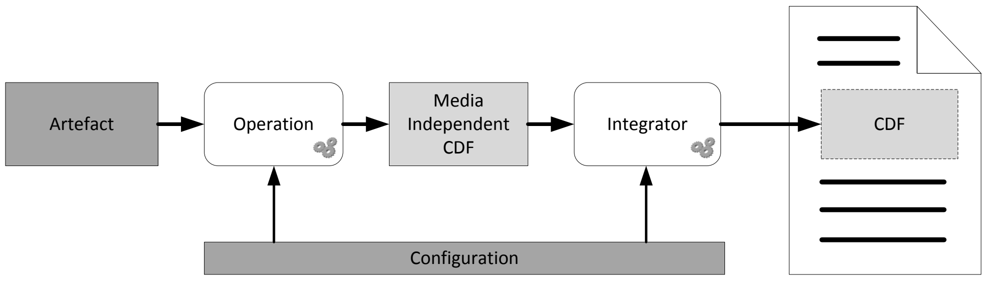


Figure 7.5: Using an integrator to insert a CDF into a document.

shows how an operation and an integrator transform an artefact and integrate the resulting CDF into a document.

If the integrator concept is used, there must be integrators for each supported document format. Furthermore, there must be an integrator responsible for the integration of error messages (see Sect. 4.4.2). All integrators must be able to handle arbitrary intermediate CDFs.

Depending on how CDFs are added to the elucidative documents (by file manipulation or via editor API, see Sect. 7.2.1), the integrators must work differently. Integrators that add CDFs by direct file manipulation are called *File Integrators.* Integrators that use the editor API are called *Editor Integrators.* The two types of integrators work differently, but they share a lot of similarities. Both must be able to insert and remove CDFs.

Add a new CDF When an active reference, or CDF, is newly added to the elucidative document, the editor will already be opened and the CDF is expected to appear at the cursor position. An editor integrator can query the current cursor position and add the CDF there. A file integrator must be told the current cursor position. It must then find the corresponding place in the document file and add the CDF there. The integrator must also add an ID to the CDF (see Sect. 7.3), so the CDF can be identified in the future.

Remove a CDF Since an integrator contains knowledge of the CDF it has previously added, it should also be responsible for CDF removal. Given an active reference ID, the integrator must find the range in the document where the CDF is located. For example, in an XHTML document it could search for an element with an `id` attribute that matches the active reference ID. In a LaTeX document it could search for comments with the artefact ID, which mark the beginning and the end of the

CDF. After the CDF has been found, the integrator removes it. Finally, it returns the original start position of the removed CDF. The position is needed if a new CDF version must be added (see below).

Integrate a new CDF version After a CDF has been removed during an update, it is necessary to integrate a new version of the CDF or, if it cannot be computed, an error message. This works like the integration of a new CDF as described above. The only difference is that the position of the CDF is now determined by the location of the removed CDF instead of the cursor position.

7.2.3 Handling Images

CDFs which contain media, such as images, must be specially handled. First, the image must be computed. If the CDF is processed by direct file manipulation, there are two alternatives. If the document format is plain text or markup, such as XML or LaTeX, the image must be stored outside the document and a link must be added to the document, pointing to the image. If the document format is more powerful, such as ODF, the image can be embedded in binary form inside the document.

If the CDF is processed via the editor API, there are also two alternatives. If the editor is a text editor, the approach is the same as if the CDF is added to a plain text file: the image is stored outside the document and a link to the image is added to the document. If the editor is a WYSIWYG editor, the image can be added to the document via the API. Storing, retrieving and deleting the image is all handled internally by the editor.

7.3 Caching the Computed Document Fragments

In the discussion of Requirement 1, we stated that a presentation layer is responsible for hiding active references and the presentation of CDFs. A naive implementation of an EDE might actually store an active reference or an active reference ID in the elucidative document and replace it with the CDF when the document is opened or published. While this would theoretically work, it is very inefficient, especially if there are many active references. Therefore, we propose an implementation that relies on caching. It has been successfully tested with DEFT.

CDF caching means, that CDFs are not only logically displayed by a presentation layer, but they are physically embedded in the elucidative document instead of active references. With CDF caching, it is not necessary to execute operations and insert CDFs every time an elucidative document is opened in the EDE. The corresponding active reference IDs are stored together with the CDFs in the document. The IDs must not be visible, i.e., they must be stored in a way that they do not appear in the final published output. The actual active references are stored in a database (see Sect. 7.1). Concrete implementations depend on the chosen document format. In Hypertext Markup Language (HTML) and XML documents, the ID can be stored in an element's `id` attribute. In ODF documents, the ID can be stored in named frames or bookmarks. In LaTeX documents, the ID can be stored in comments or dedicated macros[8].

When an artefact has been updated, or when referenced document content has been modified, the affected CDFs must be updated. For document files which are currently opened in the editor, the CDF updates must be performed instantly. For all other document files, the update can either be performed instantly, too, or when the corresponding document is opened the next time. In the following, we will compare both possibilities.

7.3.1 Instant Update

During an instant update, the EDE replaces all outdated CDFs in all document files. CDFs within document files which are not opened in the editor must be updated with file integrators. CDFs within opened document files can be either updated with file integrators or editor integrators.

7.3.2 Deferred Update

Alternatively, it is possible to defer the update of outdated CDFs until the corresponding document file is opened in the editor. In theory, every time a document is opened, each CDF must be checked if it is up to date. But with little effort a more efficient solution is possible.

Every artefact and every document file must have a revision number. An artefact revision number is increased when an artefact update has been accepted (see Sect. 4.5.1). A document file revision number is increased every time the document file is saved.

Additionally, each active reference in the database must have a revision number. The revision number of an active reference indicates the revision

[8]This exposes technical details of the ED implementation to the author, which slightly violates Requirement 1. However, the intention of that requirement was to show the author the CDFs during editing, which is still the case. The active reference IDs inside the CDFs are usually unobtrusive and we consider them an acceptable tradeoff.

of the artefact or document from which the currently cached CDF has been computed. If the active reference revision number equals the artefact or document revision, the CDF is up to date. If the active reference revision number is smaller than the artefact or document revision, the CDF might have to be updated. In that case, it is necessary to compute the CDF from the current artefact or document file and compare it to the cached CDF in the document. If they differ, the CDF in the document must be replaced by the new CDF. After the check and the possible CDF update, the active reference revision number in the database must be set to the current artefact or document revision. This indicates, that the CDF needs not be checked anymore when the document is opened the next time. If no CDF can be computed, the CDF in the document must be replaced with an error message (see Sect. 4.4.2).

7.3.3 Discussion

Instant and deferred update both have their advantages and disadvantages. In the following, we present a comparison.

Efficiency During an instant update, all affected CDFs are checked and, if necessary, updated. Depending on the number of affected document files and CDFs, this can take some time. The document format also influences the update time. More complex document formats need more time for an update.

The deferred update only updates the document files that are opened in the editor. If the editor API is used to perform the update, the speed of the editor determines the update time. In general, complex editors with styling capabilities, such as OpenOffice, need more time to insert CDFs than text editors. Some editors require that the cursor position be set to the location where content is updated. This results in scrolling to all affected CDFs, which is another performance penalty.

Consistency After an instant update, all document files are consistent again. They can be further processed, such as being edited in the editor or being published, without special consideration.

The deferred update might leave document files in an inconsistent state. Therefore, care must be taken that the document is updated (i.e., opened in the editor) before it is published.

In general, instant update is more robust and faster than deferred update. However, the final choice depends on the chosen document format and possible restrictions of the editor.

7.4 Elucidative Document Validation with Schemas

In Sect. 6.1, we showed how elucidative (XML) documents with active references can be validated against a tree grammar. The basic idea was to view active references as tree nodes, whose type corresponds to the root node type of their CDF. If the EDE displays the CDFs instead of the active references, as described in Requirement 1, and CDF caching is used, then there is actually no need to resort to the methods from Sect. 6.1. Elucidative documents with CDFs are expected to conform to the "original" tree grammar and can be validated with standard tools. However, if the author chooses to display the active reference placeholders (see Sect. 7.1) instead of the CDFs, this does not work. In that case, it is necessary to actually validate elucidative documents with active references.

In the following, we present three ways to implement validity checking for elucidative XML documents, according to Sect. 6.1. At the same time, we attempt to not reinvent the wheel and try to use standard XML tools and *schemas* as much as possible (see Requirement 4).

7.4.1 Restricting the Possible Active Reference Types

In Sect. 6.1.3, we stated that slot-nodes (which correspond to active references) must have a type. In Sect. 7.1, we stated that the actual active reference information should be stored in a database and the elucidative document should only contain a placeholder with an ID. Thus, the `div` element in Listing 7.1 can be considered an active reference (placeholder). Furthermore, the listing as a whole constitutes a valid document (according to the simplified XHTML tree grammar from Example 2 at page 82). That is, the validity of the elucidative document with the active reference can be checked with standard XML tools against a normal schema.

```
<html>
    <body>
        <div id="ref-001" />
    </body>
</html>
```

Listing 7.1: Div element in an XHTML document.

However, there is a limitation. As shown in the listing, the active reference placeholder in the document is represented by an empty XML element. Thus, only types which can be empty, according to the schema, can be used as active reference placeholders. Additionally, only types with attributes that can store the active reference ID are feasible. Types which require children, such as `table` in XHTML, or which do not allow attributes, would cause a validation error if they are used as shown in the listing.

This can be avoided by restricting the possible active reference-, and correspondingly, CDF-types to the ones that can be empty and have an attribute for the active reference ID. In XHTML, for example, this would include the `div` and `span` element types, which are very flexible containers for arbitrary content. And as all XHTML types, they can have an `id` attribute.

This solution has another great advantage: the handling of error message CDFs is greatly simplified. In Sect. 4.4.2 we said that an error message should be displayed in case a CDF cannot be computed. However, the structure of the document file should still be valid. This is clearly possible if the type of the CDF root element (i.e., the active reference type) is a flexible container like `div`. It would usually contain the actual CDF content, but in case of an error could contain an error message.

This is the preferred solution in many cases. It simplifies the validation by encoding active references as valid elements of the original tree grammar. However, for cases where this is not possible, we present two alternatives.

7.4.2 Using Subtrees as Active References

There may be cases where it is not desirable to restrict the possible active reference types. If the active reference type cannot be empty according to the schema, it must be represented by a minimal valid tree. Listing 7.2 shows an example of a table active reference.

```
<html>
    <body>
        <table id="ref-001"><tr><td/></tr></table>
    </body>
</html>
```

Listing 7.2: Table element in an XHTML document.

While this solution provides more flexibility concerning the possible CDF types, it also has drawbacks. First, it is necessary to define a minimal valid tree for each allowed type. Second, element types which do not allow an ID attribute can still not be used as active reference types. Third, handling the display of error messages would be more complex. It would be necessary to define error message operations for all possible types. For example, a CDF of type `table` would have to produce a complete table with rows (`tr`), data (`td`), and finally the message. A CDF of type `ul` (unordered list) would have to produce a list with a list item (`li`) and then the message.

7.4.3 Nillable Active References

The following elucidative document validation approach is recommended for the special case that the possible active reference types should not be restricted, but at the same time the active references should be represented by an empty XML element. This approach works only under certain conditions:

- XML Schema must be used for validation.
- The XML Schema must be available for modification[9].

XML Schema has a special property called `nillable`. The XML Schema specification[10] says: "If nillable is true, then an element may also be valid if it carries the namespace qualified attribute with local name nil from namespace http://www.w3.org/2001/XMLSchema-instance and value true [...] even if it has no text or element content despite a content type which would otherwise require content." In other words, `nillable` is a switch that makes it possible to set a `nil` attribute to an element, which in turn allows the element to be empty (without child elements or text) even though that would otherwise violate the schema. Listing 7.3 shows a simplified XML Schema declaration of an XHTML table with `nillable` set to true.

```
<xs:element name="table" nillable="true">
    <xs:complexType>
        <xs:choice>
            <xs:element maxOccurs="unbounded" ref="tbody"/>
            <xs:element maxOccurs="unbounded" ref="tr"/>
        </xs:choice>
        <xs:attribute name="id" type="xs:ID"/>
    </xs:complexType>
</xs:element>
```

Listing 7.3: Nillable table element declaration.

This declaration allows the use of the `xsi:nil` attribute on `table` elements. If `xsi:nil` is set to `true`, the table element will be considered valid

[9]We noted in Sect. 6.1.3 that modifications of the tree grammar are usually not feasible. However, the changes we propose here are minor and do not affect the actual grammar.

[10]`http://www.w3.org/TR/xmlschema-1/#Element_Declaration_details`

if it is empty. If `xsi:nil` is missing or set to `false`, the element will be considered invalid if it is empty because it requires a list of `tbody` or `tr` child elements. Listing 7.4 shows an XHTML document with a nil table as active reference.

```
<html xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <body>
        <table id="ref-001" xsi:nil="true"/>
    </body>
</html>
```

Listing 7.4: Nil table in an XHTML document.

Unfortunately, this approach has the same problems as the one from Sect. 7.4.2 w.r.t. error message handling. Additionally, `nillable` is usually not set in the XML Schemas, because it is a concept to represent `null` values for data-centric XML files, as opposed to document-centric XML files. Consequently, the `xsi:nil` attribute cannot be set in the document files.

The only solution is to modify the XML Schema. If the schema is hard-coded into the editor, we cannot use the nillable-approach for validity checking. However, if the schema can be accessed, the modification should be considered. Adding the nillable attribute to the element declarations causes only little effort and it does not affect the underlying tree grammar.

The attributes of an element must always conform to the schema, even if `xsi:nil` is set to `true`. That is, as in the solutions presented before, element types which do not permit an ID attribute cannot be used as active reference types. However, it might be worth considering adding an appropriate attribute. As the schema (probably) has to be modified anyway to add the nillable attributes, it is not much extra effort to add missing ID attribute declarations. Again, the underlying tree grammar is not affected.

7.5 Conclusion

In this chapter, we presented recommendations and issues regarding the implementation of the theoretic concepts from the previous chapters. First, we proposed to store active references as records in a database and let elucidative documents only contain a pointer to these records. We also discussed the necessary effort to keep elucidative documents and the active references in the database synchronised.

Then, we examined how CDFs can be inserted into an elucidative document. We identified document file manipulation and editor scripting as possibilities and explained the APIs that are necessary for an editor to be used as EDE editor. The motivation to reuse existing editors in the EDE came from Requirement 4. We also showed, how operation implementations can be reused if the EDE should support multiple document formats. An operation should be divided into a document format-independent and a document format-specific part, the so-called integrator. Additionally, we discussed how images can be handled by integrators.

Afterwards, we introduced CDF caching to keep expensive CDF computations to a minimum. We presented two caching strategies, instant update and deferred update, and discussed their advantages and disadvantages.

Finally, we showed that elucidative XML documents can be validated with standard tools if active references are specified as XML elements. We presented three similar approaches with different advantages and disadvantages. The first approach allowed us a very simple validation and good error message handling. This came at the cost of having only a restricted number of XML element types available as active reference types. In the second approach, validation was similarly simple and there were fewer restrictions concerning the active reference types. However, it requires more development effort and the error message handling is more cumbersome. The third approach had almost no restrictions w.r.t. the possible active reference types. However, it only works if the document is validated against XML Schema, and it even requires modifications of the Schema. Moreover, it suffers from a cumbersome error message handling. Like the editor reuse, the use of standard validation tools has been motivated by Requirement 4.

Chapter 8

Related Work

In this chapter, we present and compare work that is related to Elucidative Development (ED). We start with documentation approaches which have highly influenced ED, namely Literate Programming, Literate Modelling and Elucidative Programming. Then, we introduce consistency approaches used for ED, namely transclusion and transconsistency. Afterwards, we take a look at compound document technologies, such as OpenDoc and Object Linking and Embedding (OLE).

8.1 Related Documentation Approaches

There are several documentation approaches in the literature which had a great influence on ED. They shared the idea to reduce redundancy by reusing software views for documents. The general concept is shown in Fig 8.1. For each concrete documentation approach that we introduce in the following, we will present a similar figure. Figure 4.3, the overview figure of the Elucidative Development chapter, is also based on Fig. 8.1.

In the following, we will review the related documentation approaches, highlight their most important properties and compare them to ED. An overview of the comparison is shown in Table 8.1.

8.1.1 Literate Programming

Knuth presents in his paper about Literate Programming (LP) [31] an approach to improve program documentation significantly by considering programs work of literature. The idea is to explain to humans what a program should do instead of instructing the computer directly. A so-called literate program consists of a series of sections (sometimes called *chunks* [26, 27, 43]).

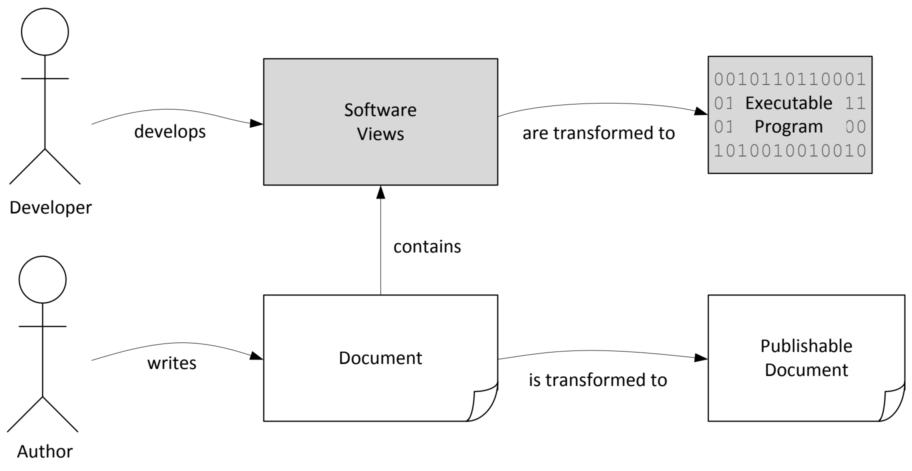


Figure 8.1: Related documentation approaches: general overview.

	documentation format	**view support**	**view location**	**operations**	**software dev. phases**
Literate Programming (LP)	typesetting language (e.g. TEX)	homogenous (source code)	integrated	weave, tangle	implementation
Literate Modelling (LM)	WYSIWYG format	homogenous (UML diagrams)	separate	embed	analysis, design
Elucidative Programming (EP)	typesetting language	homogenous (source code)	separate	embed	implementation
Elucidative Development (ED)	WYSIWYG format or typesetting language	heterogeneous (models, source code, . . .)	separate	transconsistency, weak transconsistency	analysis, design, implementation

Table 8.1: Comparison of advanced documentation approaches (from [59]).

Each section can have a *commentary*, i.e., documentation text, and *program text*, i.e., a small fragment of source code. The text is written in a document markup language, such as TEX. Knuth introduced the WEB system, which consists of two tools, `Weave` and `Tangle`. They are used to postprocess and transform the literate program. `Weave` produces a document for human readers, for example a Portable Document Format (PDF) file. `Tangle` extracts the code snippets from the literate program and produces a compilable program. Figure 8.2 shows a graphical overview.

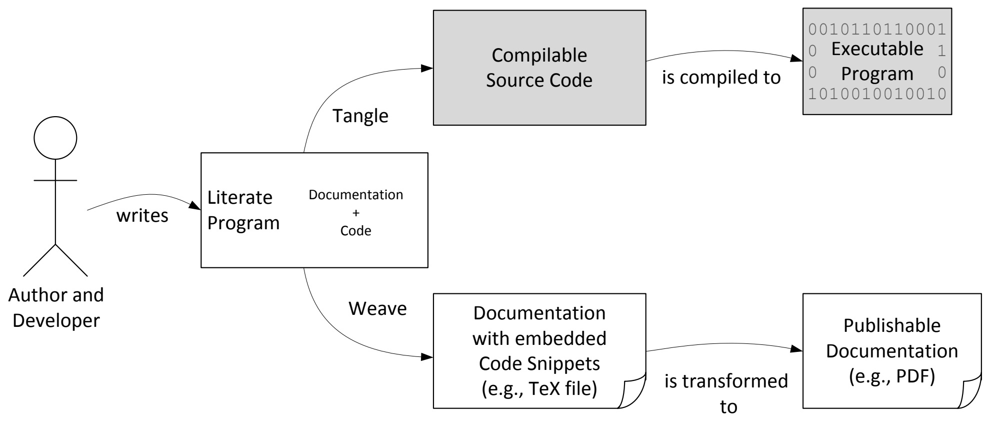


Figure 8.2: Literate Programming overview.

LP enables the programmer to write programs in an order that is best for human understanding, which can be different from the order of tangled code. Knuth found that the time to write and debug a literate program is not greater than the time to write and debug an Algol or Pascal program. Extra time spent for documentation was saved during debugging due to the higher quality of the program.

Literature Overview

In the following, we will present a literature overview about LP.

Literate Programming in an Industrial Setting [51] This paper describes the experience with LP for large programs. Development started using the *noweb* tool. Several improvements and extensions were made as needed, which finally resulted in the *dotNoweb* tool. The literate programs written in *dotNoweb* often used several programming languages and even the graph description language *dot*. *dotNoweb* automatically produces diagrams from dot graph descriptions.

Literate programs can have multiple authors, which makes collaboration tools necessary. In order to get an overview of changes, it is possible to render revision marks into the documentation.

Many programmers do not want to write text in a markup language, such as LaTeX. The authors recognise the importance of easy-to-use tools for widespread success, which includes What You See Is What You Get (WYSIWYG) word processors, such as Microsoft Word. While *Noweb* does not provide a real integration, there are scripts, which convert *Noweb* source files into Rich Text Format (RTF) and vice versa. This allows authors to write literate programs with Microsoft Word. For developers who prefer a more technical environment there is also support for Emacs.

The reported results contained both positive and negative points. It was easy to use multiple programming languages in parallel in the literate program. Small code changes could easily be made and their reason be explained. When a bug was found, it was possible to include a warning and a reference to a workaround. This easiness to update code and documentation in parallel made it unlikely for the documentation to become inconsistent. Similar to Knuth [31], the authors felt that the initial effort to write a literate program is compensated by reduced debugging time. One identified shortcoming was that it is not possible to rearrange a literal program by *weave*, which prohibits the generation of multiple different documents from one literate program. The best way to organise the text for a tutorial is not necessarily good for a reference manual. Another finding was that the amount and quality of low level documentation was very good because source code was being described. More abstract design documentation without code was comparatively sparse in contrast. Much effort was put into the development of the *dotNoweb* tool and the related editor support. The lack of standard tools is seen as one major obstacle for the spreading of LP.

Theme-Based Literate Programming [26, 27] The *WEB* system and subsequent systems gained interest, but they were not spared from criticism. A major drawback of LP was that one literate program could only be woven to one documentation. As already mentioned above, the order of the documentation was determined by the order of the chunks (models) of the literate program. However, depending on their goals, people need documentation with different focuses and levels of granularity.

Theme-based literate programming is proposed to solve this problem. It supports the definition of new kinds of chunks, e.g., for diagrams or unit tests. Furthermore, it allows the definition of theme models, which define navigation paths through the chunks. Thus, theme models allow the creation of multiple documentations from a set of chunks, providing “multiple views of a given system”.

There are more problems with LP, which have been identified. These problems are not conceptual, but merely show that tool support is insufficient. Among the mentioned problems were the following:

Little Extensibility Knuth's WEB system was written for the Pascal programming language. Before it could be used for other languages, it was necessary to define language-specific details. Ramsey presented the simpler, language agnostic tool *noweb* [42]. While it is easier to use new languages, some of WEB's power is lost. Language-dependent features, such as prettyprinting or the generation of identifier indexes, are not supported.

Debugging If the literate program contains compile-time errors, the compiler reports error locations for the tangled code (i.e., the code extracted and assembled from the literate program), but not for the actual literate program. This requires the developer to examine the tangled code and then find the corresponding location in the literate program. There are some implementations of literate programming environments, such as *noweb*, which can add location information, such as the `#line` directive in C, to the tangled program. This information can be used by the compiler to produce error messages which match the literate program. However, not all compilers can deal with this information.

Three-Syntax-Problem The literate programmer has to deal with three languages simultaneously: the programming language, the documentation language, and the interconnection language. This causes a lot of effort and might distract from the actual problem solving [39].

Object-Oriented Limitations Most literate programming tools were developed for structured programming languages. Object-oriented concepts, such as overloading and overriding, are not supported. Methods are identified by name only, there are no means to state that one method belongs to a specific class and overrides another method.

A Hypertext for Literate Programming [12] After the advent of hypertext, there were endeavours to use it as a publishing medium for LP. Chunks of a literate program have relationships to other chunks, so a direct navigation in a browser is a natural improvement. Object-orientation introduced another aspect that called for an improvement over the fixed sequential documentation of LP, which had originally been developed to document structured programs. In object-oriented programming, it is important to understand how different objects work together [40]. It is therefore often necessary to assign multiple code fragments to a single piece of text. Østerbye describes a hyperlink-based LP tool for Smalltalk. Code and documentation are written independently and connected by hyperlinks. Hyperlinks can also be used to connect documentation of different abstractions. The tool also supports management of the hyperlinks because a purely manual management would be impractical.

Literate Smalltalk Programming [40] It has been pointed out that LP can be inappropriate in some stages of the software development process, such as in initial experiments. If the code changes drastically and frequently, it is cumbersome to always rewrite the whole documentation [44]. A proposed solution is to document the changes and the rationale behind them. But while this approach helps to understand evolution of the program, it is hard to get a concise view of the current state.

Comparison with Elucidative Development

ED shares the main idea of LP that documentation and code should be close to each other. However, ED only focuses on the resulting documentation, not the way of programming. In contrast to LP, ED does not mix documentation and code in one document. Source code does not contain any documentation besides Application Programming Interface (API) documentation and comments. Documentation and code reside in different documents. They are logically connected by active references, which are embedded in the documentation and reference the code. This means, that code and documentation need not necessarily be written in parallel. Therefore, the quality of the resulting documentation is likely not as good as the quality of a real literate program. But on the other hand, developers need not change their way of programming. The LP way of programming, which is substantially different from "normal" programming, has always been an obstacle for wide adoption.

Consequently, a postprocessing tool such as *tangle* is not necessary for ED, because the software views need not be extracted from the documentation. An advanced version of *weave*, integrated into the Elucidative Development Environment (EDE), is necessary, though. It must make sure that Computed Document Fragments (CDFs) can be displayed instead of the active references when the documentation is edited (see Requirement 1). Furthermore, it is responsible for the transformation of the active references into CDFs when the documentation is published. The published result looks like a weaved literate program.

8.1.2 Literate Modelling

Inspired by LP, the goal of Literate Modelling (LM) is to explain the results of object-oriented analysis and design to different audiences. It has been observed that object-oriented analysis and design diagrams, such as UML diagrams, are often not suitable for both business experts and developers [2]. For example, use case diagrams might be not understandable by non-domain experts, because they seem arbitrary without business context. Without deep knowledge about legal rules, collaborations with partners, legacy systems and similar factors, it is hard to grasp the importance of a certain requirement. On the other hand, technical models, such as UML class diagrams or sequence diagrams, are suited for developers, but not for stakeholders.

Many stakeholders do not understand UML diagrams. One problem is that they do not know the syntax of those diagrams. Additionally, many diagram types, such as class diagrams, have no narrative structure, which makes them hard to access.

Developers, on the other hand, understand how a class diagram relates to the implementation of a system, but they will probably not see the underlying requirements and their importance. This is because the requirements cannot be explicitly expressed, so that mission-critical requirements are presented only as boxes with lines between them. They are often hidden between dozens of others with little importance. The effect is called "trivialisation of requirements" [2]. So the models' value for communication between stakeholders is limited, and therefore they are not sufficient for documentation.

Literature Overview

In the following, we will present the most notable work about LM.

Capturing Business Knowledge with the UML [2] LM stresses the importance of documentation and a common understanding about the project among all stakeholders. It proposes the creation and maintenance of so-called *Business Context Documents*. A business context document consists of explanatory text and a UML diagram. The text gives background information, which cannot be encoded in the diagram. The goal is to have documentation that gives both a business view and a technical view on the system. It is important, that text and model complement each other. Both use the same vocabulary and must always be consistent. If the model changes, the text has to be updated and vice versa.

LM in its basic form is purely conceptual and does not require the use of specific tools like LP or Elucidative Programming (EP). However, the synchronisation between the UML diagrams and the text can be very complex. Figure 8.3 shows a graphical overview of LM.

Literate Modeling Editor (LiMonE) [46] There exists a research prototype called the *Literate Modelling Editor (LiMonE)*, which supports the

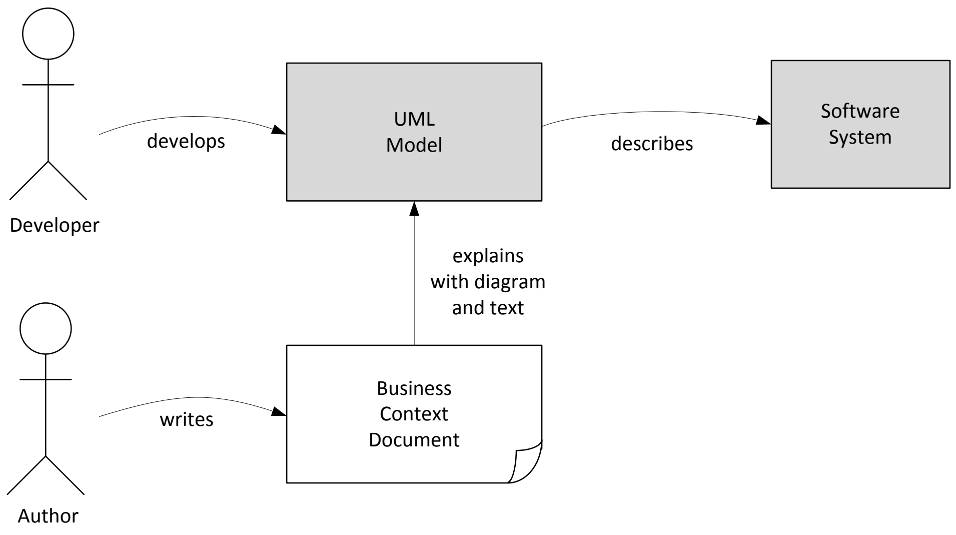


Figure 8.3: Literate Modelling overview.

synchronisation between models and text. Using natural language processing, sentences with references to model elements are mapped to Object Constraint Language (OCL) constraints. These constraints are checked against the documented UML diagram. If conflicts are found, they can be mapped back to the original sentences and highlighted in the LiMonE editing environment.

There are many ways to express a certain model property in natural language and it is not possible to provide an OCL constraint for every possibility. However, rules for a set of common expressions and phrases have been implemented. If the documentation author restricts himself to this limited set of expressions and phrases, automatic validation is possible. Detected inconsistencies are either resolved automatically or by the author.

Comparison with Elucidative Development

The goals of LM with LiMonE are similar to those of ED. Both want to keep a documented view and explanatory text consistent. But while ED aims for generality, i.e., the support for a multitude of views, LiMonE (and LM in general) are specialised solutions for the documentation of UML diagrams with the possibility to check consistency on sentence level. If the text does not correspond to the model, it is either automatically fixed or marked as error. In ED, text content is not analysed. Heuristics, such as the proximity of CDFs, are used to determine whether text might be outdated. If a CDF has been updated, the author is requested by the guidance system to proofread the changed CDF and the surrounding text. But ultimately, the author must decide whether the text is correct or not. In summary, while LiMonE and the concept of ED have some differences, they also share a lot of similarities and complement each other very well.

8.1.3 Elucidative Programming

EP [39] has been introduced as variant of LP, which addresses some of its limitations. The focus of EP is internal software documentation. EP has been defined indirectly by a number of requirements that a documentation process and the corresponding tooling must fulfil [39]. These requirements were very narrow in the beginning, but were broadened over the years. Figure 8.4 shows a graphical overview of EP.

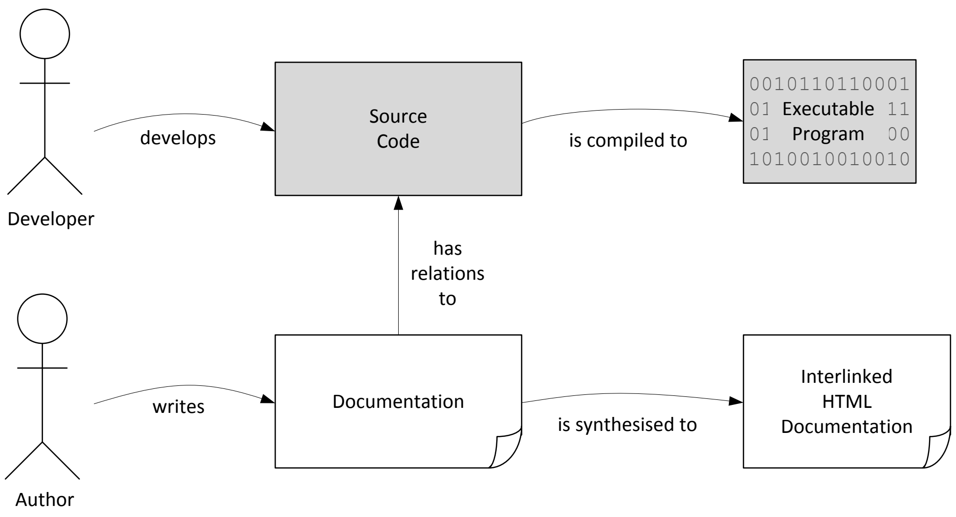


Figure 8.4: Elucidative Programming overview.

EP is very closely related to ED. In fact, it can be seen as the predecessor of ED. In the following, we will highlight some properties of EP and compare them to their ED counterparts.

Application Area

EP was originally targeted at internal documentation of software. Documentation and code should be written in parallel, so that the program understanding at the time of implementation could be recorded [39]. Future developers of a software were the intended audience of the documentation. However, it was soon discovered “that the ideas behind the approach can be used in most situations where there is a need to write about a program. This includes program tutorials, program reviews, and student reports which need to address various program details” [38]. Later works covered tutorials for frameworks and libraries [55, 57].

ED continues the way towards more general documents. The ideas behind ED can be used in most situations where there is a need to write about views of a software system. No assumptions regarding the purpose of the documents are made.

Documentation and Relations

An elucidative program is organised in a so-called *documentation bundle* [39]. The documentation bundle consists of programs, the documentation unit and a setup description. In existing EP implementations, the programs consist of one or many source code files, and the documentation unit consists of structured text files with sections and subsections. It is possible to add relations from the documentation to parts of the program files. There are four kinds of relations. So-called *weak* and *strong relations* point from the documentation to source code. A weak relation is used when a program entity is only mentioned, whereas a strong relation is used for explanations of a program entity. There can also be relations between different parts of the documentation. These are basically cross references. Finally, there are relations in the source code, namely from variable and method occurrences to their definitions. However, these are not defined by the documentation author. They are automatically found by a static program analysis and rendered in the final output. The relations from documentation to source code are language-specific. Figure 8.5, inspired by a figure from [39], shows an example of various relations in documentation and programs.

Source code is addressed by named abstractions, such as class, method, or variable names. While it is forbidden to change source code for the documentation, special comments, so-called *source markers*, are allowed. They allow the addressing of arbitrary code sections while keeping the source code syntactically valid.

An elucidative document also consists of document files and relations to the views to be documented. Relations are called active references in ED. Unlike an EP relation, an active reference has an operation, which transforms the view (via an artefact, see Sect. 4.3.1) into a CDF. Additionally, ED does also have a concept of cross references. In summary, an elucidative document

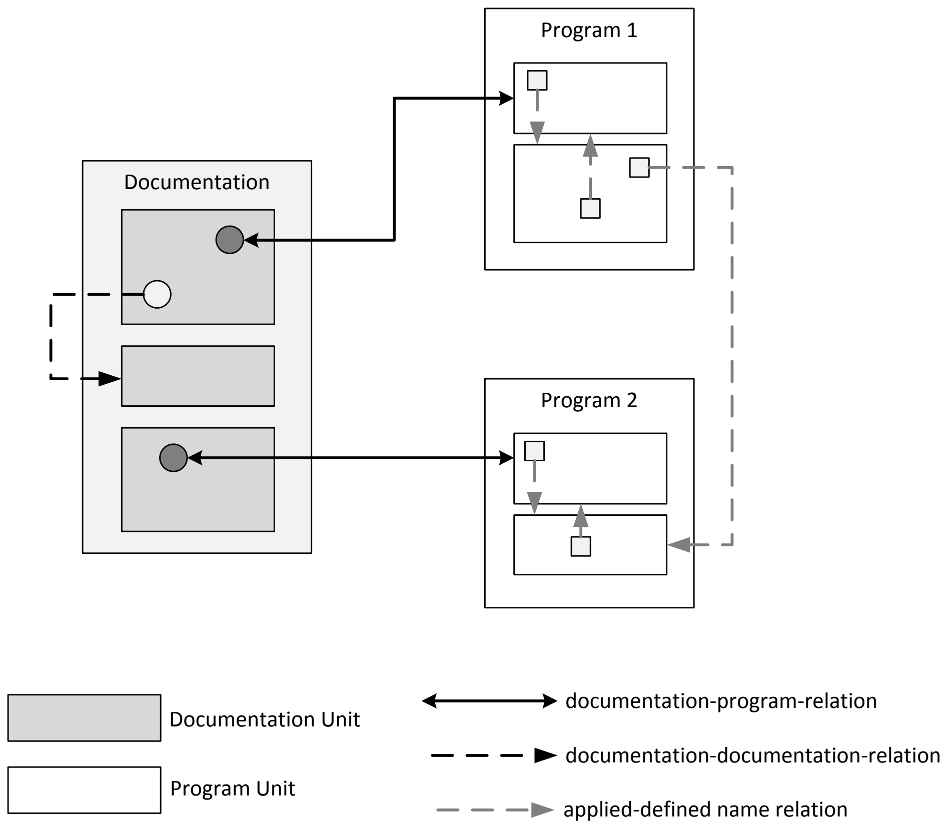

Program 1
Documentation
Program 2
Documentation Unit
Program Unit
documentation-program-relation
documentation-documentation-relation
applied-defined name relation

Figure 8.5: Relations of an elucidative program.

is very similar to an elucidative program. The main difference is that the active references of ED are more powerful and flexible.

Language and Tool Support

EP environments must be language-specific. For example, it is necessary to parse the documented source code in order to create relations to named abstractions, such as classes and methods. The parse tree can also be used to provide editing support with code completion during the creation of relations. The language-specific parts of an EP environment can be collected in one component, the so-called abstractor [38]. If the abstractor is exchanged, other languages can be documented.

Similarly, ED must also be language-specific. In order to add active references to a certain type of view, that view type must be known to the development environment. Unlike in existing elucidative programming environments, it is not necessary to exchange an abstractor in order to document another language. All views can be supported at the same time by providing the necessary operations. This makes it possible to use CDFs computed from different types of views, e.g., UML diagrams, Java code and Extensible Markup Language (XML) configuration files, in the same document.

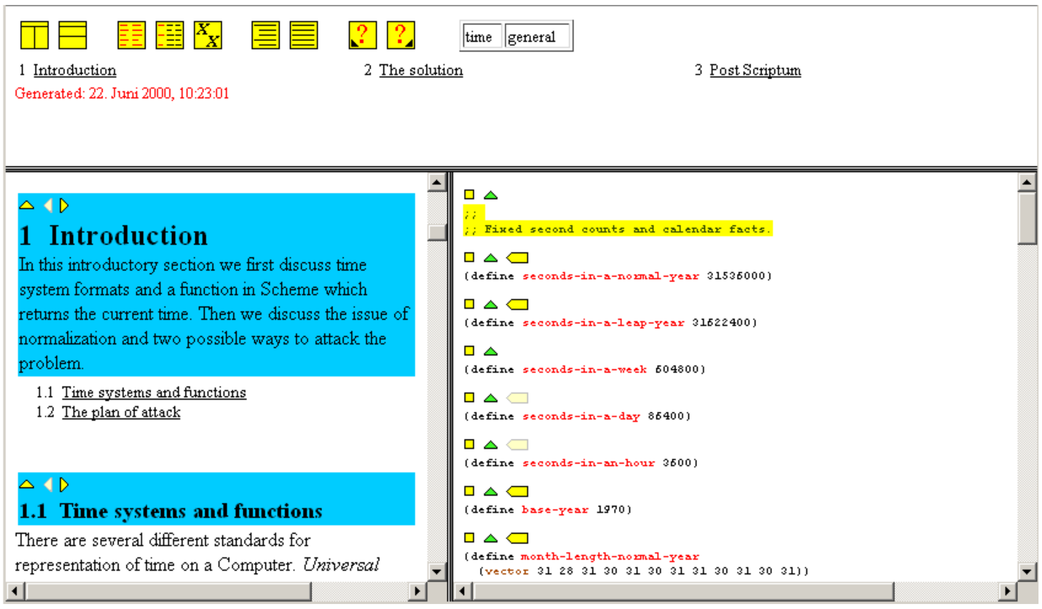
Synthesised Output

An elucidative program can be synthesised (published) to a Hypertext Markup Language (HTML) documentation. The HTML documentation has a 3-frame layout with overall navigation on the top, documentation on the left, and the source code on the right [38]. In early realisations of EP, there were only hyperlinks from the documentation to the code frame and vice versa, which allowed navigation and exploring. Different kinds of relations, so-called *weak* and *strong relations*, were rendered differently, so the reader could see whether some part of the code would be explained in detail or merely mentioned. This was an explicit alternative to the documentation created by LP, where code is embedded in the text. However, it was discovered that following hyperlinks distracts the reader [57]. Therefore, later EP approaches imitated the look of LP documentation. Relations to code were synthesised as embedded code listings, allowing the reader to read documentation and documented code together [54,56]. The hyperlinks were still added, so that the reader could see the code in its context if he wanted. The rendering is performed by a so-called *synthesiser*, which is part of the elucidative programming environment.

An elucidative document can also be published as a standalone document. But in contrast to EP, CDFs can be computed and displayed as soon as active references are added to the document (see Requirement 1). This approach is more flexible than the synthesising in EP because each CDF is computed individually by its operation instead of a common postprocessing step. Additional postprocessing is still possible, though. For example, creating an EP-like multi-frame output with a documentation frame and a code frame, which are mutually hyperlinked (see Fig. 8.6), is only possible with an explicit synthesising step. In summary, ED improves the concepts of existing EP implementations by adding flexibility without sacrificing their power.

Consistency

When a documented program evolves, the documentation ages and loses value. The relations of an elucidative program can help identify inconsistent


time general

1 Introduction 2 The solution 3 Post Scriptum

Generated: 22. Juni 2000, 10:23:01

1 Introduction

In this introductory section we first discuss time system formats and a function in Scheme which returns the current time. Then we discuss the issue of normalization and two possible ways to attack the problem.

1.1 Time systems and functions
1.2 The plan of attack

1.1 Time systems and functions

There are several different standards for representation of time on a Computer. *Universal*

```
;;
;; Fixed second counts and calendar facts.

(define seconds-in-a-normal-year 31536000)

(define seconds-in-a-leap-year 31622400)

(define seconds-in-a-week 604800)

(define seconds-in-a-day 86400)

(define seconds-in-an-hour 3600)

(define base-year 1970)

(define month-length-normal-year
  (vector 31 28 31 30 31 30 31 31 30 31 30 31))
```

Figure 8.6: Synthesised elucidative program with three frames.

parts of the documentation. Dead relations between program and documentation can be found and reported [57]. Similarly, changes in the program can be translated into suggestions, where to update the documentation. Some defects can be fixed with heuristics. For example, relations to methods which have been removed or renamed can possibly be identified. Changes within methods cannot easily be detected by default, according to [55]. With source markers, this is possible, though. EP also supports versioning, which allows for the documentation of program evolution [58]. If versioning is used, relations automatically point to an old version of the source code if their target is missing in the latest version. It is also possible to explicitly set a relation to a specific version.

ED puts much emphasis on documentation consistency during software evolution. Ideas from EP are reused and extended. For example, checking if active references have missing targets is easy. But it is also possible to identify active references whose CDFs are outdated. Every change in the referenced view causing a CDF update is a potential source of inconsistency. In many cases, only the author can decide if such a change results in an inconsistency or not, so all changes must be clearly marked for the author to check (see Requirement 3). As mentioned above, it was stated in [55] that arbitrary changes could not be easily detected. However, this is merely a matter of tool implementation. Changes between an old and a new version of a view can easily be identified if the tool keeps a version history. Another improvement w.r.t. consistency is the possibility to create CDFs from document content, including other CDFs. A versioning approach like in EP would be possible, but has not been further investigated.

8.2 Consistency Approaches

If the same information is stored multiple times in different locations, inconsistencies are possible (see Sect. 2.1). This also counts for documents, such as software documentation. LP solves this problem by using the same document for code and documentation so that both are perceived as a whole. EP and ED use a reference mechanism to automatically reuse information from views in documents. The underlying approaches are *transclusion* and *transconsistency*, which we will present in the following.

8.2.1 Transclusion

Transclusion is the embedding of document parts into other documents instead of duplicating them. “Transclusion means that part of a document may be in several places – in other documents beside the original – without actually being copied there”. The term has been coined by Theodor H. Nelson in [36]. A prerequisite for transclusion is the ability to address portions of a document. This can be done by queries, or by explicit links, to document content. In [36], such links are called *quote-links* because they are used to quote parts of other documents.

The document with the quote-link to another document can be displayed in different ways. If the referenced content is displayed directly instead of the link, we speak of transclusion. Nelson compares this with a “window in the new document [through which] we see a portion of the old”. The document with the window (i.e., the quote-link) is called *compound document* because it seemingly contains content from other documents. Compound documents can be arbitrarily nested if the referenced document content has quote-links itself. Compound documents are always up to date because at all times the current state of a linked document is seen through the “windows”. Figure 8.7 shows this approach graphically. The images are inspired by [36].

However, sometimes an author might not want the transcluded content to be updated without his consent. In this case, he can specify a quote-link to a specific revision. Later he can check “What has this passage become?” [36] and set the quote link to a more current revision.

The transclusion mechanism proposed in [36] is flexible enough to transclude any part of a document, but it cannot handle more advanced scenarios.

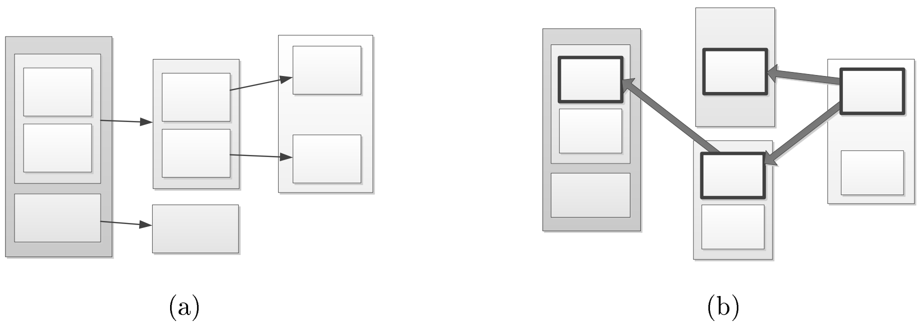


Figure 8.7: Transclusion – (a) shows a document (left) with nested windows to other documents (centre and right), (b) shows how changes are propagated along the documents to achieve immediate updates.

For example, it is not possible to filter or sort items of an enumeration. Filtering or sorting would require to transclude each item of the enumeration individually in the correct order into the compound document.

Transclusion was only one of the concepts that Nelson had in mind for the future of documents. His goal was and still is to overcome the limitations of paper. He envisioned a multi-user system that would electronically store and deliver media, e.g., text, images, music, and interconnections between them: in other words, a worldwide hypertext system. Documents could have multiple alternative versions and historical revisions, which could be compared. There was even a micropayment concept for transcluded content. The implementation of this system was called Xanadu. However, there were only few prototypes. They all took very long to develop and were ultimately not sufficient to attract enough funding for further development. In the 90s, the Xanadu approach lost ground to the conceptually simpler World Wide Web (WWW). While Xanadu had no big success, the project is still alive[1]. In 2007, Holm presented XanaduSpace [37], a program which allows a 3D-visualisation of documents and their links. However, as of 2013, the web presence has been closed down. It is still archived, though[2].

The idea of transclusion was also followed outside Xanadu. One system for transclusion in HTML is presented in [32]. The Uniform Resource Locator (URL) and the offsets of the transcluded text are stored and reevaluated whenever the document is displayed. However, this approach is not robust against changes in the transcluded document. The authors propose to inform the author of the document when transcluded documents change.

[1]`http://www.xanadu.com/`

[2]`http://web.archive.org/web/20130615044618/http://xanarama.net/`

IFrames in HTML are a lightweight version of transclusion. It is possible to embed other documents in a website, but it is always the complete document that is transcluded. It is not possible to choose a finer granularity, such as a paragraph or a sentence. Furthermore, it is not explicitly stored which documents are transcluded by which documents, which was an important factor in Nelson's concept.

ED uses some parts of the transclusion approach. It allows the inclusion of document content or views. The concept of nested compound documents has not been adopted, though. In ED, the active reference to a view automatically points to a copy of the view, i.e., the artefact. If the view changes, the affected CDFs are not automatically updated. But like in Nelson's approach, it is possible to have the change reported to the author. He can then update the artefact, so that it contains the new content of the view. This causes a recomputation of the CDFs.

Transclusion in its pure form is not suited to document software views, because transclusion can only handle displayable content. For ED, many views, such as database content, must be transformed into a displayable representation first.

8.2.2 Transconsistency and Active Documents

Transconsistency [3] is an extension of transclusion, which goes beyond simple text inclusion. It has been introduced together with the concept of active documents. An *active document* is defined as "a component-based document with a set of derived components that is computed from a set of base components. To this end, it contains or is tightly associated with software."

The associated software can either be physically embedded within the document, such as a macro inside a Microsoft Word document, or it can be outside the document, such as a Java Server Pages (JSP) script, which produces an HTML page. Active documents have an implicit form, which contains so-called *template components.* The software can expand the template components to their explicit form, the so-called *derived components.*

If the document components depend on each other and the software of the document forms an acyclic data-flow graph of operations, the document is called *active document with data-flow architecture.* Examples of such dependencies are images or tables, which are computed from other parts of the document or some external data sources.

Whenever a source changes, the corresponding components must immediately be recomputed and updated. In an active document with data-flow architecture, the components form a so-called *transconsistent dataflow graph,* which has a set of inputs, a number of arbitrary operations that modify the data, and results. The results are derived components, which can serve as input for other components and/or which are actually displayed in a document. Figure 8.8 shows an example.

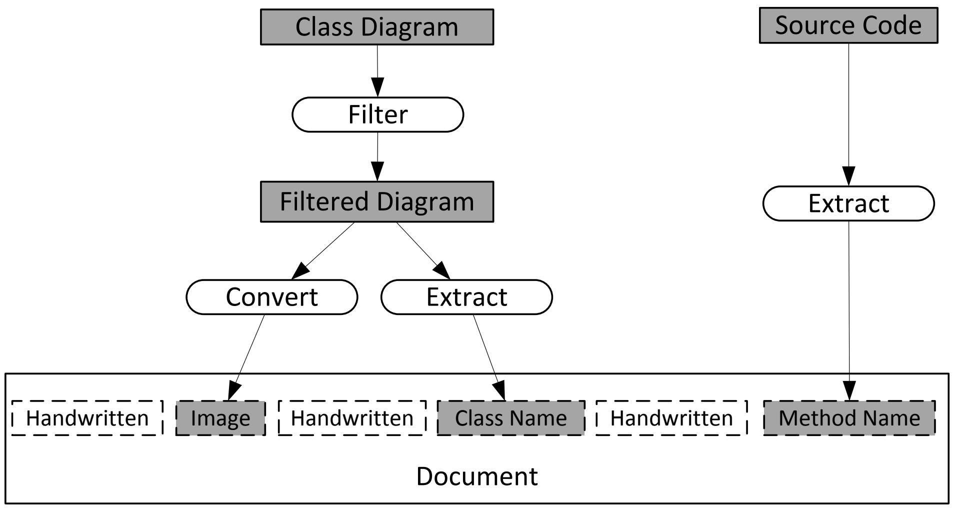


Figure 8.8: Transconsistent dataflow graph.

Elucidative documents are active documents. This relationship is the main reason for our choice of the terms *active* reference and operation in the scope of ED.

The template components of active documents correspond to active references of elucidative documents. The operations, which compute the explicit form of the active document, correspond to the operations in ED. Finally, the expanded templates of active documents correspond to CDFs.

ED uses transconsistency for document references. This means, that changes in the referenced document parts are immediately reflected in the corresponding CDFs. This is necessary to maintain local consistency.

However, transconsistency in its pure form is not used for artefact references. The changes in a view of the software system must not be applied immediately to the document (see Requirement 2). Instead, the update must be delayed until the author explicitly triggers it. We call this new concept *weak transconsistency*.

8.3 Compound Documents

There exist several approaches to embed documents or some of their parts into another document. Documents which contain embedded documents are called *compound documents*. Many of these approaches origin from the 1990s, the time when component oriented development gained momentum. Thus, many compound document approaches were not only able to put document content together, but also to embed application parts which could edit this document content. In the following, we present some of them.

8.3.1 Object Linking and Embedding

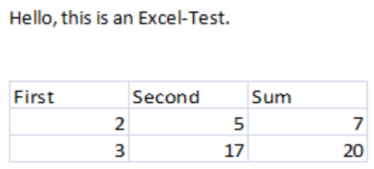
OLE is a technology from Microsoft for the creation of compound documents [15, 16]. It is based on Microsoft's Component Object Model (COM). COM is a software component architecture which allows the definition of components, which have interfaces with methods. COM objects are implemented within a *server*. A server contains the actual code that the component executes when one of its methods is called.

A compound document is a document that allows the inclusion of other documents. An example of an OLE compound document is a Microsoft Word document with an embedded Excel table. The "parent" document is called *container* (Word). The included "child" document content is provided by a different program, the server (Excel).

When the user inserts a child document into the container, the container displays a graphical representation of the child document. That graphical representation is created by the server. The container does neither know the content of the child document, nor how it should be rendered.

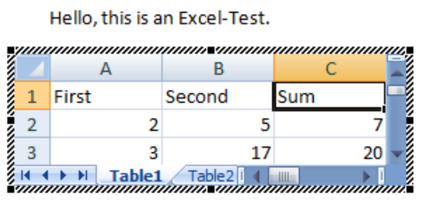
The inserted document can be activated. When an inserted document is activated, the corresponding server application is started (if it is not already running) and the document content can be edited. The server can either start as an independent application in its own window, or inside the container application. If the server application starts as an independent application, all performed changes are forwarded to the graphical representation in the container. Thus, the graphical representation always reflects the current state of the edited document. If the server application starts inside the container application, it is displayed in place of the graphical representation. This gives the user the impression of a single document being edited, e.g., an Excel table that can be edited in Word, as shown in Fig. 8.9.

There are two possibilities to include a document in a container: by embedding or by linking. If documents are embedded, they are physically stored inside the container file. This is implemented by the structured stor-


Hello, this is an Excel-Test.

First	Second	Sum
2	5	7
3	17	20

(a) Excel table: graphical representation.


Hello, this is an Excel-Test.

	A	B	C
1	First	Second	Sum
2	2	5	7
3	3	17	20

(b) Excel table: activated.

Figure 8.9: Excel table in Word: normal and activated.

age model of COM. It allows multiple COM objects to store their information in a single file. In case of an embedded OLE document, the stored information comprises the actual document data (e.g., an Excel table), its graphical representation, and management information. Saving the graphical representation in the structured storage makes it possible to load and display the whole compound document even if the servers of the embedded documents are not running.

If documents are linked, a reference to the included document is saved, together with the graphical representation. The reference contains data that describes the location of the included document. In the case of Excel, this comprises the Excel-file, the worksheet, and the cell range on that worksheet. When the referenced document is activated, the reference information is used to initialise the server application, e.g., make Excel load the right file and highlight the cell range.

8.3.2 OpenDoc

OpenDoc was a compound document framework developed by Apple and International Business Machines Corporation (IBM) in the 1990s, that aimed at compatibility and interoperability of components, written on different platforms and in different programming languages[3]. It was based on System Object Model (SOM), a Common Object Request Broker Architecture (CORBA)-compliant object request broker from IBM. The development of OpenDoc was stopped in 1997.

OpenDoc documents were containers that could contain components, so-called *parts*, including other containers. The embedded parts of an OpenDoc document would allow both editing and displaying of contents. Parts include, for example, whole word processors, but also smaller parts, such as fonts or

[3]`http://web.archive.org/web/19961225115239/http://www.software.ibm.com/clubopendoc/standish.html`

spellcheckers. There were also implementations to provide Java applets and HTML documents from the Internet as parts[4].

Components were not only logically embedded, but physically[5]. The data storage mechanism, called Bento (named after the compartmentalised Japanese lunch box), enabled data and documents to be stored within a document. Containers had to provide rendering support for their embedded components, taking care of layout and visual representation.

OpenDoc did not specify how document components had to be displayed or modified via a user interface, making documents portable in theory. Actual multi-platform support was complex, though. It required the usage of cross-platform Graphical User Interface (GUI) libraries or a re-implementation of the components on every supported platform.

OpenDoc also supported linking[6]. Whenever linked data was updated, the component was automatically updated, too. In order to keep a certain revision of linked data, it was possible to store drafts of a document and review them later.

8.3.3 The W3C Compound Document by Reference Framework

The Compound Document by Reference Framework was a World Wide Web Consortium (W3C) standard specification under development[7]. Compound documents were documents that combine multiple document formats. Documents could be combined by reference, by inclusion, or both. The hierarchy of nested documents could be arbitrarily deep.

If documents were combined by reference, the child documents existed only logically in the parent document. In order to reference child documents, existing elements of the parent document should be used, e.g., the `object` element in Extensible Hypertext Markup Language (XHTML) documents, the `foreignObject` element in Scalable Vector Graphics (SVG) or the `ref` element in Synchronized Multimedia Integration Language (SMIL). That is, references to child documents could appear anywhere in the document where the corresponding reference elements were allowed. It was possible to add parameters to the references. The specification allowed child documents to be either XML-based or native, i.e., have binary content[8].

[4]`http://web.archive.org/web/19961225075631/http://www.software.ibm.com/clubopendoc/partpaks/webpak.html`

[5]`http://www.scoug.com/OS24U/1996/scoug608.2.opdoc201.html`

[6]`http://www.scoug.com/OS24U/1996/scoug607.2.opdoc102.html`

[7]`http://www.w3.org/2004/CDF/`

[8]`http://www.w3.org/TR/WICD/`

If documents were combined by inclusion, the child documents existed physically in the parent document. XML elements from different document formats (e.g., XHTML and SVG) were distinguished by different namespaces. A *compound document profile* was used to define which tags from which namespace were allowed as child elements of an element[9].

The standard also covered hyperlinking, focus handling for keyboard navigation and media synchronisation, such as video and audio streams. The development of the standard was discontinued in 2010.

8.3.4 HotDoc

HotDoc is a SmallTalk framework for the construction of editors for compound documents [13]. A HotDoc document can contain parts, which are like "windows to applications", for example text editors or video players. Thus, HotDoc documents are at the same time documents in the classical sense and user interfaces.

Parts can contain other parts. The part developer has to specify, which part types can be inserted into the part he developed. Parts can be inserted at a certain position inside the document or another part, or they can be automatically layouted, using a layout policy.

Two or more parts of a document can share the same data model. This is achieved by "linking" them together. For example, a chart part may be linked to a spreadsheet part. When the spreadsheet data is changed, the chart is updated. Parts must be "link compatible" in order to link them.

8.4 Conclusion

In this chapter, we gave an extensive overview about work related to ED. First, we presented a number of documentation approaches. We described their basic idea, presented a literature overview, and compared them to ED. LP, the earliest of the related documentation approaches, requires a program and its documentation text being written side by side in the same document, the so-called literate program. There are tools to postprocess the literate program and extract a running program and human-readable documentation. LM, which has been introduced as an extension of LP, is an approach to keep UML models and their documentation consistent. Tool support is not necessary, but useful and encouraged. EP is also an extension of LP. It is used to keep source code and its documentation consistent. Documentation

[9]`http://web.archive.org/web/20121025121059/http://www.ibm.com/developerworks/library/x-mdcdd/`

is partly generated from source code, using generation directives. Thus, it can be seen as a special case of ED.

Then, we introduced consistency approaches which are used in ED. Transclusion is the simple inclusion of a document or some of its parts into another document. The transcluded document content is not only a copy of the other document content. Changes in the transcluded document are immediately reflected in the transcluding document. Transconsistency is similar to transclusion. The difference is, that the transcluded content can be modified by a transformation. Thus, transconsistency is more flexible than transclusion.

Finally, we presented a short overview of the history and the functionality of compound documents. Compound documents are practical applications of transclusion and transconsistency. The most prominent example is OLE, which, among others, allows the linking and embedding of Microsoft Office documents into other Microsoft Office documents.

Chapter 9

Evaluation

In this chapter, we present the evaluations of Elucidative Development (ED), that we performed with our Elucidative Development Environment (EDE) Development Environment For Tutorials (DEFT). The evaluations comprise both case studies, which measure the improvements that come with ED, and feasibility studies, which show the versatility of ED.

First, we describe our efforts and the results of turning a handwritten model specification into an elucidative document. We list the inconsistencies that we found in the original specification and examine how the situation improved with the use of DEFT.

Then, we show how ED can be used to create the Unified Modeling Language (UML) specification document, assuming that there exists a machine-readable UML metamodel. This section is based on [59].

Finally, we show several examples where we successfully used ED. This includes a part of a requirements specification, a programming tutorial for a Business Process Modelling Notation (BPMN) library, and the writing of this thesis.

9.1 Creating and Maintaining the Cool Component Specification

The CoolSoftware project[1] is a research project with the aim to optimise software w.r.t. energy consumption. A part of the project is the development and modelling of Energy Auto-Tuning (EAT) software systems. Such systems can utilise or turn off underlying resources to finish computations

[1] `http://www.cool-software.org/`

as energy-efficiently as possible. EAT systems are explicitly modelled by the CoolSoftware architecture.

One part of the CoolSoftware architecture is the Cool Component Model (CCM). The CCM is described in a specification, the Cool Component Model Specification. The specification is written in LaTeX. There exists also an Ecore implementation of the CCM metamodel.

In the past, the specification was written and maintained manually by multiple authors. This was both very time-consuming and error-prone. The Ecore metamodel allowed us to use ED with DEFT for the further evolution of the specification.

The case study had two purposes:

- Find out which errors occurred in the manually written specification and how often they occurred.
- Find out to what extent the errors are mitigated with ED.

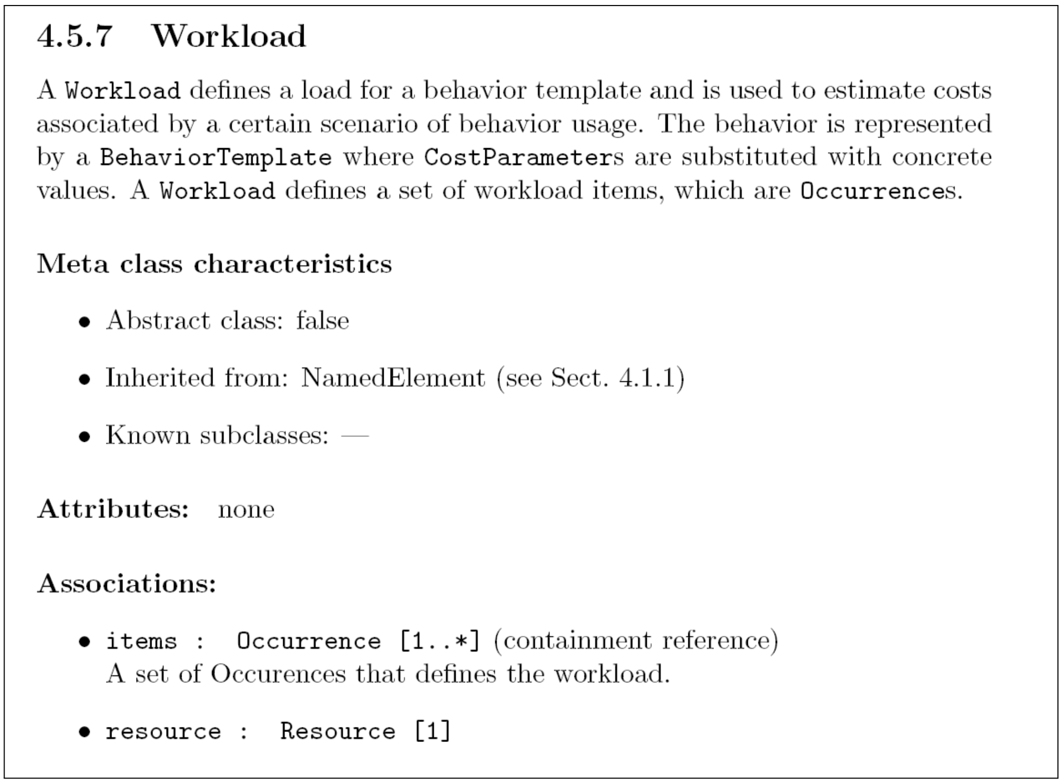
9.1.1 Rewriting the Specification with ED

The purpose of the first part of the case study was the identification of inconsistencies in the manually created specification. This was achieved by creating a consistent version of the specification with DEFT and comparing it to the manually created specification. However, there were inconsistencies between the specification and the metamodel, so we modified the metamodel to match the specification as closely as possible (see Appendix A).

The manually written specification had 441 metamodel identifiers (class names, attribute names, reference names) in the running text. Furthermore, there were 52 listings with metaclass characteristics, one listing for each metaclass. Metaclass characteristics listings are formal listings of metaclass properties, such as being abstract, the list of superclasses or the list of attributes. Object Constraint Language (OCL) constraints were specified for 13 metaclasses, and a textual constraint was defined for one metaclass. The metamodel contained 6 enumerations, which had listings similar to the metaclass characteristics listings in the specification, too. Finally, there were 13 figures with model diagram images, showing the relationship of the metaclasses graphically. Figure 9.1 shows the documentation of the `Workload` metaclass of the CCM metamodel.

In order to rewrite the specification with DEFT, we implemented operations for most of these contents.

- Ecore model to Metaclass Characteristics listing: Creates a metaclass characteristics listing.



4.5.7 Workload

A `Workload` defines a load for a behavior template and is used to estimate costs associated by a certain scenario of behavior usage. The behavior is represented by a `BehaviorTemplate` where `CostParameter`s are substituted with concrete values. A `Workload` defines a set of workload items, which are `Occurrence`s.

Meta class characteristics

- Abstract class: false
- Inherited from: NamedElement (see Sect. 4.1.1)
- Known subclasses: —

Attributes: none

Associations:

- `items : Occurrence [1..*]` (containment reference)
  A set of Occurences that defines the workload.
- `resource : Resource [1]`

Figure 9.1: Excerpt of the Cool Component Model specification.

- Ecore model to Identifier: Creates a marked up identifier (metaclass, attribute or reference name) in the running text.
- Ecore diagram to Image: Creates an image from an Ecore diagram and the corresponding LaTeX figure directives.

We did not implement operations for OCL constraints, because we did not have the necessary tools to perform consistency checks against the metamodel. Besides, we did not have enough detail knowledge of the metamodel to repair broken OCL constraints. We also did not implement operations for enumerations, because there were only few of them, they contained no errors and there were no changes from the old version of the metamodel to the new version. However, if the specification is going to be maintained with DEFT beyond the evaluation scope of this thesis, it would be useful to also support OCL constraints and enumeration listings.

After the operations were added to DEFT, we imported the first version of the CCM metamodel. We copied the running text from the manually written specification and replaced all diagram images, metaclass characteristics listings and metamodel identifiers from the running text with Computed

Document Fragments (CDFs). The result was a specification that looked like the original specification, except that it contained fewer inconsistencies.

For this case study, we added all active references individually. Active reference groups, as presented in Chap. 5 were not used, because active reference group support was not implemented in DEFT.

9.1.2 Finding Inconsistencies in the Manual Specification

We searched for inconsistencies in the manually written specification by comparing it to the specification written with DEFT. We performed the comparison manually and made detailed notes[2]. An automatic comparison was not feasible. The LaTeX source code of the two specification documents was too different because the CDFs had to be enriched with additional markup.

The comparison yielded a number of consistency issues. Most of them were minor, but we also found serious errors. The issues could be classified as follows:

Convention errors The structure of the specification should be uniform. This includes using always the same pattern for metaclass characteristics listings, the use or omission of cross references, capitalisation, abbreviations, and similar things. Deviations in the uniformity of the document are convention errors. Convention errors are minor and do not convey wrong information. However, they lower the perceived quality of the document. The convention errors we identified were not evenly distributed over the specification. The vast majority appeared uniformly throughout certain sections or subsections. This indicates that different authors used different conventions during different times of writing.

Misspelled and wrong identifiers Sometimes, the specification mentions identifiers which do not exist. In some cases, this is due to obvious typing errors, for example `Visibilty` instead of `Visibility`. They are minor and do not pose problems, but they lower the perceived quality of the document. In other cases, the wrong names are not easy to spot on first sight, for example `UserType` instead of `User`. Wrong identifiers convey wrong information and are actual errors. They are either originating from oversights or they are outdated, i.e., they result from metamodel changes that have not been applied to the specification.

[2]The detailed notes can be downloaded from `http://bartho.net/phdthesis`. A summary is presented in Appendix A.

Wrong metaclass properties Each metaclass is described by various properties, such as a list of superclasses and subclasses, attributes with type and multiplicity, and others. Missing or wrong properties are a serious problem and a severe error. The most probable reasons for wrong metaclass properties are oversights and missing specification updates after metamodel changes.

Wrong model diagrams Diagrams display the metaclasses and enumerations graphically, together with their interrelations, attributes and associations. The diagrams must contain the same information as the metaclass descriptions. They have originally been generated from the metamodel. Errors indicate that the used diagram images have been created for an earlier version of the specification and have not been updated since then.

Appendix A contains more detailed information about these errors.

9.1.3 Updating the Specification to a new Metamodel Version

The purpose of the second part of the case study was the evaluation of DEFT for a specification update. We updated the artefacts in the DEFT repository (metamodel and diagram files) to the latest available version. As a result, DEFT reported several changes in the CDFs of the specification.

The changes between the old and the new metamodel were rather comprehensive. The `datatypes` package and the `expressions` package were considerably revised, which made us replace the corresponding specification sections with completely new sections. We only added the metaclass characteristics listings to the new sections because our knowledge of the metamodel was not detailed enough to write the accompanying running text. Writing the text is left to the original authors of the specification.

The `OperationCallExpression` from the `expressions` package had 20 subclasses, such as `AdditiveOperationCallExpression`, `SubtractiveOperationCallExpression` and similar. We decided that these subclasses did not require subsections of their own. Therefore, we had to modify the operation for metaclass characteristics listings a little. We changed it so that metaclass characteristics listings only contain cross references if the corresponding metaclasses are in the specification.

9.1.4 Discussion

The evaluation showed that ED is well suited for LaTeX based specifications with textual CDFs. The corresponding operations can be implemented quickly and pay off immediately. Even the required modification of the metaclass characteristics listing operation was simple[3].

The vast amount of identifiers in the running text forced us to improve the usability of DEFT. We implemented keyboard shortcuts and an autocompletion mechanism for the insertion of identifier CDFs. This turned out to be very useful. Afterwards, the identifier CDFs could be added very quickly and the flow of writing was hardly impaired.

Operations to create images from Ecore diagrams could also be implemented easily. Unfortunately, Ecore diagrams turned out to be unsuitable for ED. Ecore diagrams contain references to Ecore models. Therefore, they do not need to store information such as attributes or references themselves. However, when they are opened (or transformed into an image), the attributes and references are not always displayed. There is no status flag indicating whether the attribute and reference sections within the Ecore classes should be expanded or not. Therefore, the Ecore diagrams added with ED are not reliable. We were not able to solve this problem.

The notification of changed CDFs is very useful, but in DEFT we only used a simple list, which was not very convenient. The number of CDF changes was very high, and it turned out that a list view is not suited to present such a large number of changes clearly. A hierarchical presentation, possibly grouped by sections and subsections, would be beneficial. Another useful improvement would be the availability of a diff-view that shows the exact CDF changes.

Apart from the minor, implementation-specific shortcomings, the specification creation and maintenance worked well.

9.2 Creating and Maintaining the UML Specification

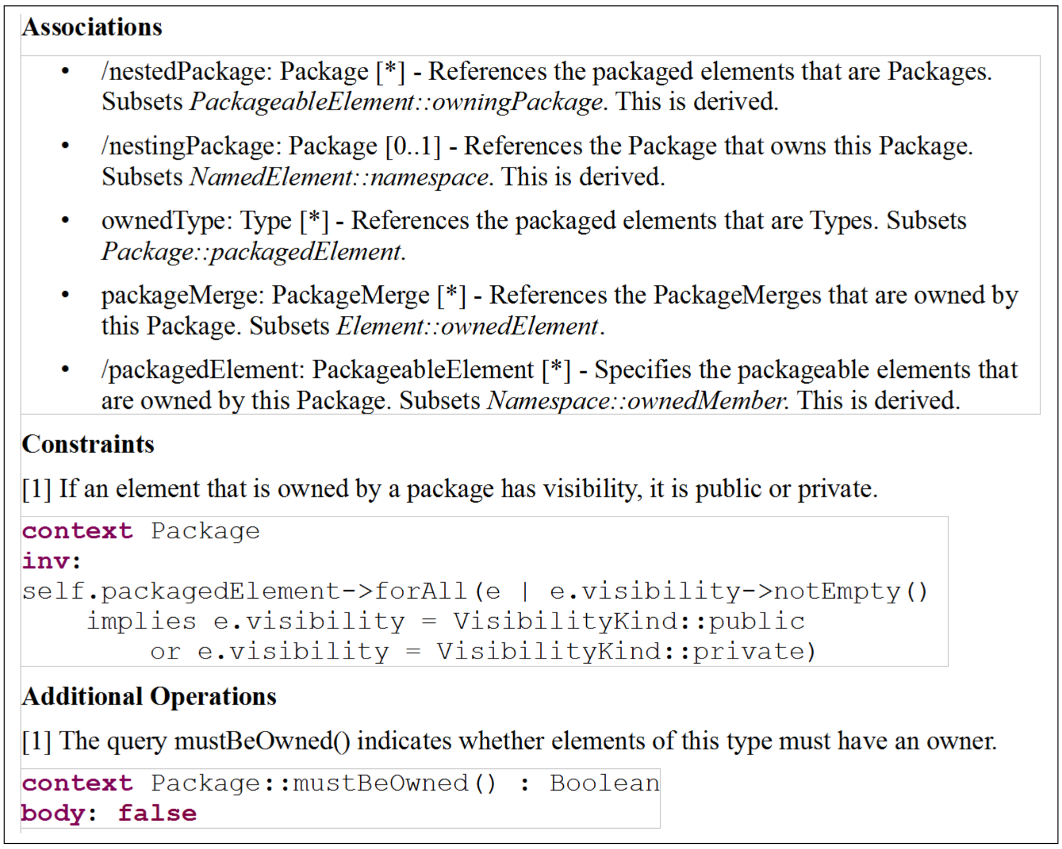
Since its first version, the UML specification has undergone a lot of revisions and has received many changes. Due to a high degree of interdependencies between the chapters of the UML specification and a lack of automatic consistency checks, current versions contain many inconsistencies. This problem

[3]It involved use of the `labelcas` package, which allows the specification of different output, depending on whether a label exists or not.

has been pointed out in various papers, e.g. [9, 61]. In [61], it was discovered that almost 50% of the OCL rules from the UML 2.3 superstructure specification[4] contain errors.

Most inconsistencies could be resolved if the UML specification did not only consist of a specification document, but also comprised an implementation of the metamodel and the OCL rules. Tools could check the model and the OCL rules for syntactic and some semantic errors. An additional benefit would be that the checked models and OCL rules could be used to partially generate the specification document with ED.

As a proof of concept, we rewrote the "Package" section (Sect. 7.3.37) of the UML 2.0 superstructure specification[5] with DEFT. Afterwards, we examined how well specification updates can be performed. We did this by updating the "Package" section to UML 2.3. Figure 9.2 shows an excerpt of the result.



Associations

- /nestedPackage: Package [*] - References the packaged elements that are Packages. Subsets *PackageableElement::owningPackage*. This is derived.
- /nestingPackage: Package [0..1] - References the Package that owns this Package. Subsets *NamedElement::namespace*. This is derived.
- ownedType: Type [*] - References the packaged elements that are Types. Subsets *Package::packagedElement*.
- packageMerge: PackageMerge [*] - References the PackageMerges that are owned by this Package. Subsets *Element::ownedElement*.
- /packagedElement: PackageableElement [*] - Specifies the packageable elements that are owned by this Package. Subsets *Namespace::ownedMember*. This is derived.

Constraints

[1] If an element that is owned by a package has visibility, it is public or private.

```
context Package
inv:
self.packagedElement->forAll(e | e.visibility->notEmpty()
    implies e.visibility = VisibilityKind::public
        or e.visibility = VisibilityKind::private)
```

Additional Operations

[1] The query mustBeOwned() indicates whether elements of this type must have an owner.

```
context Package::mustBeOwned() : Boolean
body: false
```

Figure 9.2: Excerpt of the UML specification.

[4]http://www.omg.org/spec/UML/2.3/Superstructure/PDF/
[5]http://www.omg.org/spec/UML/2.0/Superstructure/PDF/

We used the Ecore-based UML 2.3 metamodel, which came with the Eclipse Modelling Tools[6]. The contents of the metamodel and the UML specification were slightly different. We did not investigate the reasons for these differences because that was not the goal of this evaluation. We wanted to show that, given a proper metamodel, the UML specification could be created with ED. Therefore, we chose to modify the metamodel slightly, so that it matches the UML 2.3 specification. Afterwards, we created a modified copy, that corresponds to the UML 2.0 specification. The OCL constraints and operations have been copied and pasted from the specification, from both version 2.0 and version 2.3. They contained syntactic and semantic errors which were fixed, so they could be parsed with the Dresden OCL toolkit[7] and successfully applied to the Ecore-UML metamodel[8]. All identified differences between the specification and the artefacts used for our evaluation are listed in Appendix B.

9.2.1 Preparing DEFT for UML

Before we could write the "Package" section, we had to configure DEFT accordingly. First, we made it recognise the necessary artefacts. Support for the Ecore-UML metamodel could be easily added because DEFT already supports Ecore out of the box. We also integrated the EMFText[9]-based Dresden OCL toolkit into DEFT. As DEFT already comes with built-in EMFText support, the integration posed no problems.

Then, we implemented the necessary operations, which transform the Ecore-UML metamodel and the OCL code to textual representations. These operations included

- Ecore Model to Generalizations Listing: Creates a listing of all superclasses of a metaclass.
- Ecore Model to Attributes Listing: Creates a listing of all attributes of a metaclass.
- Ecore Model to Associations Listing: Creates a listing of all associations (references) of a metaclass.
- OCL to Styled Code: Creates a formatted OCL code listing from an OCL constraint or operation.

[6] `http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/junor`

[7] `http://www.dresden-ocl.org`

[8] The fixed OCL has been created by Claas Wilke for [61]. It has been reused for the joint publications [59,60], on which this section is based.

[9] `http://www.emftext.org/`

9.2.2 Writing and Updating the Package Section

After support for all artefact types and operations was provided, we imported the UML 2.0 metamodel as well as the corresponding OCL constraints and operations into DEFT. We copied the running text from the specification. The listings of generalisations, attributes and associations and the OCL code listings were added as CDFs. The result was a section that looked like the original specification, except for the OCL errors, which had been fixed.

Then we updated all artefacts from the DEFT repository to UML 2.3. This caused changes in the associations listing and some OCL code listings. These changes were reported by DEFT, which allowed us to double-check and explicitly accept them.

The result was the "Package" section of the UML 2.3 superstructure specification with consistent metaclass and OCL code listings. We did not check the running text for consistency in our proof of concept example. This task must still be performed manually and is not directly influenced by ED.

9.2.3 Discussion

The evaluation showed that ED is well suited for the presentation of class diagrams and OCL code as textual CDFs, such as listings or code listings. The corresponding operations can be implemented quickly and pay off immediately. The notification of changed CDFs is useful, but similar to the Cool Component evaluation in Sect. 9.1 a diff view would have been beneficial. This is especially the case for small changes. But all in all, the "Package" section of our proof of concept example could be quickly written and updated with DEFT.

9.3 Feasibility Studies

In this section, we show three examples of useful ED scenarios. They describe different types of views of a software system. First, we show a part of a requirements specification, created from formalised requirements stored in an ontology. Then, we show a programming tutorial that teaches how to use a Java-based BPMN refinement library. Finally, we show how we used DEFT to write this thesis as an elucidative document.

9.3.1 Visualising Requirements

In the following, we describe the results of our experiments with formalised requirements. These experiments were a feasibility study to show that even requirements can be documented with ED. First, we will shortly introduce the research work from which we received the formalised requirements. Then, we describe how we visualised the requirements.

Background

Requirements Engineering is “the process of eliciting, evaluating, specifying, consolidating, and changing the objectives, functionalities, qualities, and constraints to be achieved by a software-intensive system” [34]. The result of requirements engineering is a requirements specification. Requirements specifications are frequently used as basis for the contract between customers and software developers, and as starting point for the analysis and design of the software product to be developed.

Inconsistent or incomplete requirements lead to software which does not meet the demands of the customers. However, requirements specifications can be very complex, which makes manual consistency checking tedious and error-prone. Furthermore, requirements can evolve. Therefore, Ontology-Driven Requirements Engineering (ODRE) has been developed as a means to automatically check requirements specifications for consistency [50]. It focuses on *goal-oriented requirements engineering*.

The central element of ODRE is the Requirements Ontology (RO). The T-Box of the RO formalises requirements engineering knowledge. It can be considered the metamodel of goal oriented requirements engineering. The A-Box of the RO contains the actual requirements specification of a concrete project. This comprises goals, requirements, problems, challenges, and others, including relationships between them. Additionally, there are rules, which describe when the RO A-Box is complete or consistent. Violation of these rules means that the RO contains incomplete or inconsistent data. Figure 9.3 shows an architecture overview of ODRE, inspired by [49].

Using the Requirements Ontology for Elucidative Development

When a requirements specification should be checked for completeness and consistency, it must be formalised and encoded in the RO. Such a formalised requirements specification is well suited for ED. The reasoning facilities of ODRE are not needed for the writing of the specification, because we assume that the RO is checked for completeness and consistency before it is used for document creation. In [10], we investigated possible ways to represent requirements and their interdependencies. We found tree representations and traceability matrixes the most useful.

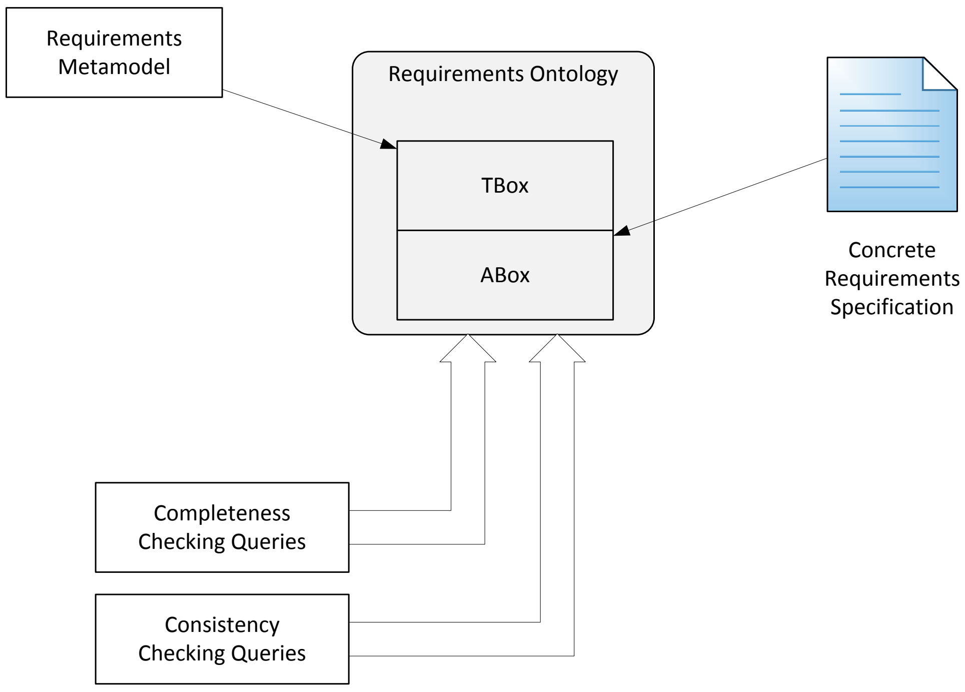
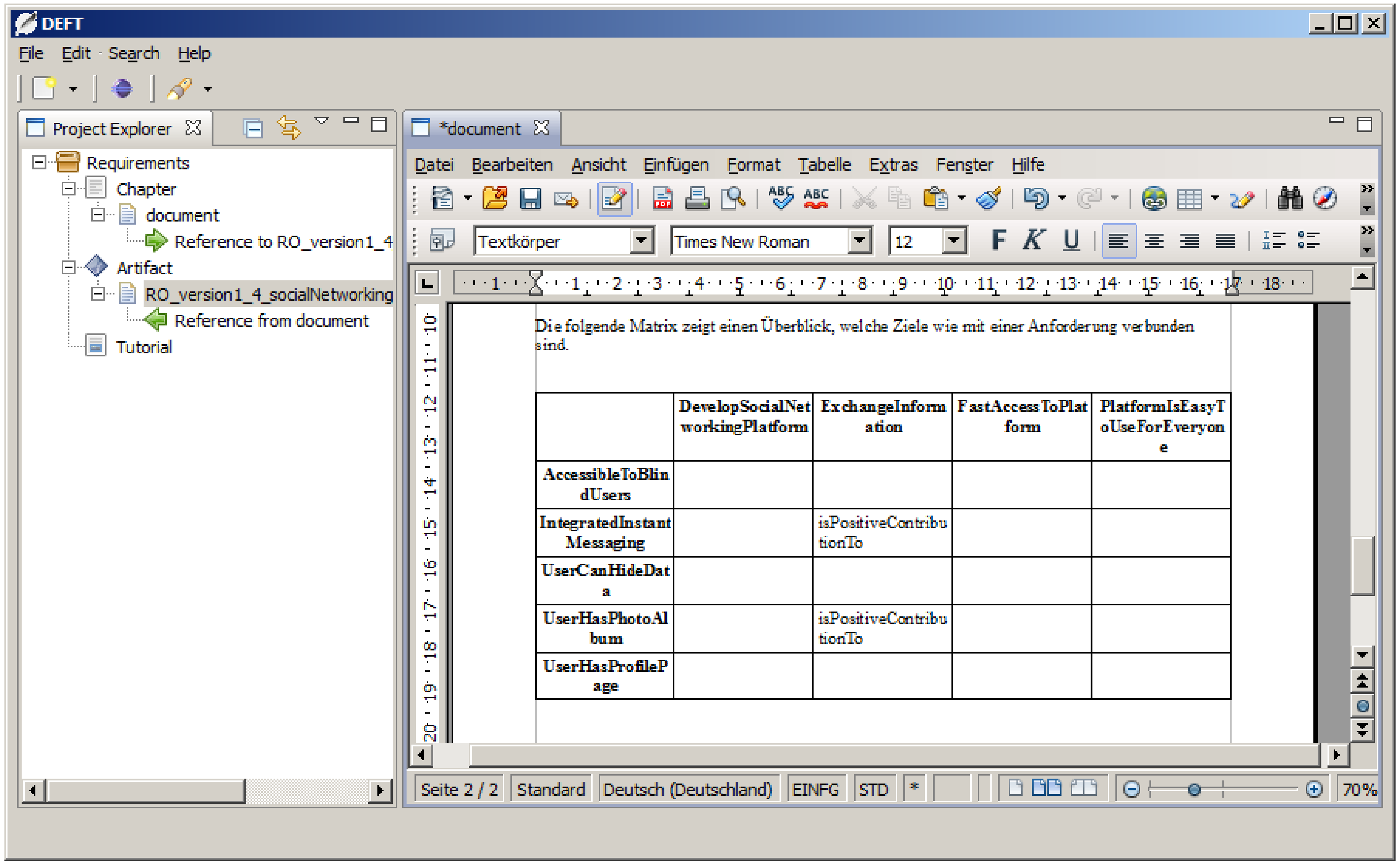

Requirements Metamodel
Requirements Ontology
TBox
ABox
Concrete Requirements Specification
Completeness Checking Queries
Consistency Checking Queries

Figure 9.3: The Requirements Ontology in Ontology Driven Requirements Engineering.

Trees can be used to display hierarchies, for example a hierarchy of goals and subgoals. Traceability matrixes are tables, which can be used to display relationships between different elements of the requirements specification. Row headings contain source elements and column headings contain target elements, or vice versa. The actual table columns indicate whether the elements have a certain relation. Figures 9.4 and 9.5 show examples from [10].

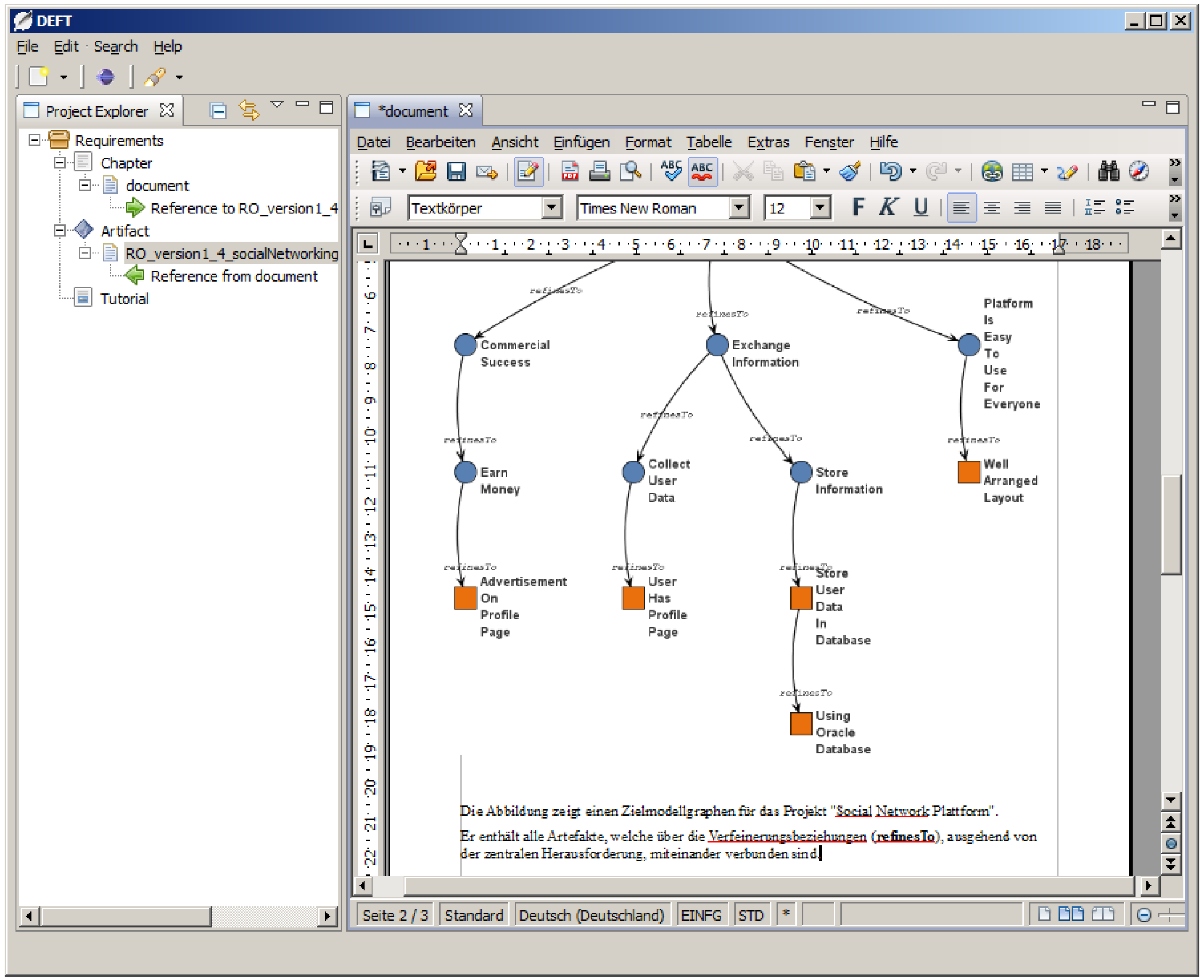
Traceability matrixes can easily become very large. The layout of large matrixes, or tables, is usually difficult. If the document format is page-based, for example Open Document Format (ODF), a table might not fit on a page. In that case, a traceability matrix must be split into multiple matrixes. This could be achieved by a CDF that comprises several tables. The corresponding operation would have to take layout information into account, such as the available page width, font sizes, or table cell margins.



Die folgende Matrix zeigt einen Überblick, welche Ziele wie mit einer Anforderung verbunden sind.

	DevelopSocialNetworkingPlatform	ExchangeInformation	FastAccessToPlatform	PlatformIsEasyToUseForEveryone
AccessibleToBlindUsers				
IntegratedInstantMessaging		isPositiveContributionTo		
UserCanHideData				
UserHasPhotoAlbum		isPositiveContributionTo		
UserHasProfilePage				

Figure 9.4: A generated traceability matrix in DEFT.



Die Abbildung zeigt einen Zielmodellgraphen für das Projekt "Social Network Plattform".
Er enthält alle Artefakte, welche über die Verfeinerungsbeziehungen (**refinesTo**), ausgehend von der zentralen Herausforderung, miteinander verbunden sind.

Figure 9.5: A generated requirements hierarchy tree in DEFT.

9.3.2 Documenting a BPMN Refinement Library

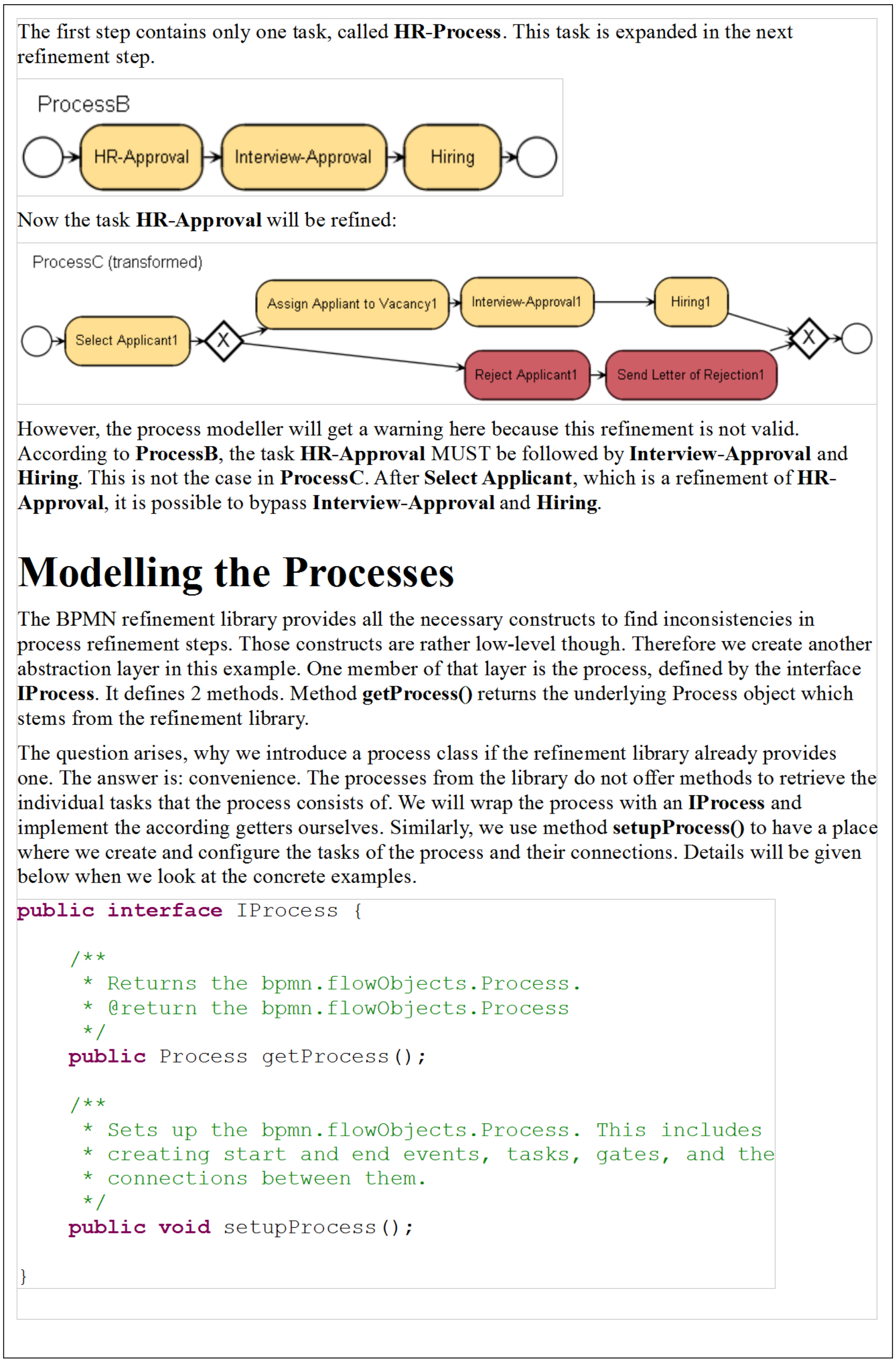
Business processes can be very complex. Often multiple people are involved in the definition of a business process, working at different levels of detail. A common technique to transform an abstract process into a more concrete one is *refinement* [8, 45].

Business process refinement is mostly a manual task. However, there are many pitfalls when refining a process, and tool support which helps validating a refinement is crucial. In the context of the Marrying Ontology and Software Technology (MOST) project[10], a BPMN refinement library was developed, which can check, whether a BPMN process is a valid refinement of another process. The refinement library is a Java library that allows for the specification of two processes and a mapping between them, written in Java code. Additionally, it has an evaluation engine that finds and reports refinement errors. For debugging purposes, the refinement library outputs the BPMN models and their mapping as *dot* graph description files[11].

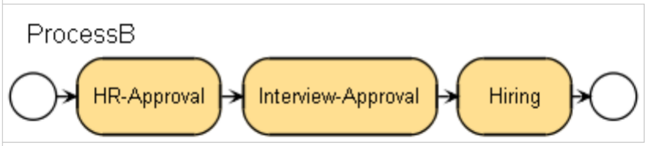
We wrote an example-based introductory documentation for the refinement library with DEFT. In the tutorial, we showed how to model the processes and the mappings in Java and how to invoke the evaluation engine. The tutorial contained mainly CDFs with code listings. In the tutorial introduction, we motivated the use of the refinement library with a refinement scenario. This scenario was illustrated by CDFs, which showed processes graphically. These image CDFs were computed from the above-mentioned debug dot files. Figure 9.6 shows an excerpt of the final documentation.

[10]An archived copy of the project homepage can be found at `http://web.archive.org/web/20120130051840/http://most-project.eu/about.php`

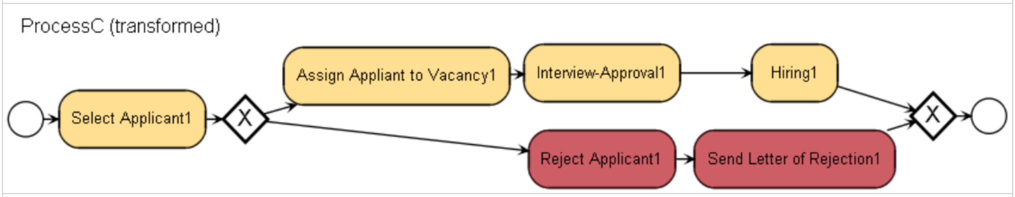
[11]`http://www.graphviz.org`



The first step contains only one task, called **HR-Process**. This task is expanded in the next refinement step.



Now the task **HR-Approval** will be refined:



However, the process modeller will get a warning here because this refinement is not valid. According to **ProcessB**, the task **HR-Approval** MUST be followed by **Interview-Approval** and **Hiring**. This is not the case in **ProcessC**. After **Select Applicant**, which is a refinement of **HR-Approval**, it is possible to bypass **Interview-Approval** and **Hiring**.

Modelling the Processes

The BPMN refinement library provides all the necessary constructs to find inconsistencies in process refinement steps. Those constructs are rather low-level though. Therefore we create another abstraction layer in this example. One member of that layer is the process, defined by the interface **IProcess**. It defines 2 methods. Method **getProcess()** returns the underlying Process object which stems from the refinement library.

The question arises, why we introduce a process class if the refinement library already provides one. The answer is: convenience. The processes from the library do not offer methods to retrieve the individual tasks that the process consists of. We will wrap the process with an **IProcess** and implement the according getters ourselves. Similarly, we use method **setupProcess()** to have a place where we create and configure the tasks of the process and their connections. Details will be given below when we look at the concrete examples.

```
public interface IProcess {

    /**
     * Returns the bpmn.flowObjects.Process.
     * @return the bpmn.flowObjects.Process
     */
    public Process getProcess();

    /**
     * Sets up the bpmn.flowObjects.Process. This includes
     * creating start and end events, tasks, gates, and the
     * connections between them.
     */
    public void setupProcess();

}
```

Figure 9.6: Excerpt of the BPMN Refinement Library tutorial.

We did not encounter any problems or unforeseen issues during the creation of the tutorial. After we finished writing the tutorial, we examined, whether the guidance works for both the code and the image CDFs. We renamed a state in one of the processes. Then, we invoked the evaluation engine on the modified process in order to have the dot files updated. The guidance system of DEFT worked as expected. The modified code file and all dot files which contained the renamed state were reported by the guidance system. We updated the artefacts, which caused the affected CDFs to be recomputed. All CDFs, both code listings and images, which somehow displayed the renamed state, were correctly updated.

9.3.3 Writing a PhD thesis about Elucidative Development with DEFT

The goal of all evaluations presented before was to demonstrate the applicability and versatility of ED. The quality and usability of DEFT, our EDE implementation, was not examined. Therefore, we chose to write this thesis in DEFT. We wanted to discover how well DEFT is suited for the creation of an extensive document, which is spread across multiple files. Since the thesis should be written in $\LaTeX$, we configured DEFT to use $\LaTeX$.

The thesis is not a document that describes a software system. Therefore, it contains only few active references. We chose to create the example Figures 4.1 and 4.13, which show excerpts of an elucidative document, with real active references and CDFs. All identifiers from the figures (framed by a black box) are CDFs. We did not generate the diagram images of the examples for the reasons explained in Sect. 9.1.4.

Additionally, CDFs appear at the following locations:

- as identifiers in Sect. 4.5.5
- as attributes and references listing for artefact metaclass (Sect. 4.3.1) and active reference metaclass (Sect. 4.3.2)
- as identifiers in the description below the attributes and references listing for artefact metaclass
- as attributes and references listing for active reference group metaclass (Sect. 5.3.2), static reference group metaclass (Sect. 5.3.3) and dynamic reference group metaclass (Sect. 5.3.4)
- as identifiers in Sect. 5.3.4

In summary, this thesis contains 35 references and CDFs. All CDFs in DEFT are cached (see Sect. 7.3). Listing 9.1 shows generated LaTeX CDFs inside manually written text.

```
The colour of the circles can be changed.
The \fbox{\deftreferenceecore{ref-R-4YAALOIotr}{Circle}} class
has a \fbox{\deftreferenceecore{ref-Tck-gALOEeOQ}{color}}
attribute (see~Fig.~II), which can be set.
After the colour has been set, the circle can be drawn.
```

Listing 9.1: Cached LaTeX CDFs.

The writing of the thesis with DEFT went smoothly. The LaTeX mode of DEFT is based on *TeXlipse*[12] and is suited to write extensive documents.

As a result of the evaluation, we identified several possibilities for improvement. Some were programming errors, which could be corrected immediately. Others turned out to be more extensive. They must be addressed before DEFT can be used productively beyond academic evaluations.

- The management of document files is still immature. It is possible to sort document files in folders and to move files and folders via drag and drop. However, inclusions such as the LaTeX directive `\input{evaluation/feasibilitystudies}` are not yet automatically updated when the referenced files are moved.
- It is not sufficient to present guidance hints as a simple list. The author must have the possibility to filter and group guidance hints.
- It was not possible to move references or CDFs from one document file into another. CDFs are not recognised by DEFT any more if they are removed from the document file to which they were originally added. This issue motivated the writing of Sect. 7.1.
- DEFT was implemented using the deferred update approach, which was described in Sect. 7.3.2. The deferred update approach works well if there is only one document file. If there are multiple document files, like in the case of this thesis, not all CDFs are properly updated when an artefact change is applied (see Sect. 7.3.3). The instant update approach would be preferable.

None of the identified issues prevented the writing of the thesis with DEFT. At no time it was necessary to resort to other LaTeX editors.

[12]http://sourceforge.net/projects/texlipse/

9.4 Conclusion

In this chapter, we presented the experiments that we performed to evaluate the ED approach and our EDE DEFT. First, we described how we turned a manually written model specification into an elucidative document and the errors that the we found in the process. We also measured, how well ED performs during an automatic update of the specification.

Then, we presented a similar scenario, using the UML specification. We turned a section of the UML 2.0 specification into an elucidative document and updated it to UML 2.3. The underlying artefact for this experiment was an Ecore implementation of the UML metamodel. We listed the errors in the analysed sections of the official UML specifications and stated that ED can prevent them.

Finally, we showed the versatility of ED by introducing a number of elucidative documents that we wrote with DEFT. This included requirements documents, where the requirements were formalised by an ontology, a programming library tutorial, where source code was described, and this thesis.

Chapter 10

Conclusion

We started this thesis by highlighting that redundancy is an integral part of software development. While redundancy is not a bad thing per se, it is the basis for inconsistency. Resolving inconsistencies is a necessary, but time-consuming and unpopular part of software development. In the thesis, we focused on inconsistencies between documents and other views of a software system, and how to minimise or resolve them.

Our main contribution was the introduction of Elucidative Development (ED) as a possible solution to the inconsistency problem. ED is an approach to create and maintain so-called *elucidative documents* by partial generation. The main idea is to generate as much document content as desired and write the remaining content manually. Content can be generated from various views of the software system, such as code or models. If the document becomes inconsistent because the views have been changed, the generated parts of the document can be regenerated. The manually written content is not affected by the regeneration and remains intact.

This contribution can be divided into several sub-contributions.

- In Sect. 2.2 we defined the formality of documents. Later, we showed that the concrete requirements of ED depend on the formality of the document to be created.

- In Chap. 4, we introduced ED for semi-formal documents. We presented important challenges that must be solved and turned them into requirements for ED. These requirements were the foundation for the explanation of:

  - the concepts that allow for the coexistence of generated and manually written content in one document,

  - the presentation layer, which hides technical details of the above-mentioned concepts from the document author,
  - a guidance system, which informs the author about actual and potential inconsistencies.

- In Chap. 5, we extended ED for use with formal documents. We call this extension *model-driven elucidative development.* Formal documents have a uniform structure, which reflects the structure of the described model. Thus, there are more possibilities for generation than in semi-formal documents. Again, we identified several challenges and requirements. Based on those, we explained the nesting and the automatic addition, removal, renaming and moving of generated content. The guidance system has also been extended.

- In Chap. 6, we presented two extensions of ED, which provide additional value. In the first extension, we examined how elucidative documents can be structurally validated. We started by explaining how normal, tree structured documents are validated against a tree grammar. Then, we modified these concepts for elucidative documents.

  In the second extension, we showed that ED can be combined with backpropagation-based round-trip engineering. This allows the modification of generated document content and the propagation of these modifications to the underlying software artefacts. We explained the concepts of backpropagation-based round-trip engineering, aligned them to ED, and presented an example to illustrate the applicability.

Our second big contribution was the evaluation of ED and the confirmation of its versatility. In Chap. 9, we described our evaluation and its results.

We performed two case studies. In Sect. 9.1, we described the transition from a manually maintained model specification to ED and showed that the quality of the specification had improved. In Sect. 9.2, we showed that it is even possible to use ED for the Unified Modeling Language (UML) specification. ED would ease the maintenance of the specification and resolve many of its inconsistencies.

After the case studies, we described several feasibility studies in Sect. 9.3. They show that ED can be applied to a wide range of documents, which can describe a wide variety of software views.

Our third big contribution was the comparison of ED with related work from different fields. The comparison shows, on which existing ideas our work is based, and how we progressed beyond them.

Additionally, there were two minor contributions worth mentioning.

- The concepts of active documents and transconsistency were not sufficient for ED. Therefore, we extended them and introduced *weak transconsistency* in Sect. 4.4.3.
- In Chap. 7, we examined possible implementations of tool support for ED. There are usually multiple possibilities to achieve a certain goal, so we often presented alternative solutions and compared their advantages and disadvantages.

Appendix A

Cool Component Specification

Here we present data from the experiment with the Cool Component Model (CCM) specification described in Sect. 9.1. This includes the steps we performed to create the first version of the metamodel, the errors we found in the specification, and the changes that were automatically performed when we updated the specification.

All revisions of the specification and the metamodel were retrieved from the Cool Software Subversion (SVN) repository. Since the Cool Software SVN is a project internal repository, the revisions relevant for this evaluation have been made available for download from `http://bartho.net/phdthesis`.

A.1 Metamodel Preparation

The starting point of the evaluation was the version of the specification from November 1, 2011. We needed both the metamodel and the LaTeX source of the specification. The LaTeX source code was available in the Cool Software repository (revision 806). The Cool Software metamodel was also taken from the repository.

Unfortunately, the specification document did not cover one single SVN revision of the metamodel, but rather a range of revisions. For example, the specification described for class `CoolEnvironment` an association called `behavior`, but this association had already been removed from the metamodel in SVN revision 761. On the other hand, the specification did not mention a class `IdentifyableElement`, but the class had been present in the metamodel starting from SVN revision 565. We believe this is due to oversights during the manual evolution of the specification.

Lacking an exact revision of the metamodel, we used the same revision as the specification (revision 806). We then added missing elements from older metamodel revisions and deleted superfluous elements, so the structure of the metamodel was as close to the specification as possible. This caused the structure of the Elucidative Development (ED) specification to be the same as the structure of the original specification, resulting in more subsections that we could compare with each other. Finally, we added the attribute and association documentation text from the specification directly to the metamodel via documentation annotations[1]. This allowed us to have attribute and association listings generated together with their documentation. Additionally, we believe that this is the best place to store the documentation.

A.2 Inconsistencies in the Manual Specification

In the following, we explain the expected content of the specification and the deviations we found.

A.2.1 Convention Errors in Running Text Cross-References

Each CCM metaclass is described in detail in a dedicated subsection. These subsections have an introductory text and a formal listing of the metaclass properties (called metaclass characteristics in the specification), such as a list of superclasses or a list of attributes. In many cases, the introductory text mentions other metaclasses in order to explain their interaction. When a metaclass is mentioned for the first time in such an introductory text, it has a cross-reference to its own defining subsection.

The metaclasses are grouped in packages. All metaclasses of one package are described in the same section. Each section has introductory text, too. It describes the aim of the package and the interaction of its classes on a more abstract level. In contrast to the subsections, the section text does not contain cross-references.

In the manually written specification we found cases where the expected cross-references were not provided in the subsections. We also found cases where the section text contained cross-references, even though they should not be there.

[1]We used the tool EMFDoc (`http://reuseware.org/index.php/EMFText_Concrete_Syntax_Zoo_EMFDoc`).

Expected cross-references	121
Correct cross-references	107
Missing cross-references	14
Unexpected cross-references	9

The existing (and expected) cross-references have the form "(Sect. x.y.z)", where x.y.z is the referenced subsection. Some cross-references have a wrong formatting, for example "(see Sect. x.y.z)" or "(sect. x.y.z)".

Cross-references with incorrect formatting	10 of 107

A.2.2 Convention Errors in Metaclass Characteristics Listings

There are 52 metaclass characteristics listings. Each metaclass characteristics listing describes one metaclass formally. We identified the following convention errors.

Cross-references with incorrect formatting	5 of 94
Metaclass characteristics heading missing	8 of 52
Inherited from has multiple entries, but is not printed as list	1 of 1
Known subclasses has multiple entries, but is not printed as list	1 of 12
Known subclasses has only one entry, but is printed as list	1 of 1
Known subclasses is empty and says *none* instead of —	9 of 13

A.2.3 Misspelled and Wrong Identifiers

Both running text and metaclass characteristics listings contain wrong or misspelled identifiers. Misspelled identifiers are identifiers with a typing error, e.g. `Transistion` instead of `Transition`. Wrong identifiers are identifiers which have no typing errors, but which do not exist in the metamodel.

Misspelled identifiers in running text	7 of 441
Wrong identifiers in running text	6 of 441
Misspelled identifiers in metaclass characteristics listings	1 of 176
Wrong identifiers in metaclass characteristics listings	7 of 176

A.2.4 Wrong Metaclass Properties

Each metaclass has various properties, such as a list of superclasses and subclasses, attributes with type and multiplicity, and others. Missing or wrong properties are a serious problem and a severe error.

Wrong attribute or association multiplicity	15 of 82
Missing attribute or association	6 of 82
Missing or wrong subclasses	9 of 48
Wrong superclasses	5 of 46
Wrong *Abstract class* definition	3 of 52
Normal association defined as containment	1 of 62
Association defined as attribute	1 of 62

A.2.5 Wrong Model Diagrams

5 of 13 diagrams contain one or two errors. These errors comprise a missing class, a missing attribute and the use of a meanwhile changed class name.

A.3 Update of the Specification

The artefacts in the Development Environment For Tutorials (DEFT) repository (metamodel and diagram files) have been updated to the latest revision from the SVN repository (revision 2367). The metamodel changes from SVN revision 806 to revision 2367 were rather big. The metamodel has evolved considerably and has undergone a comprehensive refactoring. The `datatypes` package and the `expressions` package have been completely redesigned. Therefore, we decided to abandon the corresponding subsections in the specification and recreate them from scratch. This resulted in the deletion of 18 subsections with metaclass descriptions.

The remaining parts of the specification were surprisingly stable. Besides the metaclasses in the redesigned packages, 2 metaclasses have been deleted. Their corresponding subsections have been removed from the specification. Throughout the whole specification, there were only 9 identifiers in the running text which referred to one of the 20 deleted metaclass subsections.

The new metamodel contained 48 changes in 17 metaclasses. Those changes were, for example, different multiplicities, added or removed attributes and associations, or changes in the class hierarchy. The changes were automatically identified and reported by DEFT. We accepted the artefact changes and they were applied to Computed Document Fragments (CDFs) of the specification. All diagram images were automatically updated, too.

Appendix B

UML Specification

Here, we present data from the experiment with the Unified Modeling Language (UML) specification described in Sect. 9.2. This includes the changes we made to the Ecore-based UML metamodel and the Object Constraint Language (OCL) code.

We used the Ecore-based UML 2.3 metamodel from the Eclipse Modelling Tools as artefact for Development Environment For Tutorials (DEFT). However, the metamodel and the specification did not completely match. We did not investigate the reasons for this mismatch. As our goal was to reproduce the contents of the specification with Elucidative Development (ED), we modified the `Package` class of the metamodel so that it matched the specification. From the metamodel we created a UML 2.0 metamodel afterwards. The necessary steps were also performed manually. All changes we made are listed in this appendix.

In contrast to the metamodel, the errors in the OCL constraints and operations have been fixed because otherwise it would not have been possible to parse and import them into DEFT. The errors have been fixed by Claas Wilke for [61].

B.1 Metamodel Changes

In order to make the `Package` metaclass of the Ecore-based UML 2.3 metamodel match the specification, we changed the following:

- Removed superclass `TemplateableElement`.
- Removed reference `profileApplication`.
- Changed the documentation text of reference `nestedPackage`.

- Changed the type of reference `packageMerge` from `PackageMerge` to `Package`.
- Marked reference `nestingPackage` as not derived.
- Marked reference `packagedElement` as derived.

After the content of the `Package` metaclass matched the specification, we created a copy of the metamodel and changed it to receive a metamodel which corresponds to the UML 2.0 specification. We changed the following:

- Renamed reference `packagedElement` to `ownedMember`.
- Changed property `subsets` of reference `ownedMember` to `redefines`.
- Changed the documentation text of reference `ownedMember`.
- Marked reference `ownedMember` as not derived.
- Changed the documentation text of reference `ownedType`.
- Marked reference `ownedType` as not derived.
- Added reference `package`.

B.2 OCL Changes

First, we present the OCL constraints and operations as they are printed in the UML 2.0 specification. We list the errors and say how they must be fixed. Afterwards, we list the changes that must be made for the transition to the UML 2.3 specification.

B.2.1 OCL errors

Constraint

```
self.ownedElements->forAll(e | e.visibility->notEmpty()
    implies e.visbility = #public
    or e.visibility = #private)
```

The constraint contains these errors:

- The association `ownedElements` does not exist, it should be `ownedMember` instead.
- In one place the code says `visbility` instead of `visibility`.
- The literals `#public` and `#private` are not allowed in OCL 2.0 and above. Visibilities must be expressed as `VisibilityKind::public` and `VisibilityKind::private`.

Operation mustBeOwned

```
Package::mustBeOwned() : Boolean
mustBeOwned = false
```

The operation contains these errors:

- The body starts with `mustBeOwned = ...`, but it must start with `body : ...` instead.

Operation visibleMembers

```
Package::visibleMembers() : Set(PackageableElement);
visibleMembers = member->select( m | self.makesVisible(m))
```

The operation contains these errors:

- The context declaration ends with a semicolon.
- The body starts with `visibleMembers = ...`, but it must start with `body : ...` instead.
- The association `member` requires either a cast to `PackageableElement` or must be changed to `ownedMember`.

Operation makesVisible

```
Package::makesVisible(el: Namespaces::NamedElement) : Boolean;
pre: self.member->includes(el)
makesVisible =
-- case: the element is in the package itself
(ownedMember
    ->includes(el)
) or
-- case: it is imported individually with public visibility
(elementImport
    ->select(ei|ei.importedElement = #public)
    ->collect(ei|ei.importedElement)
    ->includes(el)
) or
-- case: it is imported in a package with public visibility
(packageImport
    ->select(pi|pi.visibility = #public)
    ->collect(pi|pi.importedPackage.member->includes(el))
    ->notEmpty()
)
```

The operation contains these errors:

- The context declaration ends with a semicolon.
- The body starts with `makesVisible = ...`, but it must start with `body : ...` instead.
- The literal `#public` is not allowed. Visibilities must be expressed as `VisibilityKind::public`.
- In case 1, it says `ownedMember->includes(el)`, but it must say `ownedMember->collect(oclAsType(NamedElement))->includes(el)`
- In case 2, it says `ei.importedElement = ...`, but it must say `ei.importedElement.visibility = ...`.
- In case 2, it says `collect(ei|ei.importedElement)->includes(el)`, but it must say `collect(ei|ei.importedElement)->collect(oclAsType(NamedElement))->includes(el)`.

B.2.2 Changes for UML 2.3

The following changes were made to create UML 2.3 OCL code from the fixed UML 2.0 OCL code:

- All references to `ownedMember` have been changed to `packagedElement`.
- Removed `->collect(oclAsType(NamedElement))` from operation `makesVisible` in case 1.

Bibliography

[1] Michal Antkiewicz and Krzysztof Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. In *MoDELS*, pages 692–706, 2006.

[2] Jim Arlow, Wolfgang Emmerich, and John Quinn. Literate Modelling – Capturing Business Knowledge with the UML. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language. UML'98: Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 189–199. Springer Berlin / Heidelberg, 1999.

[3] Uwe Aßmann. Architectural styles for active documents. *Science of Computer Programming*, 56(1-2):79–98, 2005.

[4] Uwe Aßmann, Andreas Bartho, Christoff Bürger, Sebastian Cech, Birgit Demuth, Florian Heidenreich, Jendrik Johannes, Sven Karol, Jan Polowinski, Jan Reimann, Julia Schroeter, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. DropsBox: the Dresden Open Software Toolbox. *Software & Systems Modeling*, pages 1–37, 2012.

[5] Uwe Aßmann, Andreas Bartho, Falk Hartmann, Ilie Savga, and Barbara Wittek. Trustworthy Instantiation of Frameworks. In Ralf H. Reussner, Judith A. Stafford, and Clemens A. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, pages 152–168. Springer Berlin Heidelberg, 2006.

[6] Lars Bak, Jørgen Lindskov Knudsen, Ole Lehrmann Madsen, Claus Nørgaard, and Elmer Sandvad. An overview of the Mjølner BETA system. Technical report, Aarhus University, Computer Science Department, 1991.

[7] Andreas Bartho. Creating and maintaining tutorials with DEFT. In *ICPC*, pages 309–310, 2009.

[8] Andreas Bartho, Gerd Gröner, Tirdad Rahmani, Yuting Zhao, and Srdjan Zivkovic. Guidance in Business Process Modelling. In Schahram Dustdar and Fei Li, editors, *Service Engineering*, pages 201–231. Springer, 2011.

[9] Hanna Bauerdick, Martin Gogolla, and Fabian Gutsche. Detecting OCL Traps in the UML 2.0 Superstructure: An Experience Report. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen Mellor, editors, *UML 2004 - The Unified Modeling Language. Modelling Languages and Applications*, volume 3273 of *Lecture Notes in Computer Science*, pages 188–196. Springer Berlin / Heidelberg, 2004.

[10] Tom Beger. Visualisierung von Softwareanforderungen. Master's thesis, Technische Universität Dresden, 2011.

[11] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational Lenses: A Language for Updatable Views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '06, pages 338–347. ACM, 2006.

[12] M. Brown and B. Czejdo. A Hypertext for Literate Programming. In S. Akl, F. Fiala, and W. Koczkodaj, editors, *Advances in Computing and Information – ICCI '90*, volume 468 of *Lecture Notes in Computer Science*, pages 250–259. Springer Berlin / Heidelberg, 1990.

[13] Jürgen Buchner. HotDoc: A Framework for Compound Documents. *ACM COMPUTING SURVEYS*, 32, 2000.

[14] Mikhail Chalabine and Christoph Kessler. A Formal Framework for Automated Round-Trip Software Engineering in Static Aspect Weaving and Transformations. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 137–146, 2007.

[15] David Chappell. *ActiveX und OLE verstehen.* Microsoft Press Deutschland, 1996.

[16] David Chappell. *Understanding ActiveX and OLE.* Microsoft Press, 1996.

[17] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2008.

[18] Umeshwar Dayal and Philip A. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

[19] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In *1st Workshop on Model Driven Interoperability*, pages 42–51. ACM Press, 10/2010 2010.

[20] Alexander Franz Egyed. *Heterogeneous View Integration and its Automation.* PhD thesis, University of Southern California, 2000.

[21] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, DocEng '02, pages 26–33, New York, NY, USA, 2002. ACM.

[22] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 233–246. ACM, 2005.

[23] Joel Greenyer and Ekkart Kindler. Reconciling TGGs with QVT. In Gregor Engels, Bill Opdyke, DouglasC. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2007.

[24] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009)*, volume 5562 of *LNCS*, pages 114–129. Springer, 2009.

[25] Jakob Henriksson. *A Lightweight Framework for Universal Fragment Composition - with an application in the Semantic Web.* PhD thesis, Technische Universität Dresden, 2008.

[26] Andreas Kacofegitis. Theme-Based Literate Programming. Master's thesis, University of Canterbury, 2002.

[27] Andreas Kacofegitis and Neville Churcher. Theme-Based Literate Programming. *Asia-Pacific Software Engineering Conference*, 0:549, 2002.

[28] Arthur Michael Keller. *Updating Relational Databases Through Views.* PhD thesis, Stanford University, 1995.

[29] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. An Adaptable TGG Interpreter for In-Memory Model Transformation. In *Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany*, 2004.

[30] Ekkart Kindler and Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[31] Donald E. Knuth. Literate Programming. In *The Computer Journal*, volume 27(2), pages 97–111, May 1984.

[32] Josef Kolbitsch and Hermann Maurer. Transclusions in an HTML-Based Environment. In *Journal of Computing and Information Technology (CIT)*, volume 14, pages 161–174, 6 2006.

[33] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different Models for Model Matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*, pages 1–6. IEEE, 2009.

[34] Axel Lamsweerde. Reasoning About Alternative Requirements Options. In Alexander T. Borgida, Vinay K. Chaudhri, Paolo Giorgini, and Eric S. Yu, editors, *Conceptual Modeling: Foundations and Applications*, pages 380–397. Springer-Verlag, 2009.

[35] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, 2001.

[36] Theodor Holm Nelson. *Literary Machines.* Mindful Press, 3rd edition, 1993.

[37] Theodor Holm Nelson, Robert Adamson Smith, and Marlene Mallicoat. Back to the future: hypertext the way it used to be. In *Proceedings of the eighteenth conference on Hypertext and hypermedia*, HT '07, pages 227–228. ACM, 2007.

[38] Kurt Nørmark. Elucidative Programming. *Nordic J. of Computing*, 7:87–105, June 2000.

[39] Kurt Nørmark. Requirements for an Elucidative Programming Environment. In *Eight International Workshop on Program Comprehension*, June 2000.

[40] Kasper Østerbye. Literate Smalltalk Programming Using Hypertext. *IEEE Transactions on Software Engineering*, 21:138–145, 1995.

[41] Sebastian Patschorke. Konzeption und Implementierung eines Tutorialwerkzeugs für Modelle. Master's thesis, TU Dresden, 2009.

[42] Norman Ramsey. Literate Programming Tools Need Not Be Complex. Technical Report CS-TR-351-91, Department of Computer Science, Princeton Univ., Princeton, NJ, 1991.

[43] Norman Ramsey. Literate Programming Simplified. *IEEE Software*, 11(5):97–105, 1994.

[44] T. Reenskaug and A. L. Skaar. An Environment for Literate Smalltalk Programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '89, pages 337–345, New York, NY, USA, 1989. ACM.

[45] Yuan Ren, Gerd Gröner, Jens Lemcke, Tirdad Rahmani, Andreas Friesen, Yuting Zhao, Jeff Z. Pan, and Steffen Staab. Validating Process Refinement with Ontologies. In *the Proc. of the 22nd International Workshop on Description Logics (DL2009)*, 2009.

[46] Gunnar Schulze. Synchronization of UML Models and Narrative Text using Model Constraints and Natural Language Processing. Master's thesis, University of Innsbruck, 2011.

[47] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In ErnstW. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1995.

[48] Mirko Seifert. *Designing Round-Trip Systems by Change Propagation and Model Partitioning.* PhD thesis, Dresden University of Technology, 2011.

[49] Katja Siegemund, Edward J. Thomas, Yuting Zhao, Jeff Z. Pan, and Uwe Aßmann. Towards Ontology-driven Requirements Engineering. In *The 10th International Semantic Web Conference (ISWC2011)*, 2011.

[50] Katja Siegemund, Yuting Zhao, Jeff Z. Pan, and Uwe Aßmann. Measure Software Requirement Specifications by Ontology Reasoning. In *Proceedings of the 8th International Workshop on Semantic Web Enabled Software Engineering (SWESE 2012)*, 2012.

[51] Alexandre Valente Sousa. Literate Programming in an industrial setting. Technical report, Instituto Superior da Maia, 2005.

[52] George Spanoudakis and Andrea Zisman. Inconsistency Management in Software Engineering: Survey and Open Rresearch Issues. In *in Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific, 2001.

[53] Thomas Stahl and Markus Völter. *Model-Driven Software Development.* John Wiley & Sons, Ltd, 2003.

[54] Thomas Vestdam. Elucidative program tutorials. *Nordic J. of Computing*, 9:209–230, September 2002.

[55] Thomas Vestdam. Generating Consistent Program Tutorials, 2002.

[56] Thomas Vestdam. *Elucidative Programming – Tools, Patterns, and Experiments.* PhD thesis, Aalborg University, 2004.

[57] Thomas Vestdam and Kurt Nørmark. Aspects of Internal Program Documentation – an Elucidative Perspective. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 43–52, 2002.

[58] Thomas Vestdam and Kurt Nørmark. Toward Documentation of Program Evolution. *IEEE International Conference on Software Maintenance*, pages 505–514, 2005.

[59] Claas Wilke, Andreas Bartho, Julia Schroeter, Sven Karol, and Uwe Aßmann. Elucidative Development for Model-Based Documentation. In *TOOLS (50)*, pages 320–335, 2012.

[60] Claas Wilke, Andreas Bartho, Julia Schroeter, Sven Karol, and Uwe Aßmann. Extended Version of Elucidative Development for Model-Based Documentation and Language Specification. Technical Report TUD-FI12-01-Januar 2012, TU Dresden, 2012.

[61] Claas Wilke and Birgit Demuth. UML is still inconsistent! How to improve OCL Constraints in the UML 2.3 Superstructure. *ECEASST*, 44, 2011.