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

Bibliographic Information published by the Deutsche Bibliothek
The Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibiograpic data is available in the internet at http://dnb.ddb.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 practise, 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 Consequently, documents are often outdated. Outdated documents communicate wrong information about the software.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 practise, 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

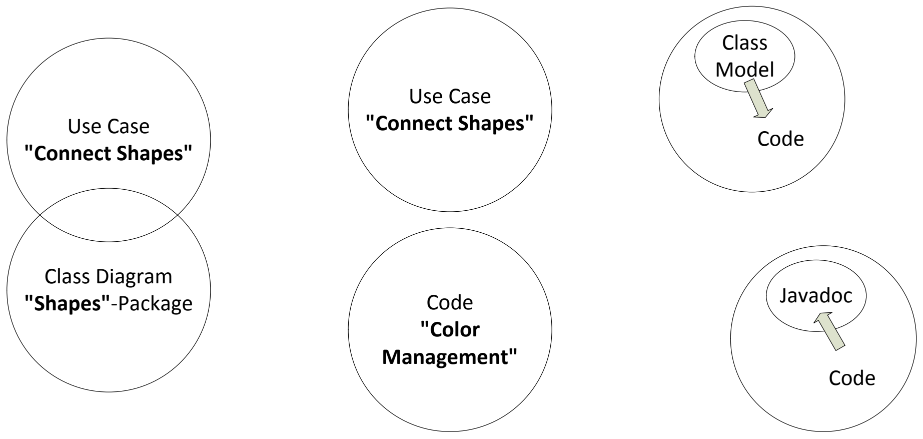
In this chapter, we explain the connection between redundancy and inconsistency, and show how generation (i.e., automatic document creation from existing data) improves consistency between documents and other parts of a software system. Since complete generation is not always possible, we present the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

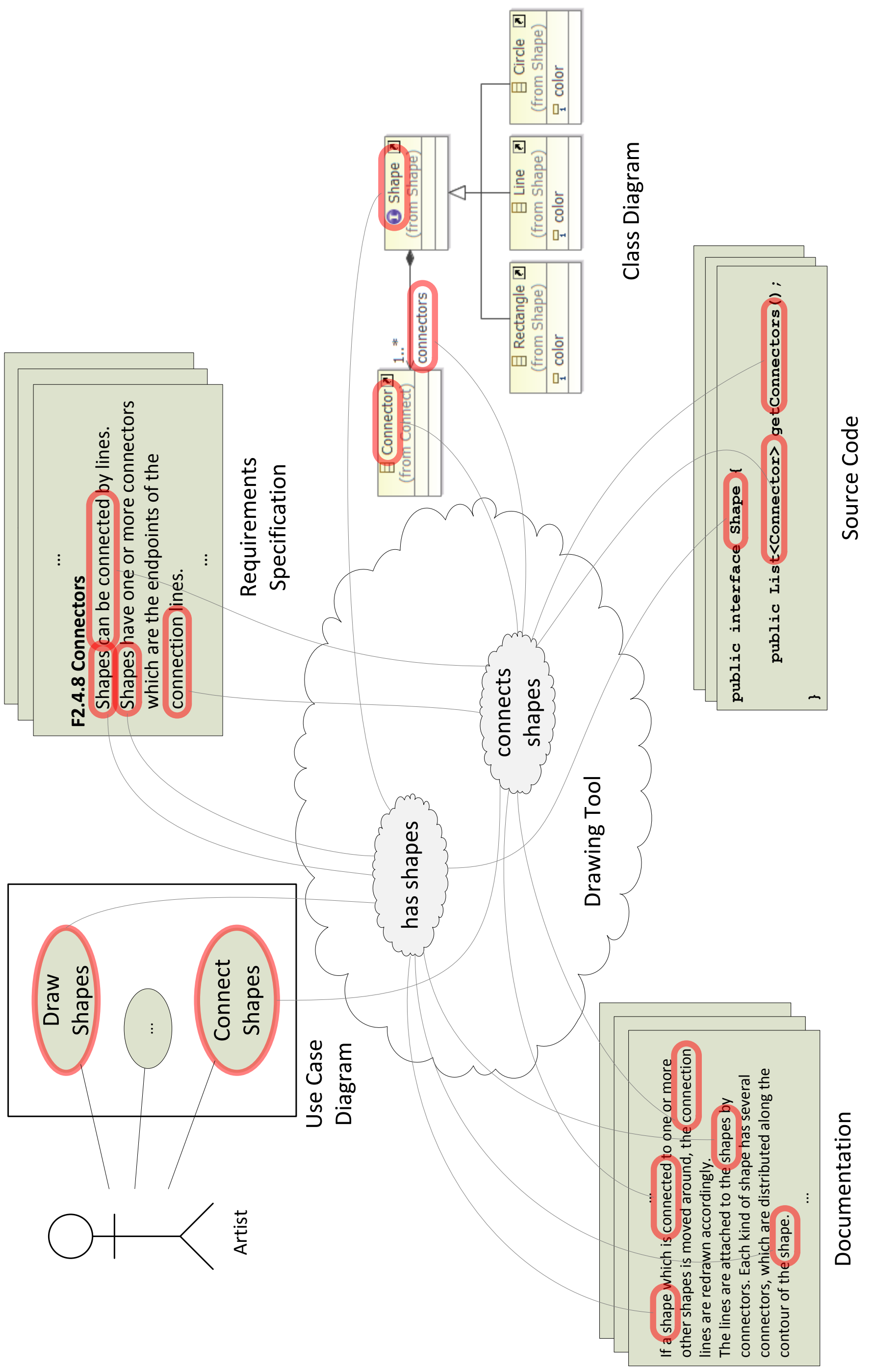


(a) Partial overlap. (b) No overlap. (c) Full overlap.

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 [3]. 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.

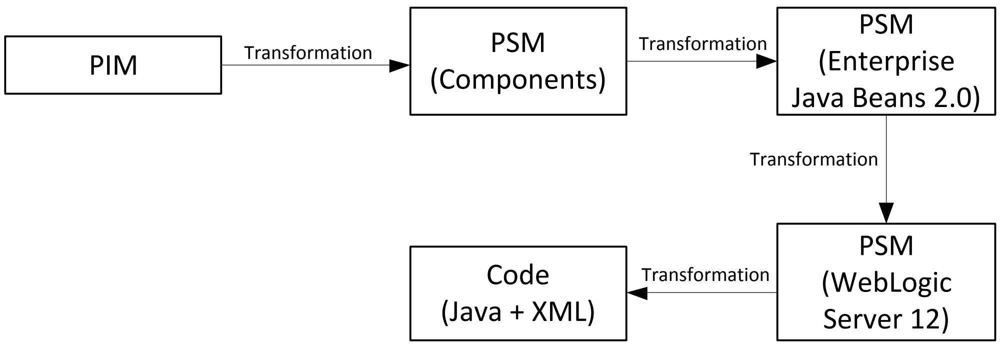


Figure 3.1: Model transformations.

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 [4], "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.

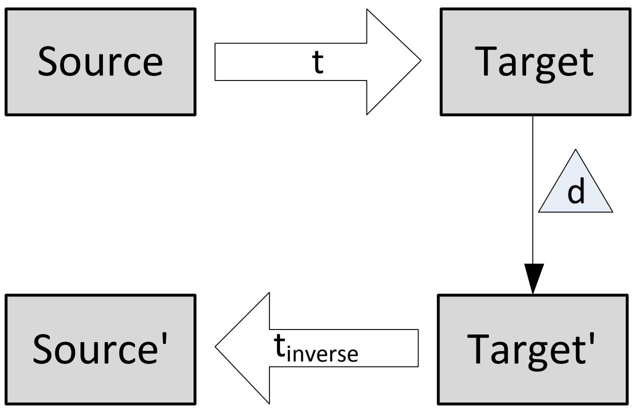


Figure 3.2: Round-trip via inverse transformation.

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)* [46]. 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

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

system, but also concrete technical realisations. Transformations are used to transform domain-specific models to more technical models and source code.

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.