# Development and Evaluation of a Graphical Notation for Modelling Resource-Oriented Applications

Masterarbeit
im Studiengang
Master of Science im Fach Informatik

von

**Dipl.-Inf. (FH) Oliver van Porten**

(Matrikelnummer: 7694849)

vorgelegt der
Fakultät für Mathematik und Informatik
der FernUniversität in Hagen

| | |
|---|---|
| **Erster Prüfer:** | Prof. Dr. Bernd Krämer<br>Lehrgebiet Datenverarbeitungstechnik<br>Fakultät für Mathematik und Informatik |
| **Beginn der Arbeit:** | 26. September 2011 |
| **Abgabe der Arbeit:** | 26. März 2012 |

Ich erkläre hiermit, die folgende Masterarbeit selbständig verfasst zu haben. Andere als die angegebenen Quellen und Hilfsmittel habe ich nicht benutzt. Wörtliche und sinngemäße Zitate sind kenntlich gemacht.

Roetgen, den 26. März 2012

Oliver van Porten

# Zusammenfassung

Die Modellierung von Anwendungen mit Hilfe von grafischen und textuellen domänenspezifischen Sprachen (DSL) ist fortwährend Gegenstand von Forschung und Entwicklung im Bereich des Software Engineering. DSLs steigern die Produktivität der Entwickler und erleichtern die Kommunkation mit den Domänenexperten.

Zur modellgetriebenen Entwicklung ressourcenorientierter Anwendungen exisitert noch keine hinreichende Unterstützung durch spezialisierte Werkzeuge. Der Einsatz von generischen Modellierungssprachen wie UML ist möglich, bildet aber die speziellen Bedürfnisse des ressourcenorientierten Ansatzes nicht hinreichend ab. Spezielle Unterstützung für die grafische Modellierung von ressourcenorientierten Anwendungen ist hier wünschenswert.

Gerade bei der Entwicklung grafischer Notationen wird aber selten Wert auf eine kognitiv effiziente Syntax gelegt. Der semantischen Modellierung wird eine weit größere Bedeutung zugemessen. In seiner Arbeit “The physics of notations” stellt Daniel Moody daher neun Prinzipen vor, die es erlauben, schon beim Entwurf einer neuen grafischen Notation das Augenmerk stärker auf gute Syntax zu legen. Diese neun Kriterien bilden dabei ein modulares Rahmenwerk.

Die vorliegende Arbeit entwickelt unter Zuhilfenahme dieser neun Kriterien eine grafische Notation zur Modellierung von ressourcenorientierten Anwendungen. Die Notation basiert auf einem gegebenen Metamodell und bietet verschiedene Sichten auf das Meta-Modell an. Diese Sichten stellen dabei verschiedene Teilbereiche der konkreten Modells der modellierten Anwendung dar.

Die Güte der Umsetzung der neun Kriterien in der entwickelten Sprache wird anschließend in Form eines Fragebogens überprüft und die Ergebnisse der Evaluation werden diskutiert.

Weiterhin wird eine auf Eclipse basierende prototypische Implementierung eines Editors zur Modellierung ressourcenorientierter Anwendungen mit Hilfe der neu entwickelten grafischen Notation vorgestellt.

# Abstract

Modeling applications using graphical or textual domain-specific languages (DSL)) is a constant subject of research and development in modern software engineering. DSLs improve productivity of the developers and aid in bridging the gap to domain experts.

However, model-driven development of resource-oriented applications is not well supported through specialized tools yet. Using generic modeling languages like UML is possible. Regrettably, this does not reflect the special needs of the resource-oriented approach to application design. Dedicated support for the development of such applications is desirable.

Especially, the development of graphical languages suffers from a lack of focus on cognitive effective syntax. Modeling semantics is regarded with far more importance than modeling syntax. In his work "The physics of notations", Daniel Moody therefore proposes nine principles that aid in the development of proper graphical syntax. These nine principles form a modular framework for the development of a cognitive effective visual language.

The work at hand presents a new graphical notation for the development of resource-oriented applications, built on the nine principles postulated by Daniel Moody. The notation is based on a given meta-model and offers distinct views on distinct parts of that meta-model. The views together form a consistent representation of the concrete model instance of the modeled application.

To assess how well the design principles were incorporated in the design of the new notation, an evaluation using a questionnaire is performed. The results of the survey are presented and discussed.

Finally, the prototypical implementation of an Eclipse-based editor for modeling resource-oriented applications based on the new notation is presented.

# Contents

# List of Figures

# List of Tables

# Listings

# 1 Introduction

Today, Software Engineering is often concerned with the development of domain-specific solutions in the form of textual and graphical domain-specific languages (DSL) [1, 2] to provide domain users with tools tailored towards their specific needs. Those tools are generally meant to help bridge the gap between domain specialists and developers.

Representational State Transfer (REST) [3] is the basic architectural style behind the Hypertext Transfer Protocol (HTTP). In the context of application development for the World Wide Web (WWW), applying REST to an application's architecture can help reduce coupling and improve overall application performance [4]. However, there is little support for modeling resource-oriented (or RESTful) applications. While there are many tools and models to support the development of object-oriented applications, such as the Unified Modeling Language (UML) [5], there are little to no tools to provide the same level of support for resource-oriented applications. UML could be used to model resource-oriented applications as well, but it is not specifically tailored towards it.

To improve support for modeling resource-oriented applications, specialized notations and tools need to be developed. The meta-model for resource-oriented applications [6] forms the semantic basis for building the required graphical or textual languages in the domain of resource-oriented applications. Generally, a meta-model defines the semantic constructs of a problem domain. In this case, these are the different types of resources and their various properties such as attributes, links and also their states and behavior.

An initial approach to define the syntax of a new graphical language could try to use UML as the basis and then redefine it to some extent to enable modeling of resource-oriented applications. However, UML has been criticized for its lack of good perceptual properties [7]. Therefore, a completely new graphical language needs to be developed that is not constrained by any of UML's undesirable properties. To improve the quality of visual languages, Daniel Moody [8] defines some key principles that establish a prescriptive theory which provides the needed guideline in such development efforts.

The objective of this thesis is to define and evaluate such a visual language that allows modeling resource-oriented applications. The notation adheres to the visual guidelines defined by Daniel Moody [8] and is based on the REST meta-model [6]. An evaluation is performed to assess how well the principles are applied to the new visual language.

Finally, a prototypical editor based on Eclipse [9] is developed that implements the new graphical notation.

The context of this thesis is provided in chapter 2 by highlighting related work from various fields of research and practice.

Chapter 3 focuses on the graphical language "Visual REST" which is developed as part of this thesis. The notational elements of the Visual REST modeling language are introduced and the different views the language uses for modeling resource-oriented applications are outlined. Additionally, a discussion of the visual properties of the notation is provided and a rationale for the design decision made is given. The chapter concludes with some examples of applications modeled using Visual REST.

Subsequently, chapter 4 presents the details of the language evaluation that was conducted. An insight into the way the questionnaire was developed, the questions that were asked and into the targeted audience is provided. A discussion of the survey results is also presented there, as are conclusions drawn from them. Appendix A contains the complete questionnaire for reference. Appendix B contains the comments that were provided by probands.

The prototypical implementation of the Eclipse-based editor and details of its design and development are covered by chapter 5. All aspects ranging from the chosen frameworks, general architecture and plug-in organization to testing and build aspects are covered there. The chapter also highlights the features of the editor that help working with Visual REST. Appendix C contains some source code statistics extracted from the editor and appendix D contains some source code examples, demonstrating various implementation aspects.

Finally, chapter 6 provides a short summary of the work presented here and sums up findings and observations made in the course of this thesis. Some ideas for future work are also presented in that chapter.

# 2 Related Work

This chapter presents an overview of related work and puts the work done here into context. Related work ranges from model-driven development aspects and REST to visual language design and questionnaire development. Also an important part is work related to the implementation of the graphical editor that is developed as part of this thesis.

## 2.1 Model-Driven Development

Model-Driven Development (MDD) [2, 10–12] is an important area in the field of software engineering where DSLs are applied. The goal of any model-driven software development project is the definition of a formal model that allows abstraction of implementation specific aspects. A model is usually not tailored towards a specific execution environment but can be used to generate runnable software using model transformations [2].

Every DSL [1] is backed by a meta-model that describes the problem domain. The DSL provides the syntax to enable practitioners to use the meta-model in one way or another. A DSL can either be textual or graphical [1, 13]. This thesis focuses on graphical notations.

## 2.2 REST

Representational State Transfer (REST) is the architectural style driving the World Wide Web (WWW) and was defined by Roy Fielding in his dissertation [3]. It defines resources and their relation to one another using links. It uses "hypermedia as the engine of application state" (HATEOAS) [3]. Resources offer a set of standard methods, have different representations and use stateless communication. The Hypertext Transfer Protocol (HTTP) [14] together with Uniform Resource Identifiers (URI) [15] and the different media types [16, 17] are a concrete manifestation of the REST architectural style [4].

Every REST resource is identified by a unique identifier, its Uniform Resource Identifier (URI) [15]. In terms of the World Wide Web this is, for instance, a Uniform Resource Loca-

tor (URL) [18] identifying a resource on a server. A URL is a sub-type of the generic URI. For example, `http://www.examples.com/article/20` identifies one particular article on the server `example.com`. Compared to object-oriented software development or classic web service approaches based on SOAP [19], the resource interfaces differ from those approaches in that they use a uniform interface. While SOAP and object-oriented software development use a method centric interface—every object defines a set of methods that can be invoked on it—REST defines a fixed set of operations that can be performed. For HTTP this includes all standard verbs like POST, GET, PUT, DELETE, HEAD and OPTIONS. There are RESTful protocols that extend those verbs with additional ones; Subversion [20] is an example [4].

While Roy Fielding's dissertation offers a rather abstract view on REST, Stefan Tilkov defines a set of certain resource types that need to be identified in a problem domain [4]. Those resource types form the fundamental elements of every RESTful or resource-oriented application.

## 2.3 The REST Meta-Model

Modeling the semantics of a certain problem domain is the first step in enabling the development of domain specific languages. The meta-model forms the semantic basis for model-driven development [21]. For REST, Schreier [6] defines the meta-model that is the basis for this work. The REST meta-model is split into two parts, the structural model and the behavioural model.

The structural model defines the overall structure of the system with its main concept being different resource types as identified by [4] (except "conceptual resources"). Figure 2.1 provides an overview of the different types.

The structural model also defines properties of the resource types. Every resource type has a unique `ResourceIdentifierPattern` that can either be a `SimpleIdentifier` or a `ComplexIdentifierPattern` containing attributes and parameters. A resource type has a defined set of `Methods` of certain `MethodTypes` which have produced and consumed `MediaTypes`. A `ResourceType` also has `ResourceElements` which can either be `Links`, `LinkCollections` or `Attributes`. Figure 2.2 gives an overview of the different relations and concepts.

As shown in figure 2.1 there are eight distinct concrete resource types (sub-types of the abstract `ResourceType`) available in the meta-model.

**Primary Resource Types** typically represent one of the main business objects of the ap-

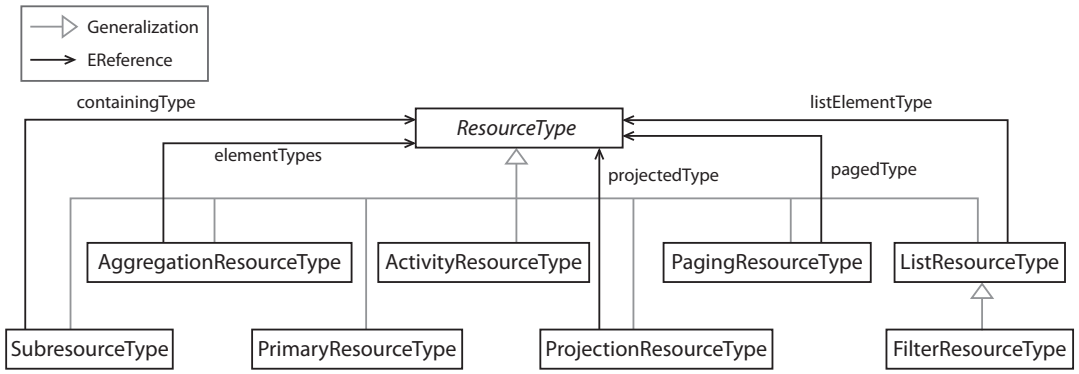



Figure 2.1: The resource types of the meta-model [6]

plication. For example, a book would be a primary resource in a book-store application.

**Paging Resource Types** provide paged access to other resources. Paged access means that it is possible to browse through a resource page by page, for example a list of all books in a book store could be the paged type of such a resource.

**List Resource Types** represent a list of other resources. As opposed to the paging resource, the list resource type does not offer paged access. The elements in such a list are of a certain resource type.

**Filter Resource Types** provide a partial view on a list of items. For instance, a filter resource could be the list of books published after 1999 in a book-store application. Filter resource is hence a subtype of list resource.

**Subresource Types** represent part of a larger whole. A subresource type is part of another resource represented as an entity on its own. For example, in a book-shop, one could have an abstract for each book on sale. That abstract can either be represented as an attribute of the book itself or as a subresource of a book.

**Activity Resource Types** are resource types representing a process or a part of it [4]. For example, when ordering books in a book-shop application, an activity resource could be entering the billing address during checkout.

**Aggregation Resource Types** aggregate properties of other resource types into a single resource type, effectively reducing the number of necessary client/server interactions [4]. An example for an aggregation resource could be the details page of a book that also shows full information of the author, thus aggregating information

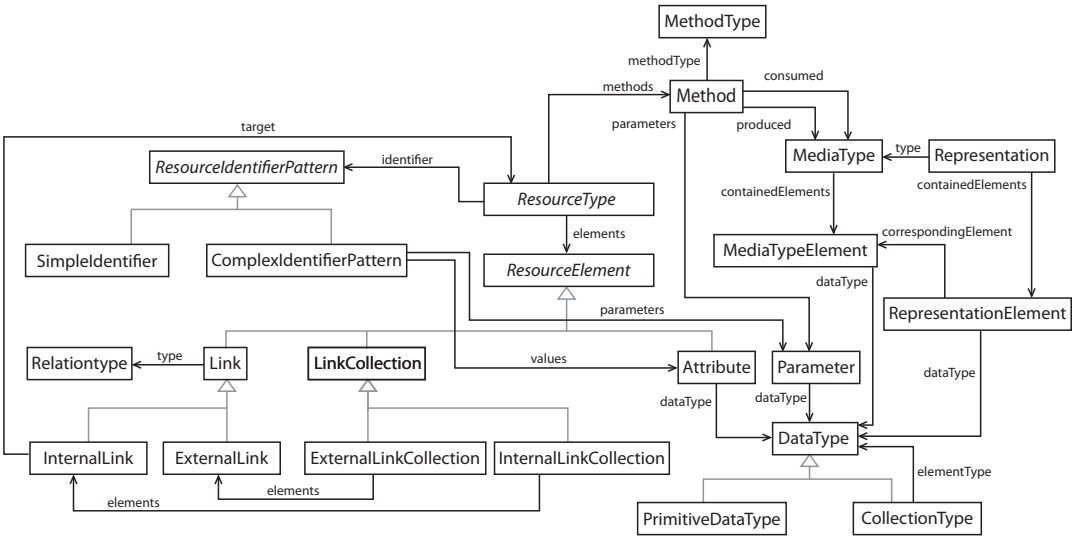



Figure 2.2: The core of the structural model [6]

about a book resource and the corresponding author resource into a single view.

**Projection Resource Types** are used to create a reduced view on another resource type, removing superfluous attributes. This reduces the amount of data that needs to be transferred [4]. For instance, a reduced view on a book could only show the title and author and omit all other information. There could then be also a list showing all books in the book-store with only their title and author instead of all information available for each single book.

The other part of the meta-model—the behavioural model—constitutes the specification of what actually happens when a request is processed. It models the `State`s and the behaviour of a resource types `Method`s by means of `Action`s executed. That is, every Method has a certain `Action` or a number of actions contained inside an `ActionSequence`. Also, there are different types of actions such as `ReturnAction` or `ConditionalAction`. Figure 2.3 gives an overview of the different relations and concepts.

## 2.4 Visual Language Design

The goal of the development of a visual language should always be **cognitive effectiveness** which "determines the ability of visual notations to both communicate with business stakeholders and support design and problem solving by software engineers" [8, p. 757].

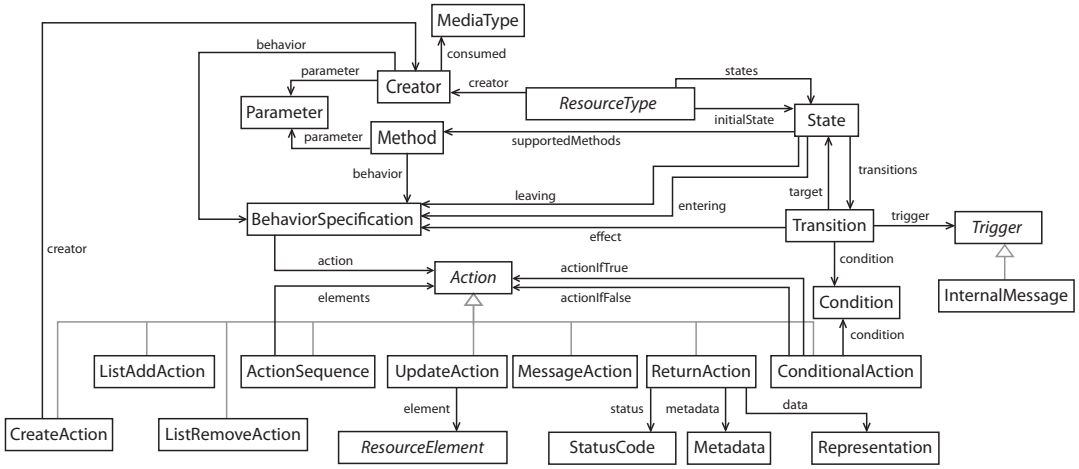



Figure 2.3: The core of the behavioural model [6]

However, the design of visual syntax is a neglected yet important issue in modeling language development [8]. A good modeling language is not only defined by proper semantics but also to a large extent by good and reasonable syntax. It would be short-sighted so assume that only because a language is semantically complete it will be easily usable by practitioners. Most visual language designers rely on their intuition when defining graphical elements of a language or simply put the issue aside completely [8, 22].

It is important to consider and to document the reasons for the design decisions made to provide traceability and justification for them [23]. Moody [8] therefore tries to put designing a visual notation onto a more scientific level by defining visual properties a good notation should have. Those criteria can be evaluated in the process of designing the notation and not only after the fact. That way a prescriptive theory for visual notations is established.

There are nine criteria which can be used in a modular fashion. Not all of them have to be applied for any notation newly designed and some can be stressed more in the design process than others. A detailed description of the criteria used in developing the graphical notation that is part of this thesis is given in chapter 3.

Besides Daniel Moody's article [8] and his work on improving existing notations such as i* [24] and UML [7] there has been work done on the subject of evaluating graphical notations before. One approach is the cognitive dimensions framework (CDs framework) [25–28]. The CDs framework however only offers a descriptive theory—it can only be used to evaluate existing notations and not to create new ones. But also other work done to provide the tools to analyze visual notations [28–31] suffers from the same fundamental

shortcomings as the CDs framework: They are merely descriptive and offer no foundation to build new notations upon.

## 2.5 Questionnaire Development and Evaluation

The development of a questionnaire is more than just asking questions [32, p. 19]. There are numerous books and articles published on the theory and practice of questionnaires development [33–37]. They provide general advice for different purposes ranging from psychological evaluation (for example, performance and intelligence tests) to opinion research.

Evaluation of data gathered through questionnaires is another aspect - although highly connected to the design phase of any questionnaire [33]. The data analysis is usually done using SPSS [33, 38, 39], but there are free alternatives to that [40, 41] which also allow statistical evaluation of the data gathered in much the same way as the commercial SPSS tool does.

There has also been work done already on the topic of graphical language evaluation. Based on the cognitive dimensions framework, Blackwell et al. developed a questionnaire they used to evaluate properties of certain visual notations [42]. Bobkowska refined and tailored that questionnaire towards her needs [43] after studying its usability [44]. She also developed a framework to support visual language evaluation [45]. Both Blackwell and Bobkowska require the user to be familiar with the system that is being evaluated to work effectively and both use mostly open questions when trying to gather a probands opinion. While this is a perfectly reasonable course of action, it does not match the scenario envisioned for the questionnaire developed here. In this case, the assumption is that probands will likely not have had the chance to work with the language yet.

Other studies on visual languages have been done as well. Johansson et al. [46] evaluate the different diagram forms, focusing on process oriented modelling languages such as Flowcharts, BMPN, EPC and UML activity diagrams using Moody's criteria. However, their evaluation is limited to a small set of researchers who were familiar with Daniel Moody's criteria and who formed a common understanding of the scales used to judge the criteria before starting the actual evaluation. Also, they directly asked the opinion of the probands on a scale of +1 to -1 for every criterion for every language. So they had a rather coarse grained scale with a rather simplified questionnaire. The basic idea is similar to the study that is done in this work in that probands are also asked to give their opinion on certain aspects of the modeling language. However, the questions used in the questionnaire presented in this thesis are more fine grained and try to mitigate the fact that probands might not have any experience with visual language design.

## 2.6 Implementation Aspects

The prototypical editor implementation is based on the Eclipse Rich Client Platform (RCP) [47] in concert with some additional components from the Eclipse [9, 48, 49] ecosystem.

The core meta-model bases on the Eclipse Modeling Framework (EMF) [50, 51]. EMF enables meta-modeling based on Ecore models and offers facilities for persistence and XMI serialization as well as a reflective Java Application Programming Interface (API). *EMF.Edit* supports the creation of editors from EMF models and *EMF.Codegen* is used for generating new editors from EMF Ecore models.

While out of the box this more or less only covers the semantic part of the model, the EMF tooling (and the Eclipse SDK, for that matter) allows pulling in additional components to facilitate the development of graphical editors for any EMF model.

The basis for developing any graphical editor with Eclipse is the Graphical Editing Framework (GEF) [52]. On top of that, there are currently basically two more or less mature frameworks that support the development of graphical editors in a more straight forward fashion than basic GEF: The Graphical Modeling Framework (GMF) [53, 54] and the Eclipse Graphiti Project [55, 56]. Both are part of the Graphical Modeling Project (GMP) [57].

The Graphical Modeling Framework uses a model-driven approach to developing graphical editors. Additional models that define the available tools, the available graphical symbols and the mapping between all these and the domain meta-model are needed in this case.

The Graphiti Project on the other hand uses a code-centric approach to developing graphical editors for the EMF models. The project offers an API tailored towards efficient and easy development of the required classes to implement an editor. It does not rely on code generation at all and also supports non-EMF models.

The editor implementation uses Graphiti instead of GMF because it seemed to be the approach that would be easier to use for a prototypical implementation. Although GMF seems to be more mature, it has a much steeper learning curve and also in the end might require modifying generated code, breaking to some extent the benefit of generating code in the first place and possibly adding additional sources for problems to the mix.

The Enhanced Editing Framework (EEF) [58] is used to generate the property sheets for the semantic elements that are used to display the elements attributes in the graphical editor. EEF offers a model-driven approach. It uses two additional models, namely the *components* and *eefgen* model, to derive rich property editors from an EMF *genmodel*.

On a more pragmatic level, there is always the need to build and test virtually any application or library, also when developing a prototype. Experience shows that it is always favourable to have a working ecosystem before any version of a product needs to be released. The editor implementation is tested on unit level by using JUnit [59] and on User Interface (UI) level by using SWTBot [60] and Google WindowTester Pro [61]. To provide a reliable and repeatable automated build, Maven [62] is used together with Tycho [63]. To catch errors in the build and failures in the tests early and regularly, Continuous Integration (CI) [64] is facilitated. The continuous integration environment is provided by the Jenkins Continuous Integration server [65] (a fork of the popular Hudson CI server [66]). Together with automated unit and UI tests and the repeatable builds provided through Maven/Tycho, this accounts for a robust build environment that catches problems early.

# 3 The Graphical Language Visual REST

In this chapter the graphical notation, called Visual REST, that is developed as part of this thesis is presented. First, criteria for a proper visual syntax are presented followed by a description of the process used to derive the new notation from the meta-model. Subsequently, Visual REST is illustrated in detail afterwards. The views and graphical symbols that are part of the language are shown and a motivation and rational for the design decisions is given, partially based on the graphical properties presented before. Afterwards, the properties of the new visual language are discussed in detail, using the criteria presented at the beginning of this chapter as focus points. Finally, examples of applications modeled using Visual REST are shown, providing an overview of the different views and usage of the different symbols in real application scenarios.

## 3.1 Criteria for Proper Visual Syntax

Visual syntax is often considered secondary when developing a new modeling language. It is a mere afterthought in the development process and not much conscious effort is put in coming up with a perceptually effective notation [8, 22]. The development of a cognitive effective visual syntax needs to become a conscious process and not only be based on intuition of the developers [8]. Criteria need to be developed to judge notations upon, so that good graphical notations become distinguishable from bad ones.

Daniel Moody [8] provides a solution to these challenges. He provides nine principles in the form of a prescriptive theory that can be used during the development of a notation.

The principle of **Semiotic Clarity** says that there should be a one to one correspondence between syntactic and semantic features. The principle is concerned with minimizing symbol redundancy and symbol overload as well as symbol excess. However, Moody states that symbol deficit is "desirable in most SE contexts" [8, p. 762]. This means that, in most notations developed for a software engineering purpose, it does not make sense to create a graphical representation for all semantic constructs, because that would lead to notations with too many different graphical symbols.

The principle of **Perceptual Discriminability** states that "different symbols should be

clearly distinguishable from each other" [8, p. 762]. Discriminability is primarily a function of the visual distance between symbols and visual distance is defined by the number of visual variables the symbols differ in. Visual variables are, for example, color, shape, brightness, texture and position. To be able to distinguish between symbols even easier, it is helpful to add redundancy: a different shape with a different color helps discriminate symbols. Ideally, each symbol should have at least one unique visual variable to pop out. They can then be detected without conscious effort according to feature integration theory [67, 68].

This is emphasized by the principle of **Visual Expressiveness** which states that the full range of visual variables and their full capacity should be used to represent the notational elements. For example, UML only uses shape to distinguish between the elements of the notation, so it only uses a single variable. A visually expressive notation should however do more. Especially color is one of the most effective visual variables [69, 70] and thus should be used in any visual notation. However, due to problems with, for instance, color blindness and black-and-white printers it should not be used as the sole criterion [8, p. 768]. A robust design of a notation adds redundancy to the information encoding to make it easier to distinguish between different notational elements based on different visual variables [8].

The principle of **Dual Coding** postulates that textual encoding ought to be used in a supportive form. Diagrams should be augmented with additional textual information. This textual information can either be an annotation that is used to explain the contents of a diagram, or it can be a hybrid representation of a symbol's meaning. An example of a hybrid representation is a multiplicity relation that is represented by different graphical symbols representing different cardinality types (for example *, 0..1, etc.). These representations then get augmented with their actual meaning in form of text, as is shown by the rightmost part of figure 3.1. An example of annotations are the annotations provided by UML—they enable UML users to add explanation directly inside a diagram so that no additional document is needed.

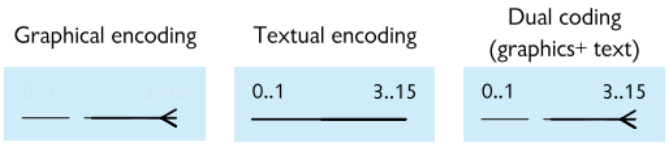



Figure 3.1: An example of dual coding [8]

The principle of **Semantic Transparency** demands that it should be possible to infer the meaning of a symbol from its appearance. The benefit is "reduced cognitive load because they have built-in mnemonics" [71] which leads to improved speed and accuracy of understanding the information contained in a diagram [72, 73]. The use of iconic rep-

resentations additionally improves the speed of recognition and recall [74], yet they are rarely used in most software engineering contexts [71].

Graphical notations should also provide a way for modularisation and hierarchically structuring to effectively represent complex situations [8]. The principle of **Complexity Management** states that these mechanisms should be supported inside the notation to manage complex use cases. Both modularisation and hierarchically structuring need to be backed by semantic constructs. An example for modularisation is the UML package since it provides the semantic construct of a module. The syntactical solution provided by UML packages however is not sufficient—more than just a representation of the semantic construct is needed to solve the problem of complexity management [8]. A hierarchical breakdown could include summary diagrams that allow navigating to lower abstraction levels or even a recursive decomposition [75] as for example is the case with Data Flow Diagrams (DFD) [76].

The principle of **Cognitive Integration** is applicable where multiple diagrams are used to represent the complete system. It can be distinguished between homogeneous integration and heterogeneous integration where the former is the use of multiple diagrams of the same type while the latter uses multiple diagrams of different types to describe a system [8]. However, using any multiple diagram approach puts additional cognitive demands on the user [77]. To mitigate this problem, cognitive integration theory [78, 79] uses *conceptual integration* and *perceptual integration*. *Conceptual integration* helps the user to build a coherent mental image of the system. An important mechanism is the use of summary diagrams. Another technique is contextualisation where the context of the current diagram (the current part of the system) is included in a diagram. Foreign elements can be included in the diagram to show which other parts of the system relate to the part shown in the current diagram. *Perceptual integration*, on the other hand, adds perceptual cues to allow easier navigation between diagrams. There are basically four questions that need to be answered. "Where am I?" (Orientation), "Where can I go?" (*Route choice*), "Am I on the right path?" (*Route monitoring*) and "Am I there yet?" (*Destination recognition*) [80]. Clear labeling of diagrams supports orientation and destination recognition. Level numbering helps with orientation as well. Using navigational cues on diagrams (Signposting) supports route choice. A navigational map (a map of all diagrams and the paths between them) helps with route choice and route monitoring.

The principle of **Graphic Economy** says that there should be only a limited number of notational elements since the number of different symbols that can be effectively recognized is limited [81]. When using only one visual variable, this limit is around 6 different symbols [82]. When increasing visual expressiveness, this limit increases almost additively [82] so that more symbols can be recognized effectively. Another way to improve graphic economy is to reduce or partition semantic complexity. Explicitly introducing symbol deficit is also an option to improve graphic economy, since some information can be more effectively encoded textually than graphically any way [27].

Different audiences or different tasks might require different views on a model. An expert and a novice practitioner might find different views on a model useful. Also, the representational medium plays an important role. Using a language on a white board, for example, might call for a different notation than using a computer program to do the modeling. The principle of **Cognitive Fit** reflects this fact in the design of modeling languages. The intended audience should be clearly defined when developing the notation. If there are distinct audiences targeted, it should be clear if there needs to be a distinct language dialect to better support them and their medium.

## 3.2 Deriving the Notation from the Meta-Model

Although Daniel Moody offers a prescriptive theory for the development of visual syntax, defining a new visual language is still not a guided process. There currently is no predefined way to follow that would allow for the structured and constructive creation of new notations. The process devised and used for the development of Visual REST given an existing meta-model is thus presented subsequently.

The development of the new syntax has to be iterative. Starting from an initial idea, that initial idea is refined and tailored using the criteria defined by Moody. The process used for the design of Visual REST is following that broad schema. Figure 3.2 provides an overview of the steps in the development process.

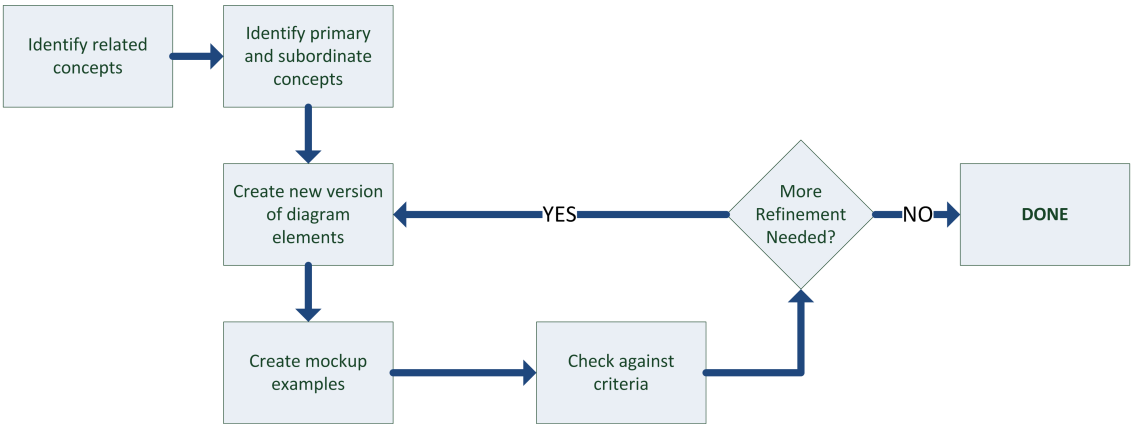



Figure 3.2: Overview of the process to derive a new graphical notation

To derive the notation, the first step is the identification of semantically related concepts. This was for a large part already done in the meta-model [6] which already defines a structural and a behavioural part, but, as section 3.3 shows, there was some refinement

needed which lead to a more fine-grained grouping, in the end.

With the semantic groups established, the second step of the process is the identification of container concepts in the groups. The resource type for example is a container concept. Resource types form the major item of concern in modeling the structure of an application. Other concepts such as attributes are subordinate to them. Connections between the container concepts were easily identifiable as well and need special treatment. The internal link for example is an attribute by nature and thus subordinate to a resource type but still receives a special representation. It is represented by a connection arrow between two resource types because it connects the resource types semantically in the model as well.

The third step is the definition of iconic representations for the primary concepts in the groups. The iconic representations are chosen by starting from the primary concepts and trying to find a representation that matches the semantic meaning of the concept while still being visually discriminable from the other symbols already used. This follows the principles of visual discriminability and semantic transparency.

After defining the icons for the major concepts, a way to include subordinate concepts in the representations needs to be found. For semantic connections this is obvious; directed connections like internal links—as mentioned before—or transitions can easily be represented by arrows. Similarly, action sequences are represented by arrows because using arrows is a semantically transparent spatial relationship between objects [8, p. 765]. A rationale for all design decisions is given when the various representations are presented later on.

Following the initial definition of all representations, the process is repeated and the representations are refined. This is done by evaluating the representations against Moody's criteria and changing or adapting things where it is needed. Representations can also be changed because of new ideas that only occurred after the previous iteration, such as a new icon for a certain resource type, for example. Besides changing representations, also other notational elements to support the various criteria are added or removed as needed. For example, in the first iteration in the definition of Visual REST there were no markers on the diagrams to support cognitive integration. They were only added in a later iteration's refinement step.

To get an idea of how well the notation created in one iteration of the process actually is regarding Moody's criteria, mockups of the notation were created using a simple drawing tool. First, generic mockups for each individual element of the notation were created to be used as blueprints. These blueprints were then utilized to manually draw—since there was no actual editor at that point—different application models. Through the actual use of these mockups in modeling a direct feedback is provided that helps to identify potential issues and problems with the new notation in relation to some of Daniel Moody's criteria

such as Cognitive Fit and Graphic Economy. This step also provides feedback beyond the mere evaluation of Moody's criteria. Through the use of the notation in a real example, a feeling for the usability of the notation is established. Although this is neither qualifiable nor quantifiable, it still helps to improve the notation.

The process stops once there is a suitable and visually sound representation for the domain concepts. The feedback loop presented earlier can also be extended with the results of the questionnaire presented in chapter 4. That way, feedback on the different notational elements can be gathered from an external source which can then be used to improve the notation even further.

## 3.3 Diagram Types of Visual REST

The REST meta-model [6] is divided into a structural and a behavioural model. While that seems a reasonable choice for the meta-model, it does not exactly match the visual language view on the model. For the sake of complexity management Visual REST offers a four different kinds of diagrams (or views) on the model.

The **Structure View** is manifested in a **Structure Diagram** which covers most aspects of the structural meta-model. It is used to display the different resource types in an application. However, a single application is not limited to having a single structural diagram. The different resources types of an application can be distributed across multiple resource structure diagrams. A single resource type can also appear on multiple diagrams to put the other resource types shown there into context. That way, the structure view offers Complexity Management by partitioning and Cognitive Integration by enabling the display of contextual information. Figure 3.3 gives an example of a simple resource structure diagram.

There is also an alternative representation of the structural view that omits many details of the resource types in favor of a clearer view on the resource structure and the connections between resource types. This view can be used as a **Structure Overview Diagram** when only those aspects are of interest, for example, to present an overview of the overall architecture of a resource-oriented application (see figure 3.4). Similarly to the structure diagram, the structure overview diagram also allows partitioning.

The behaviour of the methods supported by a resource type in a certain state needs to be defined as well. This is done using the **Resource Method Behaviour View (Resource Method Behaviour Diagram)**. A single resource method behaviour diagram can display one or more methods of a single resource type. Methods of different resource types cannot be combined into a single method diagram. However, there can be multiple

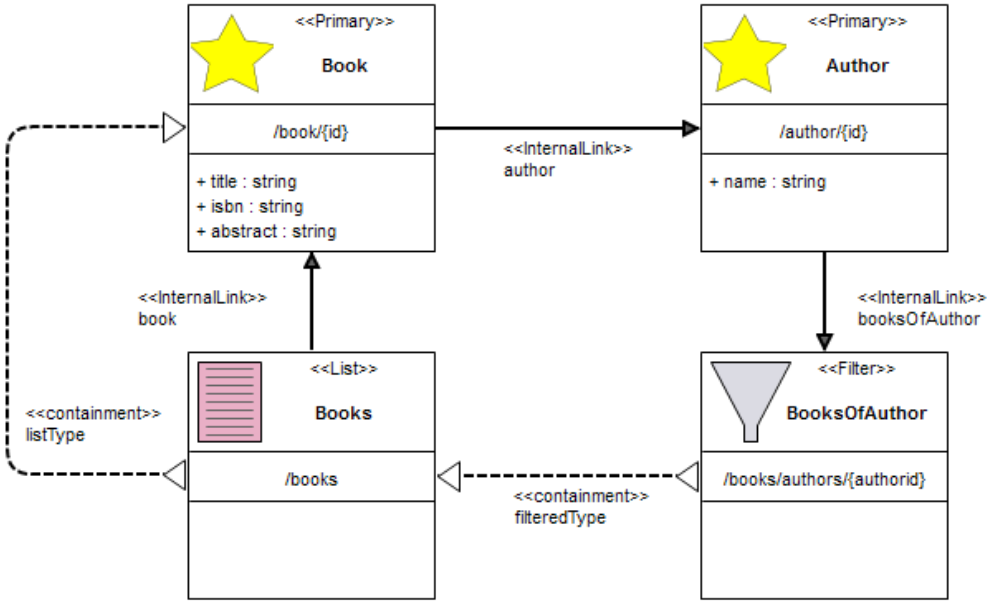



Figure 3.3: The structure view

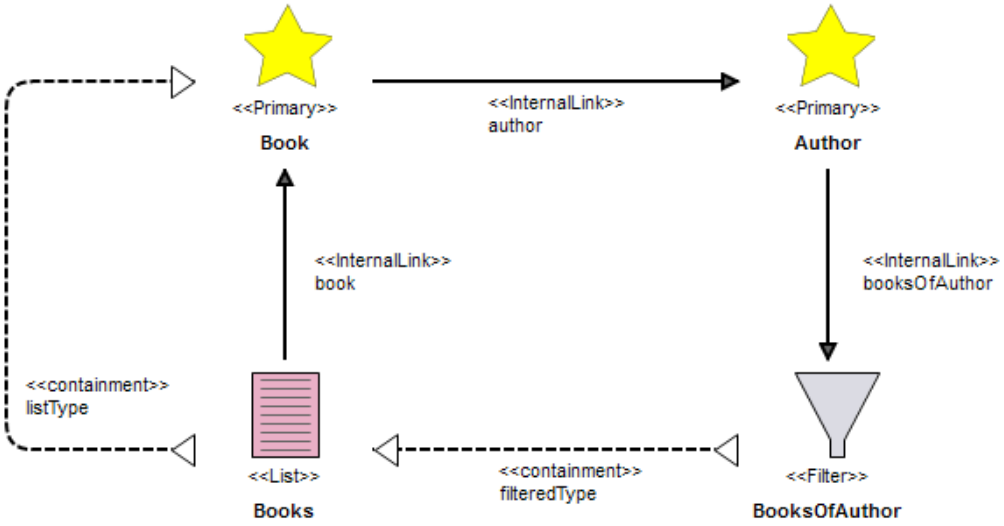



Figure 3.4: The structure overview view

method diagrams per resource type. This allows for partitioning of complex resource types to some extent in that it allows putting, for example, very complex methods on a single diagram while grouping other simple ones together into another one. Hierarchical structuring is not possible by design because it seemed to make matters overcomplicated for practitioners. Cognitive Integration is provided through the use of indicators that identify the resource type a particular method behaviour diagram belongs to. Figure 3.5 gives an example of a resource method behaviour diagram showing multiple methods.

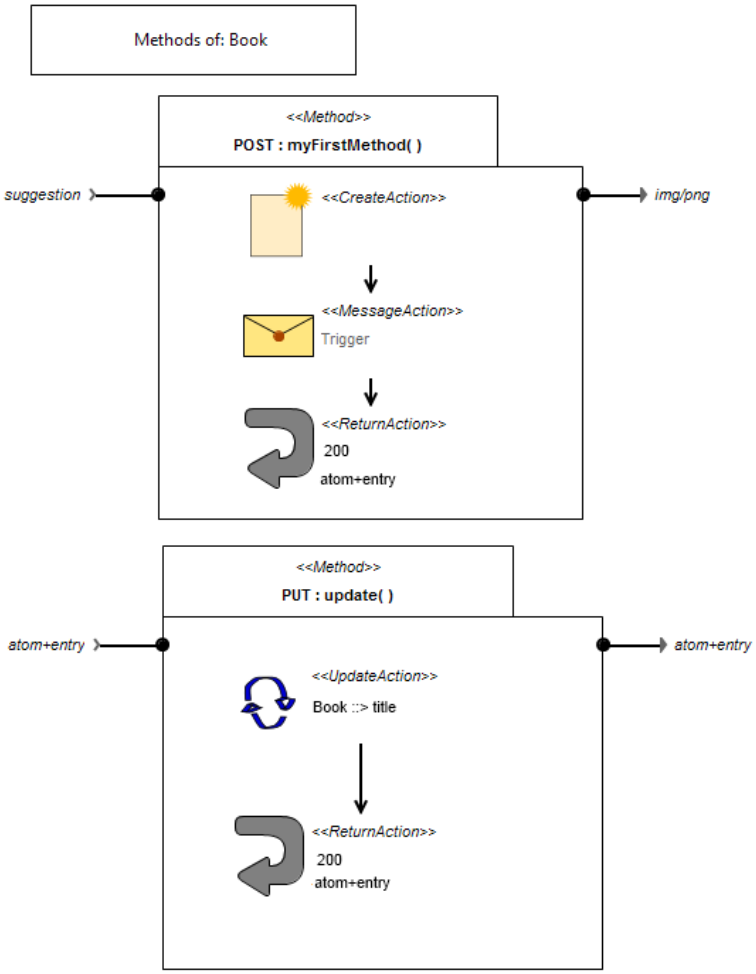



Figure 3.5: The resource methods view

The states of a single resource type are depicted inside the **Resource States View (Resource States Diagram)**. This view shows the states of a single resource type and their respective transition from and to one another. Also, the supported methods in the different states are displayed. Complexity Management on the same semantic level is not

possible; states of a single resource type cannot be distributed onto several diagram but all have to be on a single one. Complexity Management for the complete application is provided by the hierarchical structure going from resource type through resource states to method behaviour. Cognitive Integration in terms of navigation support and location awareness is provided by the indication of the resource type the state diagram belongs to. The **Resource States View** is the connection between the resource structure as defined in the **Resource Structure View** and the behaviour of individual resource methods as defined by the **Resource Method Behaviour View**. Figure 3.6 gives an example of a resource state diagram.

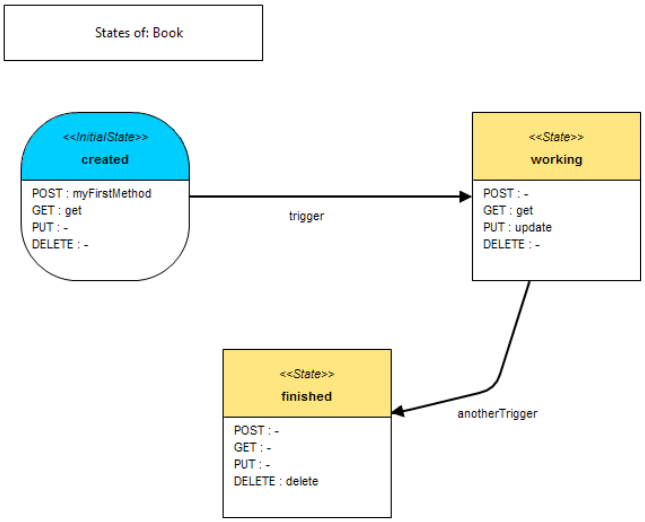



Figure 3.6: The resource states view

The visual elements of the four views of the notation and their mapping to their semantic meaning is outlined by sections 3.4, 3.5 and 3.6, respectively. Some items only received a textual representation rather than their own graphical symbols—this is also explained in those sections.

Even with all the views and their contained elements defined, there are still some items of the meta-model that are not reflected in the visual syntax—section 3.7 provides an overview. Leaving out elements is done on purpose. Adding all elements just for completeness' sake would for all intents and purposes make the notation less usable (and break with the principle of Graphic Economy). Leaving out parts of the meta-model concepts is common in a software engineering context and does not break with the principle of Semiotic Clarity [8, p. 762].

## 3.4 Elements of the Structural View

The structural view contains all elements relevant to the structure of an application. In particular, it contains all kinds of resource types, internal links, attributes and identifiers. In the following sections the different representations of the individual concepts are presented. A description of and motivation for the choice of representations is given for each symbol as well.

### 3.4.1 Resource Types

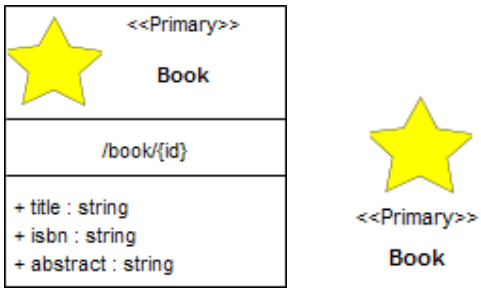
The resource types are similar in their semantics and thus are also similar in their syntax. For every resource type there is a distinct icon, though, that is used to convey the actual type. There is a collapsed and expanded version for each resource type. The collapsed version only shows the iconic representation alongside the textual type annotation (in form of a UML-like stereotype) and the resource types name. It is used on the structure overview diagram or for contextualisation on resource structure diagrams. The expanded representation additionally shows attributes and identifiers for the resource type in a UML-class like box. The iconic representation is then placed into that box. It is used on the resource structure diagram. Examples are shown when the individual resource types are discussed.

#### Primary Resource Type

The primary resource type is symbolized using a yellow star as shown in figure 3.7. In computer programs, the star is commonly used to represent favourites (e.g. in browsers). A favourite is something that is often used. The idea is that the star symbol provides a hint to that meaning. From there, the link can easily be made to a primary resource type since that as well is something that stands out of the mass—it is one of the primary concepts used in an application. The yellow star symbol differs in shape and color from all other resource types.

#### List Resource Type

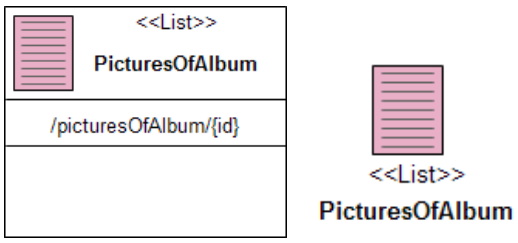
The list resource type is symbolized by a pink sheet of paper with lines on it as shown in figure 3.8. The link should be easy to make—a list in real life often looks similar. The list differs from all other symbols in shape and color. The paging resource type is closely




(a) Expanded (b) Collapsed

Figure 3.7: The primary resource type

related in terms of shape, though, but a trade-off was made here in favour of semantic transparency versus perceptual discriminability.




(a) Expanded (b) Collapsed

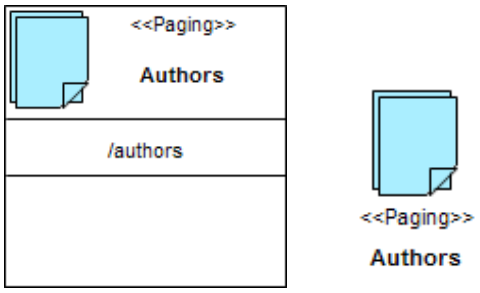
Figure 3.8: The list resource type

### Paging Resource Type

The paging resource type is symbolized using two light blue sheets of paper with the front sheet having an edge already folded away, revealing the next page. This is shown in figure 3.9. The basic idea is that this should symbolize a page being turned over. Although related in shape to the list resource type it is still somewhat different. It also differs from all other resource types in shape and color.

### Activity Resource Type

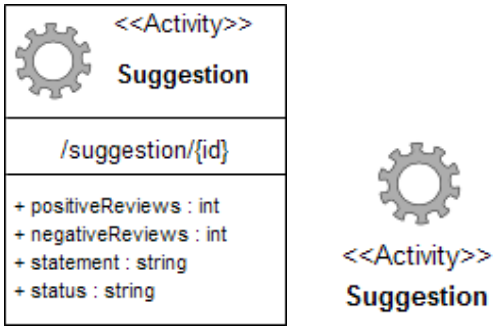
The activity resource type is symbolized using a dark grey cogwheel as shown in figure 3.10. A cogwheel was chosen because it seems to transport the meaning of an activity




(a) Expanded (b) Collapsed

Figure 3.9: The paging resource type

quite well. It can regularly be found on "go" buttons in computer programs to start actions, so out of common convention it should be easy to understand its meaning. It differs from the other resource types in shape and from all but the filter resource type in color.




(a) Expanded (b) Collapsed

Figure 3.10: The activity resource type

### Aggregation Resource Type

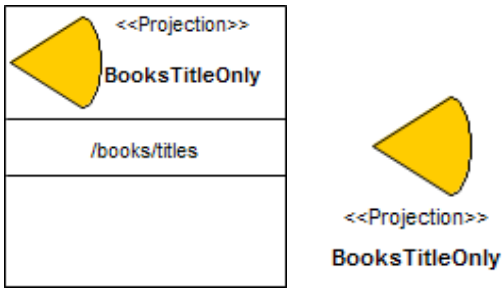
The aggregation resource type is symbolized using an accumulation of green blocks as shown in figure 3.11. This is meant to symbolize something being assembled from different parts which is basically what is happening with aggregation resource types. The green building blocks are also different from the other resource types in both shape and color.


(a) Expanded (b) Collapsed

Figure 3.11: The aggregation resource type

### Projection Resource Type

The projection resource type is represented by an orange light cone as shown in figure 3.12. The light cone is probably one of the least obvious symbol selections compared to the others. The basic idea is that a video projector throws some sort of light cone to "project" something to a surface. This is the semantic meaning that should be conveyed. The true semantics of a projection—a partial view on a resource type—are not directly conveyed by this representation but should be obvious for users that are familiar with database terms. The cone differs from the other resource types in shape and color. A semantically more transparent representation for this resource type could probably have been found, but at the cost of a likely more complex symbol. Since the notation as a whole should still be usable on a white board without the need for a dedicated editor, a decision was made to chose a simple representation at the cost of slightly reduced semantic transparency.

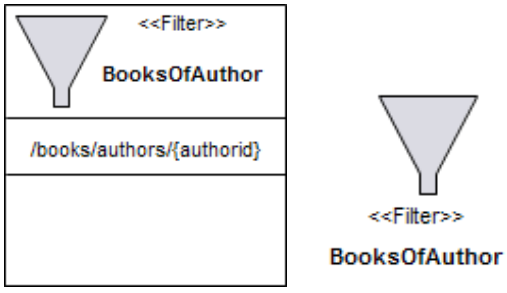



(a) Expanded (b) Collapsed

Figure 3.12: The projection resource type

#### Filter Resource Type

The filter resource type is symbolized using a light grey stylized filter as shown in figure 3.13. The filter symbol used can often be found in other application such as database front-ends. In those applications it is commonly used to support filter operations on data tables. Since this is pretty much exactly the semantics of the filter resource type, the link seems very easy to make for users. The filter differs in shape from all other symbols and in color from all but the activity resource types' cogwheel, which is colored in dark grey. Using yet an additional color seemed to reduce graphic economy to an extent where the diagram became too colorful.




(a) Expanded (b) Collapsed

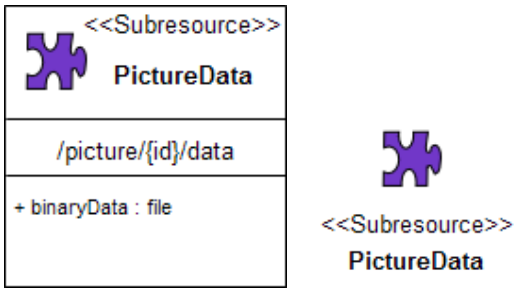
Figure 3.13: The filter resource type

#### Subresource Type

The subresource type is symbolized using a purple puzzle piece as shown in figure 3.14. The puzzle piece represents the fact that a subresource type is to another resource type what the puzzle piece is to a puzzle. It is part of something bigger. The representation of the subresource differs in both color and shape from the other resource types.

### 3.4.2 Attributes

Attributes are shown in the lower area of the expanded resource type box as shown in figure 3.15. The notation for attributes is somewhat borrowed from UML class diagrams. Attributes start with a "plus" (+) sign followed by the name of the attribute and its type. A colon separates the name from the type.




(a) Expanded (b) Collapsed

Figure 3.14: The subresource type

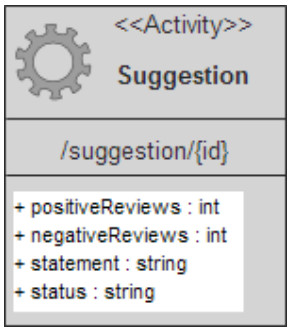



Figure 3.15: Attributes of resource types

### 3.4.3 Identifiers

Identifiers are placed in a special area inside the resource type box. They are situated right between the resource types name and its attributes. Identifiers show the resource types URL, possibly including placeholders. Placeholders are represented using curly braces. For example, to access a specific book resource by its id, the URL `/book/{id}` is used as shown in figure 3.16.

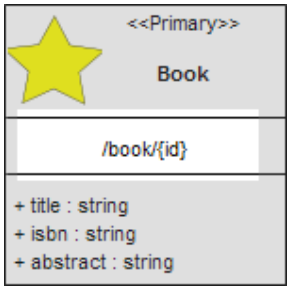



Figure 3.16: Identifiers of resource types

### 3.4.4 Internal Link

The internal link is represented using a solid black line that ends in an arrow head as shown in figure 3.17. A link is directed, so the arrow is as well, indicated by the arrow head.

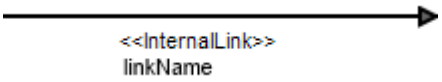



Figure 3.17: The subresource type

The name of the link is used as decoration of the arrow that represents the link. The link name is again decorated with a stereotype that indicates the type of the graphical symbol. The relation type is currently not shown in the representation.

### 3.4.5 Containment

The containment connection is special. It does not directly map to a primary semantic concept, but rather represents a semantic relation between two resource types. For ex-

ample, a filter resource type has a certain type of elements. This element type is in this sense "contained" by the filter.

A syntactically better way to represented such relations is to encode it spatially instead of by using arrows [8]. However, since many different resource types can contain a single other one, this was not feasible here. For example, a list, filter and paging resource type can all contain the same primary resource type as their element type. This makes it nearly impossible to encode the information in a form different from arrows. Only subresource types could have been represented as a contained "object" inside another resource type in a sensible way. However, since being able to separate subresource types from their parent resource types as well and for reasons of syntactic consistency also here the containment representation using an arrow was chosen.

The arrow used to show containment differs from the internal links arrow in brightness and color as well as in form. It is represented using a dashed line that starts with a solid white arrow and also ends with one as shown in figure 3.18.

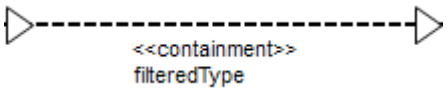



Figure 3.18: The subresource type

## 3.5 Elements of the Resource States View

The states diagram is a simple view in the sense that it contains the least number of different elements. Two types of states and a transition relation between them are supported in this diagram type.

### 3.5.1 States

A state as well as an initial state has a list of supported methods and a number of transitions to other states. There can only be one initial state per resource type. The initial state is a special case of state. It receives its own representation because it is an important part of a model. States list their supported methods and their types inside their body. Unsupported method types are also listed to allow the identification of the supported and unsupported methods at a glance; the information is encoded by spatial position rather than by text only. The position of the methods inside the body is the same

for all states, which makes their identification easy. The header of a state is annotated with the type of the state in the form of a UML stereotype followed by the name of the state.

#### State

A normal state is represented by a simple rectangle with a light orange header as shown in figure 3.19. The state differs from the initial state in that it has a sharp edge while the initial state uses rounded corners. Initial and normal state also differ in color. A box is used as representation for a state following the UML state diagram where boxes with rounded corners are used. To provide perceptual discriminability, the rounded version is used for the initial state and a version with sharp edges for the normal one. The representation should therefore work well for users familiar with UML.

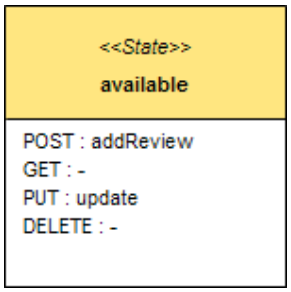



Figure 3.19: The State

#### Initial State

The initial state is represented by a rounded rectangle with a blue header as shown in figure 3.20. Again, the choice was inspired by UML and the figure differs from the normal state in color and shape. Also here, a different representation is thinkable for future versions of the notation.

### 3.5.2 Transition

A transition is represented by a simple arrow. Transitions are also directed relations as shown in figure 3.21. The semantics of a transition through the use of an arrow seem quite clear. The transition is augmented with a trigger. Currently only message triggers are supported.

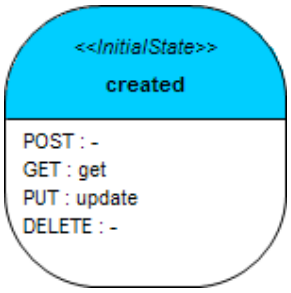



Figure 3.20: The Initial State

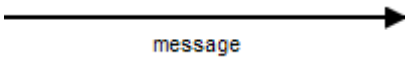



Figure 3.21: The Transition

#### Trigger

As was indicated before, triggers are properties of transitions. They are placed in as text as a decorator on the transition arrow as already shown in figure 3.21.

## 3.6 Elements of the Method Behaviour View

The method behaviour view is the most experimental view used in the notation. The metamodel itself is also still experimental and partially incomplete with regard to behaviour modeling. It is questionable if using a graphical notation to model behaviour of methods is favourable over a textual one. This is probably very much dependant on the user. The best way to model behaviour in general is also still an item of active research [83]. However, the method behaviour view presented here offers a graphical way to defining the behaviour of a resource types methods.

### 3.6.1 Method

As was already mentioned, there can be multiple methods per method behaviour diagram. Therefore, a method is the container for the actions contained within it. A method is represented by a tabbed box. The tab at the top contains a stereotype that indicates

the type represented by the graphical symbol (a method), the method name as well as its type and parameters. Figure 3.22 provides an example of such a method shape. The actions of a method are then placed inside the body of the box.

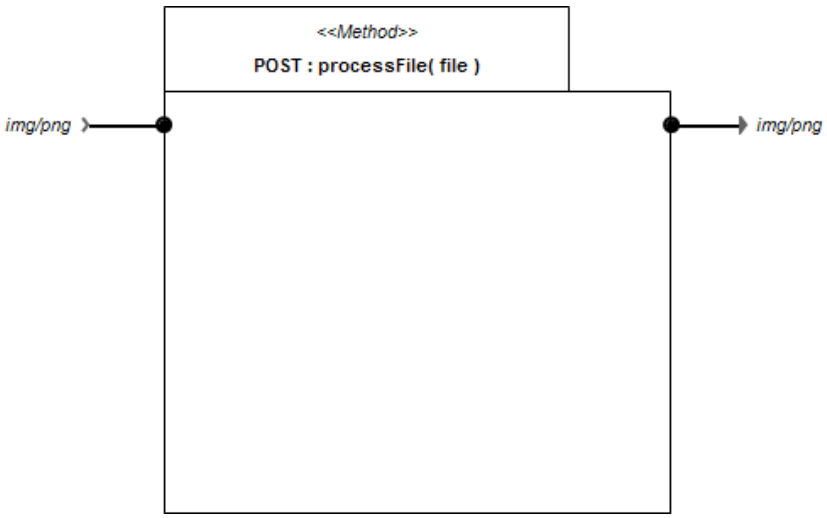



Figure 3.22: The Method

### Consumed Media Types

The consumed media type is represented by an open arrow that goes from the left to the right. It is connected to the method box via a line and a dot as shown in figure 3.23. This is meant to represent that something is moving into the method. The name of the consumed media type is written to the left of the arrow. Multiple consumed media types are placed below each other on the same side of the method.

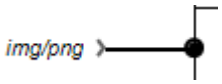



Figure 3.23: Consumed Media Types of a Method

### Produced Media Type

Similar to consumed media types, the representation should hint at the fact that something is going out of the method. As shown in figure 3.24, the produced media type is

represented by a dot followed by a line that goes into an arrow head. The dot is the connection point to the method and the produced media types name is written to the right of the arrow head.

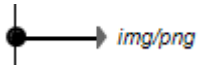



Figure 3.24: Produced Media Types of a Method

### 3.6.2 Actions

All actions except the conditional action follow the same pattern in their layout. An icon representing the action is shown to the left. To the right the textual representation is given and below that additional textual information unique to a particular action are shown. Figure 3.25 illustrates this layout. As opposed to the resource types, however, there is no additional box around the action representations in an effort to fight "boxitis" [84–86].

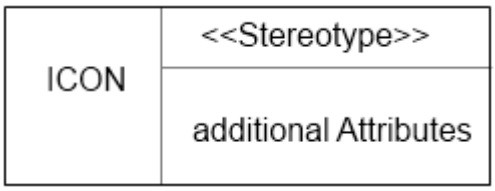



Figure 3.25: Generic Layout of Actions

#### List Add Action

The list add action is represented by a stylized list symbol augmented with a large green plus sign as shown in figure 3.26. This hints at the actual meaning of the action—it adds something to a list. A plus is commonly used as a symbol when something is added to something else. The list add action has no additional attributes that need to be displayed. It is similar in shape to the list remove and create action but not to an extent where confusion should occur, normally. It differs in color from the other actions as well.

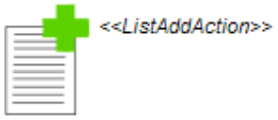



Figure 3.26: The List Add Action

### List Remove Action

Similar to the list add action, the list remove action is represented by a stylized list augmented with a red minus sign as shown in figure 3.27. A minus sign is commonly used to represent removal of something from something. The symbol is similar to the list add and create actions—it however differs enough to be easily distinguishable from the others.

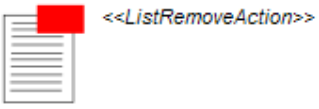



Figure 3.27: The List Remove Action

### Create Action

The create action is represented by a blank page with a stylized star as shown in figure 3.28. The representation is borrowed from the Windows XP save file dialog commonly used in many different applications. There, new folders can be created through the use of a similar symbol: a folder that has a star in its upper right corner. Also, in other applications, creating new files is often represented by a blank page that has an orange star in its upper right corner. A similar representation was chosen here because—with all those commonly used applications around—the meaning of the representation does hint at the semantics of the action quite nicely. For create actions the current version of the meta-model foresees a `Creator`. However, since the creator is not identifiable by a name at the moment, it is not noted in the textual attributes section of the create action.

### Return Action

The return action is represented by a backwards bending arrow as depicted by figure 3.29. The representation is similar to what is commonly found on any standard keyboard—the

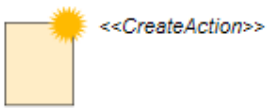



Figure 3.28: The Create Action

return key has a similar iconic representation there. The return action has a return code and a returned representation that is listed in the attribute section below the stereotype.


Figure 3.29: The Return Action

### Update Action

The update action is represented by two arrows that circle around a center point as shown in figure 3.30. This representation is commonly found in many applications on the market the allow reloading documents or updating software, for example. The icon seems common enough to convey the intended meaning. Below the stereotype the target of the update is shown.

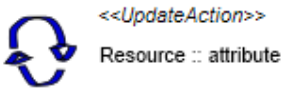



Figure 3.30: The Update Action

### Message Action

The message action is illustrated by a sealed envelope as shown in figure 3.31. The envelope is probably the prototype of a message since many years—starting in earliest years of any written communication. Even today's email and instant messaging systems still use an envelope to represent the act of sending messages. The message that is sent is shown in the attribute area of the action representation.

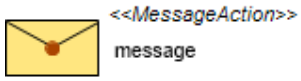



Figure 3.31: The Message Action

#### Conditional Action

The conditional action uses a special representation that is different from the other actions. For each conditional action there is a *true* and *false* path action execution can follow. Both are shown in the representation of a conditional action as shown in figure 3.32. As opposed to the other actions, the conditional action is—similarly to the method itself—a container for other actions. Instead of using arrows (from the action sequence representation) to display the different paths that can get executed this information is encoded through containment to make it easier to comprehend.

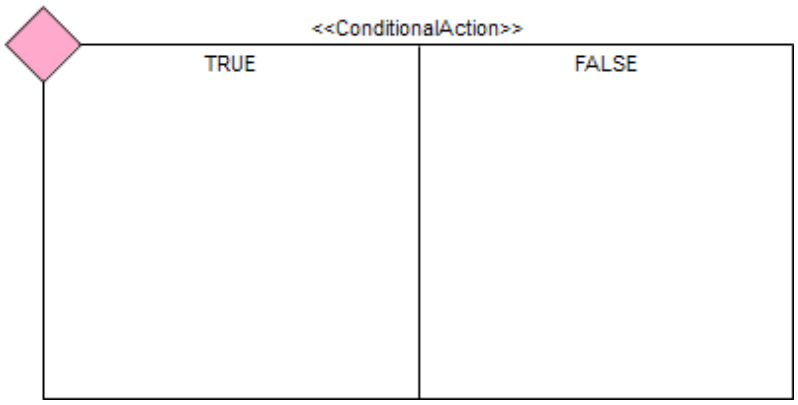



Figure 3.32: The Conditional Action

### 3.6.3 Action Sequence

The action sequence is modeled using arrows to connect the different actions in a sequence. As was mentioned before, the use of arrows to connect sequences is semantically transparent [8, p. 765]. The representation of such a sequence connection is shown in figure 3.33.

Figure 3.33: The Action Sequence

## 3.7 Elements without graphical representation

There are no graphical editors for data types, method types, relation types and bin op types because these structural components of the model cannot be meaningfully represented using symbols. That is, using graphics would only have added visual clutter and additional symbols while providing no benefit to the notation as a whole. A data type, for example, can much easier be written using structured text than using any kind of symbolic representation.

### 3.7.1 Creator

Modeling creators is currently not supported by the notation, although it is thinkable that, in a later iteration of the language, modeling of creators can follow the same scheme used for methods. Internally, a creator also has a sequence of actions that are executed. The creator was left out for the time being because it is unclear if the concept will survive a second iteration of the meta-model and also because of the experimental nature of the method behaviour view which the creator strongly relates to.

### 3.7.2 Guard Conditions

Guard conditions are currently not supported by the notation. Modeling conditions using a graphical language seems overcomplicated. Using a simple textual language to describe conditions used for conditional actions and transitions seems easier.

### 3.7.3 Internal Link Collections

Internal link collections are currently not represented in the notation through individual symbols. Similar to attributes, they could be added to the attribute list, for example. Alternatively, a representation matching that of internal links could be used, augmented with additional annotations to indicate a link collection. Perceptual Discriminability and Semantic Transparency need to be maintained, though.

### 3.7.4 External Links

External links are also currently not represented by graphical symbols in the notation. While internal links have a distinct target in a single diagram, external links do not. This means that additional target symbols are needed if an external link is also to be represented by a connection with an arrow. Also using an arrow would make sense since semantically an internal link is equivalent to an external one. It's just the target that differs. However, that would lead to a deficit in semiotic clarity because it would introduce elements that are not part of the model to the diagram (the target of the external link). An alternative would be to find a different representation for the external links arrow representation. However, that would increase graphic complexity and possibly break with semiotic clarity again since it would introduce symbol overload on some level. A last way out would be to simply add external links to the attribute list as well.

### 3.7.5 External Link Collections

Similar to internal link collections—and because external links are not represented either—the external link collection did not receive its own graphical representation.

## 3.8 Navigating the Notation

By convention, there is at least one—but possibly more—structure diagrams for a single resource-oriented application. The resource types shown in the structure diagram are linked to the state diagrams. There is exactly one state diagram for each resource type in the application. The behaviour of the supported methods of the resource type's state as shown in the states diagram can be found in a method behaviour diagrams. There can be multiple method behaviour diagrams per resource type and in each method behaviour diagram there can be multiple methods. Methods from distinct resource types cannot appear on the same method behaviour diagram, though. Figure 3.34 visualizes this relation.

Navigating this network of diagrams is mainly supported through sign posting. Sign posting means the addition of visual cues that direct the user and tell him where he currently is inside the complete network of diagrams and which related diagrams are available to him. The states as well as the method behaviour diagram contain an indicator that highlights which resource type they belong to, as shown in figure 3.35.

Navigating from a resource type to its methods is only implicitly supported. As per

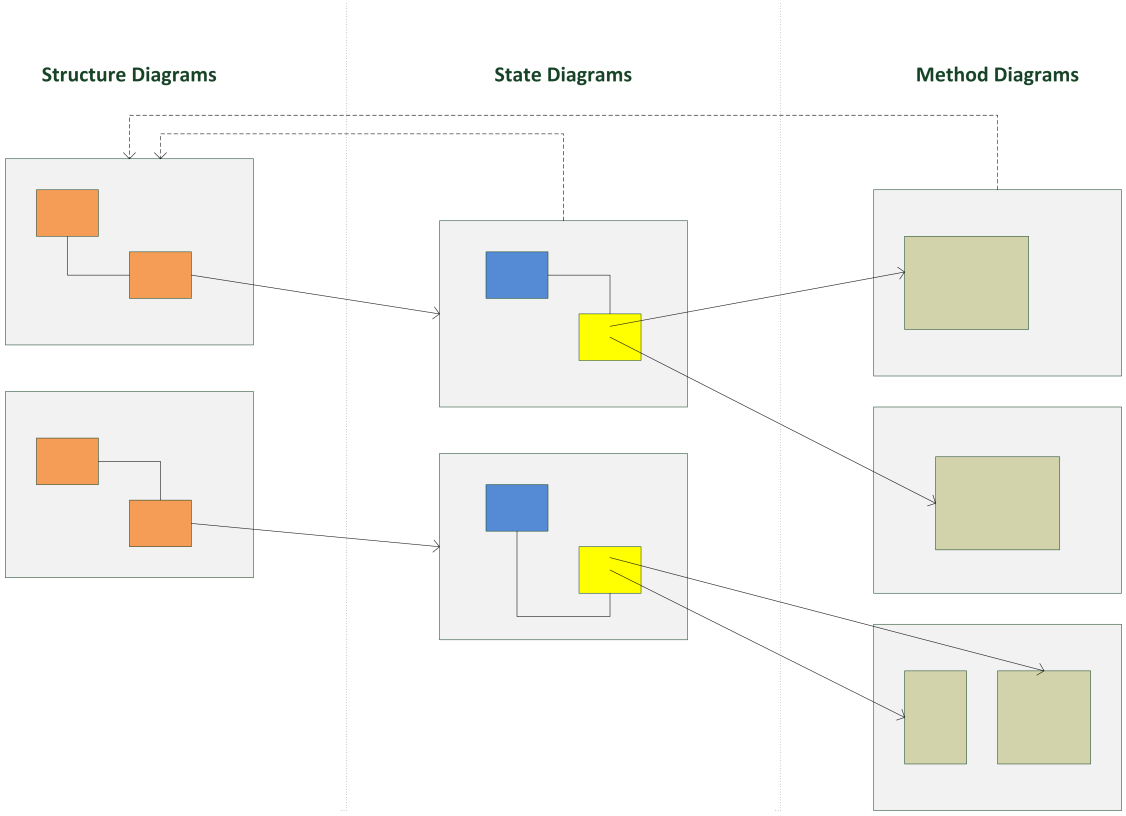
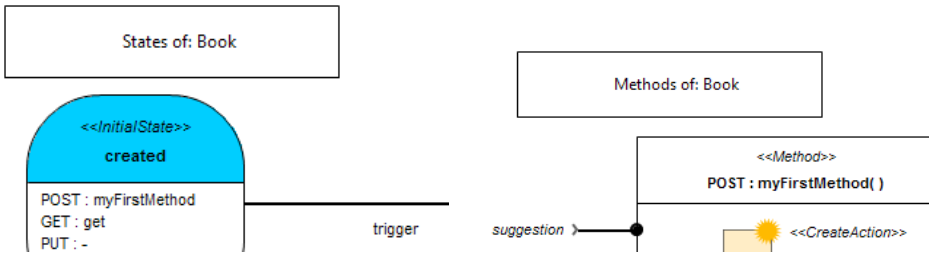



Figure 3.34: The navigation layers of the model




(a) States (b) Methods

Figure 3.35: Signposting on the diagrams

convention, there has to be exactly one state diagram for each resource type. Hence there was no need to add an additional indication in the resource structure diagram that would provide cues for that fact. Navigating from the states to the methods of a resource type is done through the display of the method names in the states. There is no direct way to navigate from the resource types to their methods. The states diagram is the link between resource types and methods.

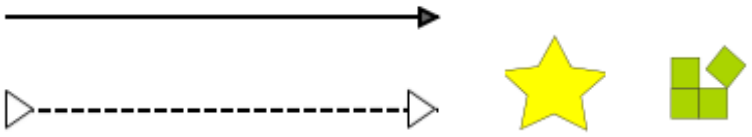
## 3.9 Moody's Criteria applied

The notation was designed to comply with Daniel Moody's criteria as presented in section 3.1 as much as possible.

**Semiotic Clarity** is given. Every semantic construct is represented by exactly one syntactical element. There is no symbol redundancy, symbol overload or symbol excess. Symbol deficit, however, was introduced intentionally, following the principle of Cognitive Fit in an effort to make the notation less overloaded. Not everything can or should be represented using graphical symbols.

**Perceptual Discriminability** is supported through the use of distinct shapes for the representation of the different semantic constructs. This is supported by using additional visual variables to create a greater visual distance between the syntax elements. The focus was on finding differing shapes inside each diagram type—the resemblance across different diagram types might partially be a bit greater.

**Visual Expressiveness** is implemented by using distinct shapes, colors and brightness. For example, the internal link connection differs from the containment connection on its shape and brightness as shown in figure 3.36(a). The primary resource types differs from, for example, the aggregation resource type on its shape and color as shown in 3.36(b).



(a) Internal Link and Containment (b) Primary vs. Aggregation Type

Figure 3.36: Visual Expressiveness

**Complexity Management** is supported on two levels. There is a hierarchical structure

such that the complete application is broken down into distinct diagrams. The hierarchy starts at the resource structure diagram and goes down to the method behaviour diagram through the resource states diagram. Additionally, modularisation is supported through the use of more than a single diagram for modeling resource structure and method behaviour. This allows users to group syntactic elements as they please.

**Graphic Economy** is kept through the partitioning into different diagram types on the one hand and through the explicit introduction of symbol deficit on the other. In addition, the symbols chosen as representation are designed to be visually expressive, which also helps in managing graphics complexity and thus increasing economy.

**Dual Coding** is implemented in the form of hybrid symbols for resource types, actions and states. For all of these elements, the symbol used to represent the semantic construct is augmented by a textual annotation in the form of a UML-like stereotype that helps interpreting the representation.

**Semantic Transparency** is achieved by designing the symbols used as representation for the semantic concepts such that they carry meaning. Although there are hardly any semantic constructs in the meta-model that have a direct representation in the real world, care was taken to choose symbols that convey semantics similar to the meta-model elements. The rationale for the selection of the different symbols is given in sections 3.4, 3.5 and 3.6 respectively.

**Cognitive Fit** can be achieved for professional users of the notation by leaving out redundant information. This includes the use of color for the symbols and the use of textual annotations. Novice users are expected to benefit from that information. They are also expected to profit by the use of overview diagrams as a reduced and less complex form to represent their applications. The notation was also designed to be usable on, for example, a whiteboard. The symbols used as representation were chosen such that, while being perceptually discriminable, they are still drawable by hand and using an editor on a computer is optional. Color will probably be left out on that medium as well and the the stereotypes used as redundant information to the iconic representations can also be omitted if professional users of the notation no longer need them.

**Cognitive Integration** is provided by visual cues on resource state and method behaviour diagrams and by contextualisation on resource structure diagrams. Visual cues are the markers on states and method diagrams that indicate which resource type the diagram (and its elements) belongs to. Contextualisation is supported for structure diagrams through the ability to add existing resource types to new diagrams to put the other resources shown on such a diagram into context. Overview diagrams can be created using the alternative resource type representations to provide additional conceptual integration.

## 3.10 Examples

In the following sections some examples for applications modeled using Visual REST are given. Not all diagrams are provided for every application because it would serve no purpose but to bloat up this thesis. The different diagram types supported by the notation are used in at least one of the examples, though, and it should be possible to get a general idea of how modeling using Visual REST works.

### 3.10.1 Modeling a Photo Album Application

The photo album example is used as an example in the definition of the meta-model [6], so it makes sense to present a version modeled using the new notation as well.

The photo album is rather simple. A single album consists of many different pictures. An album has a title and a description. A picture has a title, a description and a flag indicating if the picture was rated as excellent or not. A picture also contains binary data for the actual raw image. Suggestions can be made for excellent pictures. A suggestion has a number of positive and negative reviews, a statement and a status. The status indicates the state the suggestion is in (created, under review or done). Additionally, there is a list of all albums and a list of all suggestions as well a list of suggestions under review in the application. For sake of simplicity, media and data types are assumed to be already defined unless stated otherwise. They are part of the final application model in any case and need to be created in the modeling process.

The structural model of the application is shown in figure 3.37. The primary resource types in the application are `Picture` and `Album`. The `Album` resource types is connected to the `PicturesOfAlbum` resource type through an internal link (pictures). The `Picture` elements of that list are also reachable via an internal link (elements). The `Picture` resource type has a subresource type `PictureData` that stores the raw image data and is reachable via an internal link (data). The `Picture` links to the next and previous picture as well as to the album it is contained in (album). The list resource type `AllAlbums` has an internal link to the `Album` (elements). The `Suggestion` is modeled as an activity resource type. It links to the picture under review (picture). A list resource type `AllSuggestions` links to the `Suggestion` (elements) as does the filter resource type `SuggestionsUnderReview` (elements).

With the resource types of the application defined, the next step is the definition of the states of the different resource types as well as the behaviour of the methods supported by individual types in their respective states. For the former, resource state diagrams are used while for the latter method behaviour diagrams are utilized.

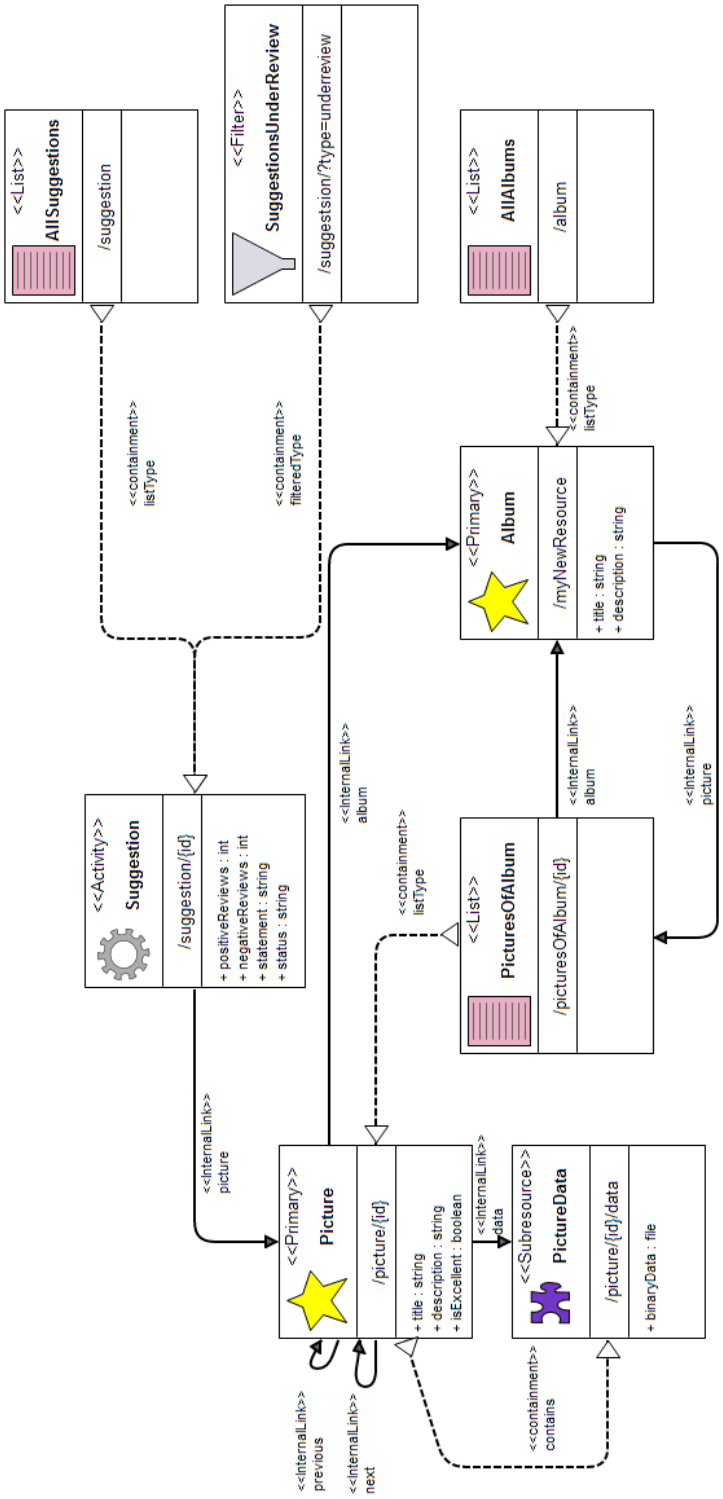



Figure 3.37: The structural view of a photo album

The states of the suggestion resource type are depicted by figure 3.38. The initial state of the resource type is *created* and `PUT`, `DELETE` and `GET` are supported. `PUT` is used to update the statement, status or picture while `GET` can be used to receive that data. `DELETE` can be used to put the suggestion into deleted state in which the status code 410 is returned. The resource moves from the *created* state to the *under review* state when `PUT` is called to set the resource's state attribute to "under review". In the *under review* state, only `GET` and `POST` are supported. `GET` returns the current state of the resource. `POST` allows adding a positive or negative review. Once there have been sufficient positive or negative reviews to mark the picture as excellent or not excellent the resource transitions into the *done* state where only `GET` is supported.

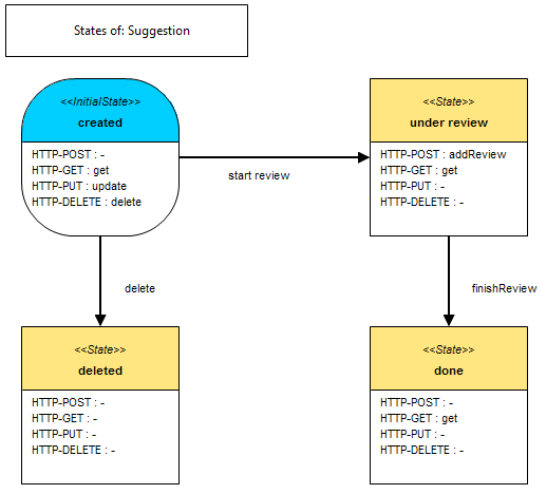



Figure 3.38: The states of a suggestion

Figure 3.39 shows the behaviour of the suggestion resource types `addReview` method. The first conditional action checks if the field `accepted` of the consumed media type `review` is true. If so, it updates the suggestions `positiveReviews`, otherwise the `negativeReviews` are updated. If after that operation the number of positive reviews is greater than two, the linked picture (`suggestedPicture` internal link) is marked as excellent and the *finishReview* internal message is sent to move the resource into the *finished* state. If the number of negative reviews is greater than two, the picture is not marked excellent and the resource is also moved to the *finished* state through the *finishReview* message. Finally, the representation of the suggestion is returned with status code 200.

Figure 3.40 shows the behaviour of the suggestion resource types `update` method. The `update` is available while the resource is in state *created*. Through the update operation, the `statement` and `status` can be update. The `status` is update if the consumed suggestion representation has it status set to "under review". This causes the conditional action

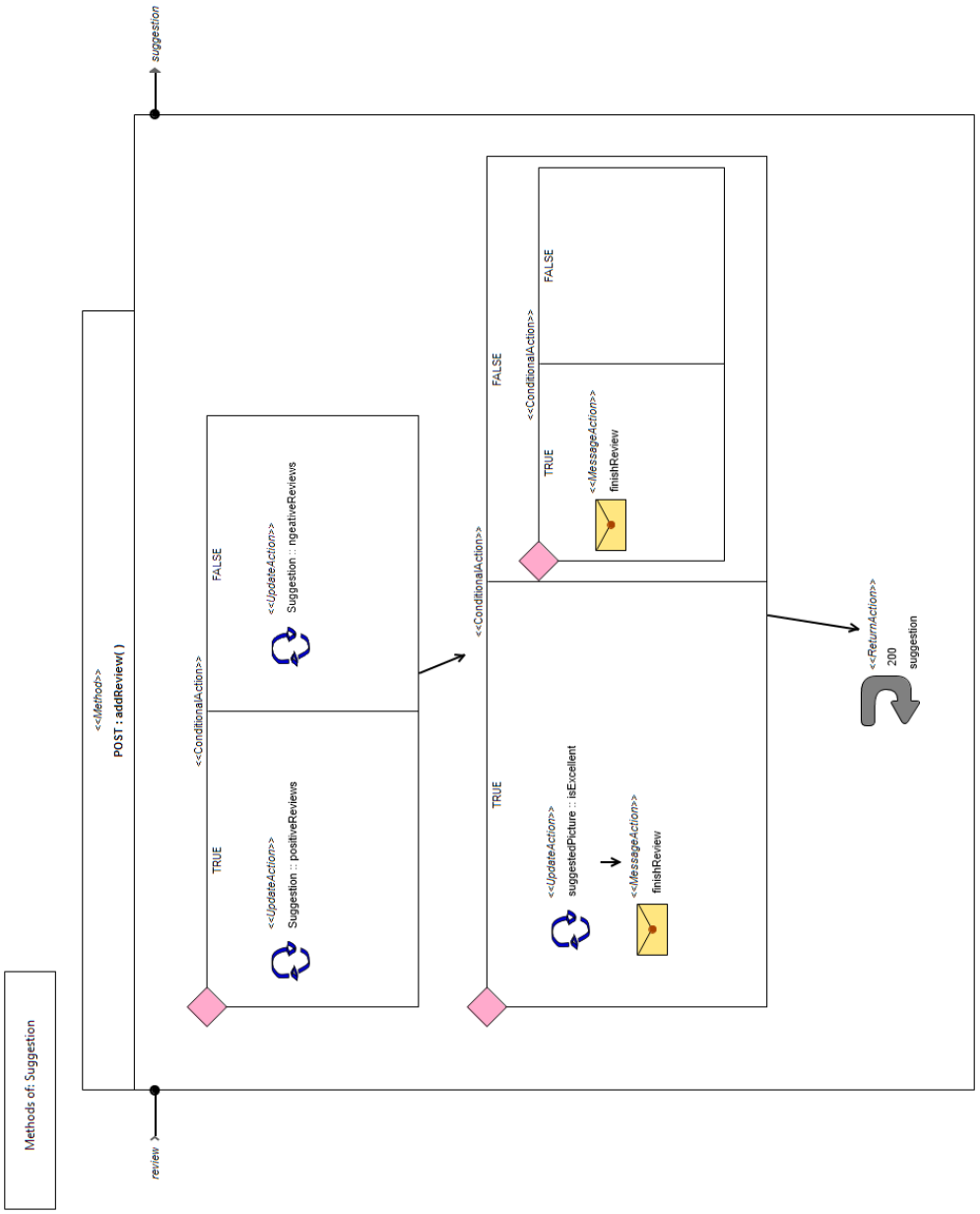



Figure 3.39: The addReview method of the pictureData subresource type

to be executed in its *true* path and set the status as well as send the internal message *start review* which transitions the resource to the *under review* state. On return, the suggestions representation is returned with status code 200.

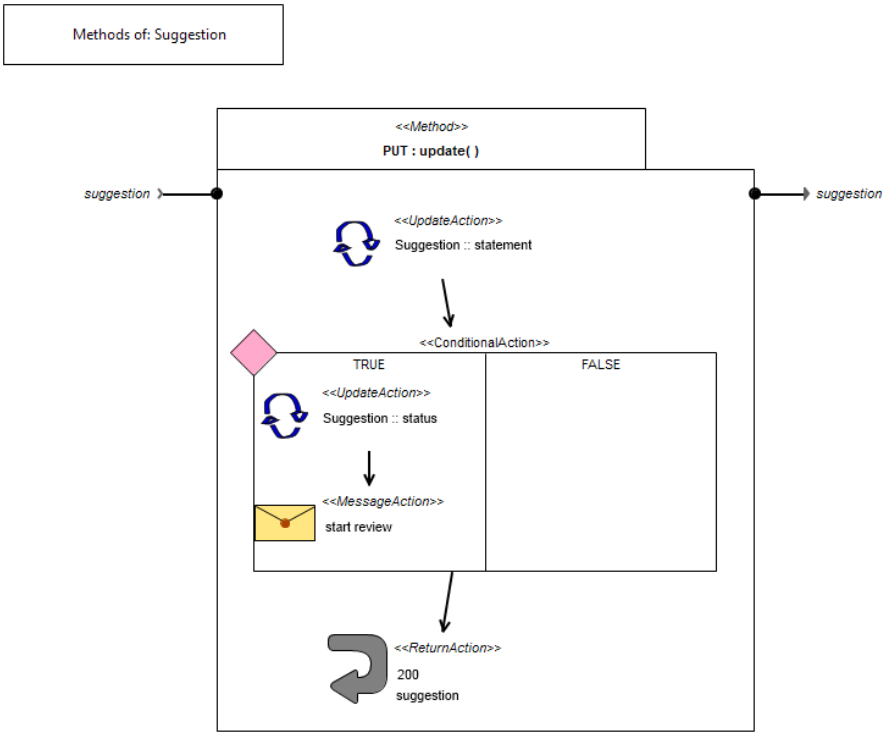



Figure 3.40: The update method of the pictureData subresource type

The states of the `pictureData` subresource type are shown in figure 3.41. The pictureData subresource type only has the initial state *created* which supports the method `get`.

The behaviour of the `PictureData` resource type's `get` method is depicted in figure 3.42. The `get` method returns the binary representation of the `PictureData` attribute `binaryData`.

Definition of the states and methods of the other resource types in the application follows the same schema presented here. There is a state diagram for every resource type and a behaviour specification for every method supported in the individual states.

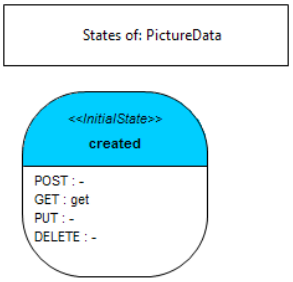



Figure 3.41: The states of the pictureData subresource type

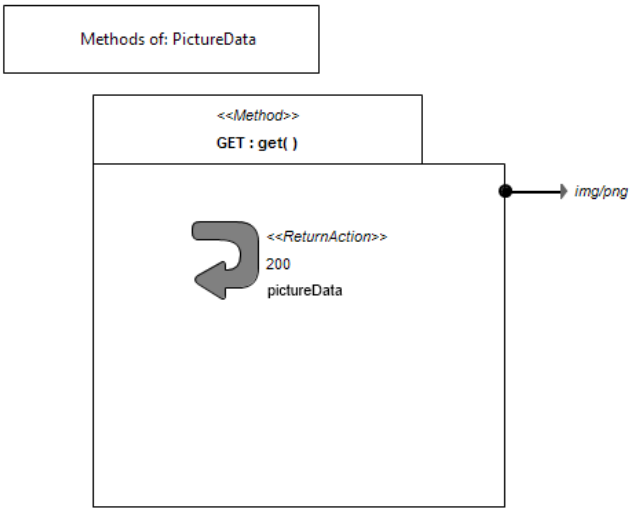



Figure 3.42: The get method of the pictureData subresource type

### 3.10.2 Modeling a Mind Map Application

A mind map is used to represent thoughts around a certain key topic. The purpose of a mind map is to organize and visualize ideas and structure information around them. Gronback [54] uses a mind map as an example for an application based on the Graphical Modeling Framework that allows creating mind maps. Such an application for creating mind maps is created here based on a resource-oriented architecture.

A resource-oriented mind map application consists of only a few resource types. Figure 3.43 shows the structure of such an application.

The only primary resource type in the application is the `Topic`. A `Topic` has a name and a description. It also has children which are found in the `ChildTopics` filter resource type. The entry point into the application is the filter resource type `StartTopics`. It lists all topics that are the root of a mind map. The `AllTopics` list resource type lists all topics, also non-root elements. The `TopicMap` aggregation resource type aggregates `Topic` and its `ChildTopics` into a single resource type.

The media types used by the application are `topic` and `application/json`. A `topic` has a `name` and a `description`, a `isRoot` flag and a list of links to its `children`. It also contains a flag to indicate if the topic can still be worked with or not (`isClosed`). The `application/json` media type is used to provide a JavaScript Object Notation (json) representation of a topic with all its children.

All resource types have an initial state *created*. The `AllTopics`, `TopicMap` and `ChildTopics` only support *get* operations in that state as shown in figure 3.44.

The behaviour of the `TopicMaps` `get` method is shown in figure 3.45. `AllTopics` and `ChildTopics` are similar and not shown here. `TopicMap::get` returns the aggregated topic and its subtopics using the `application/json` media type as representation.

To create new root topics, the `StartTopics` resource type supports POST requests using its `addTopic` method. The states and supported methods of `StartTopics` are shown in figure 3.46. The `addTopic` method is shown in figure 3.47.

The method consumes the `topic` media type. To create the actual topic a creator is used. The representation of the topic is returned afterwards. Similarly, to create a new subtopic, the `Topic` resource type supports POST requests through its `addChildTopic` method that behaves in the same way as that of the `addTopic` method of the `StartTopics` resource type. When a `Topic` resource is updated via PUT and the `updateTopic` method, the resource attributes are updated and if the `isClosed` flag of the consumed media type `topic` is true, an internal message is sent that transitions the `Topic` into the *closed*

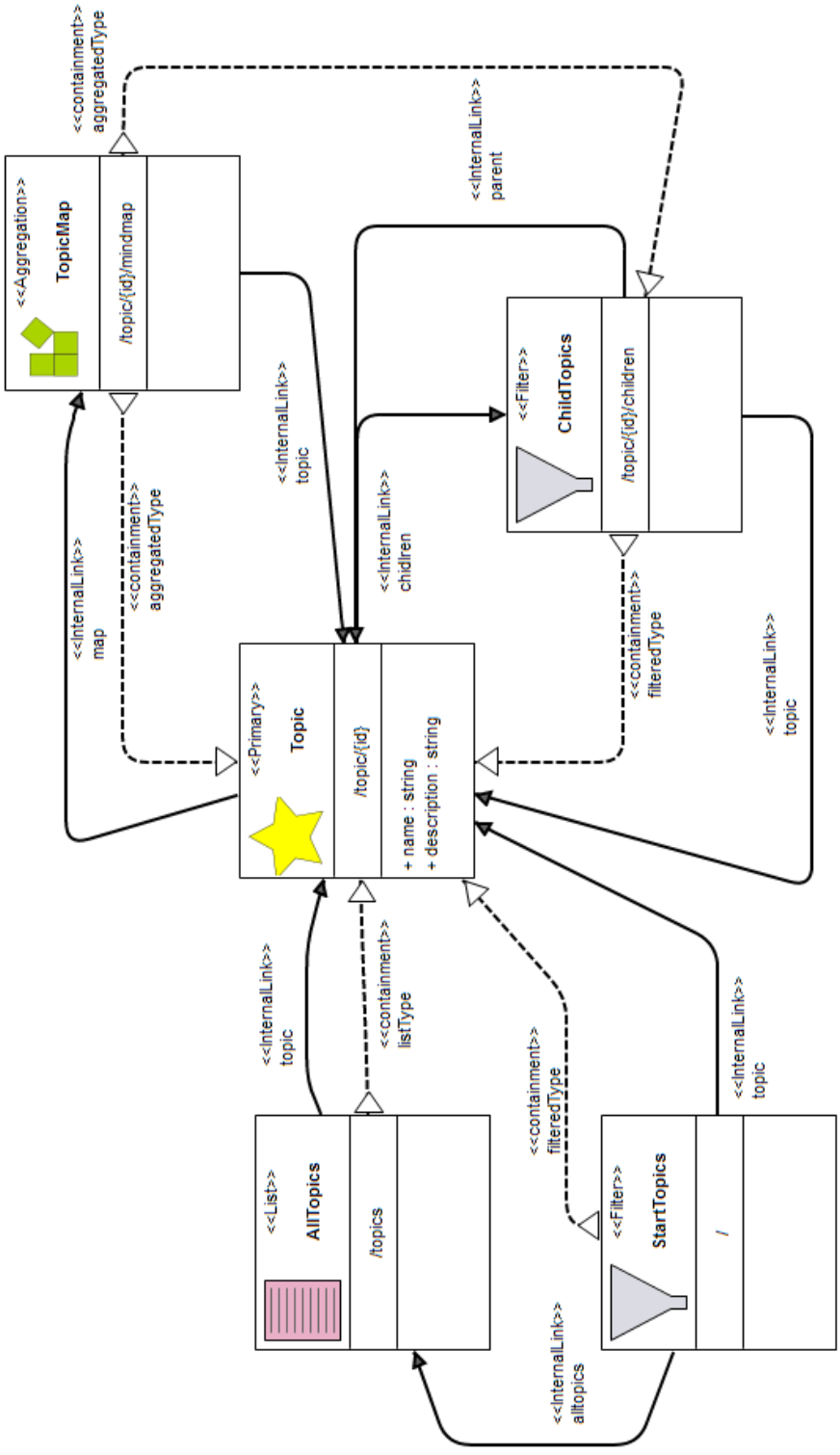
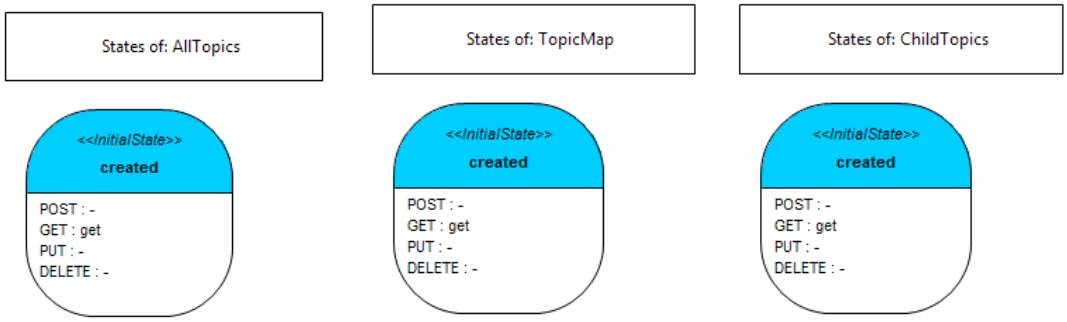



Figure 3.43: The structure of a resource-oriented mind map application




(a) `AllTopics` (b) `TopicMap` (c) `ChildTopics`

Figure 3.44: States of the `AllTopics`, `TopicMap` and `ChildTopics` resource types

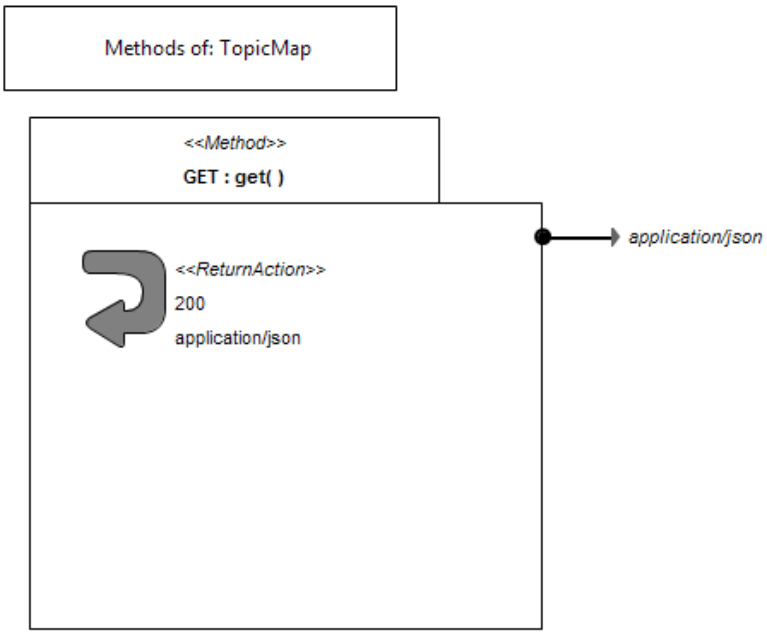



Figure 3.45: Behaviour of `TopicMap::get`

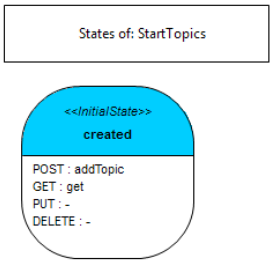



Figure 3.46: States of `StartTopics`

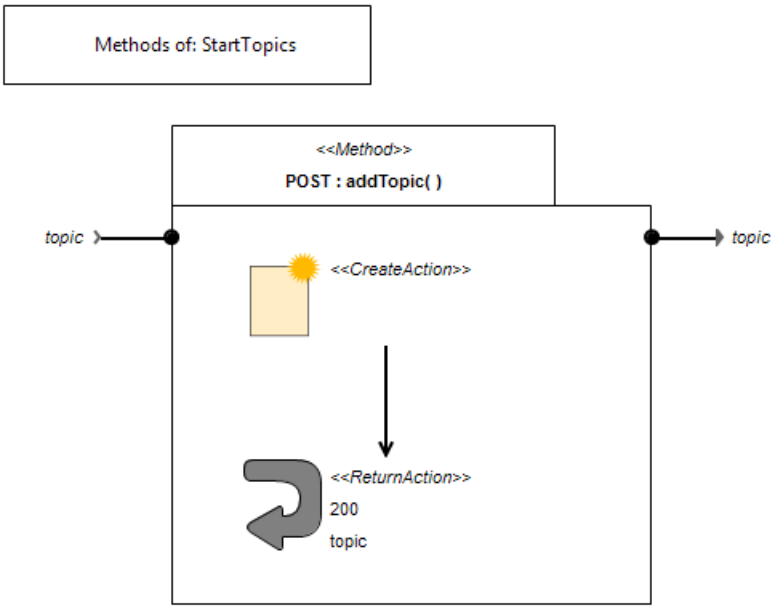



Figure 3.47: Behaviour of `StartTopics::addTopic`

state. In that state, only GET and DELETE are supported, but the resource cannot be modified any longer and no additional subtopics can be added to it. The states of the `Topic` resource are depicted by figure 3.48. The behaviour of the `updateTopic` method is shown in figure 3.49.

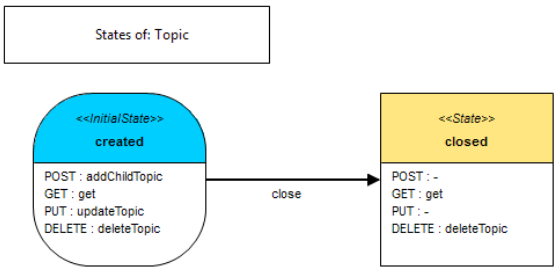



Figure 3.48: States of `Topic`

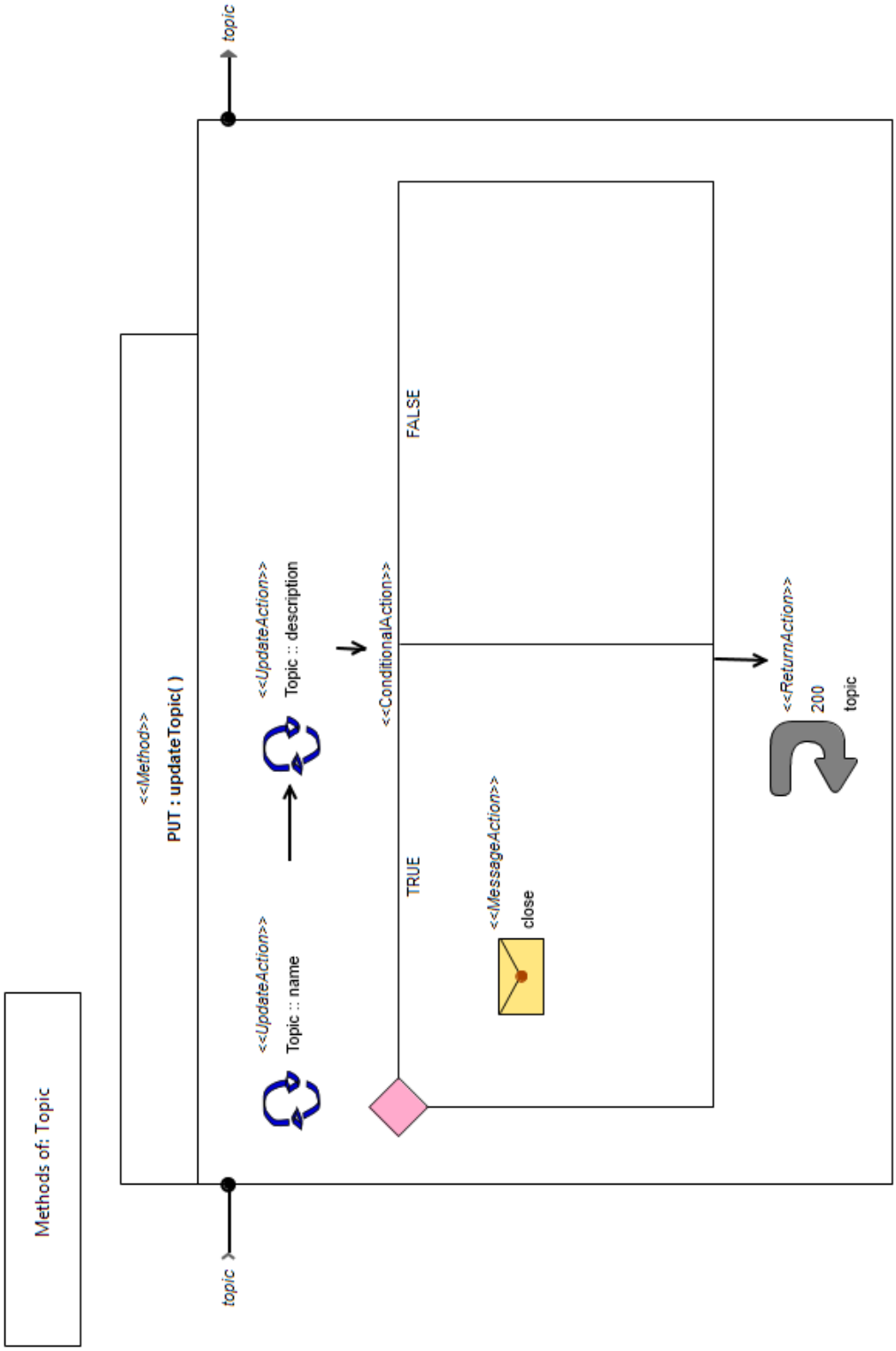



Figure 3.49: Behaviour of `Topic::updateTopic`

### 3.10.3 Modeling a Bookshop Application

The book shop example has been used already when explaining the different resource types in section 2.3. A bookshop contains three primary resource types: `Book`, `Author` and `ShoppingCart`. There is a list resource types containing `Books` and a paging resource type containing `Authors`. The `Checkout` activity resource type is connected to the `ShoppingCart` via an internal link. An `Author` has a subresource `Biographie`, holding the author's biographie. There is also a projection resource type that only lists the book titles contained in the book list resource type and an aggregation resource type that aggregates books and their authors into a single resource type. Finally, a filter resource type is available to provide a list of that aggregation, filtered by author. Figure 3.50 shows the structure overview diagram for this example. The other views are omitted here. By and large, they follow what was presented in the examples before.

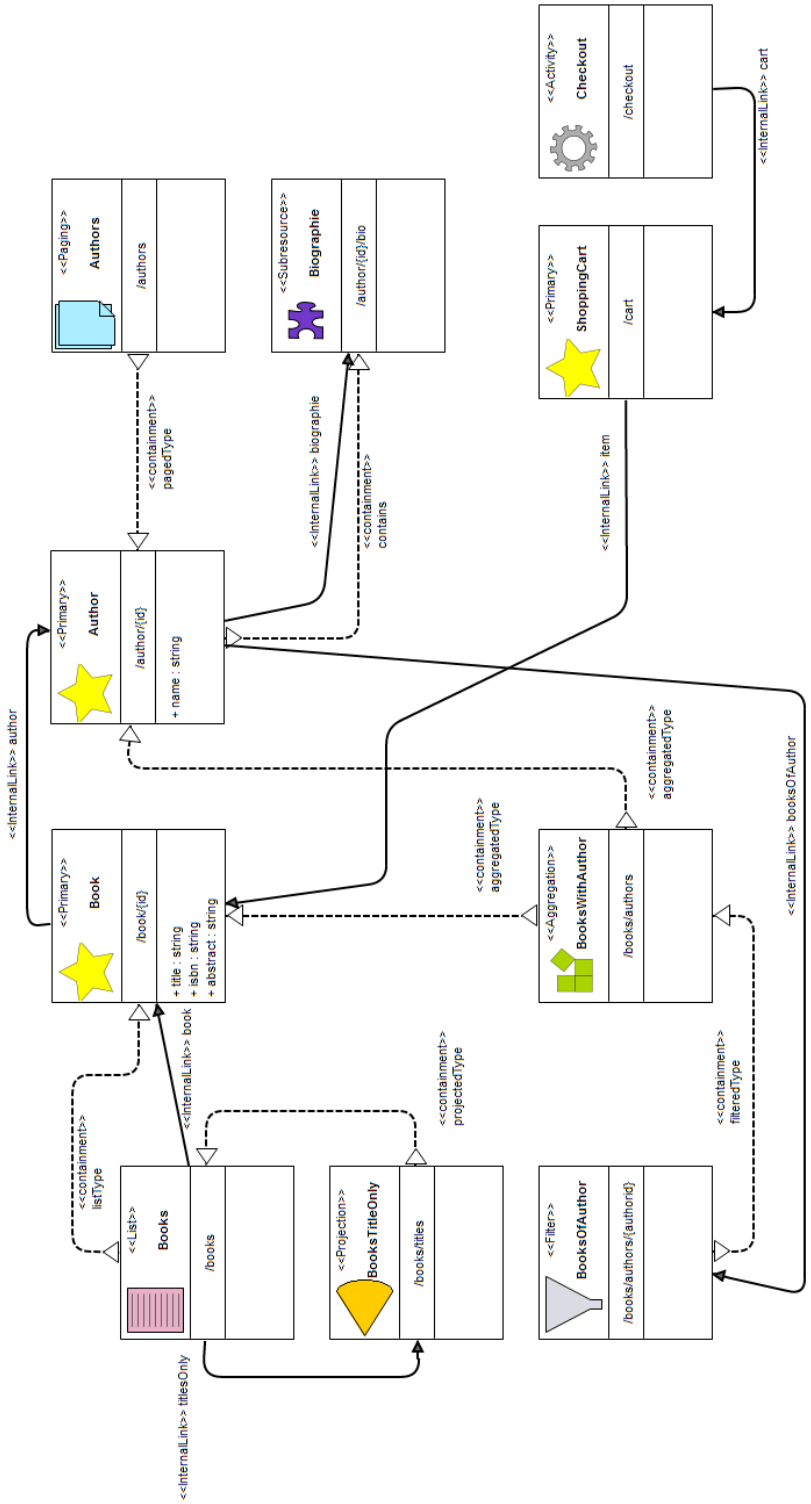



Figure 3.50: The structure overview diagram for a bookshop application

# 4 Evaluation of Visual REST

The definition of a graphical notation is just one aspect in creating a new language for resource-oriented applications. It is also necessary to evaluate the language through empirical studies to assess if Daniel Moody's criteria have been successfully applied. This chapter presents the evaluation that was performed with regard to Visual REST. It outlines the basic idea behind the questionnaire and its design. Subsequently, the results of the evaluation are presented and discussed.

## 4.1 The Research Idea

Different methods for the evaluation of graphical systems, in general [42], and visual languages, in particular [43], also with a focus on the criteria defined by Moody [46], have been published. The idea of all these previous studies is different from the study presented here, though. All of them are looking at existing languages. They are not developing new ones. They also expect users to be familiar with the system and to provide judgement based on that past experience. Blackwell et al. [42] as well as Bobkowska [43] use open questions in their evaluations which makes result analysis harder [33, p. 48]. Johansson et al. [46] use closed questions but expect probands to be familiar with Daniel Moody's work [8].

The focus of the questionnaire presented here is different on various levels. Familiarity with the system as a precondition to the actual evaluation is not an option because there is no system to be familiar with yet. The probands also cannot be expected to be familiar with Moody's work. Furthermore, the use of open questions has to be limited due to the limited amount of time available for the evaluation. Additionally, the questionnaire needs to be usable by probands without the need for monitoring by a supervisor.

The question answered by the questionnaire presented here is: "Have Moody's criteria been applied well in the development of the language?" The target audience of the questionnaire is clearly the potential users of the new language.

## 4.2 Design of the Questionnaire

The design of a questionnaire with all constraints mentioned in section 4.1 is not a task to be taken lightly. Using closed questions allows for a more straight-forward evaluation in the end, yet the questions need to be carefully designed and validated through trial runs. The limited previous knowledge of probands as well as the unsupervised execution of the survey also need to be taken into account. LimeSurvey [87] is used to provide the questionnaire to users via an online tool that can be used without supervision.

The questionnaire is divided into 7 + 2 categories. One category for each of Moody's criteria [8] except *Visual Expressiveness* and *Semiotic Clarity*. One to gather information on the proband, specifically on his or her previous knowledge in relevant fields; and another category for generic questions at the end. A question on the probands background is, for example, if the proband has any knowledge of REST and to what extent. There is only one generic question at the end; it allows entering an open comment as feedback to the language as well as to the questionnaire.

Visual expressiveness is left out of the questionnaire because there has already been plenty research done on the effect of using multiple visual variables [88–90] and it is not the intention nor goal of this questionnaire to try and repeat or even re-validate that work. Semiotic clarity is not evaluated because it is a rather binary decision. The notation either fulfills the semiotic clarity criteria or it does not. There is not much to evaluate based on user opinion.

The iconic representations of the different resource types are designed to be semantically transparent. To validate this, the proband is asked to associate the correct resource type with a symbol presented as shown in figure 4.1.

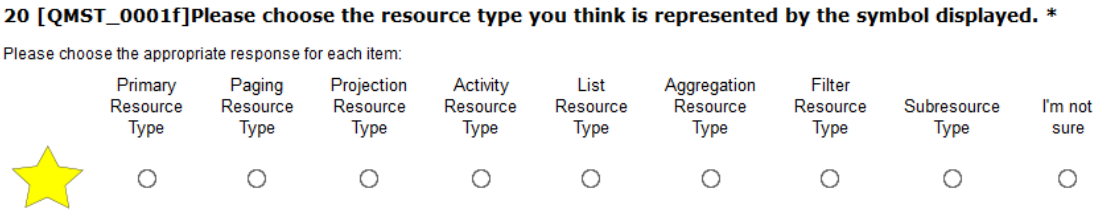



Figure 4.1: A question to evaluate semantic transparency

To decide if symbols are perceptually discriminable, the proband is asked to compare every icon against all others and judge if they can easily be discriminated or not. Icons are shown pairwise to make judging easier as shown in figure 4.2.

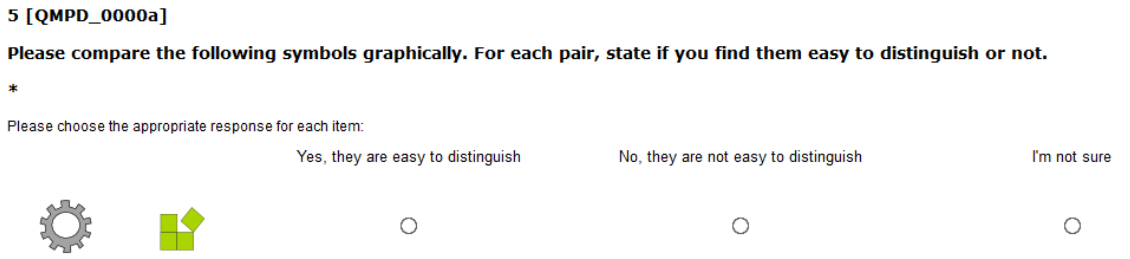



Figure 4.2: A question to evaluate perceptual discriminability

To evaluate complexity management, the proband is asked to provide an opinion on the complexity management mechanisms provided as part of the notation. This is done by presenting examples of diagrams and asking the proband to select if there is agreement with certain statements. For example, the proband is presented with a complete structural view and the same structure broken down into two distinct diagrams. The proband then has to decide if the method of breaking down the diagrams seems suitable to manage application complexity.

Dual coding is evaluated by presenting the user with representations of certain concepts with and without textual information as shown in figure 4.3. The proband is then asked to state if adding the textual representation helps or not. The proband is also asked if the UML stereotype-like representation of the textual annotation is appealing.

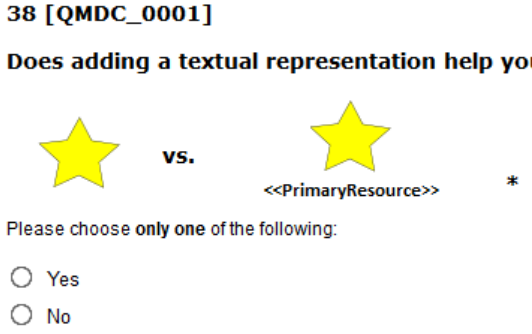



Figure 4.3: A question to evaluate dual coding

The evaluation of graphic economy is performed by presenting the proband with example diagrams that contain all possible graphical elements of any of the views available in Visual REST. The proband then has to indicate if there seem to be too many different graphical symbols in the notation.

During the evaluation of cognitive fit the proband is asked if the diagrammatic complexity seems to high for a rather complex example of an application and if the notation seems suitable for the envisioned media; whiteboard and computer.

The basic idea behind all questions is the validation of the design decisions made in the process of developing the Visual REST notation. Appendix A contains the complete questionnaire for reference.

## 4.3 Questionnaire Results

This section presents the results of the questionnaire grouped by the principle that is covered by the questions. In total, 30 opinions could be gathered through the use of the LimeSurvey [87] online survey tool.

### 4.3.1 User Background

The target audience for the questionnaire were potential users of a visual modeling language for resource-oriented applications. Thus, users are expected to have some knowledge of REST. About 93% of the probands said that they have a least heard about REST. Only two probands stated that they do not know what REST is. Figure 4.4 shows the distribution of the answers to this question.

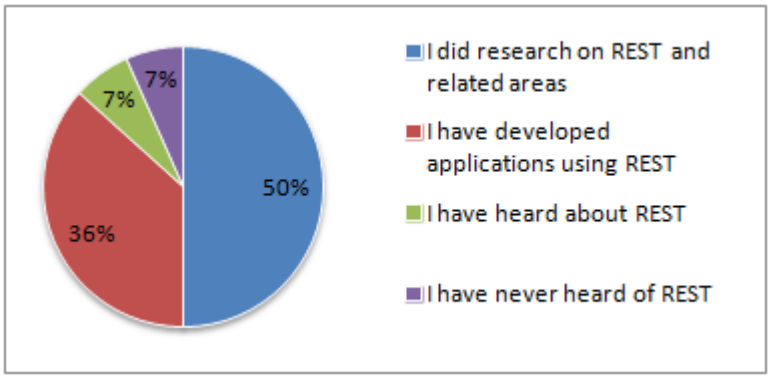



Figure 4.4: How would you rate your knowledge of REST?

When asked about their knowledge of graphical modeling languages such as UML 2 [5], BPMN [91] or state charts, 87% of the probands were familiar with at least one of these languages. The UML 2 diagram types *Activity Diagram* and *Class Diagram* had been

used or were at least known by 100% of those who were familiar with graphical modeling languages. Only 8% had neither used nor knew the *State Machine Diagram*. Figures 4.5 and 4.6 give an overview of the distribution.

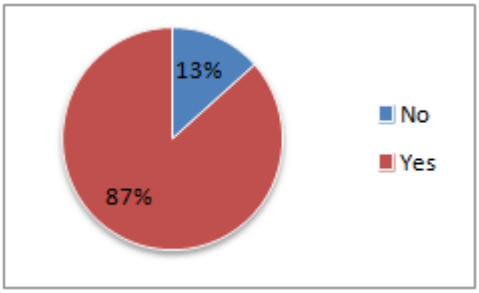



Figure 4.5: Are you familiar with graphical notations?

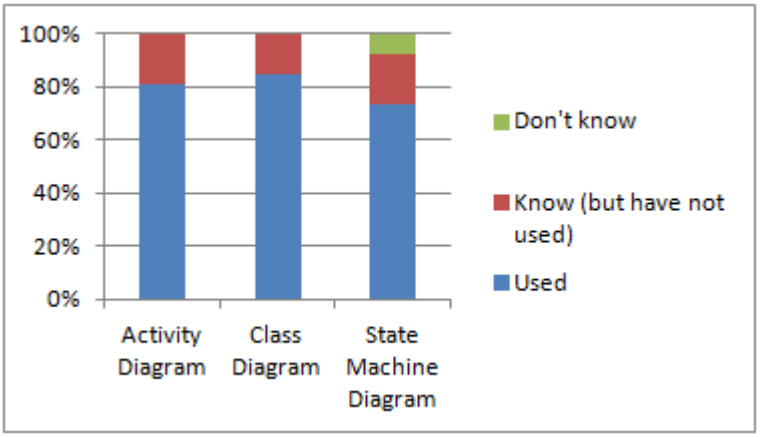



Figure 4.6: Knowledge of different UML 2 diagram types

Judgement of graphical languages can depend on the knowledge of graphical languages. Only 20% of the probands have never used a graphical editor before, the vast majority however has used them before (43%), use them regularly (30%) or even do research in the field of visual notations (7%), as shown in figure 4.7.

### 4.3.2 Perceptual Discriminability

As was stated before, the evaluation of perceptual discriminability is done through the pair-wise comparison of the graphical symbols chosen as representation for the various meta-model concepts. For each diagram type in the Visual REST notation, there is

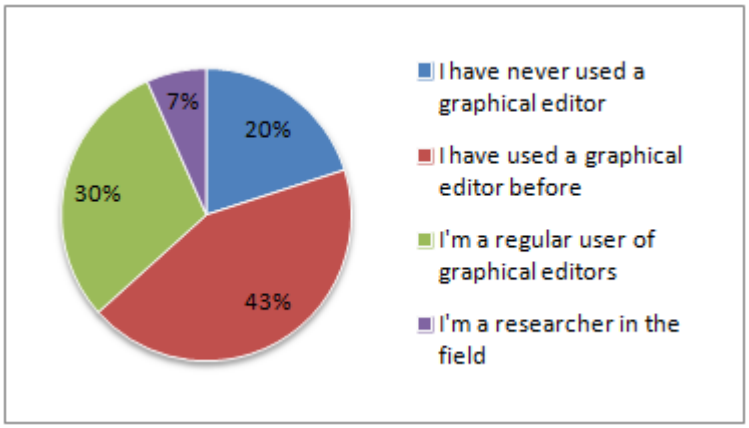



Figure 4.7: How would you rate your knowledge of visual notations?

a distinct set of visual elements. Therefore, the evaluation of the survey results also happens per diagram type instead of for the complete notation.

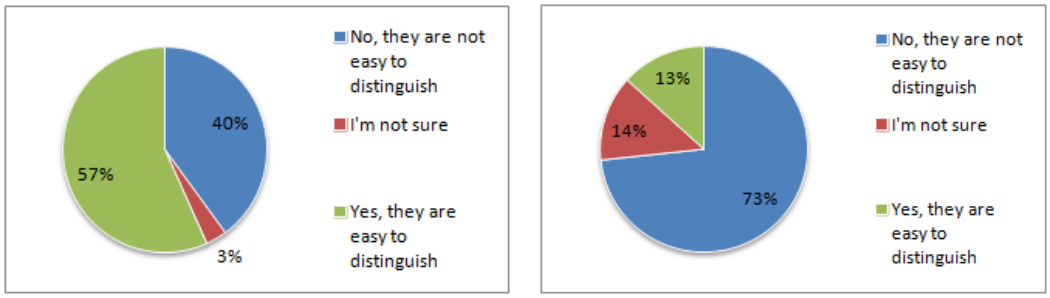
The resource structure diagram is the diagram type of the Visual REST notation that has the highest number of different graphical symbols. Table 4.1 provides an overview of the comparison matrix as produced by the pair-wise comparison of the symbols. The numbers given for each pair is the percentage of answers that found the two symbols in that matrix easily distinguishable.

| | *Activity* | *Aggregation* | *Paging* | *Filter* | *Primary* | *Subresource* | *List* | *Projection* |
|---|---|---|---|---|---|---|---|---|
| Activity | - | 100 | 96,7 | 80 | 80 | 70 | 100 | 90 |
| Aggregation | - | - | 83,3 | 86,7 | 90 | **56,7** | 80 | 76,7 |
| Paging | - | - | - | 76,7 | 93,3 | 86,7 | **13,3** | 90 |
| Filter | - | - | - | - | 83,3 | 86,7 | 83,3 | 80 |
| Primary | - | - | - | - | - | 83,3 | 90 | 73,3 |
| Subresource | - | - | - | - | - | - | 93,3 | 80 |
| List | - | - | - | - | - | - | - | 93,3 |
| Projection | - | - | - | - | - | - | - | - |

Table 4.1: Do you think the symbols are easily distinguishable?

Notably, most of the representations of the different resource types seem to differ enough to be easily distinguishable. Only Subresource and Aggregation as well as List and Paging

resource type are not well discriminable. Figure 4.8 gives an overview of the distribution of opinions on those two critical resource type pairs.




(a) Subresource vs. Aggregation (b) List vs. Paging

Figure 4.8: Rating of the discriminability of critical resource type pair

When asked to compare expanded and collapsed version of the resource type representation, the majority of the probands (57%) stated that the version without an added box (collapsed) is easier to distinguish from the other resource symbols. Only 23% found that the box made it easier to discrminiate resource types. 20% were indifferent and did not see any effect, as shown by figure 4.9.

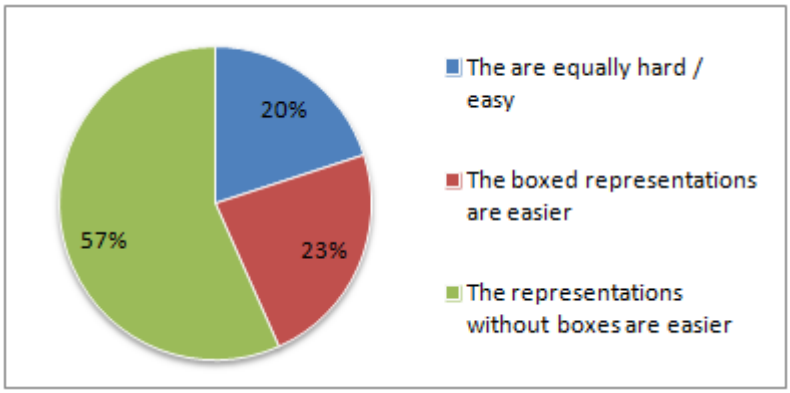



Figure 4.9: Does adding a box around the resource types have an effect on discriminability?

The two types of connectors in the resource structure diagram - internal link and containment - were also easily distinguishable as shown in figure 4.10.

The resource states diagram is the diagram type that contains the least different graphical symbols. State and Initial State differ in shape and color and are thus easily distinguishable by probands, as figure 4.11 shows.

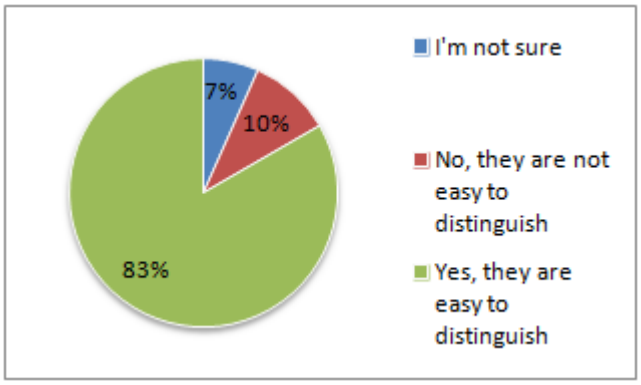



Figure 4.10: Are the connectors easily distinguishable?

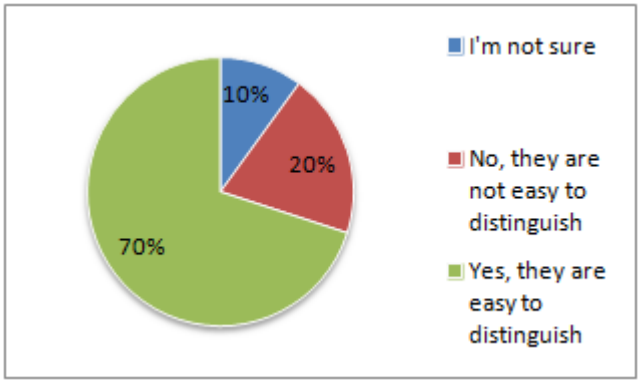


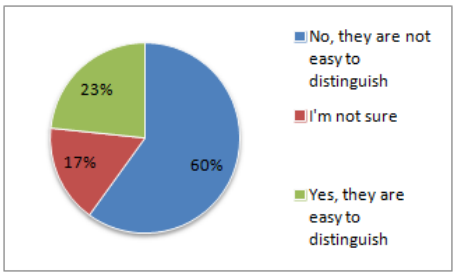

Figure 4.11: Are the state diagram elements easily distinguishable?

The resource method behaviour diagram again contains more elements and is by nature a bit more experimental. Again, probands were asked to do a pair-wise comparison of the different symbols used in that diagram type. The results are shown by table 4.2. The numbers given are again the percentage of probands who said that symbols were easily discriminable.

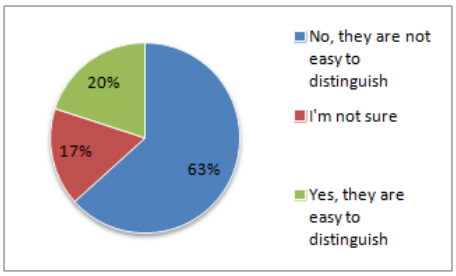
Especially the *Create Action* seems to be easily confused with both *List Add Action* and *List Remove Action*, as shown in figure 4.12. *List Add Action* and *List Remove Action* are also more easily confusable, but not to the extent they are confusable with the *Create Action* (see figure 4.12(c)).

| | *List Add Action* | *List Remove Action* | *Message Action* | *Return Action* | *Update Action* | *Create Action* |
|---|---|---|---|---|---|---|
| List Add Action | - | **46,7** | 83,3 | 90 | 90 | **20** |
| List Remove Action | - | - | 93,3 | 93,3 | 90 | **23,3** |
| Message Action | - | - | - | 90 | 93,3 | **53,3** |
| Return Action | - | - | - | - | **36,7** | 90 |
| Update Action | - | - | - | - | - | 93,3 |
| Create Action | - | - | - | - | - | - |

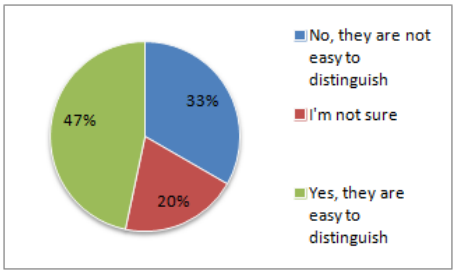
Table 4.2: Do you think the symbols are easily distinguishable?




(a) Create Action vs. List Add Action




(b) Create Action vs. List Remove Action




(c) List Add Action vs. List Remove Action

Figure 4.12: Rating of the discriminability of critical action pairs

### 4.3.3 Semantic Transparency

The evaluation of semantic transparency is achieved through an association task. Probands were asked to identify semantic concepts behind syntactic elements. The results of that association task for the resource structure diagrams resource types are listed in table 4.3. The numbers of correct associations are marked bold. Figure 4.13 provides an overview of the association distribution of the semantically least clear symbols.

| | Activity Resource Type | Aggregation Resource Type | Paging Resource Type | Filter Resource Type | Primary Resource Type | Subresource Type | List Resource Type | Projection Resource Type | I'm not sure |
|---|---|---|---|---|---|---|---|---|---|
| Activity | **76,7** | | | 3,3 | | | | | 20 |
| Aggregation | | **56,7** | 3,3 | | | 10 | | 3,3 | 26,7 |
| Paging | 3,3 | 3,3 | **63,3** | | 6,7 | 3,3 | 6,7 | | 13,3 |
| Filter | | 6,7 | | **80** | | | | 3,3 | 10 |
| Primary | 3,3 | | | | **50** | | | | 46,7 |
| Subresource | | 3,3 | | | | **46,7** | | 3,3 | 46,7 |
| List | | | 10 | | 20 | | **56,7** | | 13,3 |
| Projection | | | | 3,3 | | 16,7 | | **26,7** | 53,3 |

Table 4.3: Association of symbols with actual semantic types

The semantic transparency of the connection arrows is not that good, as shown in figure 4.14. For both internal links and containment connections more than half of the probands were unable to make the connection from syntax to semantics.

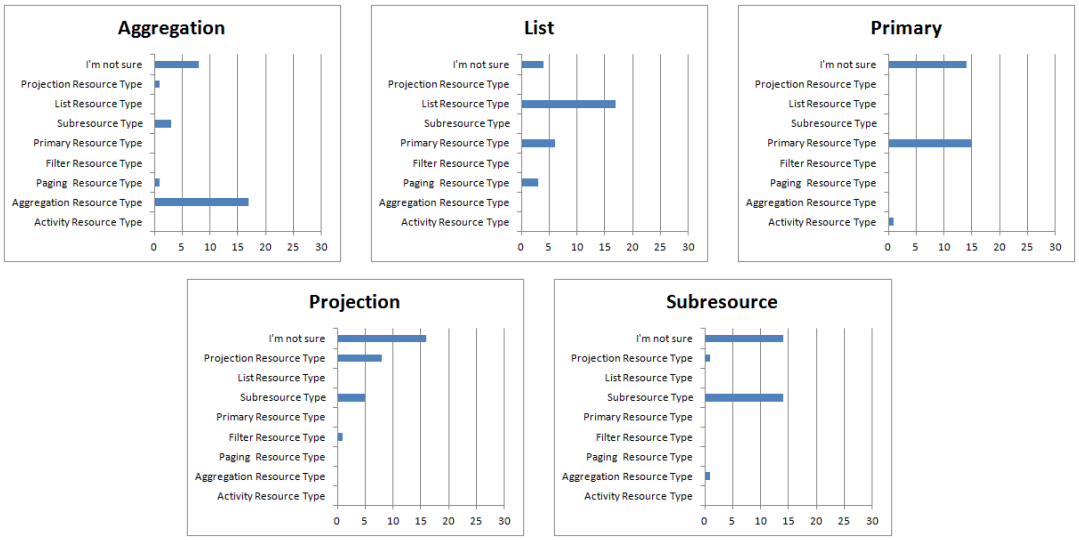



Figure 4.13: Distribution of associations for the semantically least clear resource types

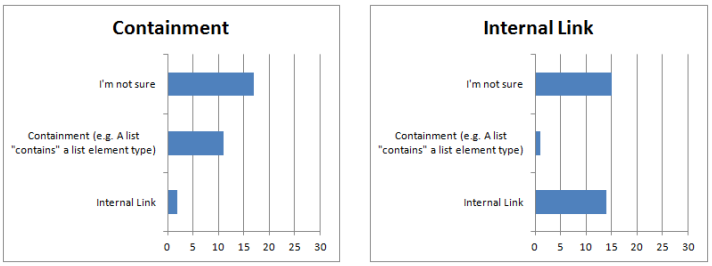



Figure 4.14: Distribution of associations for the resource type connections

Similarly, the semantics of the states as presented by the resource states diagram are not clear to probands either. It can be seen that about half are undecided, as depicted in figure 4.15.

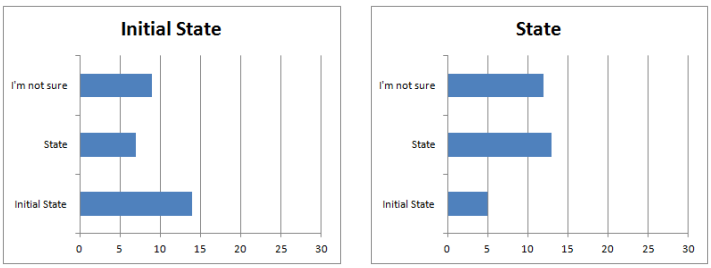



Figure 4.15: Distribution of associations for the resource states

The evaluation of the resource method behaviour diagram and its different types of actions is shown by table 4.4. Again bold values indicate correct associations made. Numbers are in percent correctly associated. As figure 4.16 shows, there is quite some uncertainty about the *Conditional Action*, *Create Action* and *Update Action*. As shown in figure 4.17, the *List Add Action* seems to be semantically less clear than the *List Remove Action*.

| | Conditional Action | List Add Action | List Remove Action | Create Action | Message Action | Return Action | Update Action | I'm not sure |
|---|---|---|---|---|---|---|---|---|
| Conditional Action | **56,7** | | | | | | | 43,3 |
| List Add Action | | **63,3** | | 26,7 | | | 3,3 | 6,7 |
| List Remove Action | | | **83,3** | | | | 3,3 | 13,3 |
| Create Action | | | | **46,7** | 6,7 | | 10 | 36,7 |
| Message Action | | | | | **83,3** | | | 16,7 |
| Return Action | | | | | | **86,7** | | 13,3 |
| Update Action | | | | | | | **46,7** | 53,3 |

Table 4.4: Association of symbols with actual semantic actions

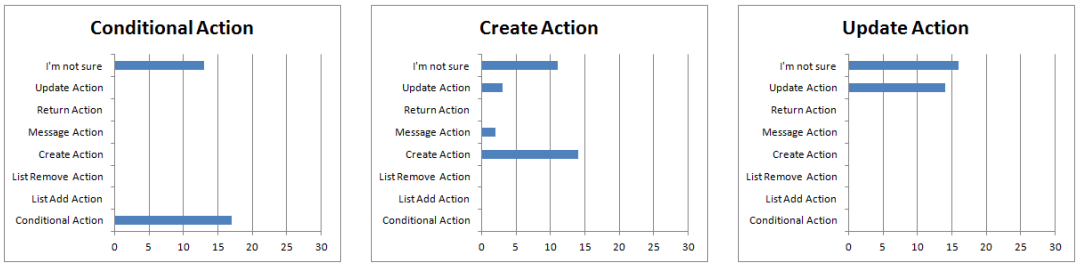



Figure 4.16: Distribution of associations for *Conditional Action*, *Create Action* and *Update Action*

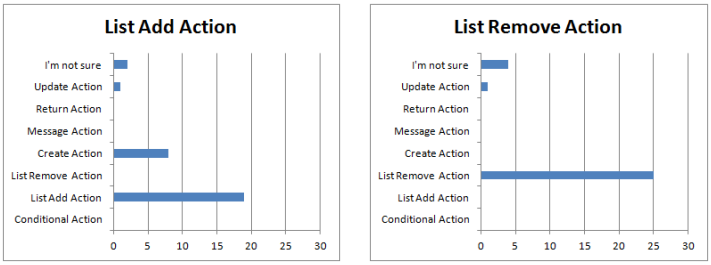
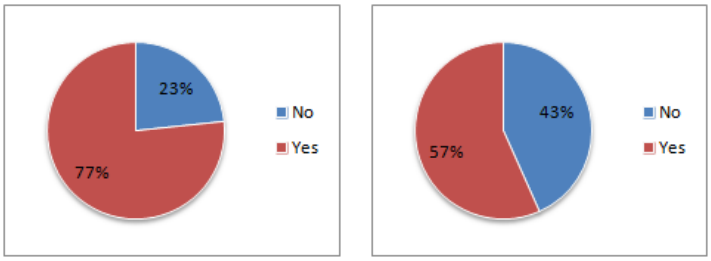



Figure 4.17: Distribution of associations for *List Add Action* and *List Remove Action*

### 4.3.4 Complexity Management

In the evaluation of complexity management, probands are asked to state if they think the complexity management mechanisms offered by Visual REST are usable. Since Visual REST offers complexity management by means of partitioning structure and method behaviour diagrams, the probands were asked to provide their opinion on those diagram types. As figure 4.18 shows, probands generally think that partitioning will help. However, this is more clear for structure diagrams, where 77% stated that partitioning will be helpful. Partitioning of method diagrams as offered by Visual REST only seems useful to 57% of the probands.



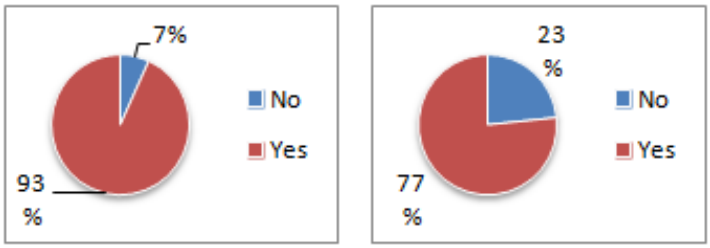

(a) Structure Diagram (b) Method Behaviour Diagram

Figure 4.18: Do you think the ability to partition the diagrams helps in managing application complexity?

### 4.3.5 Cognitive Integration

Cognitive integration is achieved by adding markers (i.e. signposting) at the top of subordinate diagrams such as the resource method behaviour and the resource states diagram. Probands were asked to state if those markers help in identifying the resource type a method or state diagram belongs to. As figure 4.19 shows, this is generally the case.

Furthermore, the resource states always present all possible method types, not only those supported by the actual state. Probands where asked to indicate if they think this is helpful or not, which is generally not the case, as shown in figure 4.20.




(a) Resource State Diagram (b) Method Behaviour Diagram

Figure 4.19: Does the marker at the top help in identifying the resource type the state/method belongs to?

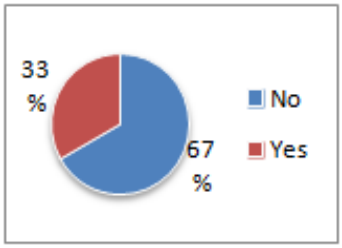



Figure 4.20: Does displaying all possible method types in the states help?

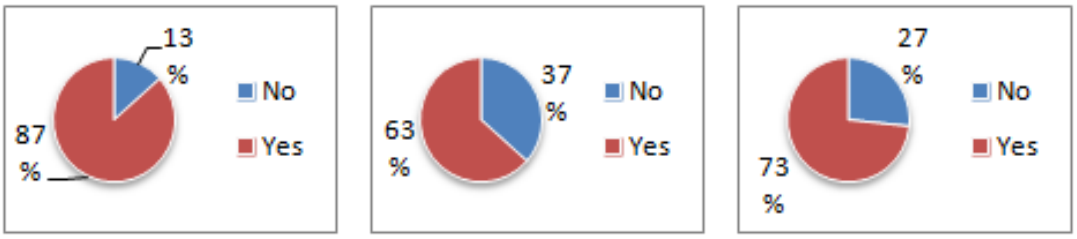
### 4.3.6 Dual Coding

When asked if dual coding helped in grasping the meaning of the iconic representations, probands generally agreed, as shown in figure 4.21.

When asked if the textual representation in the form of an UML stereotype appealed to them, probands were undecided. About half said yes, about half said no or were not sure, as shown in figure 4.22.

### 4.3.7 Graphic Economy

To identify if graphic economy is still given for the individual diagram types, probands were asked to state if they thought that there were too many graphical symbols in the different diagrams. As shown in figure 4.23, for a resource structure diagram, half of the probands thought that there were too many graphical symbols and for method behaviour diagrams more than half thought so.




(a) Resource Structure (b) Resource States (c) Method Behaviour

Figure 4.21: Does adding a textual representation help you grasp the meaning of the displayed symbol?

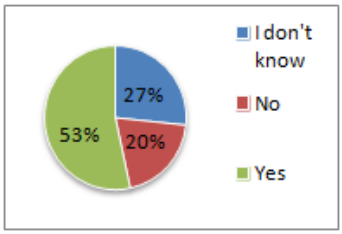
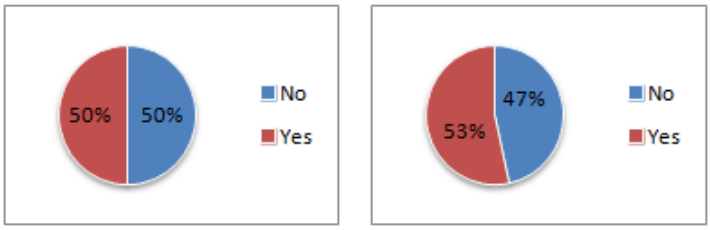



Figure 4.22: Does using UML stereotypes appeal to you?

However, for the resource state diagram, all probands except one thought that there the number of distinct graphical symbols is not too high, as shown in figure 4.24.

### 4.3.8 Cognitive Fit

When asked if they felt overwhelmed by the mass of information presented on a resource structure diagram showing a rather simple book shop application, 67% of the probands felt that way. Also 67% thought that the notation will not be usable on a whiteboard when having to draw the symbols manually. However, 77% thought that the notation will be usable if a dedicated editor was available. Figure 4.25 provides an overview of the distribution.




(a) Resource Structure (b) Method Behaviour

Figure 4.23: Are there too many different graphical symbols?

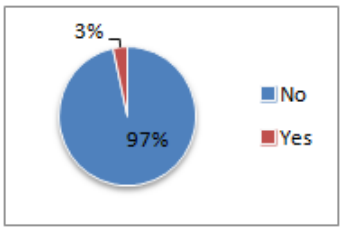


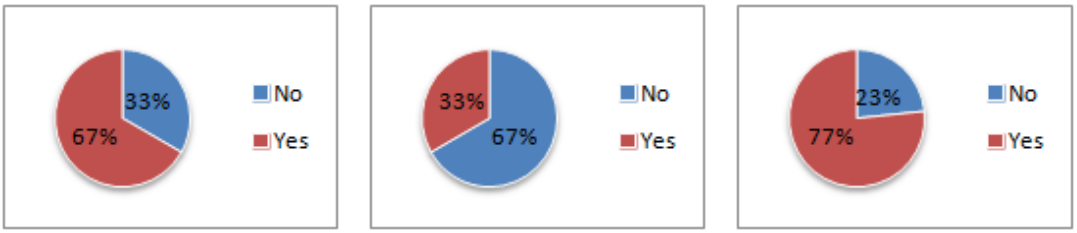

Figure 4.24: Are there too many different graphical symbols on the resource state diagram?

## 4.4 Discussion

The results presented in this chapter show that there is still a lot of room for improvement in the design of the notation. They also allow for some interpretation and speculation. Additional evaluations with a refined version of the questionnaire might prove to be valuable as the Visual REST notation evolves.

A major problem, also judging from the comments that were collected as part of the questionnaire (see appendix B), is a lack of initial understanding of the resource types as defined by Tilkov [4]. Moreover, without supervision, probands seem to drift from judging the pure syntax and also tend to judge semantics as well. Some comments indicated that probands generally thought that the distinction between the different resource types is not necessary to the extent it was done in the meta-model and thus in Visual REST. The following comment is an example:

> i think there are some fundamental flaws in the design here. for example, ”sub-resource” is not a needed representation; just ”resource” will do just fine [...]




(a) Do you feel overwhelmed? (b) Is the notation usable on a whiteboard? (c) Is it usable with an editor?

Figure 4.25: Judgement of the cognitive fit criterion

Furthermore, probands did in at least two cases mistake the notation presented to them as a client-server contract rather than as a mere server-side application model, as the following comment shows:

> [...] Don't take me to mean what you have is not useful. I only think it exclusively focuses on the server, and helps in server implementation [...]

As the proband correctly states, the server implementation is the focal point of the notation. However, this was apparently not clear when taking the questionnaire. The proband expected Visual REST to be more than just that. Another comment goes into the same direction:

> Sorry, I don't see the point. REST has a uniform interface, so every GET PUT etc will be the same. The resources will be different, as will their media types. How do these diagrams help in any way? What are they for? What problem are they solving?

Also here, the proband did not seem to grasp that Visual REST intends to build the model of a server rather than of a client-server contract. Furthermore, the proband does not seem understand the intention behind modeling the states and behaviour of resources. Lastly, some probands found the notation limiting REST to HTTP:

> [...] modeling REST using a single protocol (HTTP) is needlessly limiting [...]

In fact, Visual REST actually allows the use of arbitrary verbs. In the example presented in the questionnaire, only HTTP verbs were presented to probands, though.

Setting aside the comments, the method behaviour diagram does not work well for most

probands. It should possibly be replaced by a textual language that also integrates condition modeling in the future. This will also likely make the notation as a whole more manageable for users. Generally, the notation seems to be too complex at some points. More than 60% percent of the probands felt overwhelmed by the bookshop examples structure diagram. This goes to show that partitioning (as supported by the notation today already) or by further reduction of visual elements. Furthermore, the iconic representations need improvement. Especially the projection and subresource type but also primary, aggregation, list and paging resource type need to be revised.

However, some of the icons carry their meaning well and are perceptually easy to discriminate from the others. Filter and activity resource types are an example of excellence with regard to semantic transparency. Activity versus aggregation resource type are an example of excellence in perceptual discriminability. Moreover, probands generally found the possibility to partition the structure diagram useful and found the parent resource type indicators on the resource states and methods diagrams useful.

All in all, there seem to be some good approaches in the notation and it is generally usable, but more refinement is needed. It is interesting to see how different probands associations with certain symbols can be. These insights need to be taken into account during the refinement of the next version of the notation. The process presented for deriving the notation already accounts for such a feedback loop.

# 5 The Visual REST Eclipse Plugin

Developing an editor for the Visual REST graphical modeling language is another part of this thesis. This chapter first describes the development environment that was used in the development of the Visual REST editor. It then continues to explain the process which the development followed. The architecture and general structure of the editor is covered as well as the build and test environment used.

## 5.1 Development Environment

Development of the Visual REST editor is based on the Eclipse Rich Client Platform (RCP) [47]. RCP provides the the generic application framework all other Eclipse applications build on.

The Eclipse Modeling Framework (EMF) [50] is used to provide basic support for meta-models through ecore. EMF provides a model-driven approach to generating simple textual editors for ecore based models. Using a so called *genmodel* plus the ecore meta-model for resource-oriented applications [6] results in said editor which is already usable for simple textual editing of application models. EMF is also responsible for providing transaction support and persistence for application models through an Application Programming Interface (API).

To allow building graphical editors, the Graphical Editing Framework (GEF) [52] is available as an Eclipse project. It provides the API that is the basis for any other higher-level API such as Graphiti [55]. It is also the target of the code generation done by the Graphical Modeling Framework (GMF) [53].

Due to GMFs steep learning curve, the prototypical editor is based on Graphiti [55]. Graphiti offers a feature-based API to implement graphical modeling editors. Feature-based means that every action supported by an editor implemented upon Graphiti is represented by a feature class. Such a feature class is instantiated through a `FeatureProvider`. The `FeatureProvider` in turn is instantiated through a `DiagramTypeProvider` that is registered with a Graphiti specific extension point inside Eclipse. Examples can be found

in appendix D. For each editor type—that means for each view on the notation—there is a distinct feature provider. Distinct features need to be created to support different actions. An `AddFeature` is implemented to provide the addition of an existing meta-model element to a diagram. The `AddFeature` is thus responsible for the generation of the graphical representation. To do this, it utilizes `PictogramElements` and `Shapes` provided by Graphiti and its various factories. Examples can again be found in appendix D. To actually enable the creation of new elements, a `CreateFeature` is needed. The `CreateFeature` adds an entry to the editors palette as shown in figure 5.1.

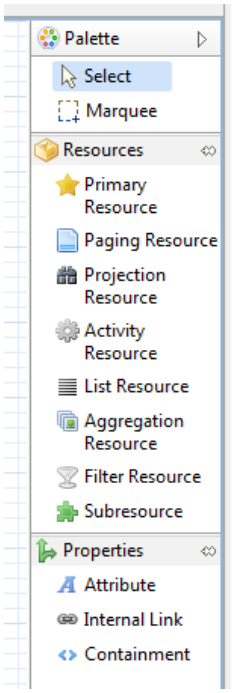



Figure 5.1: The editor palette

Additional features need to be implemented to provide additional behaviour. For example, to properly layout the graphical representations of the individual elements when the element is resized, a `LayoutFeature` needs to be implemented. The `LayoutFeature` is then called when layout of the element is done and enables the repositioning and modification of the individual parts of the representation. The `UpdateFeature` is responsible for updating the representation of the meta-model element if changes are made. Without an update feature, changing properties, for example, the name of a model element would not be reflected in the representation. The model element would have to be added to the diagram again to be recreated by the `AddFeature`. Through the `UpdateFeature` this can happen in place since it is executed when the model changes. There are additional features that can be implemented. For example, the `CopyFeature` and the `PasteFeature` together

provide copy and paste operations of graphical element. The `DirectEditFeature` is used to provide direct editing of textual elements, for instance the resource type name, as shown in figure 5.2. Again, examples of the various features and their implementations can be found in appendix D.

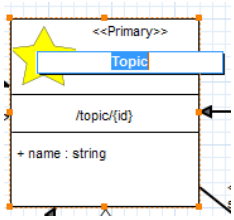



Figure 5.2: Direct editing feature

To support editing properties of graphical representations of certain domain objects, there is a need for properties views. A properties view shows the properties such as the name, attributes and identifiers of, for example, a resource type once the representation of the resource type is selected in the graphical editor.


Figure 5.3: A rich properties view

However, manually crafting properties views is a tedious and error-prone task. Therefore, for the development of the prototypical editor, the Enhanced Editing Framework (EEF) [58] is used to generate property views. EEF uses a model-driven approach. Using the EMF *genmodel*, it takes two additional models—the *components* model and the *eefgen* model—to generate rich property sheets from the information available in the models. The *components* model describes which parts of the model are mapped to which elements of the properties view while the *eefgen* model describes general generation parameters. Figure 5.3 shows an example of such a properties view for the *Book* resource type.

Both Graphiti and EEF are still very young projects and are thus still in a phase were new features and bug fixes are added in very short intervals. This sometimes leads to incompatibilities with regard to their APIs (because of changes made by the developers) or with the generated code (in case of EEF).

## 5.2 Supporting Infrastructure

A problem with building Eclipse based products—such as the Visual REST editor—is that the Eclipse Integrated Development Environment (IDE) uses a manual export mechanism to do the assembly of such a product. This poses a problem on different levels. First, this approach does not easily support building for multiple target platforms (for example Windows, Linux and MacOS X). Secondly, it is cumbersome to manually repeat the process over and over again when the code changes. This leads to development mainly happening in a non-productized version of the final environment. Only when a release is eminent, the product would be built and errors would show up—the process of building products is different from that for testing the plugins that form the final product. These shortcomings demand for a more automated and structured approach. For a long time Eclipse Buckminster [92] was the primary tool for automated building and deployment. However, Buckminster itself is quite hard to setup and work with. A rather new alternative approach is the use of Tycho [63], which is a plugin for Maven [62]. Tycho follows a manifest first approach—as opposed to the classical Maven POM first approach. This means that it simply reuses the information that is already there anyway inside the plugins—namely the manifest—to derive the build information. Additionally, Tycho is test-aware. It uses the maven surefire plugin [93] as its primary test driver for executing JUnit tests [59]. Additionally, Tycho easily supports building products for many different target platforms.

Testing an Eclipse product also needs considerable effort. The Visual REST editor is tested on different levels. Junit [59], Hamcrest [94], jMockit [95], SWTBot [60] and Google WindowTester Pro [61] are used to support testing of core and User Interface (UI) plugins. SWTBot [60] was chosen here because it is one of the standard testing tools used by many Eclipse projects; Graphiti itself is one of them. SWTBot allows implementing interactions with the Eclipse editor in the form of JUnit tests. WindowTester Pro [61] is used to enable record-and-playback development of test cases. This is especially useful when bugs are reported and regression tests need to be created. JUnit and Hamcrest provide the supporting infrastructure for both SWTBot and WindowTester Pro. JUnit and Hamcrest are also used together with jMockit to provide unit level testing for the core components. More details on testing can be found in section 5.4.

Even with all those tools defined to form a solid infrastructure for building and testing,

there is still one problem left: To have any value, tests need to be executed regularly. Although test execution happens automatically when running a Maven/Tycho build, this only moves the problem to that level: The Tycho build needs to be executed regularly—which is not the case during everyday development since it takes quite some time and is only really useful at deployment time when the products need to be built. This leads to another problem still: If the build is not executed regularly there is no guarantee it will actually work the next time it is executed. Continuous Integration [64] mitigates this problem by executing the build every time new code is committed to source control. In the context of the development of the Visual REST editor, the Jenkins [65] continuous integration server is used to provide the necessary infrastructure to enable continuous integration. Every time source code is changed and committed to source control, the server will pick up the changes and run a full Maven/Tycho build in the process. The results of the tests are evaluated and in case of failures a notification is sent via email (see figure 5.4).

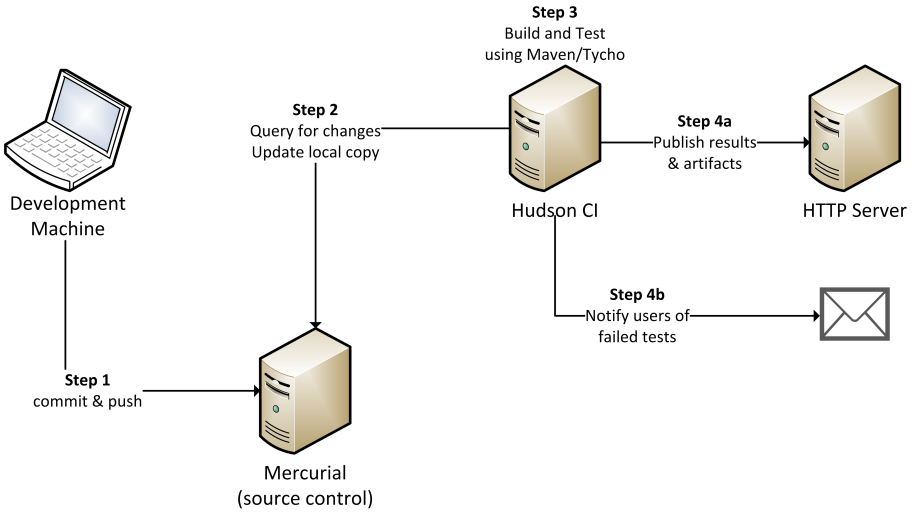



Figure 5.4: Steps in a continuous integration build

## 5.3 Development Process

Using Test-Driven Development (TDD) [96] is not easily feasible when developing visual editors using Eclipse. The main problem is that using SWTBot requires a lot of effort in the development of new tests. It is not easily possible to just specify wanted interaction but they need to be programmed in source code—without an actual editor available this is very hard. The same holds for WindowTester Pro. Jubula is an alternative that offers functional specification of test cases rather than programming them. However, due to

the nature of this editor (being a prototype) and the fact that both Graphiti and Jubula were new tools and it was unclear if they would even provide the necessary capabilities to implement the editor, Jubula was not used to specify functional tests yet.

The process used to implement the editor followed one of the basic ideas of Scrum [97]: The first step was to try out some functionality in an isolated environment (the exploration phase) and once it was deemed feasible and enough knowledge about a specific feature was built up it was implemented in the actual editor code. With more and more insight into the structure and inner workings of Graphiti and the other involved Eclipse frameworks and APIs, the structure of the plugins was refined gradually in subsequent iterations. The availability of an automated build and test system based on Maven and Tycho proved to be a huge asset in these refinement steps. The architecture and decomposition of the plugins evolved from one big monolithic plugin to a set of twelve individual plugins that form the Visual REST "feature". The details of the plugin decomposition, their dependencies and use is given in section 5.5.

## 5.4 Testing the Graphical Editor

There are three forms of tests performed: module tests for the core components as well as functional tests and regression tests for the ui components. The module tests are implemented using JUnit [59], Hamcrest [94] and jMockit [95]. JUnit provides the overall infrastructure and Hamcrest is used to provide "fluent" assertions in the test cases. JMockit is used to actually make sure module tests and not module integration tests are performed on the model classes.

For functional testing of UI components, JUnit is again used as the basic framework together with SWTBot [60]. SWTBot instruments the Standard Widget Toolkit (SWT) to provide programmable interactions with Eclipse user interface. It also uses matchers provided by Hamcrest for assertions.

For regression testing of UI components, Google WindowTester Pro [61] is used. WindowTester Pro also works in concert with JUnit and thus integrates nicely with existing tooling around JUnit itself (such as the Eclipse integration and the maven surefire plugin). WindowTester is used for regression testing since it allows recording tests instead of having to program them manually. It allows starting any application in recording mode, record some interaction with the UI and then generate JUnit code from the recording. Although recording is not flawless, it makes creating tests much easier. The problem, however, is that tests can only be recorded if there is an application to record them for, so WindowTester Pros recording feature can only be used once an editor has been implemented and is ready to be used. It is thus not suitable for test-driven development of

graphical environments.

An alternative for functional testing is Jubula [98] which has recently been made an Eclipse project. Instead of writing or recording JUnit code, Jubula uses a different approach. In Jubula interactions with the UI are provided through predefined building blocks that can be put together into functional test cases. The benefit is that this still allows a test (or feature) driven approach—because the test cases can be created with a readily available editor—while being easier to handle (for common cases) than SWTBot or WindowTester Pro. Since no JUnit code is generated or written, however, Jubula does not integrate so nicely with existing tools like Maven and Tycho, although it is possible to run Jubula tests from Maven. For future versions of the Visual REST editor Jubula is a strong candidate to be used for any functional testing and be a replacement for both SWTBot and WindowTester Pro.

## 5.5 Architecture of the Visual REST Plugin

The Visual REST editor is divided into individual plugins that form the Visual REST features. The plugins of the Visual REST editor are grouped by functionality but there are still some dependencies among them. However, the plugins were designed such that there are no circular dependencies in the dependency graph of the application. The grouping of plugins into features and the dependencies between plugins and features are depicted by figure 5.5.

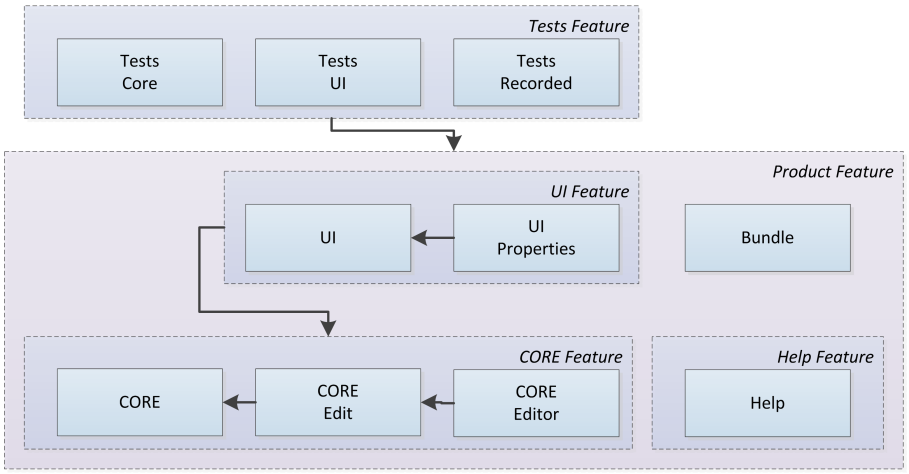



Figure 5.5: Structure of the plugins and features

The core feature includes all core plugins and is the basis for UI plugins and the UI feature

respectively. The UI properties plugins depends on the UI plugin and the core plugin to provide support for editing the model. The test plugin instruments and uses all other plugins and thus is on almost the same level a client or user would be.

Java source package names follow plugin names. For each of the plugin, the contained sources are located in sub-packages of the respective plugin name to make finding sources and debugging easier. For example, the plugin classes of the `de.van_porten.vrest.ui` plugin can be found in the Java package `de.van_porten.vrest.ui` and its sub-packages.

The Visual REST product feature **de.van_porten.vrest.feature** aggregates all features except the test feature into a single installable unit alongside the bundle plugin which is the products defining plugin. The following subsections detail the contents of the individual features and plugins.

### 5.5.1 Core Feature

The core feature **de.van_porten.vrest.core.feature** contains the plugins that provide the core functionality to be able to edit a meta-model instance at all. They do not provide any graphical editing but just textual manipulation and persistence of the model instances.

#### de.van_porten.vrest.core

The **de.van_porten.vrest.core** plugin contains the meta-model itself alongside the *gen-model*. It also contains the generated classes of the REST meta-model to be used by other plugins. The core plugin is the basis for the core.edit and core.editor plugin which both rely on it to provide the infrastructure needed by them. It uses only one emf specific extension point (`org.eclipse.emf.ecore.generated_package`).

#### de.van_porten.vrest.core.edit

The **de.van_porten.vrest.core.edit** plugin is the model editing component generated by EMF. Like the core plugin, it is also generated from the meta-model through the *gen-model*. The core.edit plugin provides the Provider and Factory classes for the meta-model classes. The extension point used is `org.eclipse.emf.edit.itemProviderAdapter-Factories`.

#### de.van_porten.vrest.core.editor

The **de.van_porten.vrest.core.editor** plugin is the textual editor component generated by EMF. The classes of this plugin are also generated from the *genmodel*. It uses the extension points `org.eclipse.ui.newWizards` and `org.eclipse.ui.editors` to register the editors and wizards generated by EMF with the Eclipse environment. The generated editor only supports basic editing operations on a textual basis. Figure 5.6 provides an example of the editor that is provided by this package.

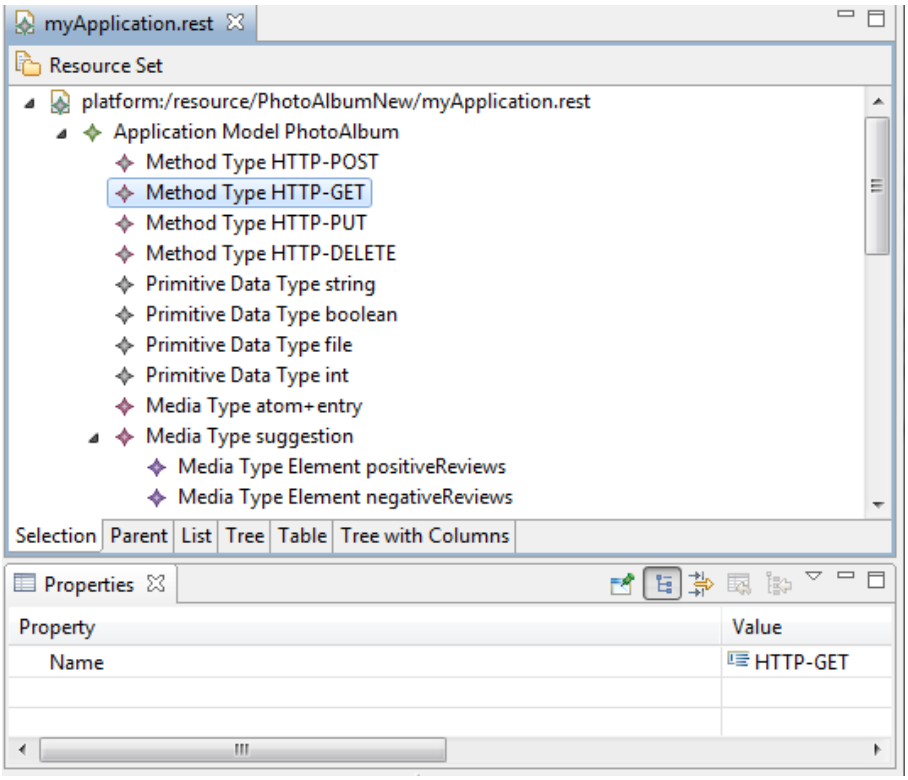



Figure 5.6: The editor provided by the core.editor plugin

### 5.5.2 UI Feature

The UI feature **de.van_porten.vrest.ui.feature** provides the graphical editors that actually implement the Visual REST language and are used to manipulate the application model in a graphical way. The UI feature consists of two plugins, the basic UI plugin and the UI properties plugin.

#### de.van_porten.vrest.ui

The **de.van_porten.vrest.ui** plugin provides the actual graphical editors based on Graphiti. There are three distinct diagram types registered with the `org.eclipse.graphiti.ui.diagramTypes` and `org.eclipse.graphiti.ui.diagramTypeProviders` extension points provided by Graphiti. These diagram types constitute the three distinct editors for the three views on the model as defined by the Visual REST language specification.

The **de.van_porten.vrest.ui** does only contain handcrafted code. The editor implementations and utility classes provided by this plugin are grouped into different Java packages. Figure 5.7 provides an overview.

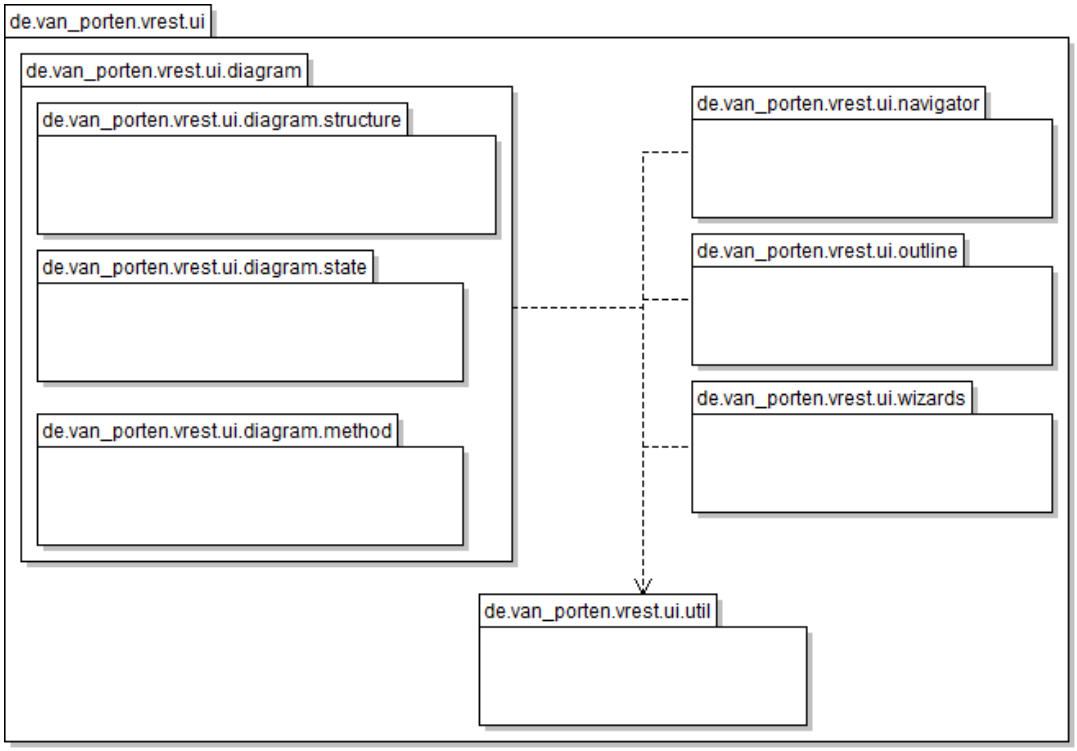



Figure 5.7: Overview of the packages in the de.van_porten.vrest.ui plugin

The `de.van_porten.vrest.ui` base package contains the project nature that is used in extending the `org.eclipse.core.resources.natures` extension point provided by Eclipse. It also contains the perspective factory used in the definition of the Visual REST modeling perspective. The Visual REST modeling perspective is tailored towards modeling Visual REST applications and provides an initial collection and layout of Eclipse views. The perspective definition makes use of the `org.eclipse.ui.perspectives` extension

point. The actual definition of the views in that perspective and their relative positions to each other happens through the `org.eclipse.ui.perspectiveExtensions` extension point.

The Visual REST perspective is used as preferred perspective for the `NewRestProject-Wizard` found in the `de.van_porten.vrest.ui.wizards` package. The new project wizard, extending the `org.eclipse.ui.newWizards` extension point, creates new projects, assigns the Visual REST project nature and switches the perspective if necessary. The Visual REST project can be created through the context menu as shown in figure 5.8. All wizards are also available through any other mechanism Eclipse offers to start individual wizards.

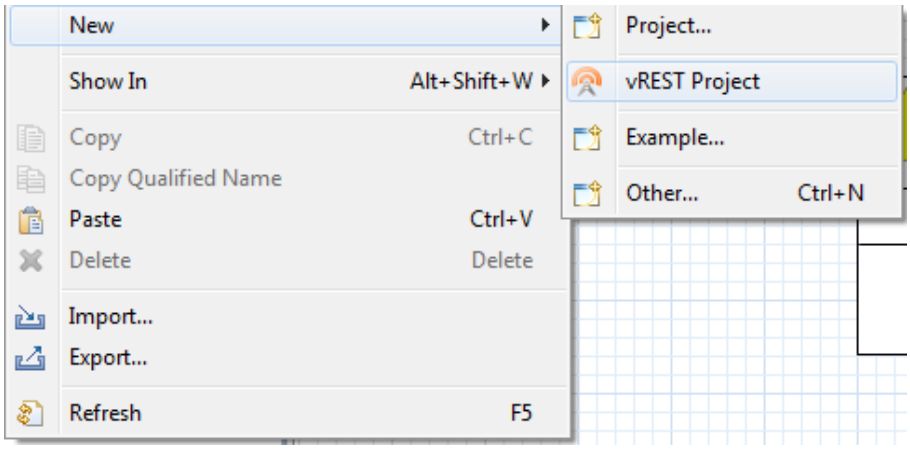



Figure 5.8: New Visual REST project

The `NewRestStructureDiagramWizard` is also found in the `de.van_porten.vrest.ui.-wizards` package and also uses the `org.eclipse.ui.newWizards` extension point. It is used to create new structure diagrams as shown in figure 5.9.

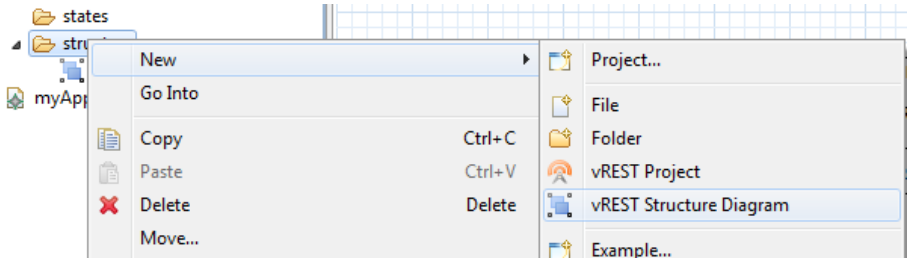



Figure 5.9: New structure diagram

The `de.van_porten.vrest.ui.diagram` package contains the image provider class `Rest-`

`ImageProvider` that is used to create the icons used in the editor palette. The image provider is shared between all editors. The `RestModelLabelFactory` is also part of that package and is responsible for creating labels for the different model elements (like resource types and states, for example). The labels are used in the display of the outline and the tree view provided as part of this plugin.

There is one top level package for each editor. The structure editor is located in the `de.van_porten.vrest.ui.diagram.structure` package. On its top level, that package contains the specific diagram type provider for the structure editor. In its sub-packages, the individual graphical elements can be found. For example, the `de.van_porten.vrest.ui.diagram.structure.resourcetype` package holds the implementation of the tool to support the creation and display of resource types in general. In this special case, there are additional sub-packages for the individual concrete resource types that implement type specific functionality. For instance, the folding (i.e. switching between detailed and condensed representation of the resource type) is implemented generically for the abstract resource type since it works the same way for all concrete resource types. There is also an abstract base class for adding the graphical representation of a resource. This abstract class utilizes open recursion and by that enables very easy implementation of concrete resource types by overriding some methods of the abstract resource type implementation.

The resource states editor is located in the `de.van_porten.vrest.ui.diagram.states` package and and the resource methods editor is located in the `de.van_porten.vrest.ui.diagram.methods` package. Both their sub-package structures follow that of the structure editor. There is a package for each individual graphical element that contains all features necessary to provide the required functionality. As with resource types, there is also a generic implementation of the features for the abstract `Action` and a concrete implementation in a suitable sub-package for concrete actions.

The `de.van_porten.vrest.ui.outline` package provides the diagram outline as shown in figure 5.10. It implements an adapter factory and registers it with the `org.eclipse.core.runtime.adapters` extension point to provide the outline view. Note that the symbols used in the outline have been taken from the *famfam silk icons* [99] and do not match the iconic representations used in the notation yet.

The `de.van_porten.vrest.ui.navigator` package contains the classes needed to provide the application tree. The application tree shows the complete model instance as an expandable tree in the project explorer. Figure 5.11 shows an example of such a tree. To provide this tree, the extension point `org.eclipse.ui.navigator.navigatorContent` is used.

The `de.van_porten.vrest.ui.util` package contains utility classes. These include classes to provide help in consistently laying out visual elements (`LayoutUtil`), a class to set and check properties of visual elements that are used to identify them (`PropertyUtil`) and

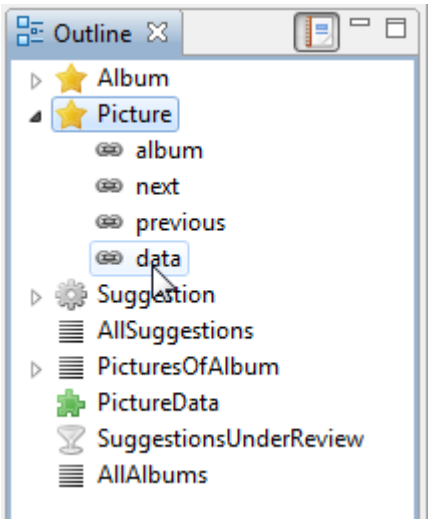



Figure 5.10: The outline view

classes used to provide persistence for the model instances created through the graphical editor (`FileService` and `ResourceUtil`).

#### de.van_porten.vrest.ui.properties

The **de.van_porten.vrest.ui.properties** plugin contains the code generated by EEF. This plugin provides rich property editing of the elements added to the diagrams using the editors provided through the ui plugin. As the name suggests, the ui.properties plugin is a sub-component of the ui plugin and depends on it. It uses the `org.eclipse.ui.views.properties.tabbed.*` and `org.eclipse.emf.edit.itemProviderAdapterFactories` extension points.

### 5.5.3 Help Feature

The help feature **de.van_porten.vrest.help.feature** contains the help system. The help system is provided by the **de.van_porten.vrest.help** plugin. It uses the `org.eclipse.help.toc` extension point to make a contribution to the Eclipse help system as shown in figure 5.12.

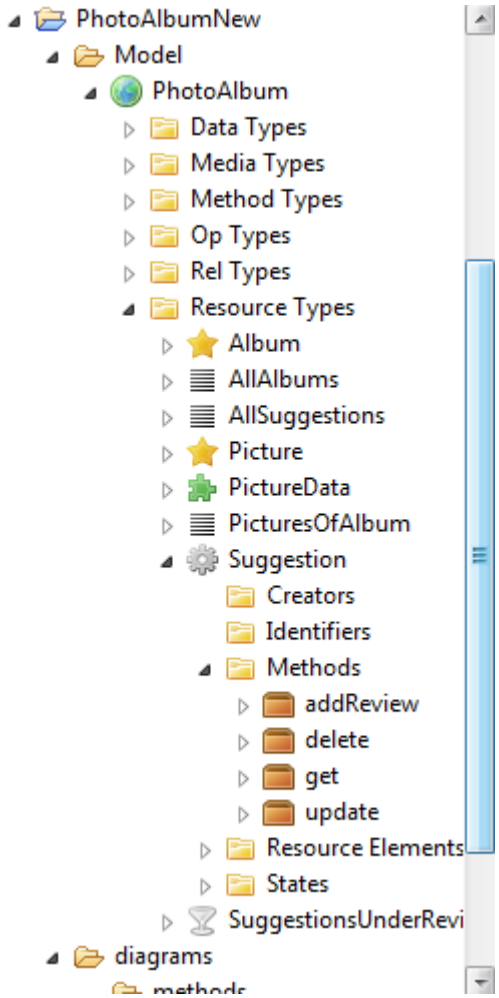



Figure 5.11: The application tree

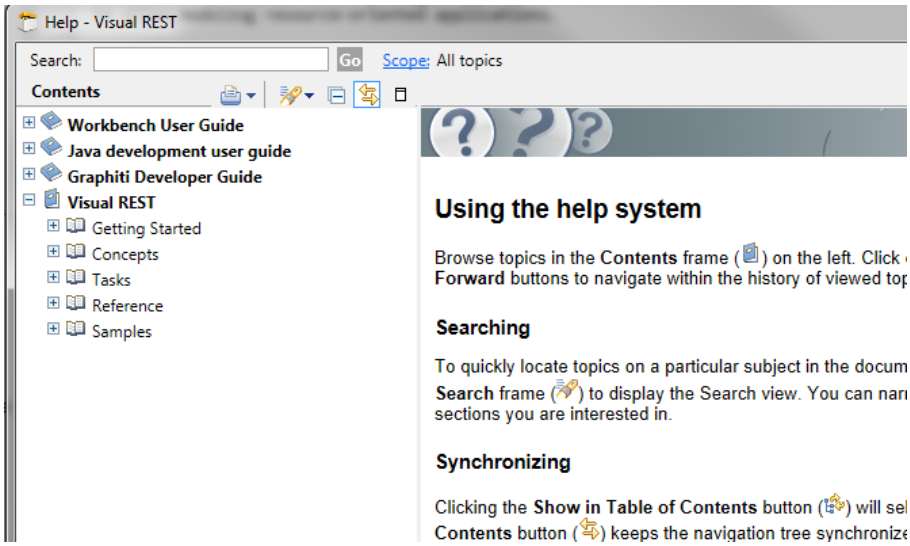



Figure 5.12: The help system of the Visual REST editor

### 5.5.4 Product Feature

The product feature is located inside the package **de.van_porten.vrest.feature**. It aggregates Core, UI and Help feature into one single installable unit. Besides the sub-features it contains just one additional plugin. The **de.van_porten.vrest.bundle** plugin provides an introduction view as well as customization of the RCP application. The customization applied is the modification of the splash screen shown while launching the editor as well as the provisioning of the welcome screen that provides a general introduction to the product.

### 5.5.5 Test Feature

Testing is an integral part of software development. Also in the development of a graphical editor testing plays an important role. There are however different areas that need testing and there are also different techniques that can or must be used. To provide some sort of structure for testing, there are three distinct test plugins. A test plugin is simply the means by which an Eclipse based application can be tested. Test-plugins are not added to the final product. The test plugins are aggregated into the **de.van_porten.vrest.-tests.feature** feature. The tests are automatically executed as part of the Maven/Tycho build. All tests are based on the JUnit framework.

#### de.van_porten.vrest.tests.core

The **de.van_porten.vrest.tests.core** plugin tests the core plugins. It is mainly used to validate that the code generated by EMF behaves as expected. It utilizes jmockit to provide unit level testing. Hamcrest is used to provide meaningful assertion based on the fluent API provided by it.

#### de.van_porten.vrest.tests.ui

The **de.van_porten.vrest.tests.ui** plugin uses SWTBot to run hand-crafted tests. SWTBot instruments SWT for those purposes and allows using the IDE in pretty much the same manner a human user would. As opposed to recorded tests, SWTBot tests can be written before any functionality is actually implemented and thus allow test-driven development to some extent—with the limitations mentioned earlier.

#### de.van_porten.vrest.tests.recorded

WindowTester Pro is used in the **de.van_porten.vrest.tests.recorded** plugin. Instead of hand-crafting the test-cases, however, a WindowTester Pro feature that enables recording of test-cases and subsequent generation of the JUnit code from those recordings is utilized. This provides for an easy way to create regression tests if errors occur.

### 5.5.6 Additional Packages

Besides features and plugins presented before, there are also additional packages that are part of the project.

The **de.van_porten.vrest.build** package is not a plugin in the usual sense. It is basically a package that contains the parent pom of all other plugins and is used as the basis for the Maven-based build.

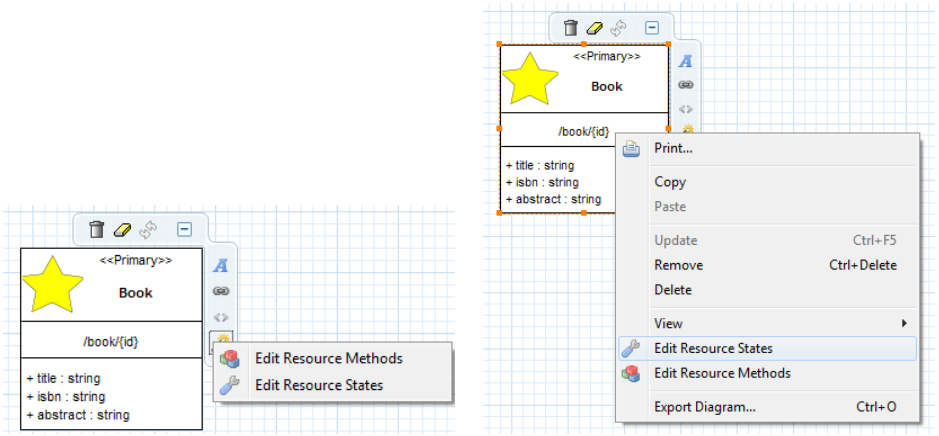
The **de.van_porten.vrest.p2-repository** package is also a Maven and Tycho specific package. It contains the product definition used to build the concrete packages of Visual REST for different target platforms. A Tycho action builds zipped releases for all supported target platforms using that product definition. The product definition is also used for development and testing purposes in the launch configuration inside the IDE.

The p2-repository also constitutes the update site. Besides building the zipped products Tycho also creates an update site that can be used from any eclipse installation via the p2 update mechanism.

## 5.6 Beyond Visual Rest

The Visual REST editor presented in this chapter goes beyond what the mere description of a visual notation could provide. The editor provides interaction support to effectively use mechanisms of the notation.

Navigating through an application model is supported via the circle menu and the context menu of resource types. Through both menus it is possible to open a resource types states and method diagram as shown in figure 5.13. The circle menu appears when the mouse hovers over a resource type; the context menu appears when the users performs a right-click on the resource type.




(a) circle menu (b) context menu

Figure 5.13: Contextual menus of resource types

The editor also provides an outline as well as a miniature view for all of the different views which make navigating complex diagrams easier. Figure 5.14 shows the outline view of a bookshop application. As can be seen, the outline shows not only the resource types in the current diagram but also the internal links as properties of the resource types. Figure 5.15 shows the miniature view miniature view of a structure overview diagram. It can be used to easily navigate on larger diagrams and provides some location awareness.

Adding foreign elements to diagrams provides contextualisation. To make adding these

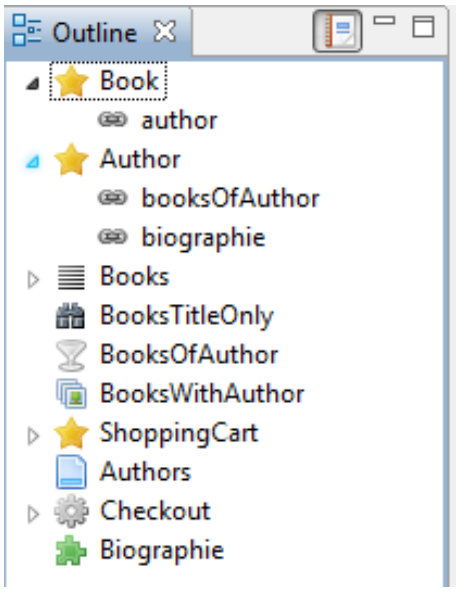



Figure 5.14: Outline of a bookshop application

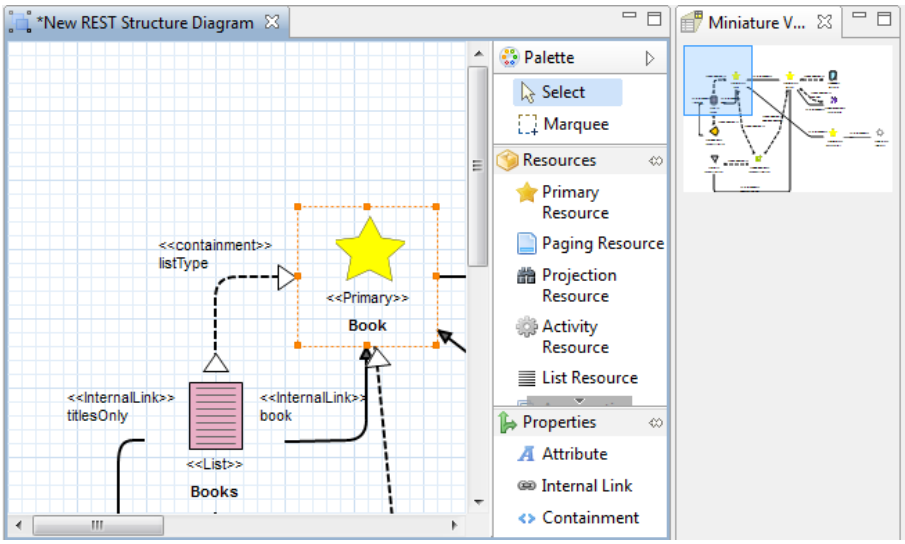



Figure 5.15: Miniature view of a bookshop application

foreign elements to diagrams easier, all elements from the the application model can be dragged and dropped from the application tree onto a diagram as shown in figure 5.16. However, only elements from the same application and of the correct type can be added to a diagram (e.g. a method cannot be added to a structure diagram).

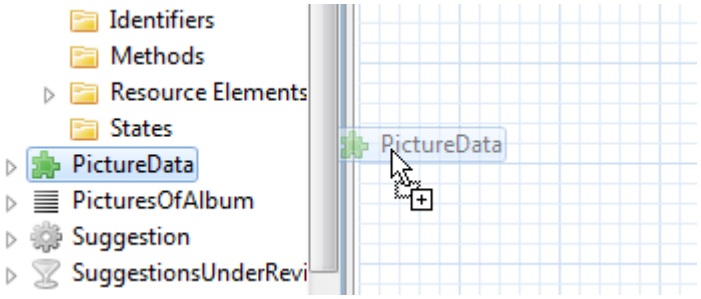



Figure 5.16: Drag and Drop operation from the application tree

To make cognitive integration even more comfortable when context is to be displayed, it is possible to collapse individual resource types into their less-detailed representations at the push of a button in the circle menu of the resource types as shown in figure 5.17. This allows adding foreign resource types to diagrams in a less intrusive way which makes reading the structure diagrams easier still.

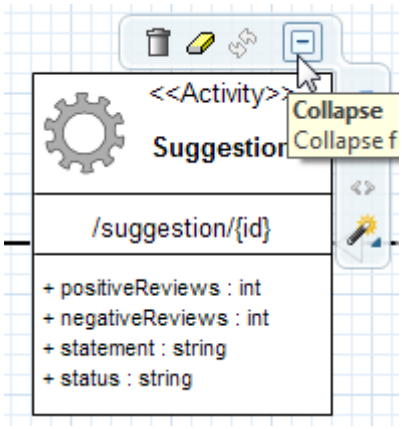



Figure 5.17: The collapse button

# 6 Final Remarks

This thesis has presented a new and unique visual language for modeling resource-oriented applications called Visual REST. The semantic basis for Visual REST is the meta-model for resource-oriented applications [6]. The language was designed to meet the criteria defined by Daniel Moody in his article "The physics of notations" [8]. Following Moody's guideline allowed building a notation that should be easier to use for modeling resource-oriented applications than, for example, UML, because it is tailored towards visual effectiveness. Visual REST offers ways to manage application complexity by providing partitioning and hierarchical organisation of the application. It uses perceptually discriminable representations for the concepts found in resource-oriented applications. The chosen representations are also semantically transparent. They carry a meaning and hint users to the actual semantics of the objects. Furthermore, there is no symbol excess, symbol overload or symbol redundancy that would break with semiotic clarity and dual coding is used to provide textual annotations to symbolic representations. Dual coding makes it easier—especially for novice users—to understand diagrams. Cognitive fit is achieved on the one hand by designing the notational elements such that they can still be used on a whiteboard as well as on a computer screen and on the other hand by allowing professional users of the notation to omit unneeded elements such as textual annotations and color. Color is an essential element in providing visually expressive representations and eases the use of the notation and the discrimination between different symbols. In addition, cognitive integration is provided through sign posting, contextualisation and overview diagrams to help users of the notation put all diagrams into context. Lastly, graphic economy is achieved through the explicit introduction of symbol deficit—there is no graphical representation for every given part of the meta-model but only for a carefully selected set.

During the development of Visual REST a process was defined that was used as the basis for iteratively creating and improving the notation (see section 3.2). This process showed how a notation can be built on top of an existing meta-model in a constructive and structured way, following the visual guidelines provided by Daniel Moody's work. It provides direction in the task of model language creation.

Visual REST was also evaluated with a focus on the properties defined by Moody. Although the notation was designed with all the different properties that are desirable in mind, the results of the evaluation still show room for improvement on various levels. For

one, the resource method behaviour diagram seems ineffective and might need replacement by a more appropriate representation. Furthermore, some of the iconic representations need to be replaced by better ones, some seemed to be hard to discriminate from others and some were not semantically transparent. The use of partitioning as a means of complexity management and the use of textual indicators for signposting seems to work well for most probands. Finally, most probands thought that Visual REST will be usable, given that a dedicated editor is available to do the modeling.

The Visual REST editor implements the new notation into an Eclipse application. The editor allows users to build new resource-oriented applications with a comfortable tool. The application model generated through the editor is the input for a number of possible code generators, some of which are in development [100] or have already been developed [101]. There is no limit to the target platform in general so any number of additional generators are possible. Ideally, they would be integrated into the Visual REST graphical editing environment to provide a complete tool set to build resource-oriented applications. The graphical editor itself is based on EMF, Graphiti and EEF which provide the foundation for modeling in general and graphical editors in particular.

Visual REST is a first step in the development of a well-defined and visually effective graphical language for the design of resource-oriented applications. It allows the definition of applications in a straight forward fashion while still providing the means to handle also larger application models well. As is the nature of first steps, there is still much room for improvement in the future, though. To give some examples, it might make sense to leave out the URLs from the model and create a separate mapping from resource name to URL. It might also make sense to change the representation of states to make them more discriminable and semantically clearer. It would also be interesting to see if a way can be found to foster reuse of elements. For the future a changed meta-model might call for a different structuring of the graphical language as well. It might also lead to additional graphical symbols. Regarding behaviour modeling, it is questionable if using a graphical notation is the right way to go or if a textual language would be better suited for the task; or maybe even a hybrid between textual and graphical language would be a good solution to the problem.

Alternative representations for the resource types could be found that are perceptually more effective. For example, the boxes around the resource types could be completely omitted. Figure 6.1 gives an example of an alternative representation of resource types without the introduction of “boxitis”. Instead of representing the URLs inside the notation they could be omitted and a map from resource type name to a list of possible URIs could be created.

Additions to the meta-model could be made. For example, it would help to be able to demarcate the entry point(s) into an application—for that there needs to be support in the meta-model, though. It would also be worthwhile to think about additional packaging


Figure 6.1: Possible alternative representations of an activity resource type

mechanisms inside the meta-model which could then be transferred into the notation. This could improve complexity management.

The process presented to derive the notation from the meta-model is still far from perfect, but also is a first step in the right direction. It is still too much tailored towards the development of Visual REST and not generic enough to be used for the development of other notations. What is missing is mainly a generic description of the steps that are necessary to build a notation on the principles of visual effectiveness. Using only those principles—despite them being a prescriptive theory—is still a destructive process. Creativity is needed to come up with new symbols and to actually build a notation around a meta-model. What would be desirable is a more constructive process. A way to build a notation "bottom-up" needs to be found—and the outcome should be a language that is visually effective.

The survey that was conducted as part of this thesis showed how well (or badly at some points) the notation follows Moody's criteria for cognitive effective visual languages. The feedback provided through the use of a questionnaire is invaluable to any visual language designer and should be included in the process of visual language development. In the limited time given for the development, implementation and evaluation of Visual REST it was not possible to let the information gathered through the survey flow back into the notations design. When thinking about processes, any feedback mechanism that is available should be part of the process itself and should be used to guide the development of the new language towards the needs of the domain users. What would also be desirable is a blue print of a questionnaire that can be tailored towards new notations. The questionnaire developed in this work is a first step. Deriving generic questions that can then again be adapted to a different notation would be a next.

The Visual REST editor is still a prototype and needs improvement. It will be made an open-source project so that everybody who is interested can participate in the creation of a development environment for resource-oriented applications. Ideally, this will build an infrastructure with different kinds of code generators and possibly additional textual languages (for example for behaviour modeling) around the meta-model for resource-oriented applications. First and foremost, the integration between the different modeling components needs to be improved, though. Although section 5.6 showed that there is

already much integration done to support the development using Visual REST, there are still options that can be exploited. To give just one example, navigation between different diagrams could be made easier through the use of additional context menu items.

The implementation of the Visual REST editing environment also needs additional work beyond adding features. Additional tests—also functional ones, possibly using Jubula—need to be developed to make the editor more robust and to provide the basis for any number of refactoring tasks. It is also thinkable to re-implement the editor from scratch with the experience gathered through the prototypical implementation. For this task, some interesting projects have been emerging in the vicinity of the Eclipse Modeling Project. EuGENia [102] is part of the Eclipse Epsilon project [103]. EuGENia allows generating the models required by GMF [53] from an annotated meta-model. However, at this point only a single diagram per model is possible so this would not have worked for the project at hand without the need to create additional meta-models and model-to-model transformations. Spray [104] on the other hand is a domain-specific language tailored towards generation of code based on Graphiti [105–107]. In the future it might prove useful for creating graphical editors using the Graphiti framework without the need to hand-craft as much code as is required today. Using either EuGENia or Spray could provide a way to create a better version of the editor either based on Graphiti again or using the Graphical Modeling Framework.

# A Visual Language Questionnaire

## Visual REST

**Visual REST** is a graphical modeling language for the development of resource-oriented applications. The language development is based on nine criteria due to Daniel Moody that establish a scientific basis for the design of graphical languages.

The purpose of this questionnaire is to evaluate how well those nine criteria have been applied in the development of the Visual REST language. Its purpose is ***NOT*** to evaluate how good the criteria are at all.

The results of this questionnaire will be part of my final Master's thesis. However, the data is anonymized completely.

Welcome to the **Visual REST** graphical notation survey.

The questionnaire tries to be as simple as possible. Only closed questions are used to make your life a little easier. At the end there is one open optional question where you can place you comments on the notation, should you have any.

At the end there is the possibility to enter your email address to **win a copy of one of the following books**:

- Bruce A. Tate - Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages
- Clinton Wong - HTTP Pocket Reference
- and for those not into IT: Hermann Maurer - Xperten Bd.1: Der Telekinet

You will need about 20 minutes to finish the questionnaire.

There are 49 questions in this survey

### User Background

The following question will gather some basic information about your knowledge of fields related to the work at hand. This will give us the means to put your answers into context and maybe also to deduce some correlation between certain "knowledge" levels and the "understanding" of the Visual REST language.

**1 [QUB_0001]How would you rate your knowledge of REST? ***

Please choose **only one** of the following:

- ○ I did research on REST and related areas
- ○ I have developed applications using REST
- ○ I have heard about REST
- ○ I have never heard of REST

REST is defined by Roy Fielding's dissertation: Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. University of California, Irvine. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

**2 [QUB_0002pre]Are you familiar with graphical modeling languages such as UML 2, BPMN or state charts? ***

Please choose **only one** of the following:

- ○ Yes
- ○ No

**3 [QUB_0002]How would you rate your knowledge of following the UML 2 diagrams? ***

**Only answer this question if the following conditions are met:**
° Answer was Y'Yes' at question '2 [QUB_0002pre]' (Are you familiar with graphical modeling languages such as UML 2, BPMN or state charts?)

Please choose the appropriate response for each item:

| | Don't know | Know (but have not used) | Used |
|---|---|---|---|
| Activity Diagram | ○ | ○ | ○ |
| Class Diagram | ○ | ○ | ○ |
| State Machine Diagram | ○ | ○ | ○ |

**4 [QUB_0003]How would you rate your knowledge of graphical notations? ***

Please choose **only one** of the following:

- ○ I'm a researcher in the field
- ○ I'm a regular user of graphical editors
- ○ I have used a graphical editor before
- ○ I have never used a graphical editor

## Moody :: Perceptual Discriminability

Questions regarding "Perceptual Discriminability" criterion

**5 [QMPD_0000a]**

**Please compare the following symbols graphically. For each pair, state if you find them easy to distinguish or not.**

*

Please choose the appropriate response for each item:

| | Yes, they are easy to distinguish | No, they are not easy to distinguish | I'm not sure |
|---|---|---|---|
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |

**6 [QMPD_0000b]**

**Again, please compare the following symbols graphically. For each pair, state if you find them easy to distinguish or not.**

*

Please choose the appropriate response for each item:

| | Yes, they are easy to distinguish | No, they are not easy to distinguish | I'm not sure |
|---|---|---|---|
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |

**7 [QMPD_0000c]**

**Again, please compare the following symbols graphically. For each pair, state if you find them easy to distinguish or not.**

*

Please choose the appropriate response for each item:

| | Yes, they are easy to distinguish | No, they are not easy to distinguish | I'm not sure |
|---|---|---|---|
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |

**8 [QMPD_0000d]**

**Again, please compare the following symbols graphically. For each pair, state if you find them easy to distinguish or not.**

*

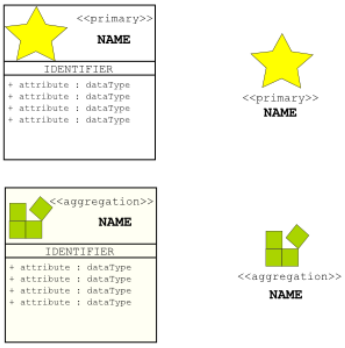
Please choose the appropriate response for each item:

| | Yes, they are easy to distinguish | No, they are not easy to distinguish | I'm not sure |
|---|---|---|---|
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |

| | | | |
|---|---|---|---|
| | ○ | ○ | ○ |
| | ○ | ○ | ○ |
| | ○ | ○ | ○ |
| | ○ | ○ | ○ |
| | ○ | ○ | ○ |

**9 [QMPD_0010]**

**When comparing the following representations of the primary resource type, does the added box around the left version have any effect on the discriminability. That is, is it easier, harder, or equally hard to discriminate the representation from, for example, the aggregation resource type as shown below.**



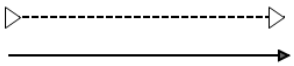

*

Please choose **only one** of the following:

- ○ The left ones are easier to discriminate
- ○ The right ones are easier to discriminate
- ○ The are equally hard / easy

**10 [QMPD_0009]Please compare the following symbols graphically. For each pair, state if you find them easy to distinguish or not. ***

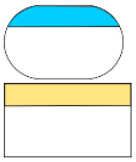
Please choose the appropriate response for each item:



| | Yes, they are easy to distinguish | No, they are not easy to distinguish | I'm not sure |
|---|---|---|---|
| | ○ | ○ | ○ |

**11 [QMPD_0011]Please compare the following symbols graphically. For each pair, state if you find them easy to distinguish or not. ***

Please choose the appropriate response for each item:



| | Yes, they are easy to distinguish | No, they are not easy to distinguish | I'm not sure |
|---|---|---|---|
| | ○ | ○ | ○ |

**12 [QMPD_0012a]**

**Please compare the following symbols graphically. For each pair, state if you find them easy to distinguish or not.**

*

Please choose the appropriate response for each item:

| | Yes, they are easy to distinguish | No, they are not easy to distinguish | I'm not sure |
|---|---|---|---|
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |

**13 [QMPD_0012b]**

**Again, please compare the following symbols graphically. For each pair, state if you find them easy to distinguish or not.**

*

Please choose the appropriate response for each item:

| | Yes, they are easy to distinguish | No, they are not easy to distinguish | I'm not sure |
|---|---|---|---|
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |

**14 [QMPD_0012c]**

**Again, please compare the following symbols graphically. For each pair, state if you find them easy to distinguish or not.**

*

Please choose the appropriate response for each item:

| | Yes, they are easy to distinguish | No, they are not easy to distinguish | I'm not sure |
|---|---|---|---|
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |
|  | ○ | ○ | ○ |

○ ○ ○

## Moody :: Semantic Transparency

Questions regarding "Semantic Transparency" criterion

**15 [QMST_0001a]Please choose the resource type you think is represented by the symbol displayed. ***

Please choose the appropriate response for each item:

| | Primary Resource Type | Paging Resource Type | Projection Resource Type | Activity Resource Type | List Resource Type | Aggregation Resource Type | Filter Resource Type | Subresource Type | I'm not sure |
|---|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**16 [QMST_0001b]Please choose the resource type you think is represented by the symbol displayed. ***

Please choose the appropriate response for each item:

| | Primary Resource Type | Paging Resource Type | Projection Resource Type | Activity Resource Type | List Resource Type | Aggregation Resource Type | Filter Resource Type | Subresource Type | I'm not sure |
|---|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**17 [QMST_0001c]Please choose the resource type you think is represented by the symbol displayed. ***

Please choose the appropriate response for each item:

| | Primary Resource Type | Paging Resource Type | Projection Resource Type | Activity Resource Type | List Resource Type | Aggregation Resource Type | Filter Resource Type | Subresource Type | I'm not sure |
|---|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**18 [QMST_0001d]Please choose the resource type you think is represented by the symbol displayed. ***

Please choose the appropriate response for each item:

| | Primary Resource Type | Paging Resource Type | Projection Resource Type | Activity Resource Type | List Resource Type | Aggregation Resource Type | Filter Resource Type | Subresource Type | I'm not sure |
|---|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**19 [QMST_0001e]Please choose the resource type you think is represented by the symbol displayed. ***

Please choose the appropriate response for each item:

| | Primary Resource Type | Paging Resource Type | Projection Resource Type | Activity Resource Type | List Resource Type | Aggregation Resource Type | Filter Resource Type | Subresource Type | I'm not sure |
|---|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**20 [QMST_0001f]Please choose the resource type you think is represented by the symbol displayed. ***

Please choose the appropriate response for each item:

| | Primary Resource Type | Paging Resource Type | Projection Resource Type | Activity Resource Type | List Resource Type | Aggregation Resource Type | Filter Resource Type | Subresource Type | I'm not sure |
|---|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**21 [QMST_0001g]Please choose the resource type you think is represented by the symbol displayed. ***

Please choose the appropriate response for each item:

| | Primary Resource Type | Paging Resource Type | Projection Resource Type | Activity Resource Type | List Resource Type | Aggregation Resource Type | Filter Resource Type | Subresource Type | I'm not sure |
|---|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

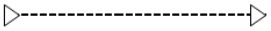
**22 [QMST_0001h]Please choose the resource type you think is represented by the symbol displayed. ***

Please choose the appropriate response for each item:

| | Primary Resource Type | Paging Resource Type | Projection Resource Type | Activity Resource Type | List Resource Type | Aggregation Resource Type | Filter Resource Type | Subresource Type | I'm not sure |
|---|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**23 [QMST_0002a]Please choose the type of link you think is represented by the arrows displayed. ***

Please choose the appropriate response for each item:



| | Internal Link | Containment (e.g. A list "contains" a list element type) | I'm not sure |
|---|---|---|---|
| | ○ | ○ | ○ |

**24 [QMST_0002b]Please choose the type of link you think is represented by the arrows displayed. ***

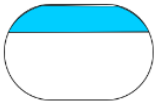
Please choose the appropriate response for each item:



| | Internal Link | Containment (e.g. A list "contains" a list element type) | I'm not sure |
|---|---|---|---|
| | ○ | ○ | ○ |

**25 [QMST_0003a]Please choose the type of state you think is represented by the symbol displayed. ***

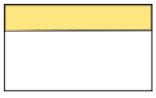
Please choose the appropriate response for each item:



| | Initial State | State | I'm not sure |
|---|---|---|---|
| | ○ | ○ | ○ |

**26 [QMST_0003b]Please choose the type of state you think is represented by the symbol displayed. ***

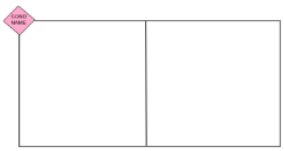
Please choose the appropriate response for each item:



| | Initial State | State | I'm not sure |
|---|---|---|---|
| | ○ | ○ | ○ |

**27 [QMST_0004a]Please try to identify which action type is represented by which symbol displayed below. ***

Please choose the appropriate response for each item:



| | Message Action | Return Action | Conditional Action | Update Action | Create Action | List Add Action | List Remove Action | I'm not sure |
|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**28 [QMST_0004b]Please try to identify which action type is represented by which symbol displayed below. ***

Please choose the appropriate response for each item:

| | Message Action | Return Action | Conditional Action | Update Action | Create Action | List Add Action | List Remove Action | I'm not sure |
|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**29 [QMST_0004c]Please try to identify which action type is represented by which symbol displayed below. ***

Please choose the appropriate response for each item:

| | Message Action | Return Action | Conditional Action | Update Action | Create Action | List Add Action | List Remove Action | I'm not sure |
|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**30 [QMST_0004d]Please try to identify which action type is represented by which symbol displayed below. ***

Please choose the appropriate response for each item:

| | Message Action | Return Action | Conditional Action | Update Action | Create Action | List Add Action | List Remove Action | I'm not sure |
|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**31 [QMST_0004e]Please try to identify which action type is represented by which symbol displayed below. ***

Please choose the appropriate response for each item:

| | Message Action | Return Action | Conditional Action | Update Action | Create Action | List Add Action | List Remove Action | I'm not sure |
|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**32 [QMST_0004f]Please try to identify which action type is represented by which symbol displayed below. ***

Please choose the appropriate response for each item:

| | Message Action | Return Action | Conditional Action | Update Action | Create Action | List Add Action | List Remove Action | I'm not sure |
|---|---|---|---|---|---|---|---|---|
| | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

**33 [QMST_0004g]Please try to identify which action type is represented by which symbol displayed below. ***

Please choose the appropriate response for each item:

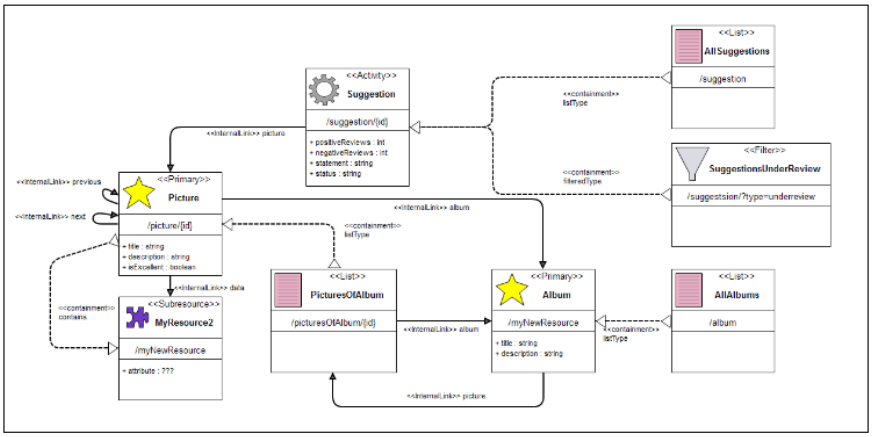
| | Message Action | Return Action | Conditional Action | Update Action | Create Action | List Add Action | List Remove Action | I'm not sure |
|---|---|---|---|---|---|---|---|---|
|  | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

## Moody :: Complexity Management

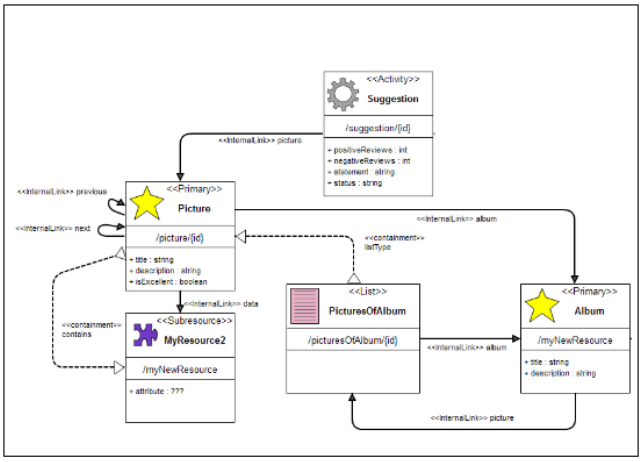
Questions regarding "Complexity Management" criterion

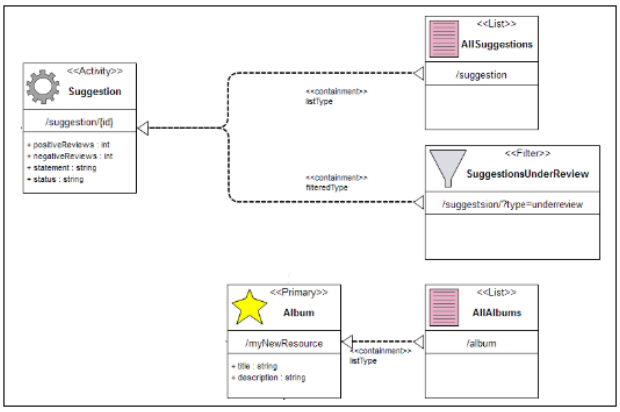
**34 [QMCM_0001]**

**The visual REST notation allows having multiple diagrams for modeling the structure. For example, consider the following diagram**




**You could model the same application using two distinct structure diagrams, like for instance so**






**Do you think having the ability to split the structural model as presented above will help in managing application complexity?**

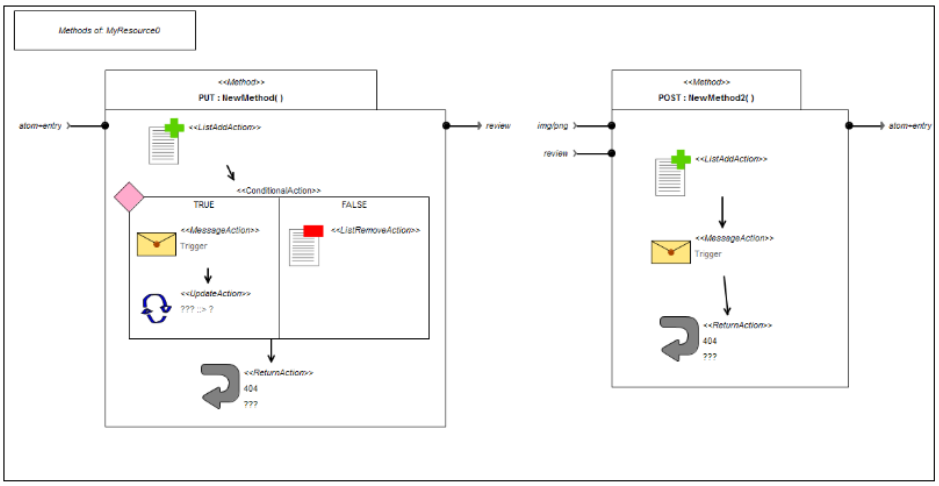
*

Please choose **only one** of the following:

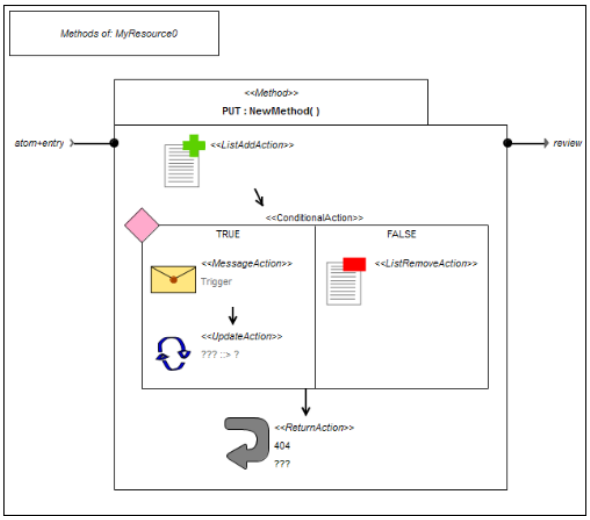
- ○ Yes
- ○ No

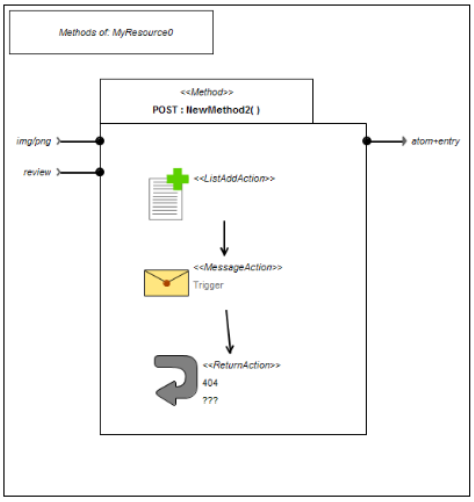
**35 [QMCM_0002]**

**Methods for a single resource type can be modeled in multiple diagrams, but all methods can also be contained in only one diagram. For example, you can have a single diagram like the following:**



**But you can also distribute the methods of a single resource type over multiple diagrams, like in the next example**




**Do you think the ability to distribute the methods of a single resource type over multiple diagrams helps in managing complexity?**

*

Please choose **only one** of the following:

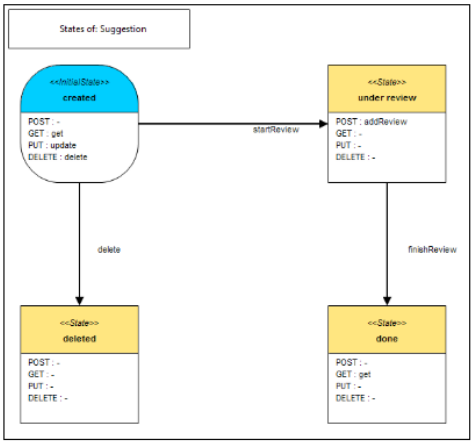
- ○ Yes
- ○ No

## Moody :: Cognitive Integration

Questions regarding "Cognitive Integration" criterion

**36 [QMCI_0002]**

**The following state diagram shows the states of the suggestion resource type.**




**Please look at the diagram and answer the following questions.**

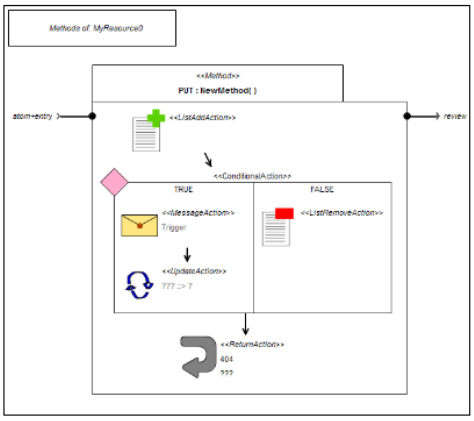
*

Please choose the appropriate response for each item:

| | Yes | No |
|---|---|---|
| The box at top helps to identify which resource type the states belong to | ○ | ○ |
| It is helpful that every state lists all possible method types instead of just those that are actually supported | ○ | ○ |

**37 [QMCI_0003]**

**Looking at the following method diagram, please answer the questions below**




*

Please choose the appropriate response for each item:

| | Yes | No |
|---|---|---|
| The box at the top helps in identifying which resource type the methods belong to | ○ | ○ |

## Moody :: Dual Coding

Questions regarding "Dual Coding" criterion

**38 [QMDC_0001]**

**Does adding a textual representation help you to grasp the meaning of the resource types?**

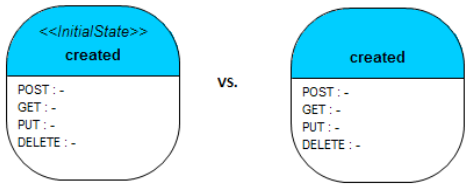

*

Please choose **only one** of the following:

- ○ Yes
- ○ No

**39 [QMDC_0001b]**

**What about the states. Does it help there?**



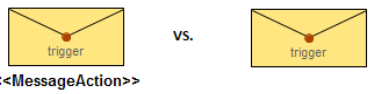

*

Please choose **only one** of the following:

- ○ Yes
- ○ No

**40 [QMDC_0001c]**

**What about the Method Behavior. Does it help you there?**




*

Please choose **only one** of the following:

- ○ Yes
- ○ No

**41 [QMDC_0002]Does adding the meaning of the iconic representations in form of UML-like stereotypes appeal to you? ***

Please choose **only one** of the following:

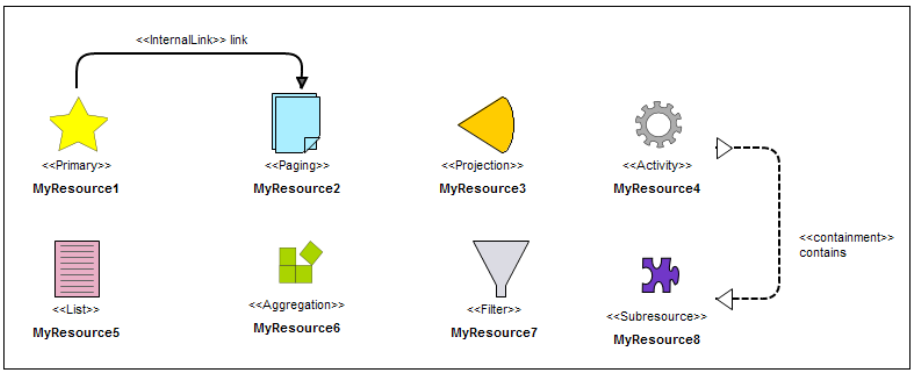
- ○ Yes
- ○ No
- ○ I don't know

**Moody :: Graphic Economy**

Questions regarding "Graphic Economy" criterion

**42 [QMGE_0001]**

**Looking at the following diagram, do you think there are too many different graphical elements?**



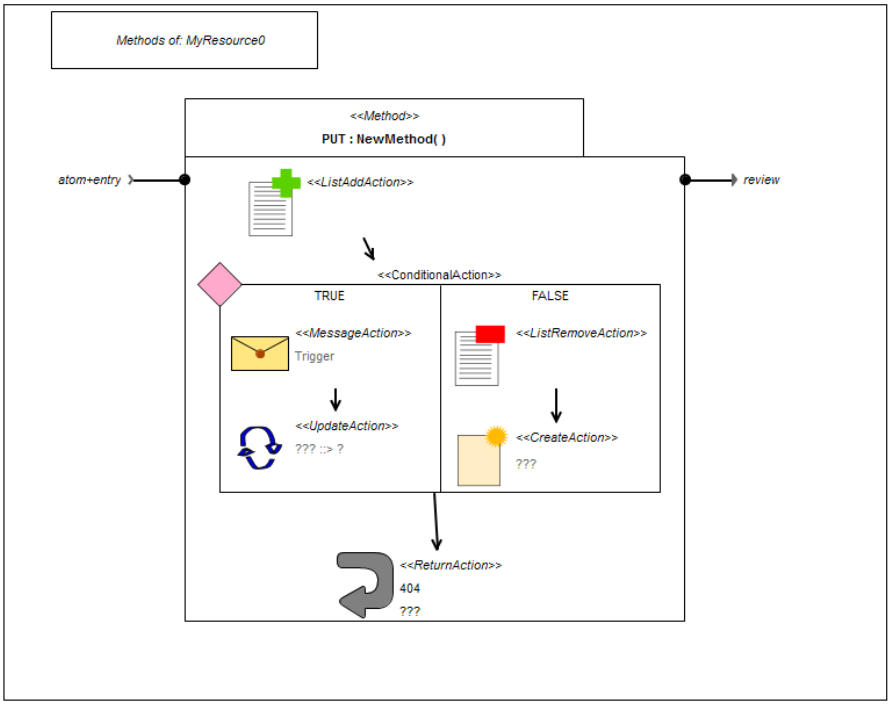

*

Please choose **only one** of the following:

- ○ Yes
- ○ No

**43 [QMGE_0002]**

**Looking at the following diagram, do you think there are too many different graphical elements?**



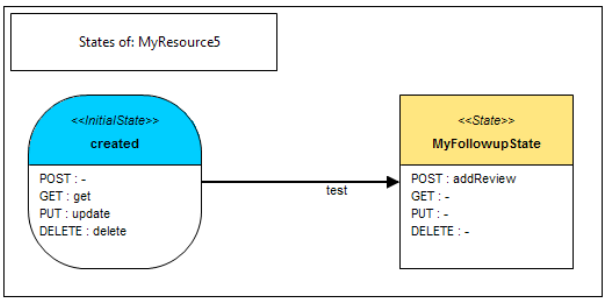

*

Please choose **only one** of the following:

- ○ Yes
- ○ No

**44 [QMGE_0003]**

**Looking at the following diagram, do you think there are too many different graphical elements?**




*

Please choose **only one** of the following:

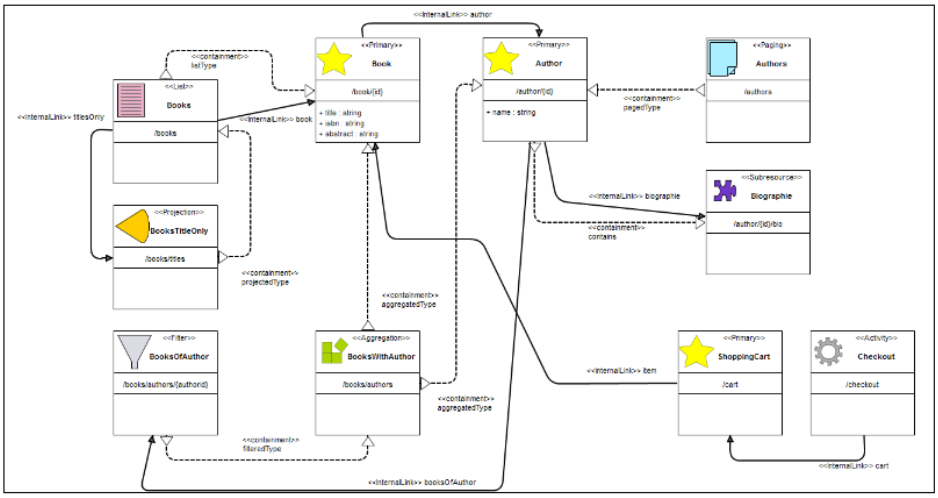
- ○ Yes
- ○ No

## Moody :: Cognitive Fit

Questions regarding "Cognitive Fit" criterion

**45 [QMCF_0002]**

**Do you feel overwhelmed by the mass of information in the following diagram?**




*

Please choose **only one** of the following:

- ○ Yes
- ○ No

**46 [QMCF_0003a]**

**Imagine you wanted to draw the model of a resource-oriented application on a whiteboard, for example. Do you think the notation presented in this questionnaire would be useable for that purpose?**

*

Please choose **only one** of the following:

- ○ Yes
- ○ No

**47 [QMCF_0003b]**

**Imagine you had a dedicated editor that would support modeling using Visual REST. Do you think the notation could be useable for you then?**

*

Please choose **only one** of the following:

- ○ Yes
- ○ No

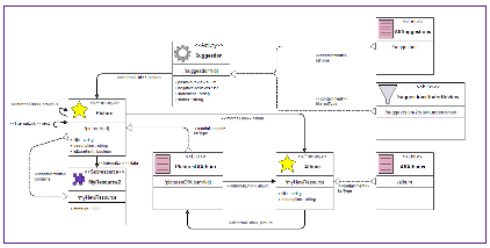
## General Comments

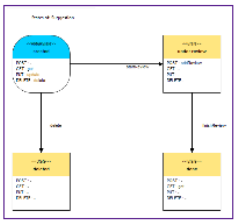
In this section you can give some general comments on the notation which are beyond what we could cover with our questions.

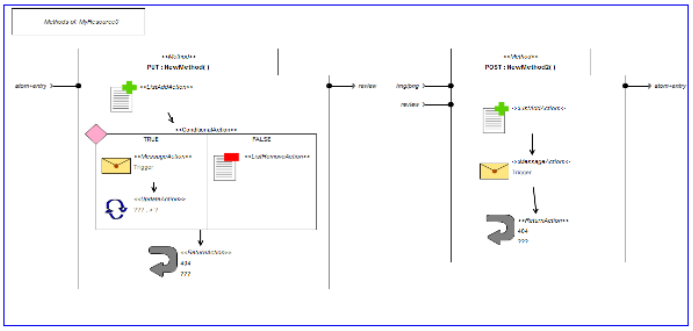
**48 [QGC_0001]**

**Do you have any comments on the notation as a whole? What do you like? What do you dislike? What would you change if you could? Are there things you do not understand or that are not clear? Would there be anything that could be done to improve the situation?**

**Here are the diagrams used as examples throughout this questionnaire again for reference. You can click on the individual diagrams to see a larger version.**







Please write your answer here:

**49 [QGC_0002]**

**As a token of gratitude for your participation you have the opportunity to win one of these books:**

- **Bruce A. Tate - Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages**
- **Clinton Wong - HTTP Pocket Reference**
- **and for those not into IT: Hermann Maurer - Xperten Bd.1: Der Telekinet**

**Just enter your email address in the following field for the chance to win. Your email address will only be used to get in touch with you should you win and will not be used for any other purpose. It will not be shared with any third party.**

**Entering your email address is optional and does not influence the evaluation in any way.**

Please write your answer here:

Thank you for participating in the **Visual REST** graphical notation survey. Your input is highly appreciated. If you are interested in the results of the evaluation please feel free to send me an email to oliver@van-porten.de.
01.01.1970 – 01:00

Submit your survey.
Thank you for completing this survey.

# B Comments given in the questionnaire

Die Diagramme mit verschiedenen Ressourcentypen erscheinen etwas "unruhig" und sind daher schwer auf einen Blick zu erfassen - es ist "zuviel los". UML-Stereotypen sind für mich schnell zu erfassen - zumindest im Moment, da die Icons noch ungewohnt sind - und sorgen auch für mehr Ruhe in der Darstellung. Die Auswahl der Icons selbst ist allerdings gut, ich kann an einzelnen Artefakten relativ schnell erkennen, was die Semantik ist. Nur dhie farbliche Komponente führt wohl in der Praxis nicht weiter, da sie oft wegfallen wird (Whiteboards, Papers, Ausdrucke ...).

I really wonder why generalisation is abused (by having the symbol doubled-up on a dependency arrow) to make containment rather than using aggregation. I'm concerned that competing arcs in the state diagram don't have real conditions. I'm concerned that the uniform HTTP interface is under-represented (especially missing PATCH).

I would remove the URI patterns - they are not of relevance. The most important part the represent between the resources are the link relations; this is for contained/embedded resources too. Fwiw, I have a generic media type for representing resources which I have visualised graphically `http://stateless.co/hal_specification.html` I have thought about producing a graphical client for browsing applications exposed with that media type - if you think there is scope for collaboration definitely get in touch Best, Mike

Ich bin eher der textuelle Typ, deshalb ist die Sprache für mich ein bisschen zu "bunt". Da die Anzahl der Symbole überschaubar ist, denke ich aber, dass man sich mit etwas Einarbeitungszeit daran gewöhnen kann. Die Bedeutungen der einzelnen Symbole ist für einen REST-"Laien" nicht immer eindeutig. Wenn man mehr Erfahrung, insbesondere mit Ressource-Typen und den verfügbaren Aktionen hat, kann ich mir aber vorstellen, dass es klarer wird. Ansonsten hoffe ich, dass diese Umfrage einen guten Beitrag zur Abschlussarbeit liefer und wünsche noch viel Erfolg dabei.

i think there are some fundamental flaws in the design here. for example,

"sub-resource" is not a needed representation; just "resource" will do just fine. modeling update as a "circle of arrows" is needlessly complex. a single icon to represent an idempotent write should be sufficient. modeling REST using a single protocol (HTTP) is needlessly limiting. REST addresses much more than resources, it also representation intermediaries (caches), code-on-demand, etc. i don't see them represented here at all. i think using color in your design is a problem. there will be users who are color-blind; black and white representations of the diagram will loose important details. in several cases the arrows seem disjointed; it's not clear where they are coming from or leading to. as REST promotes a stateless interaction, it it not clear how you plan to represent state (it is "stand-alone?", part of some resource? part of a server somewhere? which server?, etc.). i like the idea of representing an architectural style visually and i am not sure this captures the POV of Fielding's REST.

I have the attention span of about 2 seconds. I give 5 days of training course in a row on resource-oriented arhictecutres. Some of those idioms could be quite useful when drawing stuff on the white board. A lot of them I'm not sure about.

I would hardly use graphical notation for designing a REST system, but perhaps could be useful for documentation purposes. Your notation is not really describing a REST system; otherwise there will not be custom verbs, but only GET/PUT/POST/DELETE. I would much more prefer describing a REST system as suggested here `http://www.ibm.com/developerworks/rational/library/basic-profile-linked-data/index.html`.

- wirkt an manchen stellen etwas bunt - das sternchen bei create... erkennt man nicht unbedingt als solches - stern (prim resource?) suggerriert eine semantik wie "favorit"

Great job! I really like this stuff and I am sure, that it is helpful für designing RESTful apps.

In my opinion, adding icons does not help out someone who is trying to understand the semantics of the diagram. E.g. in `https://survey.van-porten.de/upload/surveys/26232/images/Photoalbum.png`, it does not help me that I see icons in understanding how the service works as a whole. However, it does help me to quickly understand which types of resources are available in the service - e.g. I see that there is a filter and several types of list resources. Therefore, I am quickly able to focus on a specific resource type or determine if the service supports filtering at all. Also, I think it would be useful to some-

how indicate which of the resources may be used as application entry points in order to navigate the resource space onward.

To use the icons on a whiteboard I need to be able to draw them myself by hand quickly. Some of the icons are hard to draw.

- Not all “Resource Types” you use have immediate Meaning to me - The transition Names in the state diagramm seem strange. You’d expect http verbs here. - The usage of “Method” feels strange. In (Http-) REST there are just four Methods: GET; PUT; POST; DELETE. “PUT: NewMethod()” does not make any sense. - In the Questionary when asking for the meaning of Conditional Action you should not have the text “COND NAME” in the symbol.

good luck interesting

Sorry, I don’t see the point. REST has a uniform interface, so every GET PUT etc will be the same. The resources will be different, as will their media types. How do these diagrams help in any way? What are they for? What problem are they solving?

I dislike the funnel as a symbol for filtering - I guessed it correctly because it’s used in other apps, but I don’t like it. Generally, I like your work. I think it’ll be useful.

The symbols are to colorful. I would prefer if the symbols would use a more consistent style. In the complexityMgmtFull.png it is not clear what is the meaning of the arrows that point to the method boxes. What does the content of those boxes mean?

Viewing resources as types with methods makes the approach seem more object oriented. Use of such a system requires one to know all the types in the system, and how each type may be used. This makes it possible that one finally ends up with a system that requires out-of-band documentation to use, and hence not hypertext driven. Don’t take me to mean what you have is not useful. I only think it exclusively focuses on the server, and helps in server implementation. Starting with a client view (define the mediatypes and the types of hyperlinks), and moving on to constructing the server components from those primitives will help build a RESTful system.

Ich würde bei den Grafiken weniger gefüllte Flächen verwenden Die Symbole

für Primary und Subressource könnten (sollten?) etwas gemeinsames haben. Ein Editor wäre cool ;-) Würd ich nutzen

From a purely aesthetic perspective, I think sequences in the third diagram could be represented in a nicer, less ambiguous manner than the use of little, short arrows. Maybe a grid could be used whereby adjacent grid squares are used to indicate progression through state? The icons themselves are a little rough and could be improved... In general, I think the state transition aspects of the diagram are useful, some aspects clutter up the diagrams too much which risks reducing the communication value of these diagrams. I don't think that any graphical representation will ever be both comprehensive and comprehensible at the same time. In my mind, the inclusions of list operations and conditional items is too detailed, particularly as important aspects about them cannot be simply represented in a graphical manner (e.g. the conditions in the conditional block). Arguably other aspects of a RESTful API such as HTTP status codes are equally important but I struggle to see how more than 2-3 different returns can be modeled without cluttering things up. Otherwise, great effort!

The most symbols can be right interpreted without a declarative text and the not directly interpretable symbols can be remembered once explained. Such for a application there should be something like a show legend function then the description of each symbol in the graphical representation can be omitted.

Simplify it.. to many different elements with subtle differences. Clearer icons might help.

I think there's too much emphasis on the internal change of state of the resources, and not enough emphasis on HATEOAS. There's also nothing said about such matters as headers (etags, caching etc); the only thing I can see is a 404 return code. There's also nothing about conneg.

# C Source Code Statistics

| Plugin | # Lines | Generated |
|---|---|---|
| de.van_porten.vrest.ui | 10,292 | 0% |
| de.van_porten.vrest.ui.properties | 40,489 | 98% |
| de.van_porten.vrest.core | 11,348 | 100% |
| de.van_porten.vrest.core.edit | 6,453 | 100% |
| de.van_porten.vrest.core.editor | 4,814 | 100% |
| de.van_porten.vrest.bundle | 24 | 0% |
| de.van_porten.vrest.help | 6 | 0% |
| Total | 76,223 | 76% |

Table C.1: Source lines of code of the Visual REST plugins

| Plugin | # Lines | Generated |
|---|---|---|
| de.van_porten.vrest.tests.core | 2,117 | 80% |
| de.van_porten.vrest.tests.ui | 651 | 0% |
| de.van_porten.vrest.tests.recorded | 59 | 95% |
| Total | 2,827 | 62% |

Table C.2: Source lines of code of the Visual REST tests

# D Graphiti Code Samples

Listing D.1: Graphiti FeatureProvider

```
1 public class RestStructureFeatureProvider
2     extends CommonFeatureProvider {
3   public RestStructureFeatureProvider(
4       IDiagramTypeProvider dtp) {
5     super(dtp);
6   }
7
8   @Override
9   public IAddFeature getAddFeature(IAddContext context) {
10    Object addedClass = context.getNewObject();
11    if (addedClass instanceof IRestModelContainer) {
12      addedClass = ((IRestModelContainer) addedClass)
13        .getModel();
14    }
15
16    if (addedClass instanceof PrimaryResourceType) {
17      return new AddPrimaryResourceTypeFeature(this);
18    } else if (...) {
19      ...
20    }
21    return super.getAddFeature(context);
22  }
23
24  @Override
25  public ICreateFeature[] getCreateFeatures() {
26    return new ICreateFeature[] {
27        new CreatePrimaryResourceTypeFeature(this),
28        new CreatePagingResourceTypeFeature(this),
29        ... };
30  }
31
32  @Override
33  public IUpdateFeature getUpdateFeature(
```

```
34        IUpdateContext context) {
35      PictogramElement pictogramElement = context
36          .getPictogramElement();
37      Object bo =
38        getBusinessObjectForPictogramElement(
39          pictogramElement);
40      if (bo instanceof PrimaryResourceType) {
41        return new UpdatePrimaryResourceTypeFeature(
42          this);
43      } else if (...) {
44        ...
45      }
46      return super.getUpdateFeature(context);
47    }
48
49    ...
50
51    @Override
52    public ICustomFeature[] getCustomFeatures(
53        ICustomContext context) {
54      return new ICustomFeature[] {
55          new DrillDownResourceTypeStatesFeature(this),
56          new DrillDownResourceTypeMethodsFeature(this),
57          new CollapseResourceTypeFeature(this),
58          new ExpandResourceTypeFeature(this) };
59    }
60  }
```

Listing D.2: Graphiti DiagramTypeProvider

```
1  public class RestStructureDiagramTypeProvider extends
2      AbstractDiagramTypeProvider {
3
4    private IToolBehaviorProvider[] toolBehaviorProviders;
5
6    public RestStructureDiagramTypeProvider() {
7      super();
8      setFeatureProvider(new
9        RestStructureFeatureProvider(this));
10   }
11
12   @Override
13   public IToolBehaviorProvider[]
14       getAvailableToolBehaviorProviders() {
```

```
15     if (toolBehaviorProviders == null) {
16       toolBehaviorProviders = new
17         IToolBehaviorProvider[] {
18           new RestStructureToolBehaviorProvider(this)
19        };
20     }
21     return toolBehaviorProviders;
22   }
23 }
```

Listing D.3: Graphiti AddFeature

```
1 public class AddStateFeature
2     extends AbstractAddShapeFeature {
3   ...
4   public AddStateFeature(IFeatureProvider fp) {
5     super(fp);
6   }
7
8   @Override
9   public boolean canAdd(IAddContext context) {
10    if (context.getNewObject() instanceof State) {
11      if (context.getTargetContainer()
12            instanceof Diagram) {
13        return true;
14      }
15    }
16    return false;
17  }
18
19  @Override
20  public PictogramElement add(IAddContext context) {
21    State addedClass = (State) context.getNewObject();
22    Diagram targetDiagram = (Diagram)
23      context.getTargetContainer();
24
25    ...
26
27    IPeCreateService peCreateService =
28      Graphiti.getPeCreateService();
29    IGaService gaService = Graphiti.getGaService();
30    ContainerShape containerShape =
31      peCreateService.createContainerShape(
32        targetDiagram, true);
```

```
33
34      if (addedClass.eResource() == null) {
35        getDiagram().eResource()
36          .getContents().add(addedClass);
37      }
38      link(containerShape, addedClass);
39
40      int width = context.getWidth() <= 0 ? 150
41        : context.getWidth();
42      int height = context.getHeight() <= 0 ? 150
43        : context.getHeight();
44      PropertyUtil.setStateShape(containerShape);
45      ...
46      {
47        foregroundColor = COLOR_STATE_FOREGROUND;
48        backgroundColor = COLOR_STATE_BACKGROUND;
49        textColor = COLOR_STATE_TEXT_FOREGROUND;
50
51        Rectangle rect = gaService
52          .createRectangle(containerShape);
53        rect.setFilled(true);
54        rect.setForeground(manageColor(foregroundColor));
55        rect.setBackground(manageColor(backgroundColor));
56        rect.setLineWidth(1);
57        rect.setStyle(StyleUtil
58          .getStyleForState(getDiagram()));
59        gaService.setLocationAndSize(rect,
60            context.getX(), context.getY(),
61            width, height);
62      }
63
64      ...
65
66      /*** BEGIN: Name Label ***/
67      {
68        Shape shape = peCreateService
69          .createShape(containerShape, false);
70        Text text = gaService.createDefaultText(
71            getDiagram(), shape,
72            addedClass.getName());
73        text.setForeground(manageColor(textColor));
74        text.setHorizontalAlignment(
75            Orientation.ALIGNMENT_CENTER);
```

```
76        text.setVerticalAlignment(
77            Orientation.ALIGNMENT_CENTER);
78        text.setFont(gaService.manageDefaultFont(
79            getDiagram(), false, true));
80        gaService.setLocationAndSize(text, 0, 30, width, 25);
81
82        PropertyUtil.setStateNameShape(shape);
83
84        /* create link and wire it */
85        link(shape, addedClass);
86      }
87      /*** END: Name Label ***/
88
89      ...
90
91      /* add a chopbox anchor to the shape */
92      peCreateService.createChopboxAnchor(containerShape);
93
94      /* call the layout feature */
95      layoutPictogramElement(containerShape);
96
97      return containerShape;
98    }
99 }
```

Listing D.4: Graphiti CreateFeature

```
1  public class CreateStateFeature
2      extends AbstractCreateFeature {
3    public CreateStateFeature(IFeatureProvider fp) {
4      super(fp, "State",
5        "Create a new State for the
6          underlying ResourceType");
7    }
8
9    @Override
10   public boolean canCreate(ICreateContext context) {
11     return context.getTargetContainer() instanceof Diagram;
12   }
13
14   @Override
15   public String getCreateImageId() {
16     return RestImageProvider.IMG_STATES_STATE;
17   }
```

```
18
19   @Override
20   public Object[] create(ICreateContext context) {
21     State newClass = RestBehaviorFactory
22         .eINSTANCE.createState();
23     newClass.setName("NewState");
24     try {
25       ResourceUtil.saveToModelFile(newClass, getDiagram());
26     } catch (CoreException e) {
27       e.printStackTrace();
28     } catch (IOException e) {
29       e.printStackTrace();
30     }
31     ResourceType parent = (ResourceType)
32       getBusinessObjectForPictogramElement(getDiagram());
33     parent.getStates().add(newClass);
34     addGraphicalRepresentation(context, newClass);
35     return new Object[] { newClass };
36   }
37 }
```

# List of Abbreviations

| | |
|---|---|
| **DSL** | domain-specific language |
| **REST** | Representational State Transfer |
| **RCP** | Rich Client Platform |
| **EEF** | Enhanced Editing Framework |
| **EMF** | Eclipse Modeling Framework |
| **GEF** | Graphical Editing Framework |
| **GMF** | Graphical Modeling Framework |
| **GMP** | Graphical Modeling Project |
| **SDK** | Software Development Kit |
| **API** | Application Programming Interface |
| **WWW** | World Wide Web |
| **URI** | Uniform Resource Identifier |
| **URL** | Uniform Resource Locator |
| **SOAP** | formerly *Simple Object Access Protocol*. No longer an acronym since version 1.2 |
| **HTTP** | Hypertext Transfer Protocol |
| **OOD** | object-oriented software development |
| **UML** | Unified Modeling Language |
| **MDD** | Model-Driven Development |
| **IDE** | Integrated Development Environment |
| **UI** | User Interface |
| **TDD** | Test-Driven Development |
| **CI** | Continuous Integration |
| **DFD** | Data Flow Diagram |
| **SWT** | Standard Widget Toolkit |
| **json** | JavaScript Object Notation |

# Bibliography


[1] M. Fowler, *Domain-Specific Languages*, 1st edition. Boston, Massachusetts, USA: Addison-Wesley Professional, 2010.

[2] T. Stahl, M. Völter, S. Efftinge, and A. Haase, *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*, 2nd edition. Heidelberg, Germany: dpunkt.verlag, 2007.

[3] R. T. Fielding, "Architectural styles and the design of network-based software architectures", Thesis (PhD), University of California, Irvine, 2000.

[4] S. Tilkov, *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*, 2nd edition. Heidelberg: dpunkt.verlag, 2011.

[5] The Object Management Group. (2005). Unified Modeling Language 2.0, [Online]. Available: `http://www.uml.org/` (visited on 02/20/2012).

[6] S. Schreier, "Modeling RESTful applications", in *WS-REST*, R. Alarcón, C. Pautasso, and E. Wilde, Eds., ACM, 2011, pages 15–21.

[7] D. Moody and J. van Hillegersberg, "Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Suit of Diagrams", *Information Systems Journal*, volume 5452, pages 134–163, 2008.

[8] D. L. Moody, "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering", *IEEE Transactions on Software Engineering*, volume 35, number 6, pages 756–779, Nov. 2009.

[9] *Eclipse*, `http://www.eclipse.org/` (visited on 02/20/2012).

[10] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap", in *Future of Software Engineering FOSE 07*, IEEE, IEEE Computer Society, 2007, pages 37–54.

[11] B. Selic, "The pragmatics of model-driven development", *IEEE Software*, volume 20, number 5, pages 19–25, Sep. 2003.

[12] G. Pietrek, J. Trompeter, B. Niehues, T. Kamann, B. Holzer, M. Kloss, K. Thoms, J. C. F. Beltran, and S. Mork, *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*, 1st edition, J. Trompeter and G. Pietrek, Eds. Frankfurt: entwickler.press, 2007.

[13] D. R. Bertolami, M. Rüedlinger, F. Buchli, and M. Hofer, "Domänenspezifische Sprachen - Verschiedene Ansätze im Vergleich", *ObjektSpektrum*, volume 03, pages 74–79, 2011.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. (1999). Hypertext Transfer Protocol – HTTP/1.1, [Online]. Available: `http://www.ietf.org/rfc/rfc2616.txt` (visited on 02/20/2012).

[15] T. Berners-Lee, R. Fielding, and L. Masinter. (1998). Uniform Resource Identifiers (URI), [Online]. Available: `http://www.ietf.org/rfc/rfc2396.txt` (visited on 02/20/2012).

[16] N. Freed and N. Borenstein. (1996). Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, [Online]. Available: `http://www.ietf.org/rfc/rfc2046.txt` (visited on 02/20/2012).

[17] J. Postel. (1994). Media Type Registration Procedure, [Online]. Available: `http://www.ietf.org/rfc/rfc1590.txt` (visited on 02/20/2012).

[18] T. Berners-Lee, L. Masinter, and M. McCahill. (1994). Uniform Resource Locators (URL), [Online]. Available: `http://www.ietf.org/rfc/rfc1738.txt`.

[19] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. (2007). SOAP Version 1.2, [Online]. Available: `http://www.w3.org/TR/soap12/` (visited on 03/17/2012).

[20] *Apache Subversion*, `http://subversion.apache.org/` (visited on 02/20/2012).

[21] C. Atkinson and T. Kuhne, "Model-driven development: a metamodeling foundation", *IEEE Software*, volume 20, number 5, pages 36–41, Sep. 2003.

[22] M. Lankhorst, *Enterprise Architecture at Work: Modelling, Communication and Analysis*, 2nd edition. Berlin, Germany: Springer, 2009.

[23] J. Lee, "Design Rationale Systems: Understanding the Issues", *IEEE Expert*, volume 12, number 3, pages 78–85, 1997.

[24] D. L. Moody, P. Heymans, and R. Matulevicius, "Improving the Effectiveness of Visual Representations in Requirements Engineering: An Evaluation of i* Visual Syntax", in *IEEE 17th International Requirements Engineering Conference*, Washington, DC, USA: IEEE, Aug. 2009, pages 171–180.

[25] A. F. Blackwell, C. Britton, A. L. Cox, T. R. G. Green, C. A. Gurr, G. F. Kadoda, M. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young, "Cognitive Dimensions of Notations: Design Tools for Cognitive Technology", in *Proceedings of the 4th International Conference on Cognitive Technology: Instruments of Mind*, London, UK, UK: Springer-Verlag, 2001, pages 325–341.

[26] T. Green and A. Blackwell. (Oct. 1998). Cognitive Dimensions of Information Artefacts: a tutorial (Version 1.2), [Online]. Available: `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf` (visited on 03/17/2012).

[27] T. Green, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework", *Journal of Visual Languages and Computing*, volume 7, number 2, pages 131–174, Jun. 1996.

[28] T. R. G. Green, "Cognitive Dimensions of Notations", in *People and Computers*, A. Sutcliffe and L. Macaulary, Eds., Cambridge, UK: Cambridge University Press, 1989, pages 443–460.

[29] G. Costagliola, A. Delucia, S. Orefice, and G. Polese, "A Classification Framework to Support the Design of Visual Languages", *Journal of Visual Languages and Computing*, volume 13, number 6, pages 573–600, Dec. 2002.

[30] G. Costagliola, V. Deufemia, and G. Polese, "A framework for modeling and implementing visual notations with applications to software engineering", *ACM Transactions on Software Engineering and Methodology*, volume 13, number 4, pages 431–487, Oct. 2004.

[31] M. Petre, A. F. Blackwell, and T. R. G. Green, "Cognitive Questions in Software Visualisation", *Software Visualization: Programming as a Multi-Media Experience*, pages 453–480, 1998.

[32] S. Kirchhoff, S. Kuhnt, P. Lipp, and S. Schlawin, *Der Fragebogen. Datenbasis, Konstruktion und Auswertung*, 3rd edition. Stuttgart, Germany: UTB, 2006.

[33] E. Raab-Steiner and M. Benesch, *Der Fragebeogen - Von der Forschungsidee zur SPSS/PASW-Auswertung*, 2nd edition. Stuttgart, Germany: UTB, 2010.

[34] M. Bühner, *Einführung in die Test- und Fragebogenkonstruktion*, 3rd edition. München, Germany: Pearson Studium, 2010.

[35] G. A. Lienert and U. Raatz, *Testaufbau und Testanalyse*, 6th edition. Weinheim, Germany: BeltzPVU, 1998.

[36] J. Rost, *Lehrbuch Testtheorie - Testkonstruktion*, 2nd edition. Bern: Huber, Feb. 2004.

[37] H. Moosbrugger and A. Kelava, *Testtheorie und Fragebogenkonstruktion*, 1st edition. Heidelberg, Germany: Springer, 2007.

[38] IBM, *IBM SPSS Statistics 20*, `http://www.ibm.com/software/de/analytics/spss/` (visited on 11/02/2012).

[39] F. Brosius, *SPSS 19*, 1st edition. Heidelberg, Germany: mitp, 2011.

[40] *GNU PSPP*, `http://www.gnu.org/software/pspp/` (visited on 02/20/2012).

[41] *The R Project*, `http://www.r-project.org/` (visited on 02/20/2012).

[42] A. F. Blackwell and T. R. G. Green, "A Cognitive Dimensions Questionnaire Optimised for Users", *Proceedings of the 12th Workshop of the Psychology of Programming Interest Group*, pages 137–154, 2000.

[43] A. E. Bobkowska, "A Methodology of Visual Modeling Language Evaluation", in *SOFSEM*, P. Vojtás, M. Bieliková, B. Charron-Bost, and O. Sýkora, Eds., volume 3381, Springer, 2005, pages 72–81.

[44] —, "Cognitive Dimensions Questionnaire Applied to Visual Modelling Language Evaluation - a Case Study", in *Proceedings of the 15th Workshop of the Psychology of Programming Interest Group*, I. M. Petre and D. Budgen, Eds., Keele UK, Apr. 2003, pages 367–378.

[45] —, "A framework for methodologies of visual modeling language evaluation", in *Proceedings of the 2005 symposia on Metainformatics - MIS '05*, New York, New York, USA: ACM Press, 2005.

[46] L.-O. Johansson, M. Wärja, H. Kjellin, and S. Carlsson, "Graphical modeling techniques and usefulness in the Model Driven Architecture: Which are the criteria for a "good"Computer indepedant model?", *Proceedings of The 31st Information Systems Research Seminar in Scandinavia (IRIS31)*, V. Asproth, Ed., 2008.

[47] *Eclipse Rich Client Platform*, `http://www.eclipse.org/rcp/` (visited on 02/20/2012).

[48] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns and Plug-Ins*, E. Gamma, L. Nackman, and J. Wiegand, Eds. Amsterdam, Netherlands: Addison-Wesley Longman, 2003.

[49] E. Clayberg and D. Rubel, *Eclipse: Building Commercial-Quality Plug-ins*, 3rd edition, E. Gamma, L. Nackman, and J. Wiegand, Eds. Amsterdam, Netherlands: Addison-Wesley Longman, 2009.

[50] *Eclipse Modeling Framework*, `http://www.eclipse.org/modeling/emf/` (visited on 02/20/2012).

[51] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclispe Modeling Framework*, 2nd edition, E. Gamma, L. Nackman, and J. Wiegand, Eds. Amsterdam, Netherlands: Addison-Wesley Longman, 2009, page 704.

[52] *Graphical Editor Framework*, `http://www.eclipse.org/gef/` (visited on 02/20/2012).

[53] *Graphical Modeling Framework*, `http://www.eclipse.org/gmf/` (visited on 02/02/2012).

[54] R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, 1st edition, E. Gamma, L. Nackman, and J. Wiegand, Eds. Amsterdam, Netherlands: Addison-Wesley Longman, 2009.

[55] *Eclipse Graphiti*, `http://www.eclipse.org/graphiti/` (visited on 02/20/2012).

[56] C. Brand, M. Gorning, T. Kaiser, J. Pasch, and M. Wenz, "Graphiti - Entwicklung hochwertiger grafischer Modelleditoren", *Eclipse Magazin*, volume 1, pages 67–72, 2011.

[57] *Graphical Modeling Project*, `http://www.eclipse.org/modeling/gmp/` (visited on 02/20/2012).

[58] *Enhanced Editing Framework*, `http://www.eclipse.org/modeling/emft/?project=eef` (visited on 02/20/2012).

[59] *jUnit*, `http://www.junit.org/` (visited on 02/20/2012).

[60] *SWTBot*, `http://eclipse.org/swtbot/` (visited on 02/20/2012).

[61] *Google WindowTester Pro*, `http://code.google.com/javadevtools/wintester/html/index.html` (visited on 11/18/2011).

[62] *Maven*, `http://maven.apache.org/` (visited on 02/20/2012).

[63] *Tycho*, `http://www.eclipse.org/tycho/` (visited on 02/20/2012).

[64] P. M. Duvall, *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston, Massachusetts, USA: Addison-Wesley Professional, 2007.

[65] *Jenkins Continuous Integration*, `http://www.jenkins-ci.org` (visited on 02/20/2012).

[66] *Hudson Continuous Integration*, `http://hudson-ci.org/` (visited on 02/20/2012).

[67] P. T. Quinlan, "Visual feature integration theory: past, present, and future", *Psychological Bulletin*, volume 129, number 5, pages 643–673, 2003.

[68] A. M. Treisman and G. Gelade, "A feature-integration theory of attention", *Cognitive Psychology*, volume 12, number 1, pages 97–136, 1980.

[69] J. Mackinlay, "Automating the design of graphical presentations of relational information", *ACM Transactions on Graphics*, volume 5, number 2, pages 110–141, Apr. 1986.

[70] W. Winn, "An Account of How Readers Search for Informations in Diagrams", *Contemporary Educational Psychology*, volume 18, pages 162–185, 1993.

[71] M. Petre, "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming", *Communications of the ACM*, volume 38, number 6, pages 33–44, 1995.

[72] J. Sweller, "Cognitive load theory, learning difficulty, and instructional design", *Learning and Instruction*, volume 4, number 4, pages 295–312, 1994.

[73] R. Mayer and R. Moreno, "Nine Ways to Reduce Cognitive Load in Multimedia Learning", *Educational Psychologist*, volume 38, number 1, pages 43–52, 2003.

[74] C. S. Peirce, *Charles S. Peirce: The Essential Writings (Great Books in Philosophy)*. Amherst, New York, USA: Prometheus Books, 1998.

[75] T. DeMarco, *Structured Analysis and System Specification*. Yourdon Press, 1979, pages 409–424.

[76] P. Bruza and T. P. Van Der Weide, "The semantics of data flow diagrams", *Information Systems Journal*, pages 1–13, 1993.

[77] K. Siau, "Informational and Computational Equivalence in Comparing Information Modeling Methods", *Journal of Database Management*, volume 15, number 1, pages 73–86, 2004.

[78] J. Hahn and J. Kim, "Why are some diagrams easier to work with? Effects of diagrammatic representation on the cognitive intergration process of systems analysis and design", *ACM Transactions on Computer-Human Interaction*, volume 6, number 3, pages 181–213, 1999.

[79] J. Kim, J. Hahn, and H. Hahn, "How Do We Understand a System with (So) Many Diagrams? Cognitive Integration Processes in Diagrammatic Reasoning", *Information Systems Research*, volume 11, number 3, pages 284–303, 2000.

[80] K. Lynch, *The Image of the City*. Cambridge, Massachusetts, USA: The MIT Press, 1960, ch. 16.

[81] A. v. K. Lemon and O. v. K. Lemon, "Constraint Matching for Diagram Design: Qualitative Visual Languages", in *Proceedings of the First International Conference on Theory and Application of Diagrams*, London, UK: Springer-Verlag, 2000, pages 74–88.

[82] G. A. Miller, "The Magical Number Seven, Plus or Minus Two", *Psychological Review*, volume 63, number 2, pages 81–97, 1956.

[83] A. McNeile and N. Simons, "Methods of Behaviour Modelling. A Commentary on Behaviour Modelling Techniques for MDA", *Architecture*, 2004.

[84] E. Tufte, *Envisioning Information*. Cheshire, Connecticut, USA: Graphics Press, 1990.

[85] —, *The Visual Display of Quantitative Information*, 2nd edition, 2. Cheshire, Connecticut, USA: Graphics Press, Jun. 2001.

[86] A. White, *The Elements of Graphic Design: Space, Unity, Page Architecture and Type*. New York: Allworth Press, 2002.

[87] *LimeSurvey*, `http://www.limesurvey.org/` (visited on 02/20/2012).

[88] D. Green and J. Swets, *Signal Detection Theory and Psychophysics*. New York: Wiley, 1966.

[89] G. Lohse, "The Role of Working Memory in Graphical Information Processing", *Behaviour and Information Technology*, volume 16, number 6, pages 297–308, 1997.

[90] C. Shannon and W. Weaver, *The Mathematical Theory of Communication*. Urbana, Illinois, USA: University of Illinois Press, 1963.

[91] The Object Management Group, *Business Process Modeling Notation 2.0*, `http://www.bpmn.org/` (visited on 02/20/2012), 2011.

[92] *Eclipse Buckminster*, `http://eclipse.org/buckminster/` (visited on 02/20/2012).

[93] *Maven Surefire Plugin*, `http://maven.apache.org/plugins/maven-surefire-plugin/` (visited on 02/20/2012).

[94] *Hamcrest*, `http://code.google.com/p/hamcrest/` (visited on 02/20/2012).

[95] *JMockit*, `http://code.google.com/p/jmockit/` (visited on 02/20/2012).

[96] K. Beck, *Test Driven Development: By Example*, 1st edition. Boston, Massachusetts, USA: Addison-Wesley Professional, 2002.

[97] K. Schwaber and J. Sutherland. (2011). Scrum Guide, [Online]. Available: `http://www.scrum.org/scrumguides/` (visited on 03/17/2012).

[98] *Jubula*, `http://eclipse.org/jubula/` (visited on 02/20/2012).

[99] *famfam Silk Icons*, `http://www.famfamfam.com/lab/icons/silk/` (visited on 02/20/2012).

[100] W. Schneider, "Entwicklung eines Codegenerators, der Modelle ressourcenorientierter Anwendungen in eine funktionale Sprache umsetzt", Masterarbeit am Lehrgebiet Datenverarbeitungstechnik, FernUniversität in Hagen, 2012.

[101] J. Brüwer, "Entwicklung eines Java-Codegenerators für ressourcenorientierte Anwendungen", Masterarbeit am Lehrgebiet Datenverarbeitungstechnik, FernUniversität in Hagen, 2012.

[102] *EuGENia*, `http://www.eclipse.org/gmt/epsilon/doc/eugenia/` (visited on 02/20/2012).

[103] *Eclipse Epsilon*, `http://www.eclipse.org/gmt/epsilon/` (visited on 02/20/2012).

[104] *Spray*, `http://code.google.com/a/eclipselabs.org/p/spray/` (visited on 02/20/2012).

[105] M. Boger. (2011). A DSL for Graphical Editors - A Proposal for the Spray project, [Online]. Available: `http://spray.eclipselabs.org.codespot.com/files/A%20DSL%20for%20graphical%20editors-2011-05-31.pdf` (visited on 03/17/2012).

[106] Gentleware AG. (2011). Poseidon for DSLs User Guide, [Online]. Available: `http://www.gentleware.com/fileadmin/media/archives/userguides/poseidon-for-dsls/user-guide/Poseidon_for_DSLs_Documentation.html` (visited on 03/17/2012).

[107] J. Warmer, K. Thoms, and J. Reichert. (2011). Spray User Guide, [Online]. Available: `http://spray.eclipselabs.org.codespot.com/files/SprayUserGuide.pdf` (visited on 03/17/2012).