universität wien

# Masterarbeit

Titel der Arbeit

## Guidelines for the Design and Implementation of Projectional Editors for Domain Specific Languages

verfasst von
Daniel Tieber, BSc

angestrebter akademischer Grad
Diplom Ingenieur (Dipl.-Ing.)

Wien, 2015

Studienrichtung lt. Studienblatt: Wirtschaftsinformatik
Studienkennzahl lt. Studienblatt: 066 926
Betreuer: Univ.-Prof. Dr. Uwe Zdun

# Content

*dedicated to my mother*

**Acknowledgements**

At this point I would like to express my gratitude to my advisor Prof. Uwe Zdun for the excellent and continuous supervision through the creational process of this master thesis. His guidance helped me throughout researching and writing this thesis.

Furthermore I would like to thank my friend Thomas Hochgatterer for his almost inexhaustible patience while reading and discussing this thesis with me as well as all the inspiring discussions throughout my studies. Equally important I would like to thank my friend and fellow student Benedikt Pittl for his support during my studies and especially for his review on this thesis.

Last but not least I would like to thank my loved ones, who have supported me throughout the entire process, especially my girlfriend Nadine, by keeping me harmonious and helping me putting pieces together. Without you, believing in me, this would have been so much harder.

# 1. Introduction

The predominant approach to software development and maintenance is to write source code line by line within a text editor. Since information systems become more and more important in everyday reality, the number of people participating in software development processes will increase steadily. Moreover, software becomes ever more complex, which is one of the various reasons for the success of frameworks and emerging languages, both for general or domain specific purpose. Therefore more convenient ways of writing and maintaining code could play a vital part in prospective software engineering. A promising approach to cope with these challenges may be seen in projectional editing, which intends to adapt principles of model driven development to the approach of editing code. In contrast to the classical approach, source code editing, the logic of a computer program is not edited directly, but rather via projections that focus on a particular viewpoint of the software, projected from the abstract syntax tree of the internal representation of an application. Although the idea of projectional editing is not a novel one, open questions in regard to the concrete implementation of such projectional editors are still remaining. The circumstance that all enterprises behind market-leading integrated development environments (like Eclipse or IntelliJ) work on projectional editors acknowledges the importance of this topic. This master thesis focuses on the implementation of projectional editors and provides the reader with an initial guidance model comprised of recurring design decisions including practicable alternatives and decision drivers.

## 1.1. Motivation

This master thesis aims to provide valuable scientific insights in the design of projectional editors with the aid of a guidance model, a meta-model for the architecture of projectional editors. This ought to be reached through a comprehensive literature research as well as the implementation of a prototype of a projectional editor, whereby the insights gained from both, the literature and during the iterative development cycles are used to create the intended guidance model.

In order to explain the relevance of this thesis, a look at the project reality and the way software projects are designed and implemented has to be taken. Since information technologies become increasingly important for almost every business, or above that take on a key role and therefore may represent a competitive advantage, domain specific knowledge is required continuously to maintain and adapt software. This is why domain experts are tightly integrated in the software development process, as visible in approaches like behavior driven development or the outplacement of business logic in business rules engines or workflow management systems. That implies that the infrastructure of software development environments has to change in a way that allows hassle-free participation of non-technicians. Projectional editors may be a step in the right direction, since different abstraction layers and therefore appropriate views are an obvious advantage in comparison to text editors. From another point of view, software developers could also profit from projectional editors, by organizing code in separated views and therefore reducing complexity when possible. Applications interact closely with each other and are increasingly integrated which leads to more complex architectures. Code separation and organization may already be stretched to their limits which is why

new approaches to master the complexity of these projects have to be found.

## 1.2. Objective

The main purpose of this thesis is to create a guidance model for projectional editors, with the goal to provide software architects with a starting point when designing a projectional editor. Therefore the key issues occurring during the development process will be identified and appropriate options are going to be presented. Furthermore the mutual interdependence of decisions within the guidance model will be depicted and initialized. This initial guidance model is not entitled to be final and static, but should rather lead to further discussion and improvement.

## 1.3. Approach

In order to create the guidance model a two legged approach has been pursued. On the one hand relevant scientific literature and practice reports have been studied and analyzed to obtain a comprehensive overview of the underlying theory. With this in mind not only the theoretical background of projectional editing is of concern but also a profound understanding of guidance models which is why this approach is explained in detail in Chapter 2.3. On the other hand a prototype of a projectional editor has been implemented with the purpose of gaining experience in order to discover recurring design issues. Concretely, a web-based projectional editor for mathematical formulas has been designed and implemented, since this field of applications is comparably narrow, easy to understand and is suitable for useful projections. Both, theoretical and practical insights

aim to discover the most substantial issues when designing a projectional editor.

## 1.4. Structure of the thesis

This thesis is arranged in four parts. The first part covers the current state of research and consists of two subchapters; the first is about the conceptual background of projectional editors, whereas the second deals with guidance models as supportive artifacts acting in an advisory capacity for software architects. The second part explains all details of the prototype, with a primary focus on the architecture of the editor including emerging design decisions and chosen alternatives. In the third part a guidance model has been designed, followed by a discussion and a prospect for further research.

# 2. Related Work

The following chapter consists of two major blocks of information. The first is concerned with the characteristics as well as opportunities and weaknesses of projectional editing. It starts with a short review including the background followed by an evaluation and demarcation of both editing methods, source code editing and projectional editing. The second section covers background information regarding reusable architectural design decisions in form of a guidance model. A clear understanding of the proposed decision-modelling framework is inevitable, due to its central importance within the scope of this thesis, since it is the chosen framework to capture the findings of this work. Moreover the underlying approach of decision identification and recording used in this thesis is described.

### *2.1.1. Historical Background*

Since the beginning of software engineering, researchers and software developers try to create abstraction layers for software code [1] in order to create software in an easier, more convenient way. In this context the term easier addresses multiple characteristics of software development. On the one hand this targets general quality attributes of software code (e.g. readability, maintainability, complexity), on the other hand the before mentioned abstraction layers are aimed at reducing complexity for the software engineer. Since complexity and possibility space are mutually dependent in a way that with increased options comes increased complexity, several levels of abstraction are used in today's software development as seen in the different usage of languages (e.g. ISO C++ shields the majority of complexity of storage management in comparison to ANSI C) and

frameworks (object relational mapper help abstracting the persistence layer and provide an interface). What all these different abstraction approaches have in common is that they tend to support the process of translating the abstract ideas of a computer program to computer language which is eventually binary code. In the early 80s computer aided software engineering (CASE) [22] was the predominant technical term for this process, comprising different tools that try to “improve both quality and efficiency, resulting in a net improvement in maintenance and development productivity” [17]. Computer aided software engineering attracted considerable attention in research and professional literature, however it has never played a major role in practice. Computer aided software engineering had its origins in the analysis of graphical computer representations, more precisely its major goal was to reduce complexity of code in comparison to dominant general purpose languages back then. One of the major problems with CASE was, that it did not support concurrent development and in addition the solution space was narrowed during the continuous development of it, which lead to the drawback that it did not support many application domains in the end [2]. Yet other researches claimed that the CASE approach does not go far enough and that benefits arising from the automation of development do not justify the burden posed by the approach [18]. Computer aided software engineering reached its peak in the early 90s followed by the ultimate downfall which was accompanied by severe criticism by opponents of the approach, calling it a “disastrous infatuation” [2]. Despite all that, the desire for easier, more convenient way to develop code endured through the 90s [22], expressed by a vast number of emerging languages and frameworks, as for instance Java, which has five major goals; one of them clearly addressing the simplicity [19].

Another driver for the need of simple programming of applications was the growing demand for domain specific languages, which are more or less “[…] concise, processible languages for describing a specific concern when building a system in a specific domain. The abstractions and notations used are tailored to the stakeholders who specify the particular concern“[3]. Domain specific languages not only aid software engineers developing software easier [22], but also enable non-technicians of a particular domain to easily participate in the construction process of software [3]. For instance business rules engines and behavioral driven development became increasingly popular, since domain experts can communicate in simple manner with software experts in order to improve the quality and essentially the accuracy of a computer aided problem solution. Once the classical waterfall model fell into disuse more agile approaches like SCRUM or XP [20, 21] tried to diminish the problems of their predecessors, which were varying design mistakes due to unprecise communicated or just unknown requirements to the software solution that were often excessively costly to correct [32, 33]. Domain specific languages can be a supportive way of actively including domain experts in the design process and therefore reduce software design flaws which can result e.g. out of communication misunderstandings.

Self-evidently domain specific languages do not attempt to be Turing complete in most cases [3] whereby it should be noted that they are often on top of Turing complete languages or can become Turing complete if the domain specific language becomes as big and general as a general purpose language [3]. If a domain specific language is based on another language it is called external domain specific language and if it is independent from any other language it is referred to as an internal domain specific language [4].

Both approaches combined almost directly lead to model-driven development, an interesting approach which has its focus on creating software on basis of models [23]. The approach undertakes, that problem solving concepts can be generalized in form of models which can be reused, independent of a specific domain. Nevertheless, domain specific languages are well suited to serve as modelling languages.

To understand the core concepts of model-driven development, it is necessary to know the difference between software development and programming. On one side software development “includes aspects such as requirements engineering, development processes, software design, and documentation” [7], on the other side programming is a much more narrow term, describing the very writing of software code. Traditionally, models are created before (or in the industry during or after) the implementation of software. This is the reason why the architecture of an application in form of models, or more precise diagrams is not consistent to its implementation, even though it should be. An advantage of model driven development is that it is capable of closing this gap between design and implementation by insurmountably connecting design and implementation [24].

As already stated, a primary goal of model driven development is to create code from models, which entails further advantages, as well as disadvantages. Besides the fact that code can be faster generated than written, if it is reusable in different problem solutions, another great benefit is the before mentioned overcame gap of design and implementation through an inevitable synchronization of software documentation in form of the models and the actual software code by generating code from models or vice versa. Someone could argue that the generation of code is, despite of the remarkable

progression of code generators, still in the fledgling stage, the increasing attention in both industry and research communities emphasizes the future relevancy of this topic for software developers [34].

With the emergence of model driven engineering and domain specific languages, user friendly editors became increasingly popular. Since “[...] the notation is especially important for languages targeted at non-programmers […]” [5], the demand for supporting editors is on the rise.

This can be seen in the state of the art features of almost every popular integrated development environment, where smart code completion, syntax highlighting, on-the-fly code analysis and automated refactoring functionalities help developers to work with the textual representation of a computer program. Although these features make significant contribution to read and understand software code, not all difficulties can be eliminated this way. For instance complexly nested conditional structures as well as asynchronous execution threads cannot be represented by plain text. The first mentioned issue could be tackled by representing the logic in form of decision trees or tables, whereby the latter mentioned can be represented clearly in form of diagrams. The alert reader will ask himself, if there is a solution for a clear representation of an application faced with both challenges, complexly nested conditional structures as well as asynchronous execution threads. The answer is no, it is not possible with the classical approach of source editing which reveals the very hurdles that prevent non-technicians from active participation in the development process. In the industry this is a crucial issue since the majority of business logic of an application, where domain expert knowledge is needed, deals with decisional structures.

The aforesaid arguments outline the need for different ways of editing a computer program, which leads to the approach of projectional editing. Projectional or structured editing separates the editable presentation of an application from the storable presentation of the respective information. The subsequent chapter explains projectional editing in detail and points out the differences to source code editing.

## 2.2. Projectional Editing

As mentioned previously, the classical and more common approach to edit a computer program is called “source editing” [25]. This approach is the state-of-art way of writing and maintaining source code information of applications and is hallmarked by several source files containing the program logic in textual form. All information is stored in those source code files, as well as directly edited if an alteration of the underlying logic is necessary. Later on these two facets will be treated separately as storage and editable presentation [25]. Depending on the language, whether it is compiled or interpreted, the information gets transformed before or during execution into an executable representation of the application which is also called runnable.

During the so called lexical analysis, the source code, more precise the source code in form of a stream of characters, is split into tokens by a lexer. A token is the smallest unit and consists of a type (e.g. number, literal, operator ...) and a corresponding value. Every token represents a so called lexeme, a concrete representation of a basic unit of meaning in computer language. Subsequent a parser takes this array of tokens, gained from the former step and orders them into a syntactically meaningful form, the so called abstract syntax tree, which is also referred to as the abstract representation. Thereby the parser

checks whether the sequence and combination of tokens fits the expected grammar. The abstract representation of a program can be turned into an executable file or representation which is also called executable representation [25]. Figure A comprises the different representations and their relations.

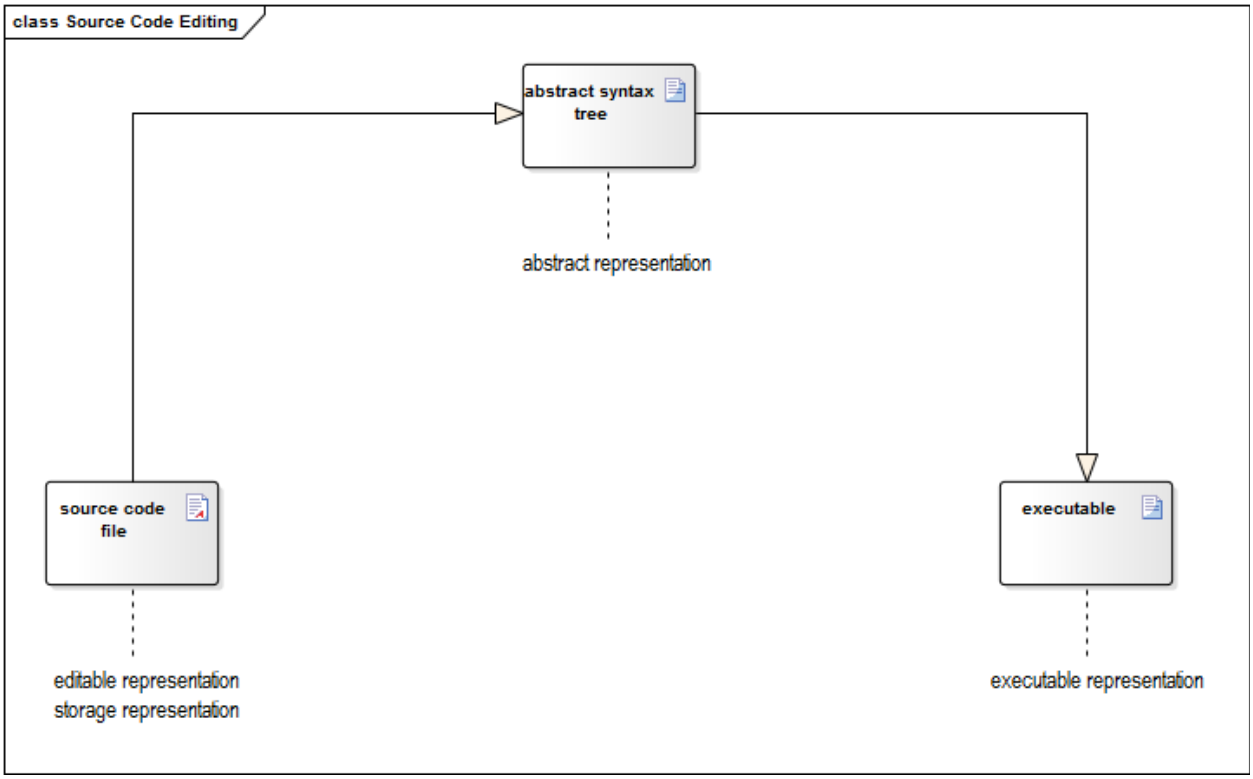



*Figure A: Representation forms of source code editing (adapted from martinfowler.com [6])*

The most important fact for the purpose of explaining the difference between projectional and source editing is that in source editing, the storage and editable representation are identical. This is distinct from projectional editing, where multiple editable representations, the so-called projections, from the same storage representation are possible. This implies that different projections or views can be used to edit the very same information or parts of that information of an application. The storage representation is different from these

projections due to various reasons, which are going to be discussed in the following section. The storage representation can but must not equal the abstract representation. Figure B illustrates projectional editing and its representations.

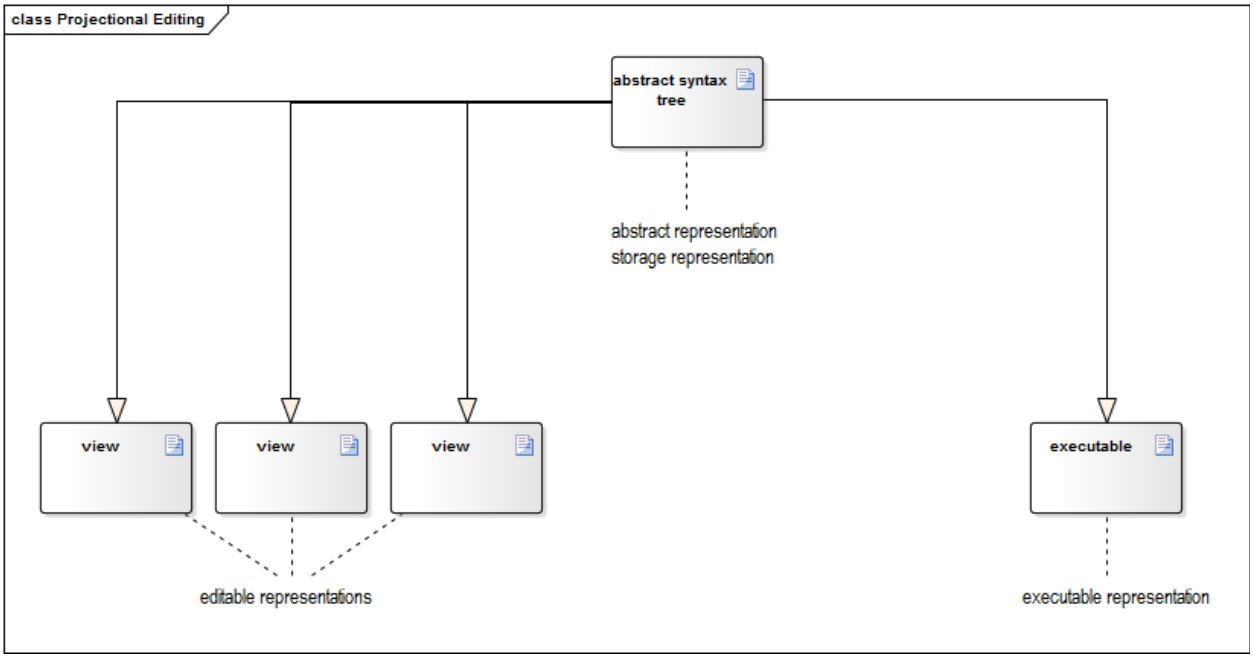



*Figure B: Representation forms of projectional editing (adapted from martinfowler.com [6])*

Another way of explaining the differences between source code and projectional editing is to analyze the refined and strong distinctions from an editor's point of view. Whereas the abstract syntax tree is the core concept of the abstract representation and stores all the information in a way which is favorable for the processing in a computer, the editable representation is also called concrete syntax which is more favorable for the developer because it focuses on a concrete aspect and therefore is able to shield complexity. From the viewpoint of usability, the major differences of source code editing and projectional editing are based on their technical distinctions. In editors based on source code editing

the concrete syntax is the single point of editing and viewing the total information. In projectional editing the abstract syntax tree is edited directly whereas the concrete syntax is used to display the information to the user, as shown in Figure C.

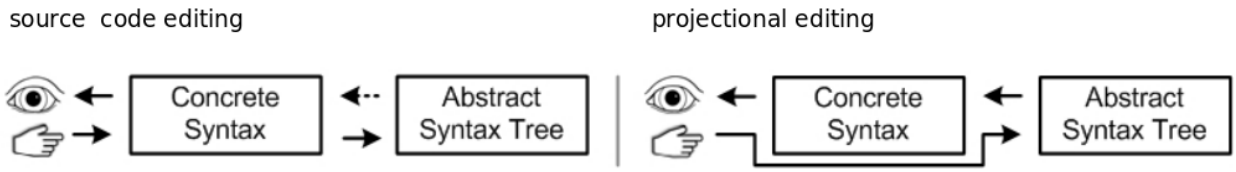



*Figure C: The difference of source code editing and projectional editing (image source: [5])*

Multiple projections, as already noted, enable software engineers to analyze and applications using different points of view in order to “construct a subjective mental model” [27]. The scope and profundity of this mental model of an application is crucial for a complete understanding of the mechanics of a problem domain. Projectional editors divide the problem domain in different projections, leading to an easier navigation in the code and a shortened learning curve [26, 27].

Projectional editing leads to major adaptions to the way a software engineer can create and modify an application. For instance a major problem that comes with projectional editing is, that it is almost impossible to enable efficient insertion and manipulation of textual code [5]. As projectional editing relies on direct manipulation of an abstract syntax tree in comparison to parsing characters from a text buffer to build a new abstract syntax tree on every change in source code editing, all manipulation have to be strictly defined actions. Manipulations have to be validated in order to keep the abstract syntax tree valid. This is significantly in contrast to the idea of freely entering code. Not only that text allows invalid statements, disambiguation can occur and manipulations have to be recognized

by the editor. Moreover, an abstract syntax tree contains information in tree form which is structurally different to textual statements. This is one of the reasons why developers claim that projectional editing is not suitable for industrial software development, because it is disproportionate time-consuming to think in abstract syntax trees. This overhead results in longer development cycles and thus higher costs of development. Because of the drawbacks of textual insertion, projectional editors generally do not provide the insertion of freely entered code. One may argue that famous projectional editors do provide a textual projection, but in fact these projections do not support unaided insertion of code but rather guided insertion of tokens, which might seem from a user's point of view very similar to autocomplete features known from integrated development environments. However, even if this might look familiar to developers, since integrated development environments include convenience features like type ahead, autocomplete or even syntax enforcement, the differences between both approaches when considered in detail are significant, even if modern integrated development environments are able to construct an abstract syntax tree in real time [26]. From the user's point of view, the difference is obvious. Classical editors suggest available options, but leave it to the software engineer whether she takes the advice or not. On the contrary projectional editors enforce the usage of available tokens and do not permit any deviations from the grammar of the language. Moreover, source code editors have to parse the total textual information on any alteration of the source code to create an abstract syntax tree which can be used to determine all valid operations possible to be in accordance with the context free grammar of the language subsequently. It is quite evident that it is impossible to offer this feature for every alteration of a software engineer, which implies that grammatically

invalid statements are still possible. Projectional editors offer the same feature, though in another way, since the abstract syntax tree is already in sync with the concrete syntax. In the final analysis this means that a projectional editor is able to predict all valid manipulations flawlessly for all nodes in the abstract syntax tree. As mentioned previously, projectional editing requires an unfamiliar way of mentally conceptualizing code. What this means will be explained by an adapted example taken from “Towards User-Friendly Projectional Editors” [5] in more depth below.

### *2.2.1. Manipulation of Code*

The familiar way of entering information in form of code is, simplistically spoken, to enter the information similar as it would be spoken just with another set of grammar rules. This is easily comprehensible when considering the following example which has been slightly adapted from “Towards User-Friendly Projectional Editors”. Consider the insertion of the mathematical formula x = 6 + 3. In a source code editor the user would insert the information in the same sequence as she would express it verbally. “X” followed by an equal sign, followed by “6”, the addition sign and finally a “3”. Remember that a parser would analyze this sequence of characters to build the abstract syntax tree shown in Figure D.

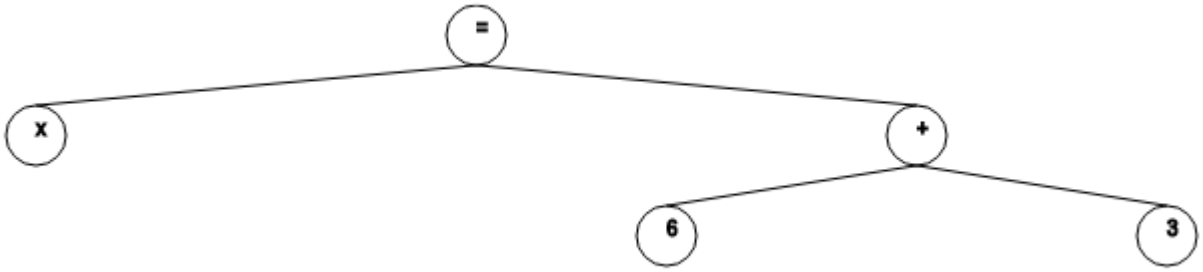



*Figure D: Abstract Syntax Tree of the formula "x = 6 + 3"*

As Figure D shows, the information is not stored in a one dimensional sequence of strings but in a tree with a root node containing the equals sign. If a user wants to enter this exact information in a projectional editor she has to think of the formula in abstract syntax form. This leads to another difficulty in projectional editing which is the handling of changes in the code. If the formula x = 6 + 3 has to be altered to x = 6 + 3 * 2 the additional information can be entered easily in a source code editor. Subsequently, the parser would take the whole formula to parse it into an abstract syntax tree as illustrated in Figure E. If the user had to execute the same manipulation directly on the abstract syntax tree, he or she would have to delete the leaf with the number 3 in it first, before he or she would be able to add the new subtree. This subtree has a parent node with the value multiplication sign and two leafs with the numbers 3 and 2 in it.

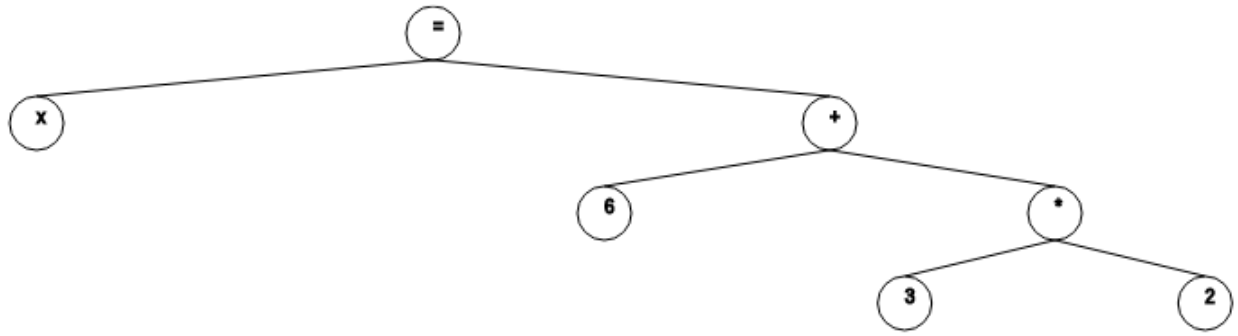



*Figure E: Abstract Syntax Tree of the formula “x = 6 + 3 * 2”*

Another barrier of using projectional editors for industrial software development, regardless of projection techniques, is that accustomed approaches of commenting code including commenting code out are not transferable to projectional editors [5]. Where the centralization of editable and storage representation in source code editing brings the problem of missing abstraction possibilities, the centralization of abstract and storage representation entails that there is no efficient way of adding unnecessary information, as comments are from the viewpoint of a computer, to the logic of an application. On the other hand, commenting code out is difficult, because nodes in their symbolic form for computers are inseparably connected to their parent and children nodes. If a subtree placed in the middle of an abstract syntax tree shall be removed, all leafs beneath this subtree dangle in the air and are unreachable and therefore superfluous. An automatic recreation of the lost edges is not only due to the unfitting cardinalities impossible.

Additional to this inconvenience comes that pasting code snippets is almost impossible. Textual code acts like a self-contained system that incorporates all details of a reusable block of information and is therefore conveniently transferable. On top of that, even the selection of code is fraught with problems due to the circumstance that a projection does

not have to display the total information of a model. To demonstrate how savage this cut is for software engineers, remind that gluing code is the preferred alternative to rewrite existing functionality for the majority of problems. This implies that software engineers who highly depend on the reuse of existing code are faced with a fundamental shift in the approach of developing applications.

### *2.2.2. Coding Infrastructure*

Infrastructure as code is not only a universally known buzzword with reference to the operation of information technology, but becomes increasingly relevant for the infrastructure used by developers. The productivity of engineers is highly dependent on the integrated development environment and the tools used within the team. With the different mechanisms among both editing methods in mind it is self-evident that version control systems like Git or SVN cannot rely on their usual operating principles by what collaborative software development is further hampered. The majority of supporting tools is built on the file-based nature of structured editing and relies on superficial analysis of the information, independent from the underlying logic or grammar, which is incompatible and inapplicable to projectional editing. With that said a considerable proportion of infrastructure tools is not compatible to the approach of projectional editing and cannot be easily ported [5]. A solution for this problem is likely to be found in a different approach for collaborative work. Today's version control system heavily depend on file comparisons, enriched by algorithms which recognize differences accurately and provide solutions for the merge process of diverging files. Operational transformation [30, 31], the key mechanism underlying collaborative text editors like Google's Docs or Microsoft's

Word Online, works on basis of single manipulations which are transformed for every single collaborator's context in order to be faced by a synchronized state of a document [30]. This approach is technically much closer to projectional editing, which works on basis of manipulations too, than the idea of compare abstract syntax trees entirely after every manipulation.

### *2.2.3. Prospects*

Despite everything, it should be mentioned that the comparison of source code editing and projectional editing in the literature is almost exclusively based on criteria favorable to source code editing. Since projectional editors are despite their longstanding presence a niche product in comparison to source code editors, only a few software engineers are as experienced in both editor types and can therefore rely on the same level of information. Moreover projectional editors lack of time and monetary investments on the one hand and empirical values on the other hand to be as mature as modern source code editors are these days. Hence it is questionable if an indicated inherent superiority can be concluded from the mentioned advantages.

On the sunny side of projectional editing, the error rate of coding due to semantical and syntactical errors can be diminished impressively [22]. This does not prevent software engineers from creating logical flaws, of course, but can still be exploited for one's favor. The key advantage, though, lies dormant in the potential capabilities of multiple projections. If this advantage is able to outweigh the downsides of projectional editing, it could change the way of creating applications revolutionarily. Different views on a software system make it possible to fully comprehend even outstandingly complex

architectures due to suitable forms of representation. Multiple views are basically nothing new to software architects, who are already used to different diagrams for different purposes. In further consequence this leads to a stronger and more natural connection between architecture and implementation which in turn shortens training periods, enhances software documentation and cuts costs. Participation on projects can be further encouraged, since bulky coding guidelines which are a barrier for new and unexperienced engineers eventually become in a great measure redundant. This might be thought too far ahead, but as version control systems promoted open source communities, projectional editing could lead to another uplift for collaborative engineering.

To sum it up it can be said, that today's literature focusses on today's level of experience, circumstances, habits and solution processes. It is hardly surprising that projectional editing performs poorly on these criteria. However, software systems become increasingly complex and projectional editing could be a part of a future solution strategy if popularity grows and if internal oppositions can be overcome.

## 2.3. Reusable Architectural Design Models

In this section the approach used to capture the findings of this thesis will be described in greater detail. Findings gained from the design and implementation of the prototype as well as insights from the literature are used in Chapter 4 to create a generally valid guidance model for the design and implementation of a projectional editor. A clear understanding of this approach is necessary to interpret the outcome of this thesis correctly.

Software documentation is a key deliverable not only in IBM's business reference

architecture framework, but also to stakeholders with high quality expectations [9]. In practice software documentation is implicitly built around models [9], with the consequence being that the chosen alternative is quite well documented, but the road to it is covered and therefore mystical. This circumstance is most unfortunate as software documentation should not be autotelic, as rather end in long-term benefits due to reduced development times and simplified solution analysis in the future. If a decision's alternatives and drivers are unmentioned, the effort, necessary to study the pros and cons of every alternative possible, is all in vain in respect of prospective decisions.

A way to tackle this issue is to register decisions in form of formalized design decision templates where design decisions alternatives are listed and it is explained why one alternative was preferred over others as illustrated in the metamodel in Figure F.

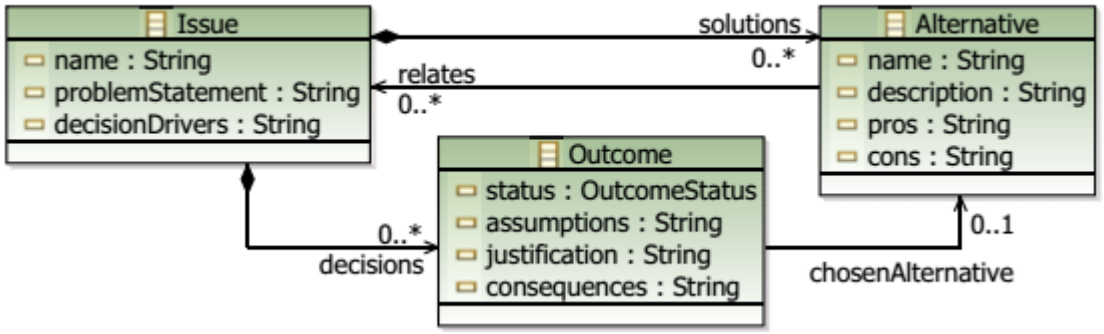



*Figure F: Design decision metamodel, taken from [15]*

This upgrades retrospective documentation artifacts to design guides [10] which can bring considerable beneficial input to similar decisions. This further supports software documentations' long term benefits, which have to justify the expended effort. An example for such a design decision template can be seen in Table A. There are many ways to

document design and this template is only one possible example, based on an IBM template and supplemented by two additional fields [9].

| Design Decision Template | |
|---|---|
| Issue | Describe the architectural design issue you're addressing, leaving no questions about why you're addressing this issue now.<br><br>Following a minimalist approach, address and document only the issues that need addressing at various points in the life cycle. |
| Decision | Clearly state the architecture's direction—that is, the position you've selected. |
| Status | The decision's status, such as pending, decided, or approved. |
| Group | You can use a simple grouping—such as integration, presentation, data, and so on—to help organize the set of decisions. You could also use a more sophisticated architecture ontology, such as John Kyaruzi and Jan van Katwijk's, which includes more abstract categories such as event, calendar, and location. For example, using this ontology, you'd group decisions that deal with occurrences where the system requires information under event. |
| Assumptions | Clearly describe the underlying assumptions in the environment in which you're making the decision—cost, schedule, technology, and so on. Note that environmental constraints (such as accepted technology standards, enterprise architecture, commonly employed patterns, and so on) might limit the alternatives you consider. |
| Constraints | Capture any additional constraints to the environment that the chosen alternative (the decision) might pose. |
| Positions | List the positions (viable options or alternatives) you considered. These often require long explanations, sometimes even models and diagrams. This isn't an exhaustive list. However, you don't want to hear the question "Did you think about ... ?" during a final review; this leads to loss of credibility and questioning of other architectural decisions. This section also helps ensure that you heard others' opinions; explicitly stating other opinions helps enroll their advocates in your decision. |
| Argument | Outline why you selected a position, including items such as implementation cost, total ownership cost, time to market, and required development resources' availability. This is probably as important as the decision itself. |

| | |
|---|---|
| Implications | A decision comes with many implications, as the REMAP metamodel denotes. For example, a decision might introduce a need to make other decisions, create new requirements, or modify existing requirements; pose additional constraints to the environment; require renegotiating scope or schedule with customers; or require additional staff training. Clearly understanding and stating your decision's implications can be very effective in gaining buy-in and creating a roadmap for architecture execution. |
| Related Decisions | It's obvious that many decisions are related; you can list them here. However, we've found that in practice, a traceability matrix, decision trees, or metamodels are more useful. Metamodels are useful for showing complex relationships diagrammatically (such as Rose models). |
| Related Requirements | Decisions should be business driven. To show accountability, explicitly map your decisions to the objectives or requirements. You can enumerate these related requirements here, but we've found it more convenient to reference a traceability matrix. You can assess each architecture decision's contribution to meeting each requirement, and then assess how well the requirement is met across all decisions. If a decision doesn't contribute to meeting a requirement, don't make that decision. |
| Related Artifacts | List the related architecture, design, or scope documents that this decision impacts. |
| Related Principles | If the enterprise has an agreed-upon set of principles, make sure the decision is consistent with one or more of them. This helps ensure alignment along domains or systems. |
| Notes | Because the decision-making process can take weeks, we've found it useful to capture notes and issues that the team discusses during the socialization process. |

*Table A: Design Decision Template taken from [9]*

Architectural or design decisions should cover the system's key structural elements and still be as few as possible in order to "maintain a balance between guiding and constraining the technical organization" [9]. Furthermore, fewer key decisions are easier to be socialized in the sense of communicated to all architects of a team. The term socialized addresses the difficulty to not only communicate insights from prior decisions

but also to keep them present and in mind. This requires a well-defined process of storing decisions in a way that is well-suited to keep a mass of documents sorted and retrievable. It is questionable if the state-of-the art way of keeping decisions in decision logs has a promising future.

A more comprehensive option to file decisions can be found in so-called guidance models. A guidance model [10] is a metamodel, listing and organizing design decisions for a particular application genre and architectural style. The basic idea behind a metamodel is to identify decisions required and make them reusable, so that they are reusable in terms of versatile artifacts which can be consulted when the same fundamental type of problem reoccurs. This can be easily explained by a fictive example. When building a large scale system with a huge amount of inhomogeneous data expected, there is always the question how to persist the information. IT service organizations are faced with exactly the same question over and over again, with the result that the selection of a database technology has to be evaluated multiple times. In a guidance model for large scale systems this issue would be one among other reoccurring issues and if a new large scale system has to be build, the corresponding design decision could light the way.

A concrete instance of a guidance model is called decision model. The interplay between guidance models and decision models can be explained this way: On the one hand guidance models aim to identify decisions required, on the other hand decision models aim to log decisions made [10]. This relationship is illustrated in Figure G. As illustrated a decision model can be based on multiple guidance models if the granularity of the guidance models allows it. Some might say that this concept is close to the concept of

multiple inheritance of interfaces in C++. With that said, the future of industrial software development could be based on a selection of several guidance models, which guide through the occurring design decisions for the software to be. In an ideal situation, all occurring design decisions would be covered.

One might ask, what the difference between guidance models and patterns, like the GOF patterns, is. In the papers [13] and [14] it is explained like this: neither guidance models nor patterns are able to solve all design issues in an architectural process. A pattern “is a proven solution to a problem in a context, resolving a set of forces” [13]. Patterns are not domain specific, but rather abstract solutions for recurring technical challenges. On the other hand guidance models and with that design decisions focus on expert knowledge which is usually related to domain knowledge. Moreover, design decisions offer multiple options including respective advantages as well as disadvantages. Decision drivers have to be considered in order to pick the accurate option, whereas patterns offer advice for technical challenges. For instance, the selection of a database technology for an online shop is inseparable from the domain specific requirements, which is why this issue should be documented with all available alternatives including their advantages and disadvantages, as well as related decisions and decision drivers. On the contrary, the separation of business logic and code used for the presentation layer is a more general, technical issue which can be recorded in pattern language, because it is not related to a specific domain. The proposed approach in [13] combines pattern languages and architectural decision models in order to make use of both advantages.

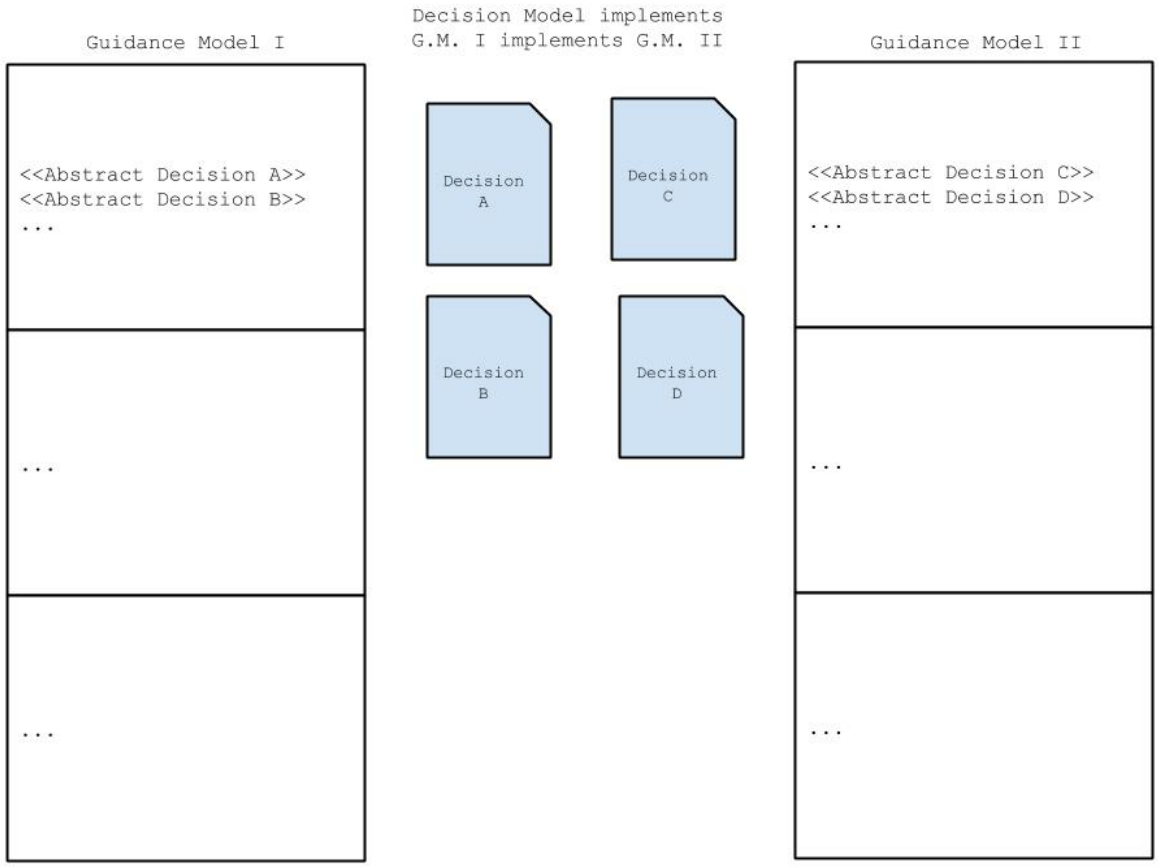



*Figure G: Interplay between guidance and decision models*

A well-known example for a guidance model is the one for service oriented architectures called SOA guidance model [11], which will be discussed in more detail in order to provide a better and deeper understanding of such guidance models. A major goal of guidance models is to prevent bad design decisions which can be are costly to change [11] or in other words the guidance model is aimed to provide concrete advice for practitioners when designing a software oriented architecture. This includes information about architectural patters and their advantages and disadvantages in regard to software oriented architectures as well as recurring problems and proven solutions with due regard

to available options and drivers affecting the selection of an appropriate alternative. For example, a recurring decision in the architectural process of a service oriented architecture is the selection of a service protocol which could be either REST or SOAP. Of course there are advantages and disadvantages on both sides, but above that this decision significantly affects further decisions like the standard of exchanged messages which could be either XML or JSON, for instance. So the guidance model not only supports the architect in choosing the best options, but additionally foresees upcoming decisions which have an impact on the current decision. To bring it back to the metamodel depicted in Figure F, this decision as a concrete instance of the model could look like the diagram in Figure H. Kindly note that design decisions can be linked to, or even affect other decisions.

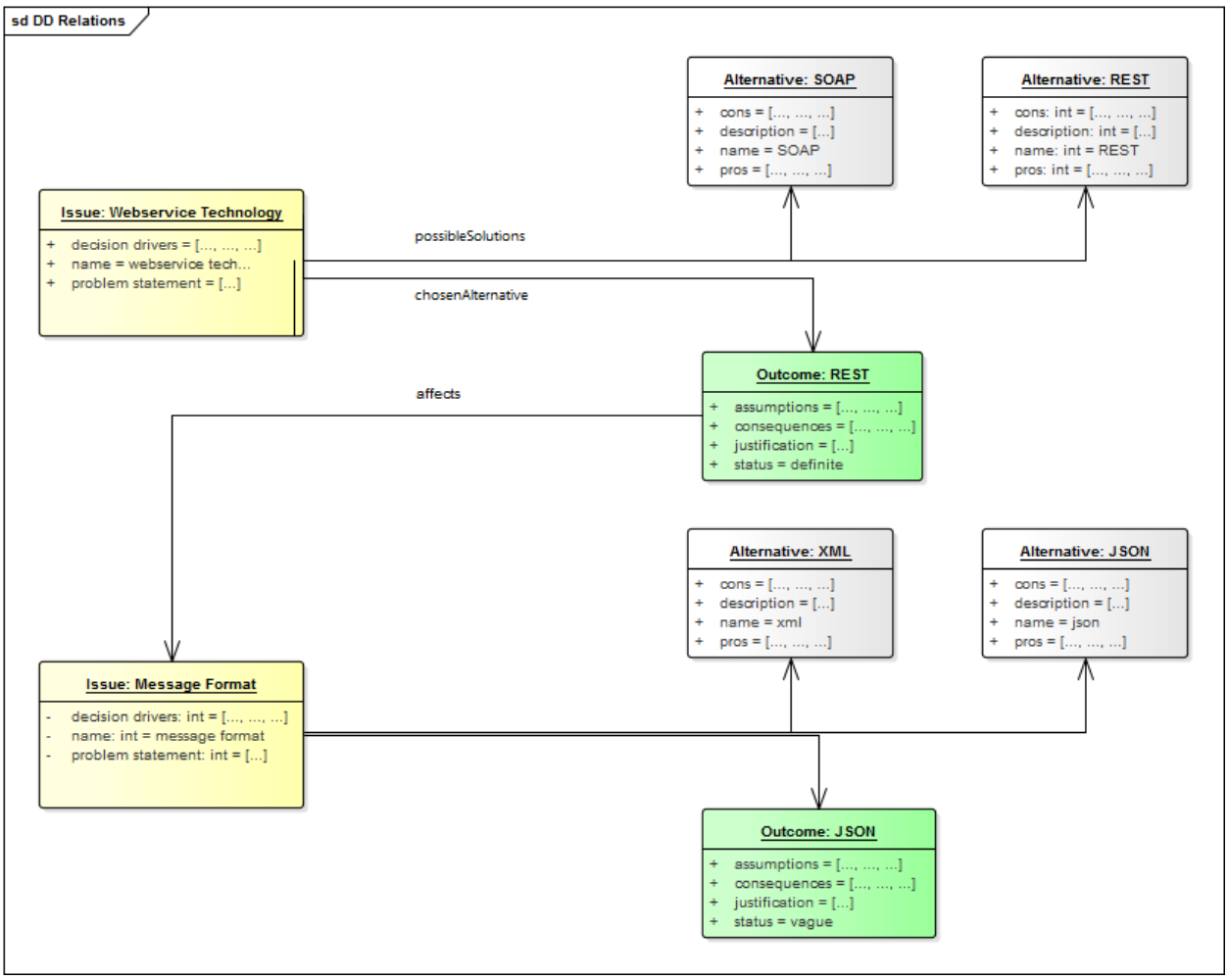



*Figure H: Exemplary instance of related design decisions*

This chapter will be completed by a process, proposed in [12] explaining a conceptual framework for “decision modelling with reuse”, which will be used in Chapter 4 to create a guidance model for the implementation of projectional editors. The process consists of three consecutive steps as illustrated in Figure I.


*Figure I: Decision making steps, adapted from [12]*

- In the first step of the process, the so-called “decision identification”, decisions have to be identified during the design phase of a software project. Possible triggers for decisions are requirements or earlier decisions, which lead to new issues and therefore necessary decisions. Paper [12] proposes a so-called “push approach” in response to drawbacks of the “pull approach”. This means that necessary decisions are identified on basis of a requirements model as well as a reference implementation upfront and not during the design phase, solely based on the architect’s experience.
- Subsequently alternatives are selected according to respective decision drivers in the “decision making” step. Another task of this step is to evaluate and select the most appropriate alternative for the project. The authors of [12] criticize software architects for selecting alternatives solely based on personal preferences and experience made in the past, hence, they propose different options to support an unbiased and objective decision.
- The last step “decision enforcement” serves the purpose to communicate and share the knowledge gained during the process. This is especially important for achieving a sustaining benefit that justifies the expended effort. The first step and the selection of alternatives during the decision making can be seen as attached to the guidance model, whereas the selection of an alternative in context of the

actual software project belongs to the concrete instance of the guidance model, or in other words the decision model of the project. The decision enforcement cannot be attributed to one of the two models since communication of both models is equally important but may address different stakeholders.

# 3. Prototyping a projectional editor

For this thesis a prototype has been created to exemplify the implementation of a projectional editor. This approach promises to gain experience in accordance to the literature in order to identify occurring design decisions. These decisions are going to be used later on to create a universally valid guidance model.

The first question arising was: which domain could possibly be suitable for the demonstration purpose of a projectional editor? Having carefully weighed the advantages and disadvantages of several domains, the selection fell on mathematical formulas due to various reasons. These include a manageable amount of operations, the possibility of multiple useful projections and the circumstance that this domain does not have to be explained. Moreover insights gained are transferable to other domains due to the generality of this field of application. In order to progress as quickly as possible, dynamically typed languages were preferred over statically typed ones.

Furthermore the implemented editor should be delivered in form of a web application in contrast to prevalent desktop editors due to various reasons, which are discussed in detail in this chapter as well as in DD-001 (design decision about the software delivery platform). Web-based applications enjoy a massive upturn in comparison to desktop application [37, 38], which is not exclusively based on the emergence of mobile devices. Since the turn of the millennium “the first signs of using the web browser itself as an application platform started emerging” [37] leading to a transformation of the software industry [36] which is keeping on and is furthermore accelerated through the appearance of new browser technologies like HTML5, WebGL and ECMAScript 6 and 7 in the past decade [37]. Nevertheless, this decision is not merely based on the rising popularity of web

applications over desktop applications. General valid advantages of web applications like the cross-platform nature and the capabilities supporting rapid prototyping [35] perfectly fit the purpose of this thesis, but also specific needs of editors are encouraged by the use of web technologies. These include a high amount of user input and interaction as well as the role of user interfaces in this application. Last but not least, this prototype of a web based editor is a novelty and can therefore potentially open up entirely new possibilities for projectional editors, emphasizing collaborative interaction of multiple editors, which could easily be implemented using operational transformation, as already stated in Chapter 2.

Subsequently different projections had to be selected, where the choice fell on a textual representation of the abstract syntax tree as well as a binary tree representation. The idea of a latex based representation of the formula has been rejected, because of the strong similarity to the textual presentation. The importance of the selection of projections will be discussed later on. However, for the purpose of this thesis, the relevance of this decision is highly limited due to the lack of real application of the prototype. Instead of a business decision, the selection was mainly based on a technical perspective. More information about the reasons behind this decision can be found in DD-002 in Chapter 3.3.4.

Agile development approaches were chosen in order to maintain and moreover support the flexibility of prototyping which allows a certain amount of uncertainty during the process. Since the software to be built is only vaguely outlined more classical approaches would not be appropriate for the purpose of conducting this research. Findings gained during the design and implementation process could not be used since classical

approaches command a strictly consecutive order of design and implementation, without the possibility of iterative loops. Therefore the architecture and the way of documentation have to be suitable, which is why different techniques based on Kanban, Scrum and Extreme Programming were used to put the prototype into writing. Design decisions fit in seamlessly in this approach. In order to keep the effort in a reasonable ratio to the scope of this paper, some documentation requirements have been simplified. From the author's perspective, this serves the scientific method which is clearly different from business approaches.

The requirements model in the next chapter continues the vague specification defined in this paragraph and extends it to a comprehensive requirements model, which is the basis for the architecture of the prototype.

## 3.1. Requirements model

In order to gather the different requirements to the software, a requirements model in form of epics has been created. This approach has been chosen to draw the outlines of the application, because it supports agile software development methods and therefore it seems most appropriate.

### *3.1.1. Architecture Epics*

In this chapter a product backlog is used to define the functional as well as the nonfunctional requirements to the prototype. Each epic aims at describing functionality for a certain stakeholder plus the respective goal. The central role is assigned to the editor, who is working with the projectional editor. All other stakeholders, like system

admins and/or devops teams, are negligible for the scientific approach.

| As a(n)... | I want to be able to... | So that I can... |
|---|---|---|
| editor | create a new formula | work on mathematical formulas on my computer |
| editor | see my formulas | have an overview |
| editor | persist my formula | view it again later |
| editor | reopen my formula | work on it again |
| editor | work with a textual projection | read/write using a textual projection as I am used to |
| editor | work with a binary tree projection | read/write using another projection |
| editor | switch between projections | use different projections flawlessly |
| editor | see the abstract syntax tree | see how operations affect the structure of it |
| editor | validate my formula | write formally valid formulas |
| editor | execute my formulas | check the validity and functionality of my formula |
| editor | use my browser to read/write formulas | work on them everywhere, on every suitable device |

*Table G: Architectural Epics*

This requirement model goes without a split of epics into user stories and tasks, since the main purpose is to sketch the outline.

## 3.2. Architecture

For a better, although still low granular, understanding of the different views on the prototype, the 4+1 view model [16] design approach has been chosen. The 4+1 view

model does not focus on the form of representation and notations, but is understood as a generic approach which can be adapted by the software architect.

### *3.2.1. Logical View*

In this chapter the data structure of an abstract syntax tree is illustrated, which is the most important diagram of the structure of information within the system. The data is kept in JSON format, which can be used at the client, server and the database. The selected technologies Node.js and MongoDB are perfectly fitted for this.

Each abstract syntax tree starts with a root element, commonly an assignment, with a reference to one or multiple child elements. The formula object, holding the abstract syntax tree, contains additional information about the author, a name of the formula as well as information about the last alteration of the formula. The abstract syntax tree itself is in form of a tree consisting of multiple nodes. There are in total seven different types of nodes, which are: Assignment, Identifier, Number, Expression, Function, Binary and Unary. With these types of nodes any formula can be expressed.

An Assignment contains a name (name of the dependent variable), a value (the structure of explanatory variables) and a unique identifier. An Identifier has a name and a unique identifier. A Number contains a name and a unique identifier. Expressions have a value (the actual expression) and a unique identifier. Functions consist of a name, a list of arguments and a unique identifier. Binary nodes have an operator (basic mathematical operations are possible), a left and right side as well as a unique identifier. Unary nodes consist of an operator and an expression, as well as a unique identifier. The unique identifier of all nodes is necessary to target operations on specific nodes.

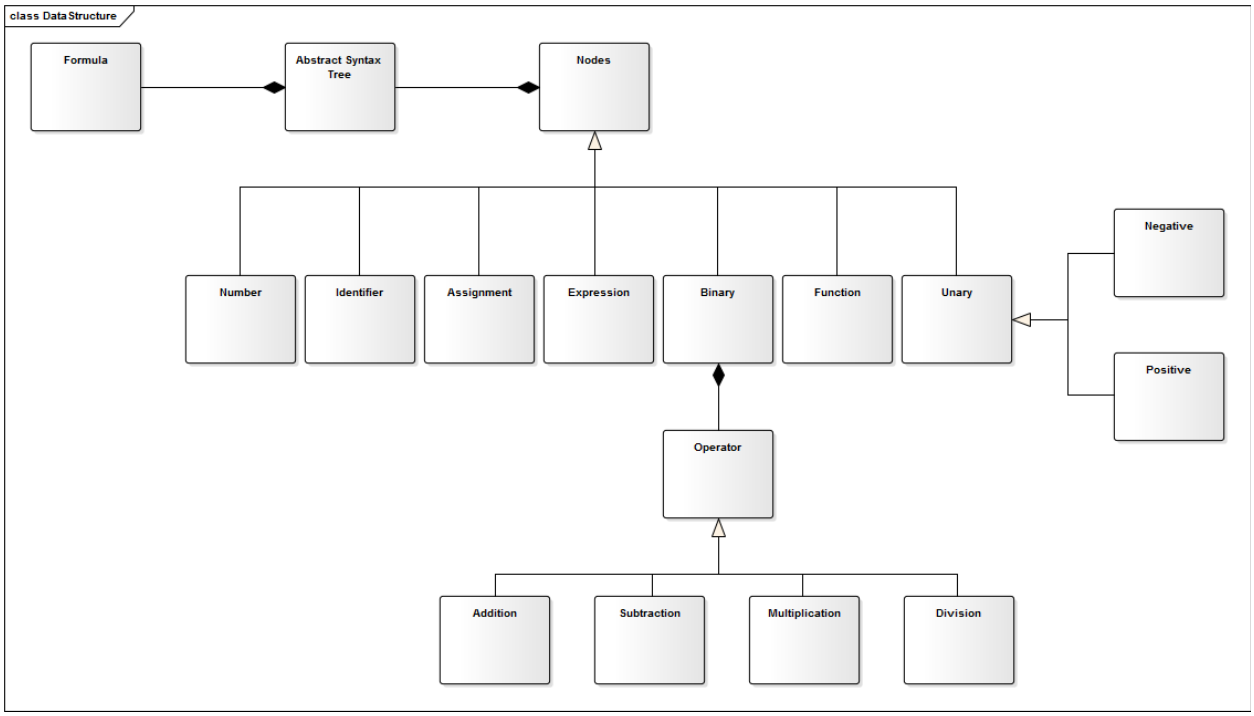



*Figure J: Data structure of the abstract syntax tree*

### *3.2.2. Process View*

An overview of the basic communication flows is depicted in Figure K. As it can be seen in the illustration, the abstract syntax tree is requested from the client at the server and all alterations are made on the client and are sent back to the server afterwards. This brings the benefit of reduced necessary communication between server and client, but faces the drawback that collaborative work would not be possible. If this would be a requirement to the software, all manipulations on the abstract syntax tree had to be sent isolated. This would also be very appropriate for the rest of the architecture, since the event driven nature of the technology stack is perfectly fitted for this purpose, but goes

hand in hand with the difficulties raised by the additional communication.

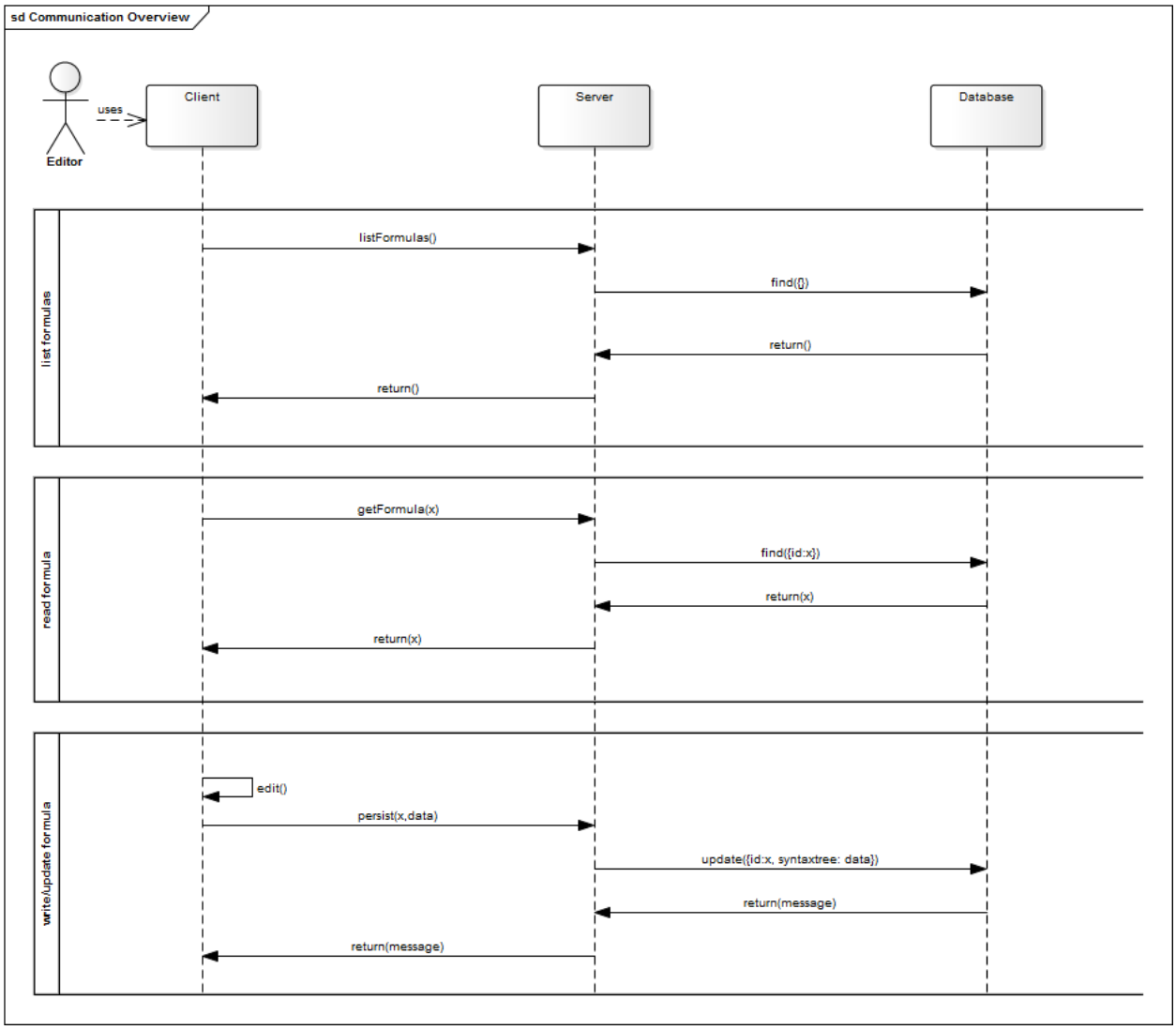



*Figure K: Communication between client and servers*

### *3.2.3. Development View*

The system overview depicted in Figure L shows the structure of the client and server application and the information flow between client and server as well as server and database server. The server cluster is basically responsible for the persistence of

formulas as well as for a final validation of the data structures received from the client. However, the majority of business logic is executed in the client application. The central class of the client application is the ASTController, which handles the communication with the projections of the editor, which can either be the textual projection or the tree projection. Moreover it handles the communication with the executer, which is responsible for executing the abstract syntax tree. Each Projection has an UI Component, which is responsible for handling the user's interaction with the projection. Additionally the textual projection is dependent on a lexer and a parser, which is the simplest approach to work with textual projections. On the other side, the tree projection uses a TreeHelper class providing functionality for handling trees.

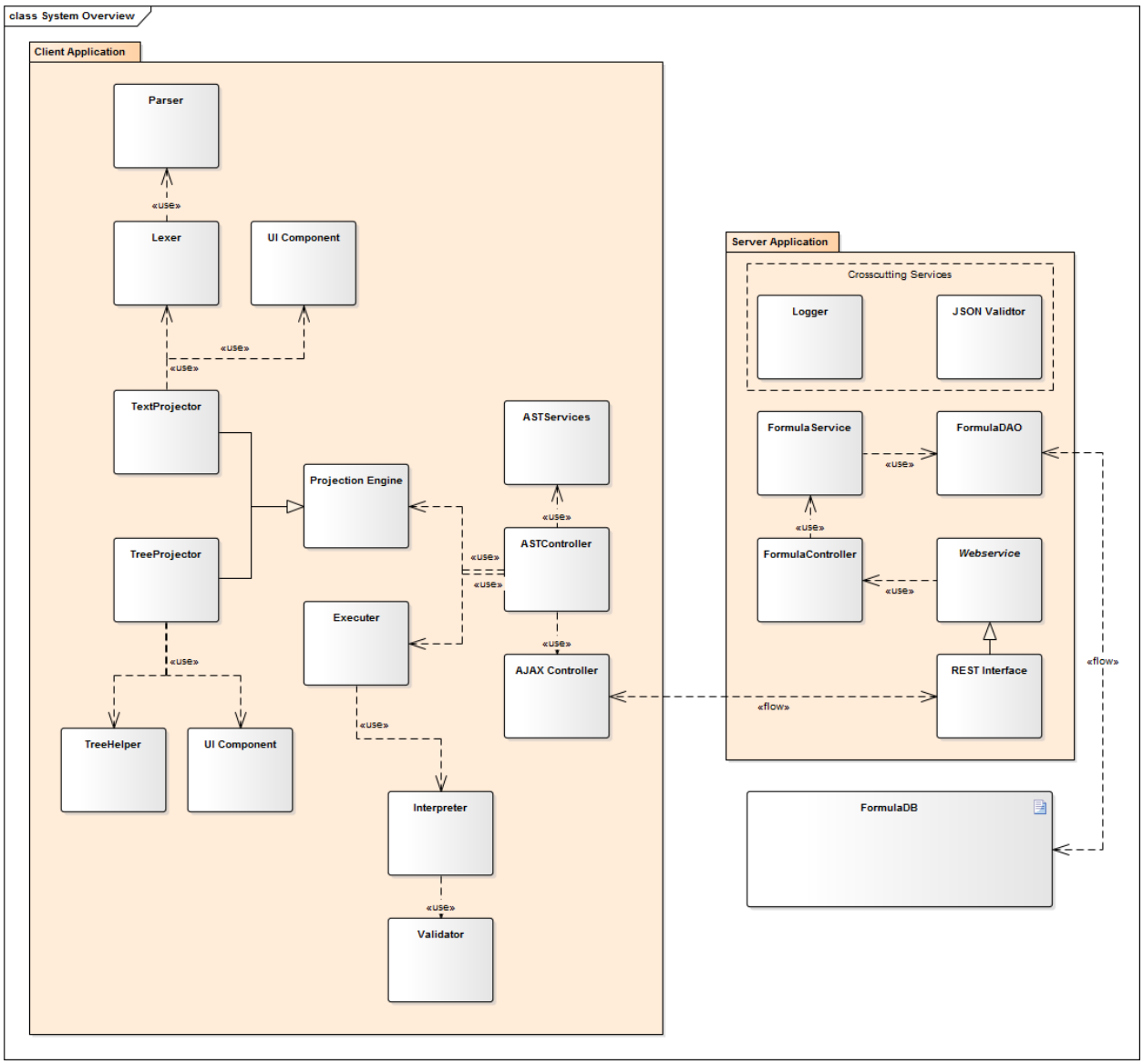



*Figure L: System Overview*

### *3.2.4. Physical View*

The physical view is hallmarked by three units. The web server hosts the backend that manages the interaction with the user as well as the persistence functionalities of the system. Therefore it has implemented a RESTful interface which is the single point of communication as well as a persistence layer communication with the database server.

The second unit is, as already mentioned, the database server, which provisions a MongoDB instance containing the database wherein the formulas are persisted. It can be deployed either on a separate device or on the same device as the server. The third unit provides the user with an interface for the interaction with the server, so it is the client of the editor. The communication between web server and browser is exchanged over HTTP, more precisely the RESTful webservice of the server. The client uses AJAX to communicate with the server and all connections established are initialized by the client. The infrastructure can be provisioned in the cloud using Chef Recipes.

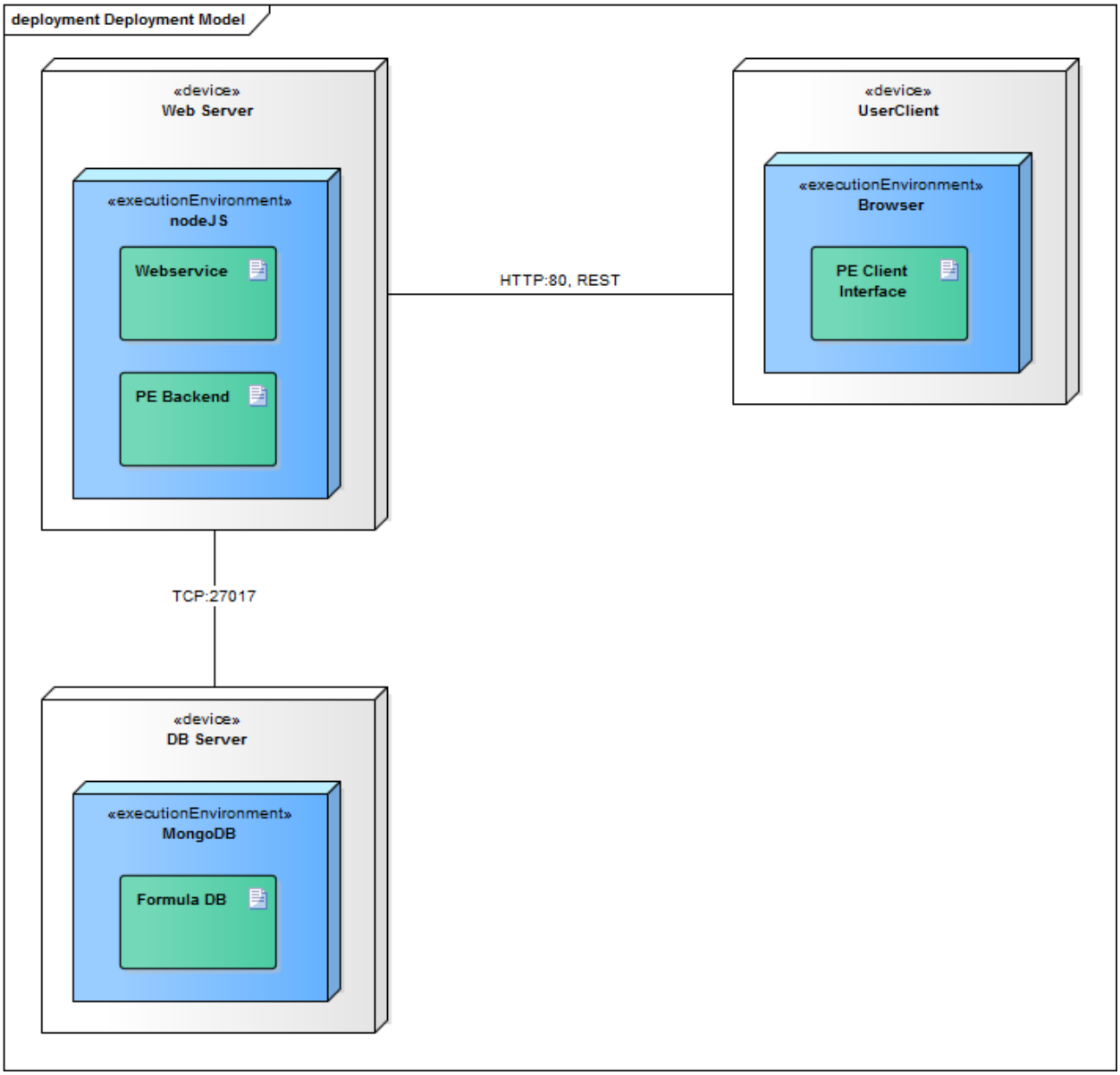



*Figure M: Physical View*

### *3.2.5. Scenarios*

The scenario view is based on the backlog covered in the previous chapter. However, the last epic is missing since it is a nonfunctional epic. The simplistic appearance is caused by the lack of different roles and the strong focus on the editor to the detriment of supportive features as, for instance, a role based access system or grouping and tagging

capabilities.

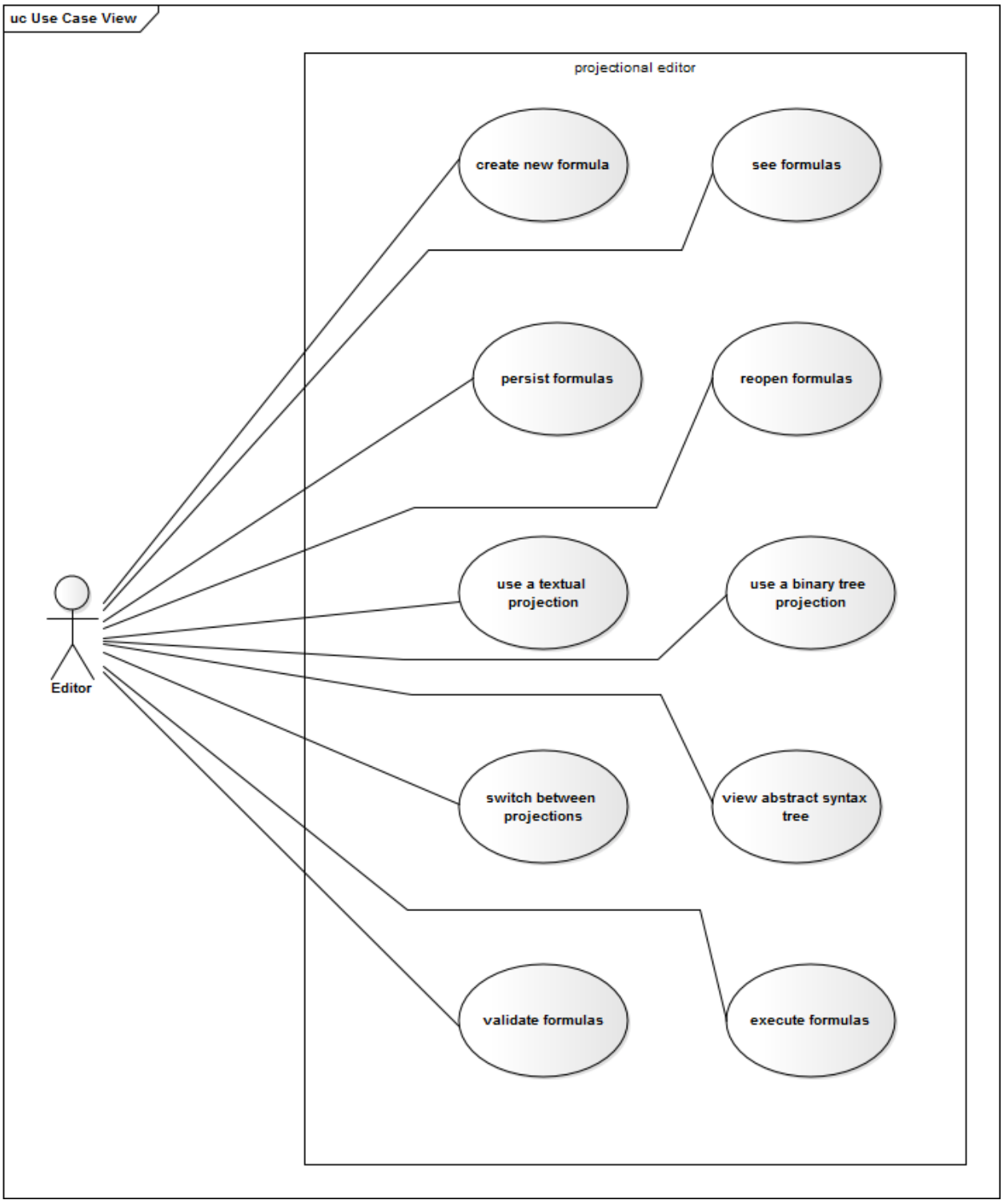



*Figure N: Scenarios View*

### *3.2.6. Technology Stack*

In this chapter the chosen technology stack is explained and all frameworks are listed with their respective licenses in Table B-D. On overview of the functional libraries, frameworks and components is depicted in Figure O. Infrastructure libraries are not listed in Figure O, as they are included in the tables.

The evolution of the selected frameworks and an explanation can be found in the following chapter about the implementation of the prototype.

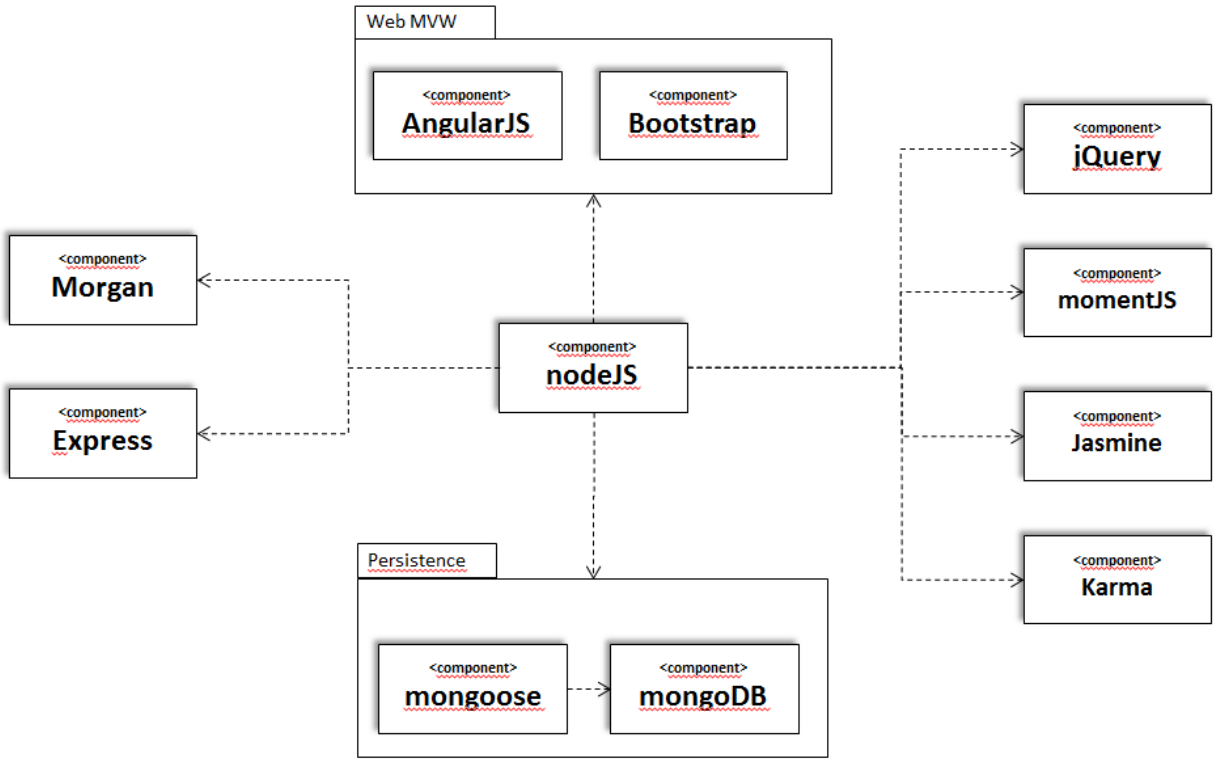



*Figure O: Technology Stack (Overview)*

Server

| Name | Version | Description | License |
|---|---|---|---|
| Node.js | 4.0.0 | Javascript Runtime Environment | MIT License (MIT) |
| Express | 4.12.2 | Web Framework | MIT License (MIT) |
| Mongoose ODM | 4.1.6 | Object Modelling Tool | MIT License (MIT) |
| Morgan | 1.6.1 | Logger | MIT License (MIT) |
| Gulp | 3.9.0 | Javascript Task Runner | MIT License (MIT) |

*Table B: Server components including licenses*

Database Server

| Name | Version | Description | License |
|---|---|---|---|
| MongoDB | 3.0 | noSQL Database | GNU AGPL v3.0 |

*Table C: Database server components including licenses*

Client

| Name | Version | Description | License |
|---|---|---|---|
| AngularJS | 1.4.5 | Web Application Framework | MIT License (MIT) |
| Bootstrap | 3.3.5 | CSS Framework | MIT License (MIT) |
| Moment.js | 2.10.6 | Javascript Library | MIT License (MIT) |
| JQuery | 2.1.4 | Javascript Library | MIT License (MIT) |
| Bower | 1.4.1 | Javascript Package Manager | MIT License (MIT) |
| Jasmine | 2.3 | Testing Framework | MIT License (MIT) |
| Karma | 0.13 | Test Runner | MIT License (MIT) |

*Table D: Client components including licenses*

## 3.3. Implementation

This section covers the implementation of the prototype created within this thesis. For detailed information about the architecture please take a look at Section 3.2, which contains a system overview as well as the technology stack for the prototype. The prototype has been developed using IntelliJ's IDEA version 14.1. JSHint has been utilized as code quality tool, to detect errors and potential problems in the code. Packages are with Node.js's npm package manager version 2.11.2 and frontend dependencies are included and managed with bower 1.4.1. The whole build process including code quality checks as well as unit and integration tests is modelled with Gulp 3.9.0.

### *3.3.1. Phase 1*

The first version of the prototype was based on SpringBoot and its embedded Tomcat servlet container instead of Node.js, which was rejected after a short period due to the unhandy communication between server and client. The handling of data transfer objects was fraught with problems, since the structure of an abstract syntax tree does not follow a strict schema, which would be more appropriate for Java's POJOs. The mapping between POJOs and JSON objects was not only tedious but also slow and led to a pile of mapping classes, thus confusing code. In addition to that, the communication between database server and application server was too slow to achieve an acceptable user experience, neither with Spring Data nor with MondoDB's drivers, which was hardly surprising since messages from the client in JSON format were mapped on DTO objects on the server and back to the JSON format for the database. Further, in this early stage of prototyping, all manipulations on the abstract syntax tree were sent encapsulated to

the server. This offers the advantage of keeping the client application lightweight and ensures a clear separation of logic and user interface related code, but suffers from the drawback of high latencies when interacting with the projections from the user's perspective. To take on these challenges, a shift in the use of technology was necessary, which led to following questions, answered in design decisions.

- How should the server/client architecture be structured regarding the separation of concerns? (DD-003 and DD-004)
- How can the selected projections be displayed providing the user with the best experience possible? (DD-005)
- Should the abstract syntax tree be persisted as JSON object in a noSQL database, or would a relational database using a generic approach lead to better results? (DD-006, DD-007 and DD-008)

### *3.3.2. Phase 2*

The decision making process for DD-003 could not be finished without a further test run, since the performance of a rich client application remained unclear. Therefore the client application of Phase 1 was ported to Python's SimpleHTTPServer in order to integrate the majority of business logic in the client application and consequently remove this functionality on the server. The usability of the prototype based on python was markedly improved, which is why this path was pursued further. The following questions were dominant in this phase, which is why they were documented in form of design decisions:

- Which platform is most appropriate for the architecture? (DD-009)
- Which messaging pattern fits best for the selected web service technology? (DD-

010 and DD-011)

### 3.3.3. Phase 3

In this phase the outline of the application is clearly drawn on basis of the design decisions made in the previous phases. A Node.js based server handles the communication between client and database server and is responsible for reading and writing formulas into the database. The Express framework was used to create the server-side interface for the message exchange between server and client. On the client-side Angular.js ensures a clean separation of logic and view. A minimalistic but still functional interface was built on top of Twitter's Bootstrap.

### 3.3.4. Design Decisions

| DD-001: Software delivery platform | |
|---|---|
| Issue | The editor can be either implemented in form of a desktop installation, which would be the most common distribution form for editors or in form of a web-based application. |
| Decision | Web-based application |
| Status | Decided |
| Group | Business Decision, Technology |
| Assumptions | - |
| Positions | ● Desktop application<br>the editor is delivered in form of a desktop application and has therefore to be downloaded and installed upfront. This choice would inevitably raise the question which operating systems should be supported.<br><br>● Web-based application<br>the editor is delivered online which implies that it is always |

| | |
|---|---|
| | available and has not to be installed. This choice implies that the editor has to deal with the downsides of the browser sandbox in concerning the access to the user's hardware. |
| Argument | With respect to the domain a web-based editor is not only possible but furthermore well-suited, since all representations can be edited, stored and executed in a web-based environment. Furthermore, this choice explores the opportunities of a web-based editor with all its advantages and disadvantages and is therefore especially interesting from a scientific point of view. |
| Implications | The technology stack has to be aligned with this decision. Moreover a server/client architecture has to be designed, along with consequent decisions. |
| Related Decisions | ● All other decisions are related to this one |
| Notes | - |

*Table E: Design Decision with ID: DD-001*

| DD-002: Selection of projections | |
|---|---|
| Issue | Multiple projections for the projectional editor have to be selected in regard to their match to the business domain. |
| Decision | Textual projection, binary tree projection |
| Status | Decided |
| Group | Business Decision |
| Assumptions | - |
| Positions | ● Textual projection<br>The formula can be displayed and written as freely enterable text<br>● Binary tree projection<br>The formula can be displayed and edited in a binary tree<br>● Latex projection<br>The formula can be displayed and edited in form of latex code<br>● Token projection<br>The formula can be edited using tokens |
| Argument | This editor will support textual as well as binary tree projections due to various reasons. The textual projection is used to explore the options of a real textual representation for projectional editors. The binary tree projection is significantly different to the textual representation and seems to be well fitted for the domain of mathematical formulas. A latex |

| | |
|---|---|
| | projection would face the same opportunities and weaknesses as the textual projection and the token projection has been discarded due to its lacking scientific contribution. |
| Implications | All projections can be implemented using the chosen technology stack |
| Related Decisions | ● DD-005: Presentation layer technology |
| Notes | - |

*Table F: Design Decision with ID: DD-002*

| DD-003: Client Server Architecture | |
|---|---|
| Issue | The editor in form of a web application can be implemented with different approaches of handling the workload, with either a stronger emphasize on the client or on the server. This DD is ought to clarify the separation of logic. |
| Decision | Smart client architecture |
| Status | Decided |
| Group | Technology |
| Assumptions | ● Browsers support all approaches |
| Positions | ● Thin client<br>Only little data processing is done on the client. Every operation is sent to the server to be carried out. This offers the advantage that crucial logic is executed on the server instead of the client which reduces the risk of malicious behavior caused by evil users. On the other hand this leads to fat servers which can become unfavorable when the need for scalability and maintainability is strong. Moreover the user experience suffers badly from this due to directly perceived latency between server request and response. Although template engines are a practicable way to separate business logic from presentation logic, this approach does not exert a force of separation of concerns. Generally speaking, this approach seems to become more and more outdated, although network bandwidth has rapidly increased while network latency has significantly decreased in recent years.<br><br>● Smart, thick or fat client<br>Smart clients are often used within a SOA landscape. The majority of data processing is executed on the client, which necessitates the browser to offer enough resources in terms of |

| | |
|---|---|
| | memory and CPU time, which can become crucial with complex, graphically enhanced user interfaces. This disburdens the server and eases the scaling of the application, but also leads to redundant code since malicious behavior has to be checked and identified twice - on the server and on the client. Moreover the possibility space for attacks is enlarged. The advantages of smart clients include a clean separation of code as well as a greatly improved user experience. This is achieved by asynchronously exchanged messages with the server via AJAX or similar technologies. In all, this leads to a desktop-application-like behavior of the web application. |
| Argument | A smart client seems to be the most appropriate option, because the principal focus is on the development of an editor, which clearly requires an extensive user experience like on desktop applications. Moreover a service oriented architecture perfectly fits into the need for multiple presentations which are decoupled from the abstract syntax tree. |
| Implications | A smart client necessitates a clear separation of functionality carried out at the server and functionality carried out at the client, depending on the actual domain in order to meet the requirements of the user. |
| Related Decisions | ● DD-010: Selection of a SOA pattern<br>● DD-011: Separation of server and client functionality |
| Notes | - |

*Table G: Design Decision with ID: DD-003*

| DD-004: Selection of an architectural pattern | |
|---|---|
| Issue | An architectural pattern has to be chosen for the structure of the code. |
| Decision | MVC |
| Status | Decided |
| Group | Pattern selection |
| Assumptions | ● The selected pattern has to fit the delivery platform of the editor |
| Positions | ● MVC / MVP / MVW<br>● Multitier architecture<br>● Layer<br>● Event-driven architecture<br>● Pipes and filters<br>● Service oriented architecture |

| | |
|---|---|
| Argument | MVC fits best the chosen technology stack and is very appropriate for the application since business logic is independent from the presentation related code of a projection. The model is manipulated using controllers, which fit the idea of manipulating an abstract syntax tree in a projection, but keep both sides encapsulated and independent [39]. |
| Implications | A logicless template engine would support this pattern. |
| Related Decisions | ● DD-001: Software delivery platform<br>● DD-003: Client Server Architecture<br>● DD-005: Presentation layer technology |
| Notes | - |

*Table H: Design Decision with ID: DD-004*

| DD-005: Presentation layer technology | |
|---|---|
| Issue | A bundle of presentation layer technologies has to be chosen to create the views for all projections. |
| Decision | Bootstrap (Sass) as CSS framework, Handlebars as template engine, jQuery for DOM operations |
| Status | Decided |
| Group | Technology |
| Assumptions | ● Browser support is of little importance |
| Positions | Since the editor will be deployed as web-based application, the near endless amount of technologies for online presentations available is relevant. For the purpose of this prototype at least three different building blocks have to be found:<br>• A CSS Framework for the basic layout<br>• A template engine for the rendering of views<br>• A Javascript library for DOM operations |
| Argument | Bootstrap is used to create the core layout of the editor. Handlebars serves as logicless template engine to enforce a clean separation of business logic and presentation logic. |
| Implications | The binary tree has to be created manually, without using a specific library |
| Related Decisions | ● DD-001: Software delivery platform<br>● DD-002: Selection of projections<br>● DD-004: Selection of an architectural pattern |

| | |
|---|---|
| Notes | For this decision a vast amount of options is available. The choice is based on technical characteristics as well as the Author's preferences and experience. |

*Table I: Design Decision with ID: DD-005*

| DD-006: Selection of a data storage | |
|---|---|
| Issue | To be able to persist and load formulas, a technology is needed which meets the requirements and offers the best fit. |
| Decision | For the purpose of this prototype NoSQL document databases are the most appropriate forms of persisting data. This perfectly fits into the selection of the message form as well as the technology stack. |
| Status | Decided |
| Group | Technology, Persistence |
| Assumptions | ● No technology is well-established and experience does not influence this decision<br>● The effort which is necessary to operate the persistence technology is not considered<br>● License costs are not considered |
| Positions | ● SQL Database<br>Data is stored among one or more tables and therefore has to be parsed into a predefined target schema. Since the schema of an abstract syntax tree is not fixed, a generic approach (like EAV modelling) is inevitable with this alternative. SQL Databases are well-tried and there is a large choice of tools available (Reporting, ORM Mapper ...). Additionally SQL is widespread and standardized.<br><br>● NoSQL Database<br>There are different types of NoSQL database technologies, which have in common to be at a certain degree schema free and do not use the relational model. Data is stored encapsulated which can lead to redundancy, since joins are expensive. For this decision only document-based databases are considered, because graph databases, column-value and key-value-stores are apparently less suitable. As already stated, document databases are schema free, which means that the data structure has not to be defined upfront and can vary from dataset to dataset; in our case from formula to formula.<br>NoSQL databases can be scaled easily and offer a high availability. Different vendors offer dissimilar qualities according to the CAP theorem. There is no standardized query language, |

| | |
|---|---|
| | even if some vendors offer a SQL-like syntax.<br><br>● Persistence File<br>Another option is to store formulas in persistence files, which is not completely different from NoSQL databases, where each document (=dataset) is stored in a separate file. In comparison to NoSQL databases, this approach offers even more flexibility and a higher degree of control options. On the other hand, this implies a manual handling of the persisted data, which leads towards a custom document based NoSQL database. |
| Argument | NoSQL databases perfectly fit into the selection of the message form as well as the technology stack. An EAV-model would be the only alternative way to store the information adequately. On the contrary, NoSQL databases offer a way to store the information “as it is” without the need of parsing and administration. |
| Implications | The persistence technology is highly connected with the format of the messages that are transferred between client and server. Furthermore the selection of frameworks and libraries effects this decision, since not all persistence technologies are equally supported by different platforms. |
| Related Decisions | ● DD-008: Message format<br>● DD-009: Selection of a server platform |
| Notes | MongoDB will be used as document-based database, since it is well documented and supports completely schema less documents. Moreover it is easy to use and does not require an extensive configuration. |

*Table J: Design Decision with ID: DD-006*

| DD-007: Data model | |
|---|---|
| Issue | The structure of the abstract syntax tree (data model) has to be suitable for the domain. |
| Decision | The model |
| Status | Decided |
| Group | Persistence |
| Assumptions | ● The structure has to support an evaluation of the formula using depth-first traversal |
| Positions | Each element in the abstract syntax tree is encapsulated in a discrete value object with specific attributes describing the characteristics of the |

| | |
|---|---|
| | node. Following model-classes can be used to capture the information of a node: Assignment, Identifier, Number, Expression, Function, Binary and Unary. All objects are extended by an identifier for specific DOM operations on the nodes. |
| Argument | The chosen model is capable of retaining the information of an abstract syntax tree for formulas |
| Implications | This structure can be used as it is for persistence and communication reasons |
| Related Decisions | ● DD-008: Message format<br>● DD-011: Selection of a messaging protocol |
| Notes | - |

*Table K: Design Decision with ID: DD-007*

| DD-008: Message format | |
|---|---|
| Issue | Information has to be exchanged between server and client. Therefore a standardized data structure format is needed to transmit human-readable data objects. |
| Decision | JSON |
| Status | Decided |
| Group | Persistence |
| Assumptions | ● Formulas do not have to be in a predefined structure in order to be portable to other applications<br>● All message formats are equally supported in all browsers |
| Positions | ● JSON<br>JavaScript Object Notation is a widespread standard used for client-server communication. JSON structures are readable and compact, but cannot be validated against a schema even if it is theoretically possible (JSON hyper schema). In addition, JSON is said to work well with REST [29].<br><br>● XML<br>Ideal for highly structured data, since data can be validated (against an XSD) and transformed easily (using XSLT). The overhead is higher and the readability slightly worse in comparison to JSON. XML is the basic message format for SOAP [28]. |

| | |
|---|---|
| Argument | JSON is the chosen message format, since data is highly unstructured which would contradict XML's advantages of validation and transformation. Moreover JSON documents are used in many NoSQL databases, which means that a transformation step can be axed. |
| Implications | - |
| Related Decisions | ● DD-007: Selection of a data storage<br>● DD-009: Selection of a server platform<br>● DD-011: Selection of a messaging protocol |
| Notes | - |

*Table L: Design Decision with ID: DD-008*

| DD-009: Selection of a server platform | |
|---|---|
| Issue | A server platform has to be selected which is suited for the architecture including the message and data formats. |
| Decision | Node.js |
| Status | Decided |
| Group | Technology |
| Assumptions | ● The knowledge base of the developer is irrelevant for this decision<br>● The operation of the server does not influence the decision |
| Positions | ● Node.js<br>Node.js's single threaded approach is particularly suitable for micro services faced by a large amount of short requests using Google's V8 JavaScript engine. Since JavaScript can be used on the server as well as on the client, JSON is natively supported on both ends. Moreover validation rules can be used on the server and on the client, leading to a reduced expenditure.<br>Disadvantages include a lack of mature modules as well as the drawbacks of the asynchronous non-blocking nature of Node.js, like the so-called callback-hell or bubbled up errors.<br><br>● Java (+ Spring)<br>JavaScript and Spring is well established in the world of web applications. Modules are mature and IDEs highly support the development of Spring applications. This implies that there is already an answer to every possible question leading to a way of gluing third party software together instead of writing own code. However, JSON has to be mapped to objects, which can become |

| | |
|---|---|
| | very complex and unhandy since the structure of an abstract syntax tree can vary substantially.<br><br>● PHP<br>Another option could be to use PHP, which enjoys great popularity when it comes to prototyping. The language is very simple and modules are highly available. It performs well with MySQL databases and can be coded object oriented and procedural. However, PHP is not suitable for large applications and code quality of PHP projects leave much to be desired. |
| Argument | A dynamically typed language seems to be more appropriate, which is why Node.js and PHP were considered further. Since Node.js offers a native handling of JSON documents as well as great support for service oriented architectures, the choice fell against PHP. |
| Implications | A database-driver has to be found. |
| Related Decisions | ● DD-007: Selection of a data storage<br>● DD-008: Message format |
| Notes | A build process should be set up beforehand, leveraging the JavaScript environment. Express can be used as web application framework to design the web interface. |

*Table M: Design Decision with ID: DD-009*

| DD-010: Selection of a SOA pattern | |
|---|---|
| Issue | A SOA pattern has to be selected for the client/server architecture. |
| Decision | SOFEA |
| Status | Decided |
| Group | Pattern selection |
| Assumptions | ● A web application is preferred to a desktop application |
| Positions | ● SOFEA (Service Oriented Front End Architecture)<br>After web templating engines and AJAX enhanced websites, SOFEA, often referred to as the concept of single page applications, is an architectural pattern where the UI functionality is implemented in the client, completely separated from the server. This step upgrades websites to web applications and offers the opportunity of interoperability. |

| | |
|---|---|
| Argument | The SOFEA pattern offers the best fit for a web application that provides an equally good user experience as desktop editors do. Furthermore it offers the opportunity to use the editor offline (HTML 5 offline storage). |
| Implications | The MVC pattern is implemented in the client, not the server. Instead of rendering views on the server, the application is downloaded as a whole or partially (asynchronous module definition) and executed in the browser. All communication is sent and received via webservices (either REST or SOAP). |
| Related Decisions | ● DD-003: Client Server Architecture<br>● DD-008: Message format<br>● DD-011: Selection of a messaging protocol |
| Notes | - |

*Table N: Design Decision with ID: DD-010*

| DD-011: Selection of a messaging protocol | |
|---|---|
| Issue | To transmit messages in the service oriented architecture, an appropriate protocol respectively webservice pattern has to be selected. |
| Decision | REST |
| Status | Decided |
| Group | Webservice Architecture |
| Assumptions | ● Libraries and frameworks are available for both options |
| Positions | ● REST (Representation State Transfer)<br>Unlike SOAP, REST is not a protocol but an architectural style for webservices, which relies on basic HTTP 1.1 verbs and therefore uses the HTTP protocol. REST works on URL basis and is very lightweight. It does not offer any possibility to provide information of the webservice, which is why third party libraries like swagger are used to describe the interface. The message format is not determined, whereas JSON is quite popular.<br><br>● SOAP (Simple Object Access Protocol)<br>SOAP relies exclusively on XML to transmit messages. SOAP works on operation basis, which is why it offers WSDL (web service description language) files providing a definition how the webservice can be consumed. |
| Argument | Although SOAP offers useful functionality to secure and document the |

| | |
|---|---|
| | web interface, this advantage is outweighed by the unhandy consumption of SOAP services using a web application. Moreover the editor's webservice is used for private communication between two parts of a single system in contrast to publicly offered services where SOAP shines. |
| Implications | - |
| Related Decisions | ● DD-007: Selection of a data storage<br>● DD-008: Message format |
| Notes | - |

*Table O: Design Decision with ID: DD-011*

### *3.3.5. Final state of the projectional editor*

The prototype has been adapted to be consistent with the design decisions and requirements to the editor. In this chapter, the application is presented along with technical details of the implementation.

Figure P shows the textual projection of a sample formula. As explained in Section 2.1, a textual projection causes difficulties in the manipulation of single operations on the abstract syntax tree. In this prototype, text is scanned 300ms after the last keystroke of a user, which implies that multiple operations can occur at the same time. Therefore the whole abstract syntax tree has to be completely replaced, which is not problematic in our case since MongoDB does not offer atomic operations on data anyways. The buttons on the upper corner of the textbox are for demonstration and debugging purposes and we will look closely on them in the next paragraph.

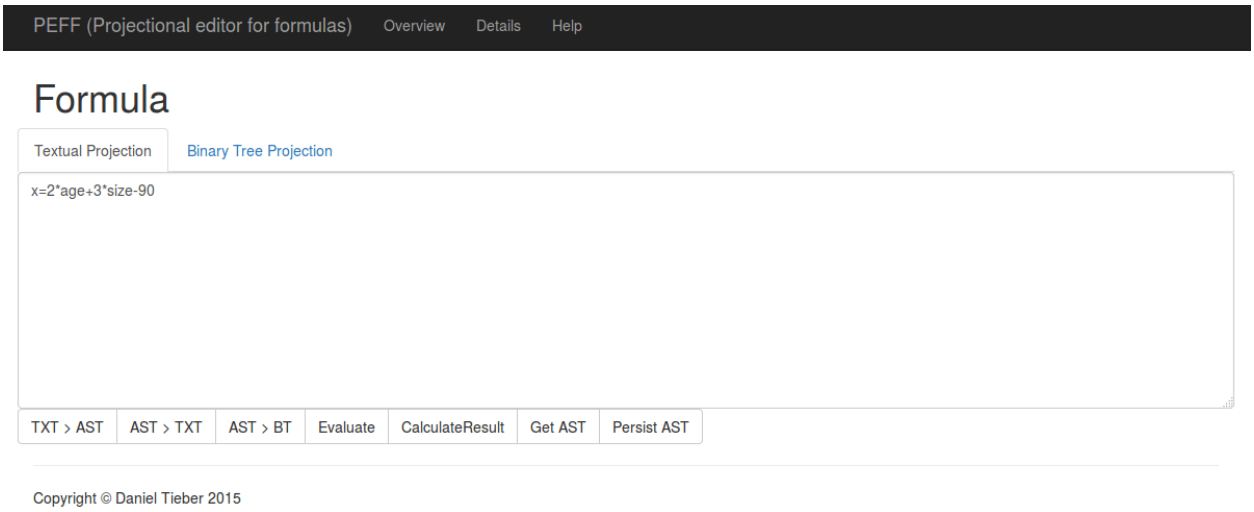



*Figure P: Sample formula in textual form*

The abstract syntax tree underneath this formula can be created either after a timeout or with a click on the first button “TXT>AST”. The abstract syntax tree is in JSON format and is presented in Code Snippet A. This transformation can also be vice versa, as the “AST>TXT” button indicates.

```
{
    "Assignment": {
        "name": {
            "Identifier": "x",
            "id": "f94a6483-ba43-d28d-c589-79616384b432"
        },
        "value": {
            "Binary": {
                "operator": "-",
                "left": {
                    "Binary": {
                        "operator": "+",
                        "left": {
                            "Binary": {
                                "operator": "*",
                                "left": {
```

```
                                    "Number": "2",
                                    "id": "e606fe38-cd72-ab1e-31de-
203bb652a97b"
                                },
                                "right": {
                                    "Identifier": "age",
                                    "id": "6839c3d0-e8e4-2636-884f-
40e190245ba2"
                                },
                                "id": "b998e7a7-9185-296e-a11d-
b47960499f22"
                            }
                        },
                        "right": {
                            "Binary": {
                                "operator": "*",
                                "left": {
                                    "Number": "3",
                                    "id": "e62c206e-b5ce-ef85-f5e2-
2bd27be67096"
                                },
                                "right": {
                                    "Identifier": "size",
                                    "id": "ca4a96c4-ff92-570c-9700-
bb7a33c80f78"
                                },
                                "id": "c2a6a502-650e-7283-818f-
cc7f400b94e1"
                            }
                        },
                        "id": "bb6e5e36-7665-9ef1-a2ef-06e89e8feb30"
                    }
                },
                "right": {
                    "Number": "90",
                    "id": "8b851155-2455-a5e5-16e3-0ae041474bb2"
                },
                "id": "c8cc3da6-fd59-aff7-6d4b-0d98295aa731"
            }
        },
        "id": "827cad57-1e0f-0cb4-0465-aedecc189e81"
    }
}
```

*Code Snippet A: An exemplary abstract syntax tree, created by the application*

The very same formula can also be displayed and edited in binary tree form, as depicted in Figure Q.

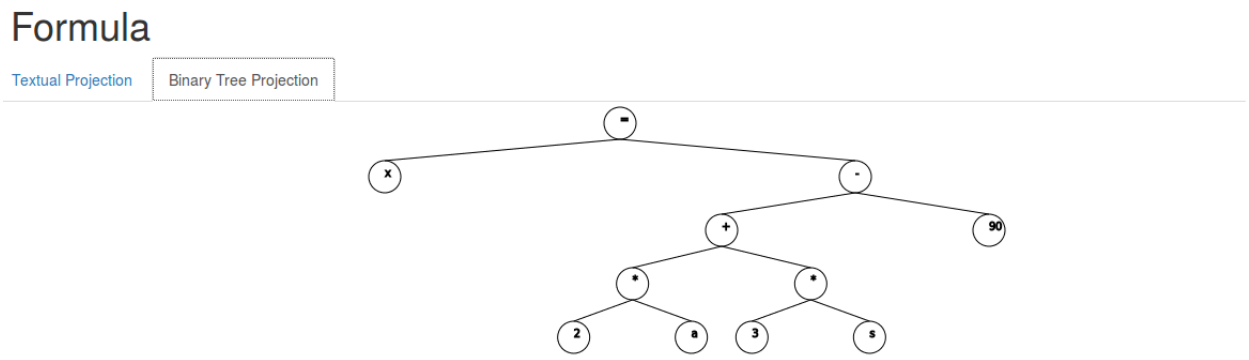
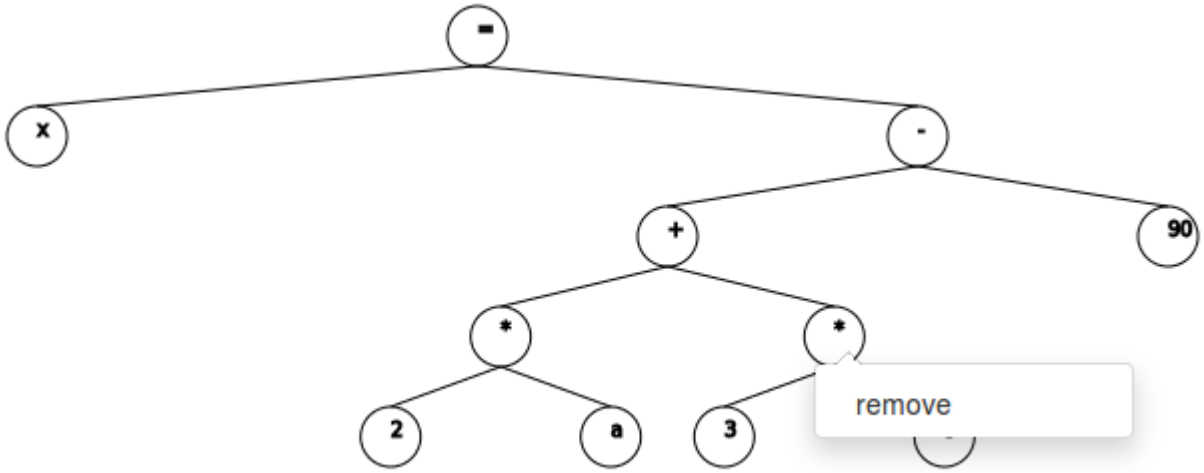
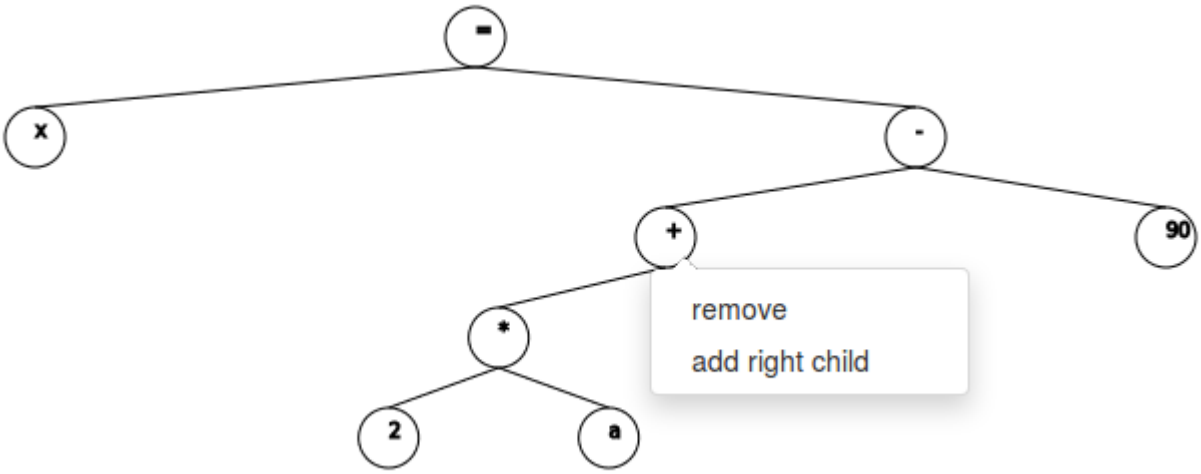
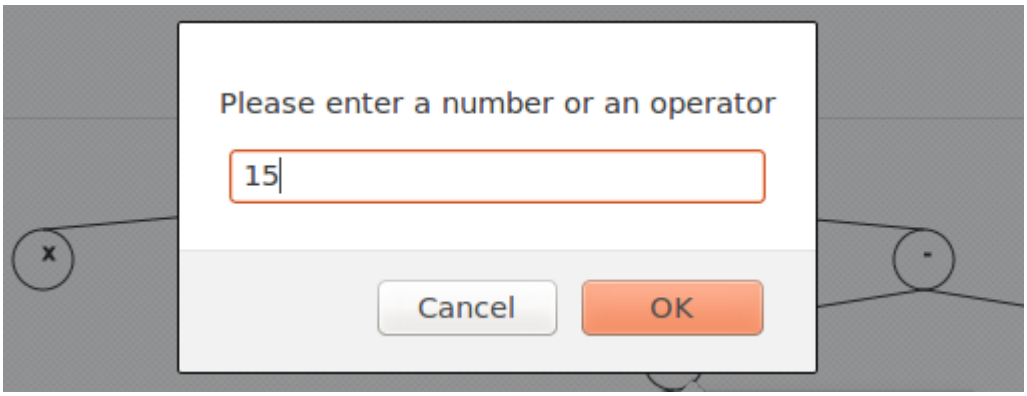



*Figure Q: Sample formula in binary tree form*

The illustration is rendered in an interactive HTML5 canvas element. Operations can be carried out based on nodes, as illustrated in the following image sequence (Figure R). A right-click on a node opens a context menu, containing all available operations.


=
x
-
+
90
*
*
2
a
3
remove


=
x
-
+
90
*
remove
add right child
2
a


Please enter a number or an operator
15
Cancel
OK
x
-

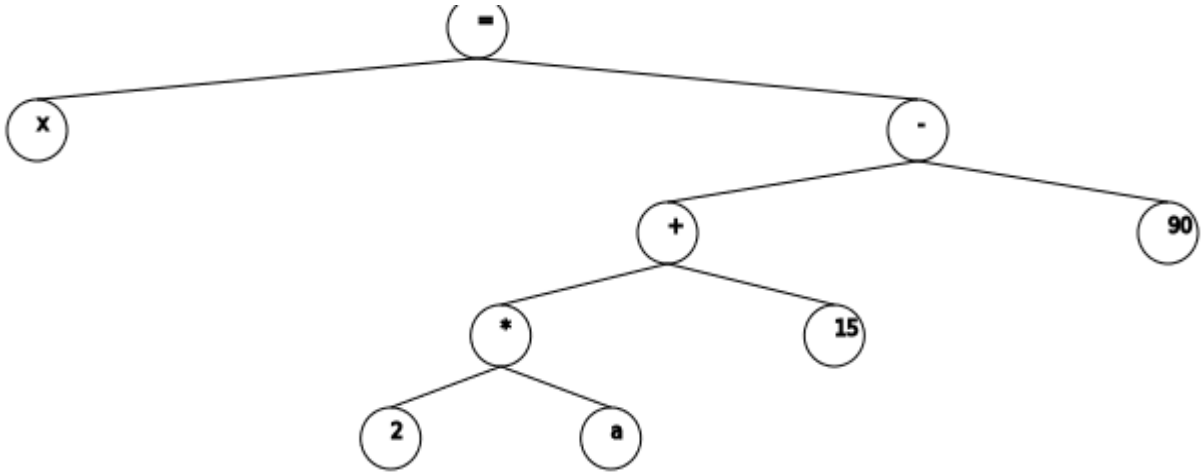



*Figure R: Editing a formula through the binary tree projection*

The corresponding textual projection of the abstract syntax tree is shown in Figure S.

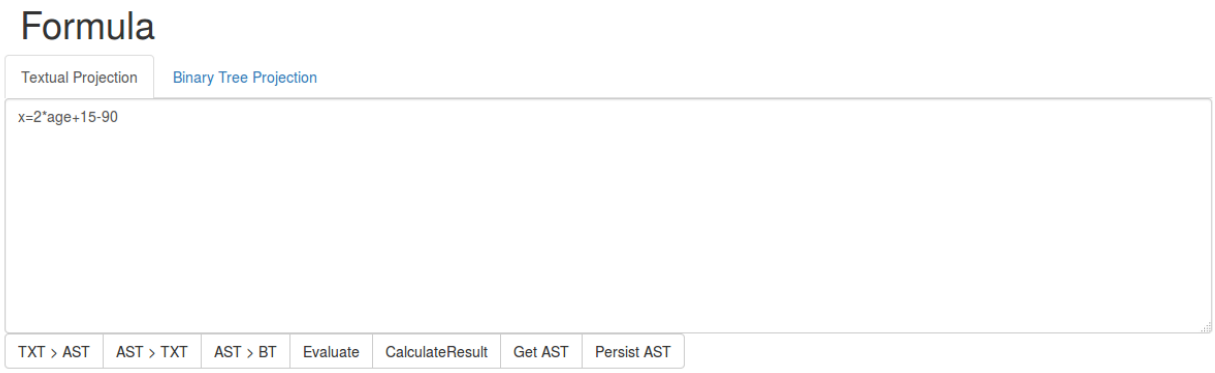



*Figure S: Textual projection of the updated formula*

The “Evaluate” Button displays a form-group containing input boxes for each dependent variable. After submitting values for each parameter the editor solves the equation on basis of the abstract syntax tree. This functionality can be seen as a transformation of an editable to an executable representation of the abstract syntax tree, comparable to the execution of an algorithm. The traversal used to execute the abstract syntax tree is called

inorder traversal. A pseudocode snippet of the algorithm can be found in Code Snippet B.

```
inorder-traversal(node) {
 if(node != 0) {
   inorder(node:left-child);
   process(node:value);
   inorder(node:right-child);
 }
}
```

*Code Snippet B: Inorder Traversal Pseudocode*

Finally, the formula can also be persisted using the “Persist AST” button, as shown in Figure T. A query on the formula collection is depicted in Figure U.


*Figure T: Persisting a formula*

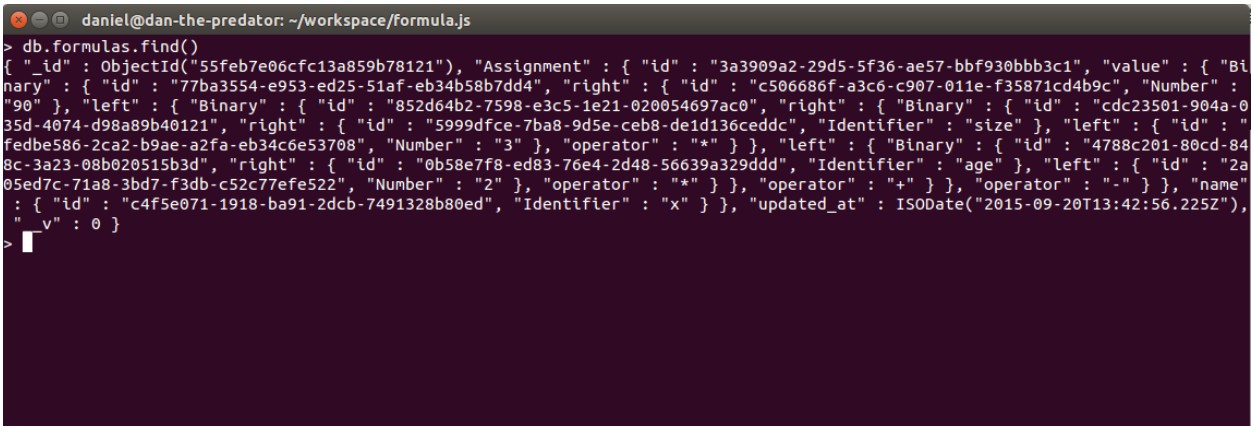



*Figure U: Database Query*

# 4. Guidance Model for the design and implementation of structured editors

This chapter synthesizes and summarizes the theoretical and empirical findings of the previous chapters as well as the key insights gained from the practical work on the prototype, in form of a guidance model. First, the segmentation of decisions is presented followed by the introduction of the guidance model. The guidance model is, as already said, a first approach to collect, sort and connect the occurring design decisions when designing and implementing a projectional editor. The author of this thesis strongly emphasizes the novelty of such a guidance model for projectional editors, concomitant with an early stage of maturity. Scientists and practitioners are encouraged to extend and modify this model.

## 4.1. Segmentation of decisions

According to the posed process in Chapter 2 it is regarded as imperative to define decision types in order to structure the architectural decisions of the model. The segmentation of topics is inspired by the conceptual decision model of the SOA design space [12] and is applied to the idiosyncratic characteristics of a projectional editor. In contrast to the categorization for decisions of the SOA guidance model, the column names have been adapted in order to express an increased focus on the decisions, accompanied by a less sharp focus on the communication and enforcement of the decisions made. Table P shows the segmentation used for this guidance model. The decisions are ordered top-down in regard to their severity, implying that wrong decisions

are most costly for the priory mentioned decision types, as already explained in previous chapters.

| Name | Examples | Audience | Phase |
|---|---|---|---|
| Executive decisions (ED) | Platform Selection, Architectural Style, Backlog Refinement | Project Lead, Enterprise Architects | Before project starts |
| Architectural decisions (AD) | Logical and Physical Separation, Persistence Strategy | Lead Architects, Technical Leads | Early design phase |
| Implementer's decisions (ID) | Patterns, Ordinary and Trivial Decisions | Lead Programmer Software Engineers | Throughout the project |

*Table P: Decision types (adapted from [12])*

## 4.2. Model

Having a categorization of decisions, the guidance model is ready to be created. As already mentioned, the model shown below in Table Q is based on theoretical and practical insights gained in the course of this thesis. The most common and reoccurring architectural issues are captured within the guidance model, others which are more domain-related and therefore more specific, have been omitted due to their lack of universality. Nevertheless, they are explained in Chapter 3.3.4 in detail. Since the guidance model attempts to be as general as possible it offers support in the decision making process when designing a projectional editor, through identified decisions, their alternatives and decision drivers.

| Type and ID | Description (Selection of…) | Alternatives (subset) |
|---|---|---|
| ED-01 | Software Delivery Platform | Web Application, Desktop application |

| | | |
|---|---|---|
| ED-02 | Selection of Projections | Textual, tree, tags, graphical models, tables, mixed |
| AD-03 | Technology Stack | Java, Node.js, C++, C#, PHP, Scala, Ruby and respective frameworks and tools |
| AD-04 | Persistence Strategy | NoSQL Database, SQL Database, XML-Files |
| AD-05 | Separation of Concerns | MVC, N-Tier, Layered |
| ID-06 | Presentation Layer Technology | Juce, Swing, HTML5, ECMAScript, CSS+Preprocessor, Template Engine, CSS Framework |
| ID-07 | Abstract Syntax Tree Structure | Serialization Strategy, Structure of the tree |

*Table Q: Guidance Model for Projectional Editors*

## 4.3. Related Design Decisions

In this chapter the decisions within the guidance model are explained in more detail. Each decision is captured in a table containing a more precise description of the actual issue along with alternatives, decision drivers and information about relationships to other decisions.

| **ED-01: Software Delivery Platform** | |
|---|---|
| Scope | Executive Decision |
| Phase | Requirements Specification |
| Description | This decision is related to the question whether the editor should be delivered in form of a web or desktop application. |
| Decision Drivers | Importance of: Updates, Platform Independence, Availability, Hardware Support, User Experience |
| Alternatives | • Web application<br>Editor is deployed on a server and accessible using browsers, which implies it can be accessed everywhere. |

| | |
|---|---|
| | Updates (and with that bug fixes) can be deployed easily on a centralized server and the client interface is platform independent “out of the box”. No installation is needed which lowers the user’s barriers to use the software. Nevertheless, it is browser dependent and has to deal with browser’s sandboxed nature. High level of user experience can be provided using modern web technologies.<br><br>• Desktop application<br>Has to be downloaded and executed on the user’s machine. This brings the advantage that the application is able to communicate with the client’s hardware. A permanent internet connection is not required. Bug fixes and updates have to be installed on every client. |
| Questions raised | • Web application<br>  ○ Client/Server Architecture<br>  ○ Message format/protocol<br>  ○ Server platform<br>• Desktop<br>  ○ Platform independence<br>  ○ Update strategies |
| Relationships | - |

*Table R: ED-01 Software Delivery Platform*

| **AD-02: Selection of Projections** | |
|---|---|
| Scope | Architectural decision |
| Phase | Requirements Specification |
| Description | The selection of projections is especially crucial for the success of a projectional editor. The advantages gained through miscellaneous projections have to outweigh the disadvantages of the use of an unfamiliar editor. |
| Decision Drivers | Domain related aspects of the underlying data, user experience |
| Alternatives | It is not possible to list all available possibilities. An incomplete list of options could look like this:<br>• Assisted text insertion<br>Works on basis of tokens that enforce a valid syntax<br>• Tabular insertion<br>Used for multi-branched decision making structures<br>• Graphical notation<br>Uses symbols and icons to display information |

| | |
|---|---|
| | • Tree based manipulation<br>Information arranged in tree structures |
| Questions raised | - |
| Relationships | ID-06: Which presentation layer technology is appropriate to provide the user with the selected projections?<br>ID-07: How should the storage representation be structured in order to appropriate for the selected projections? |

*Table S: ED-02 Selection of Projections*

| **AD-03: Technology Stack** | |
|---|---|
| Scope | Architectural decision |
| Phase | Requirements Specification |
| Description | Which technology stack serves best the needs of the editor? |
| Decision Drivers | Experience, license costs, user (experience) requirements, availability of support, interface compatibility, operations, performance, testability |
| Alternatives | All available technologies able to fulfill the objectives of the application to-be |
| Questions raised | - |
| Relationships | ED-01: Depending on whether the application should be delivered as desktop or web application, the technology stack varies widely |

*Table T: AD-03 Technology Stack*

| **AD-04: Persistence Strategy** | |
|---|---|
| Scope | Architectural decision |
| Phase | Requirements Specification |
| Description | To be able to persist and load the abstract syntax tree, a technology is needed which meets the requirements and offers the best fit. |
| Decision Drivers | Available Technologies, Experience (usage and operations), License Costs |

| | |
|---|---|
| Alternatives | • SQL Database<br>Data is stored among one or more tables and therefore has to be parsed into a predefined target schema. Since the schema of an abstract syntax tree is not fixed, a generic approach (like EAV modelling) is inevitable with this alternative. SQL Databases are well-tried and there is a large choice of tools available (Reporting, ORM Mapper ...). Additionally SQL is widespread and standardized.<br><br>• NoSQL Database<br>There are different types of NoSQL database technologies, which have in common to be at a certain degree schema free and do not use the relational model. Data is stored encapsulated which can lead to redundancy, since joins are expensive. For this decision only document-based databases are considered, because graph databases, column-value and key-value-stores are apparently less suitable. As already stated, document databases are schema free which means that the data structure has not to be defined upfront and can vary from dataset to dataset; in our case from formula to formula.<br>NoSQL databases can be scaled easily and offer a high availability. Different vendors offer dissimilar qualities according to the CAP theorem. There is no standardized query language, even if some vendors offer a SQL-like syntax.<br><br>• Persistence File<br>Another option is to store formulas in persistence files, which is not completely different from NoSQL databases, where each document (=dataset) is stored in a separate file. In comparison to NoSQL databases, this approach offers even more flexibility and a higher degree of control options. On the other hand, this implies a manual handling of the persisted data, which leads towards a custom document based NoSQL database. |
| Questions raised | • Storage format (data model) |
| Relationships | AD-03: Depending on the selected technology stack one persistence strategy or the other may be better or worse supported in respect of drivers, libraries and frameworks.<br>ID-07: The structure of the abstract syntax tree could also have an impact on this decision. |

*Table U: AD-04 Persistence Strategy*

| AD-05: Separation of Concerns | |
|---|---|
| Scope | Architectural decision |
| Phase | Requirements Specification |
| Description | How can the code of modules of the application be separated best in regard of the maintenance, expandability and testability of the application? |
| Decision Drivers | Structure of the application, experience, existing modules |
| Alternatives | Different approaches are possible; for instance:<br>• MVC / MVP / MVW<br>• Multitier architecture<br>• Layer<br>• Event-driven architecture<br>• Pipes and filters<br>• Service oriented architecture |
| Questions raised | - |
| Relationships | ED-01: Is the technology stack suited for the delivery platform?<br>AD-03: Is the technology stack appropriate for the chosen architectural style? |

*Table V: AD-05 Separation of Concerns*

| ID-06: Presentation Layer Technology | |
|---|---|
| Scope | Implementer's decision |
| Phase | Requirements Specification |
| Description | Which (bundle of) presentation layer technologies is suitable to display the selected projections? |
| Decision Drivers | User experience requirements, experience |
| Alternatives | There is a huge amount of presentation layer technologies available, for almost every kind of platform. Some popular choices include:<br>• Web template engines<br>Haml, Mustache, handlebars, Jade, Thymeleaf, Apache Velocity, Django<br>• CSS frameworks<br>Bootstrap, Compass, 960gs, Foundation, Jeet<br>• CSS preprocessors |

| | |
|---|---|
| | Stylus, Sass, Scss<br>• GUI frameworks<br>in the Java world: AWT, Swing, SWT, JavaFX, Apache Pivot<br>in the C++ world: Qt, Kiwi, gtk, Juce<br>in the C# world: GTK#, QtSharp, Qyoto |
| Questions raised | - |
| Relationships | AD-02: Self-evidently this decision is related to the selection of projections which have to be supported |

*Table W: ID-06 Presentation Layer Technology*

| **ID-07: Abstract Syntax Tree Structure** | |
|---|---|
| Scope | Implementer's decision |
| Phase | Requirements Specification |
| Description | How can the data of the abstract syntax tree be modelled and organized? |
| Decision Drivers | Performance requirements, readability, underlying information |
| Alternatives | Depends on the domain |
| Questions raised | - |
| Relationships | AD-04: The data model has to emphasize the advantages of the chosen persistence strategy. |

*Table X: ID-07 Abstract Syntax Tree Structure*

# 5. Discussion and Summary

The acquired guidance model for the implementation of projectional editors has to be seen as an entrance point for software engineers, especially software developers, designers, and architects who are planning the development of such an editor. Up to this point the author knows of no other implementation of a projectional editor in form of a web application, which in turn explains the focus on web-based applications. In contrast to the SOA guidance model, which is the guidance model for service oriented architectures and has already been mentioned in a previous chapter, there is neither a reference model nor a reference implementation for projectional editors available, which is why the guidance model is based primarily on the prototype enriched by insights gained from the literature review in order to draw a complete picture of the arising decisions. This may be caused by the circumstance that an highly technological driven decision model, as the SOAD is, is much more based on domain-independent decisions, rather than preferences depending on business or domain related drivers. Moreover the design options for a projectional editor are significantly influenced by previous decisions, leading to a variety of decision paths where decisions might or might not arise, contingent on decisions made before. Under these given conditions, it was necessary to increase the degree of abstraction in order to be able to capture the complexity of the architectural design process of a projectional editor. This circumstance is shown through the differences between the design decisions made for the prototype and the more abstract guidance model for projectional editors in the subsequent chapter, which covers less, but more general decisions.

With reference to projectional editing in general, the author takes the liberty at this point,

to emphasize the possibilities of projectional editors with the emergence of new technologies on the one hand as well as changed needs and requirements to information technology on the other hand. As analyzed in this thesis, the raise of web technologies and increased capabilities will lead and already led to a radical shift in the landscape of applications; away from desktop applications used by a single user, to collaborative tools which are available instantaneously and everywhere. Beyond that, from a commercial and economical point of view, information technology undergoes a rapid change from a vertical division to a horizontal, crosscutting parts of many industries. This causes a higher need for participation of domain experts which are likely to be non-technicians, unable to work with today's tools of software engineering. Projectional editors would be able to close this gap between software engineers and domain experts. Nevertheless, the improvement of projectional editors has been sluggish in the past, maneuvering projectional editors in the corner of niche products. However, this is, according to the opinion of the author, caused by the comparison of projectional to classical editors on basis of characteristic qualities favoring classical editors while ignoring exciting opportunities of projectional editing, which in turn is significantly based on different levels of experience.

From an implementer's point of view, the realization of a projectional editor was accompanied by several challenges. The selection of a platform, for instance, seemed to be not of tremendous importance in the beginning, but a wrongly taken decision became a blocker in the implementation leading to a shift in technology. The selection of a platform technology should be highly connected with the separation of concerns and the implications arisen. One of the most important and challenging architectural decision was

the design of the abstract syntax tree, since it has to store the information in its entirety, regardless of projections. Mistakes made during the data modelling will lead to further impurities in the persistence layer as well as effects on the performance of manipulation and execution of the abstract syntax tree. Last but not least, the selection of projections is not only crucial for the success of the editor, but is also a decision which has to be exceptionally elaborated from the viewport of the target audience and the environment in terms of domain specific characteristics. The crucial questions for the selection of projections could be: “Which projections offer a useful abstraction of information?” or “How can certain aspects of data be simplified”. As a side node, a textual projection implies a tradeoff between projectional and classical editing as well as highly sophisticated techniques ensuring a good user experience.

In conclusion, this thesis envisions the opportunities of projectional editing on the one hand and is aimed to provide support for practitioners and scientists facing the challenges occurring when designing a projectional editor on the other hand. This is the first attempt of implementing a projectional editor in form of a web application, according to the knowledge of the author.

Further research work might use this guidance model derived from theoretical as well as practical insights, to enhance it and to explore undiscovered options. Moreover, collaborative editing, or technically speaking operational transformation, bears an obvious and remarkable resemblance to projectional editing. Therefore, the collaborative manipulation of information, like already prevalent in text processors like Microsoft’s Word or Google’s Docs, could also become reality for integrated development environments, which is why it could play a major role in further research.

# List of Tables

# List of Code Snippets



# List of Figures

# List of Literature

[1] Schmidt, Douglas C. 2006. Model-driven engineering. Computer 39 (2): 25-31.

[2] Yourdon, Ed. 2001. Can XP Projects Grow?. Computerworld 35 (30): 28-29.

[3] Völter, Markus. 2003. DSL Engineering: designing, implementing and using domain-specific languages.

[4] Fowler, Martin. 2010. Domain-Specific Languages.

[5] Völter, Markus, Siegmund, Janet, Berger, Thorsten, Kolb, Bernd. 2014. Towards User-Friendly Projectional Editors.

[6] Fowler, Martin. "Projectional Editing." Martinfowler.com. January 14, 2008. Accessed August 23, 2015. <http://martinfowler.com/bliki/ProjectionalEditing.html>

[7] Völter, Markus. 2011. From Programming to Modeling - and Back Again. IEEE Software 28 (6): 20-25.

[8] Vogel, Oliver, Arnold, Ingo, Chughtai, Arif, Ihler, Edmund, Kehrer, Timo, Mehlig, Uwe, Zdun, Uwe. 2009. Software-Architektur. Grundlagen - Konzepte - Praxis. Spektrum Akademischer Verlag Heidelberg

[9] Tyree, Jeff, Akerman, Art. 2005. Architecture Decisions: Demystifying Architecture. Software, IEEE 22 (2): 19-27.

[10] Zimmermann, Olaf. 2011. Architectural Decisions as Reusable Design Assets. Software Architectures. 4880: 15-32.

[11] Zimmermann, Olaf. 2010. An Architectural Decision Modeling Framework for SOA and Cloud Design. Presentation. IBM Research – Zurich.

[12] Zimmermann, Olaf,  Gschwind, Thomas, Küster, Jochen, Leymann, Frank, Schuster, Nelly. 2003. Reusable Architectural Decision Models for Enterprise Application Development.

[13] Zimmermann, Olaf, Zdun, Uwe, Gschwind, Thomas, Leymann, Frank. 2008. Combining Pattern Languages and Reusable Architectural Decision Models into Architectural decisions as

reusable design assets. Proceedings of the WICSA/ECSA 2012 Companion Volume (WICSA/ECSA '12).

[14] Zimmermann, Olaf, Zdun, Uwe, Gschwind, Thomas, Leymann, Frank. 2008. Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method. Software Architecture: 157-166.

[15] Könemann, Patrick, Zimmermann, Olaf. 2010. Linking Design Decisions to Design Models in Model-Based Software Development. Springer Verlag Berlin Heidelberg.

[16] Kruchten, P. B. 1995. The 4+1 View Model of architecture. IEEE Software 12 (6): 42-50.

[17] Case, Albert. 1985. Computer-aided software engineering (CASE): technology for improving software development productivity. Data base 17 (1): 35-43.

[18] Chen, M. 1989. Computer-aided software engineering: present status and future directions. Data base 20 (1): 7-13.

[19] "The Java Language Environment." The Java Language Environment. Accessed October 16, 2015. <http://www.oracle.com/technetwork/java/intro-141325.html>.

[20] Nathan-Regis, Bodje N. 2012. EVALUATION OF THE MOST USED AGILE METHODS (XP, LEAN, SCRUM). International journal of engineering science and technology 4 (1): 23.

[21] Trepper, Tobias. (2012). Agil-systemisches Softwareprojektmanagement. Springer Fachmedien Wiesbaden.

[22] Schmidt, D. C. 2006. Model Driven Engineering. Computer 39 (2): 25-31.

[23] Pastor, Oscar. 2008. Model-Driven development. Informatik-Spektrum 31 (5): 394.

[24] France, Robert. 2014. Model-Driven Development of Complex Software: A Research Roadmap.

[25] Fowler, Martin. 2011. Domain-Specific Languages. Pearson Education Inc.

[26] Voelter, Markus. 2015. Projecting a Modular Future. IEEE Software 32 (5): 46-52.

[27] Juhar, Jan, Vokorokos, Liberios. 2015. Understanding source code through projectional editor. 13th International Conference on Engineering of Modern Electric Systems,

[28] Mintert, S. 2005. Implementation of Web services - REST vs. SOAP?. Wirtschaftsinformatik 47 (1): 63-65.

[29] Richardson, Leonard. 2007. Web Services mit REST. O'Reilly.

[30] Imine, Abdessamad, and Farn Wang. 2005. Towards synchronizing linear collaborative objects with operational transformation. Formal Techniques for Networked and Distributed Systems - FORTE 2005. Springer Berlin / Heidelberg.

[31] Sun, Chengzheng. 1998. Operational transformation in real-time group editors: issues, algorithms, and achievements. In Computer supported cooperative work Proceedings of the 1998 ACM conference. ACM.

[32] Osorio, Jorge A. 2011. Moving from Waterfall to Iterative Development: An Empirical Evaluation of Advantages, Disadvantages and Risks of RUP.

[33] Mitchell, Susan M. 2009. A comparison of software cost, duration, and quality for waterfall vs. iterative and incremental development: A systematic review.

[34] Fu, Jicheng. (2011). Model-Driven Development: Where Does the Code Come From? 2011 IEEE Fifth International Conference on Semantic Computing.

[35] Chen, Jimq. 2001. Building Web Applications: Challenges, Architectures, and Methods. Information systems management 18 (1): 68-79.

[36] Wasserman, Anthony. 2011. How the Internet transformed the software industry. Journal of Internet Services and Applications 2 (1): 11-22.

[37] Taivalsaari, Antero. (2011). The Web as an Application Platform: The Saga Continues. 37th EUROMICRO Conference on Software Engineering and Advanced Applications.

[38] Taivalsaari, Antero (2008). Web Browser as an Application Platform. 34th Euromicro Conference Software Engineering and Advanced Applications.

[39] Fowler, Martin. "GUI Architectures." Martinfowler.com. July 18, 2006. Accessed October 25, 2015. <http://martinfowler.com/eaaDev/uiArchs.html>

# Appendix

## German Abstract

Die vorherrschende Vorgehensweise um Software zu entwickeln und warten nennt sich Source Code Editing und basiert auf dem Ansatz Applikationen Zeile für Zeile zu bearbeiten. Da Informationssysteme zunehmend komplex einerseits und zunehmend bedeutend anderseits werden, ist davon auszugehen, dass in Zukunft unterschiedlichste Akteure verstärkt in den Schaffungsprozess einer Applikation eingebunden werden müssen. Projektionales Editieren stellt eine viel versprechende Alternative zu Source Code Editing dar, die darauf basiert, dass Software unter Zuhilfenahme unterschiedlicher Projektionen bearbeitet werden kann. Dies ermöglicht es einerseits auch Nicht-Technikern an dem Schaffungsprozess zu partizipieren und unterstützt andererseits Techniker durch die Abstraktionsmöglichkeit komplexer Information. Basierend auf einer Literaturrecherche und der Implementation eines Prototypen, legt diese Masterarbeit den Schwerpunkt auf die Implementation von Projektionalen Editoren und bietet dem Lese ein Guidance Model aus wiederkehrenden Design Decisions inclusive einer Vorstellung von verfügbaren Alternativen und Entscheidungstkriterien.

## English Abstract

The predominant approach to software development and maintenance is to write source code line by line within a text editor. Since information systems become more and more important in everyday reality, the number of people participating in software development processes will increase steadily. Projectional editing is a promising alternative to source code editing which enables software engineers to create and maintain software using different projections of the same application. This enables non-technicians to participate on the one hand and supports engineers by shielding complexity on the other hand. This master thesis is based on related work as well as the implementation of a prototype and focuses on the implementation of

projectional editors, providing the reader with an initial guidance model comprised of recurring design decisions including practicable alternatives and decision drivers.