

Jörn Friedrich Dreyer

INSTANT STORYBOARDING

Immediate visualization of ontology learning for acceptance tests
with graph transformations in web applications

JÖRN FRIEDRICH DREYER

INSTANT STORYBOARDING

IMMEDIATE VISUALIZATION

OF ONTOLOGY LEARNING

FOR ACCEPTANCE TESTS

WITH GRAPH TRANSFORMATIONS

IN WEB APPLICATIONS

DISSERTATION submitted in fulfillment of the requirements for the degree of Doktor der Naturwissenschaften (Dr. rer. nat.) to the department FB 16 Electrical Engineering / Computer Science, Software Engineering Research Group at the University of Kassel.
Disputation in Kassel / Witzenhausen on November 11th 2014

Copyright © 2015 by Jörn Friedrich Dreyer
All rights reserved.
ISBN: 1-51524-095-9
ISBN-13: 978-1-51524-095-2

A digital version of this thesis is freely available in KOBRA (**K**asseler **O**nline**B**ibliothek, **R**epository und **A**rchiv), the e-library for scientific contents from the University of Kassel:
<https://kobra.bibliothek.uni-kassel.de/>

DAS MUSS DOCH AUCH EINFACHER GEHEN!

ALBERT ZÜNDORF

Acknowledgements

My profound thanks to Albert Zündorf for the advice on graph transformation theory as well as extra history lessons on the subject. His continued support and guidance led me the right way.

Equally profound thanks go to Pieter Van Gorp for his thorough reviews and support before this thesis was even started. His constructive feedback allowed me to polish key sections in my theory.

I would also like to extend my appreciation to Leif Geiger and Andreas Hotho, who guided me on the first paper that sparked off this whole thesis. Their deep knowledge in Fujaba and natural language processing allowed me to work on bringing the two research groups a little closer to each other.

Special thanks to Christoph Eickhoff, Sascha Müller and Bernhard Grusie for their assistance in the software development. Special extended thanks go to Christian Schneider, who invited me to Kassel in the first place.

I would also like to express my deepest gratitude to my wife Nina as well as Alexandra and Benedikt. Without their competitive ideas it would have taken me a lot more time to complete the thesis, if ever.

Abstract

This thesis aims at empowering software customers with a tool to build software tests themselves, based on a gradual refinement of natural language scenarios into executable visual test models. The process is divided in five steps:

1. First, a natural language parser is used to extract a graph of grammatical relations from the textual scenario descriptions.
2. The resulting graph is transformed into an informal story pattern by interpreting structuring rules based on Fujaba Story Diagrams.
3. While the informal story pattern can already be used by humans the diagram still lacks technical details, especially type information. To add them, a recommender based framework uses web sites and other resources to generate formalization rules.
4. As a preparation for the code generation the classes derived for formal story patterns are aligned across all story steps, substituting a class diagram.
5. Finally, a headless version of Fujaba is used to generate an executable JUnit test.

The graph transformations used in the browser application are specified in a textual domain specific language and visualized as story pattern. Last but not least, only the heavyweight parsing (step 1) and code generation (step 5) are executed on the server side. All graph transformation steps (2, 3 and 4) are executed in the browser by an interpreter written in JavaScript/GWT.

This result paves the way for online collaboration between global teams of software customers, IT business analysts and software developers.

Contents

<i>An introduction to Storyboarding</i>	11	<i>Instant Examples</i>	131
<i>Foundations of instant storyboarding</i>	19	<i>Conclusion</i>	149
<i>Open Questions</i>	29	<i>Outlook</i>	153
<i>Instant Storyboarding</i>	31	<i>Bibliography</i>	159
<i>Web based storyboarding</i>	38	<i>List of Figures</i>	167
<i>Instant grammatical relations</i>	49	<i>List of Listings</i>	172
<i>Instant graph visualization</i>	61	<i>List of Tables</i>	176
<i>Instant informal story patterns</i>	74		
<i>Instant formal story patterns</i>	98		
<i>Instant storyboards</i>	110		
<i>Instant acceptance tests</i>	124		

An introduction to Storyboarding

In this thesis on *Instant storyboarding*, we will discuss *immediate visual feedback on ontology learning for acceptance tests with graph transformations in web applications*. Reading the title we can identify two main topics of this thesis:

Storyboarding is a film industry standard that visualizes the movie script scene by scene. The Fujaba community has developed a process that expresses requirements in a similar graphical way. In the context of software engineering storyboarding formalizes requirements by adding story patterns for start, end and intermediate situations, helping developers to spot potential problems before they occur.

Instant storyboarding has been the main goal of my research work. Current tool support not only requires interested users to download, install, run, configure and navigate through the UI of the Fujaba tool suite, but is also limited to managing storyboarding artifacts. In comparison to pen and paper, they are inaccessible and lack recommendation capabilities for the core aspect of the process: deriving a story pattern from a textual scenario.

To achieve this goal, I created a prototype that demonstrates the advantages of instant storyboarding. As already mentioned in the subtitle of this thesis I researched solutions to implementation details in five related topics:

Immediate Visual Feedback Spreadsheet software automatically recalculates the depending formulas whenever a cell is changed. With *Instant storyboarding* we are striving to achieve the same level of liveness for storyboarding.

Ontology Learning is the task of learning concepts and relations from natural language in ontology engineering. There, the subtask of instance learning resembles the task of deriving story patterns from textual scenario descriptions described by the storyboarding process.

Behavior Driven Development The storyboarding process of Fujaba produces JUnit tests that cover the requirements defined by the textual scenario descriptions. They capture the desired behavior of a system in the same spirit as Behavior Driven Development as proposed by Dan North.

Graph Transformations Instant storyboarding uses graph transformations to convert grammatical relations into informal story patterns, complement type information and add further implementation details. Each of these steps is an ideal candidate for visualization because of the graph nature.

Browser Applications To maximize the accessibility of *Instant storyboarding*, we implemented the prototype as a web application. Instead of downloading an installer, an executable or even source code trying out *Instant storyboarding* is as easy as pointing your web browser to <http://instant-storyboarding.de>.

In the following sections, I will explain these seven aspects in more detail and give an overview of the structure of this thesis and its contributions.

Storyboarding by Example

“Scenarios have a plot; they include sequences of actions and events, things that actors do, things that happen to them, changes in the setting, and so forth.”

~ Rosson and Carroll¹

Storyboarding is the process of formalizing a textual scenario by deriving a story pattern for each step in the *plot*. Let us examine the process with a simple chess example as in [listing 1](#).

```

1 Start situation:
2 Alice and Bob are playing chess. Alice
3 moves her white pawn from field c2 to c4.
4
5 End situation:
6 Alice's pawn is now on field c4.

```

In order to formalize this scenario Diethelm, Geiger, and Zündorf² recommend the following steps to derive a story pattern:

1. Identify nouns and underline them in the text.
 - (a) Draw an object for every noun.
 - (b) Use the noun as the name of the object.
 - (c) Map noun attributes to the object.
2. Identify verbs and use a dotted line to underline them in the text.
 - (a) Draw a link between the corresponding subject and object of every verb.
 - (b) Use the verb as the link name.

Interpreting these steps word by word would result in a simple object diagram as in [figure 1](#). Links for the ownership of the pawn or the place of it are not yet taken into account.

¹Mary Beth Rosson and John M. Carroll. *Usability Engineering: Scenario-Based Development of Human-Computer Interaction*. San Diego, CA: Academic Press, 2002. ISBN: 1-55860-712-9

Listing 1: This textual scenario for a simple chess situation describes one of the possible opening moves for Alice. We will revisit the scenario several times, so if you are unfamiliar with chess I recommend reading the wikipedia article to get an idea of the game: Wikipedia. *Chess*. 2001. URL: <http://en.wikipedia.org/wiki/Chess> (visited on 03/24/2014).

²Ira Diethelm, Leif Geiger, and Albert Zündorf. “Systematic Story Driven Modeling, a case study”. In: Edinburgh, Scotland, May 24 - 28, 2004. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/DGZ04.pdf>

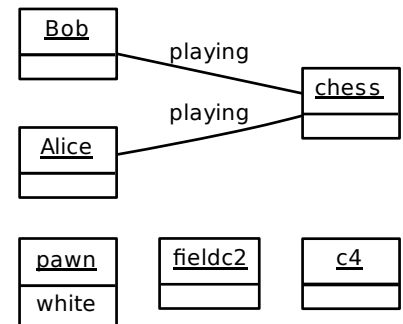


Figure 1: The informal object diagram lacks class names and attribute types. Nevertheless, it captures the actors and relations of the scenario in a visual representation.

While the missing class names make the object diagram technically incorrect it is sufficient for starting a modeling discussion between developers.

In this discussion, developers familiar with object oriented development and the collaboration diagram notation can use their world knowledge to add type information to the scenario model. They may add class names like `Player` or `Human` for *Alice* and *Bob*, use `Counter` as the class for *pawn*, model *white* as a `color` attribute of a counter or a player, and maybe add a `move()` method. These additional modeling decisions would transform the object diagram into a story pattern as shown in [figure 2](#).

Figure 2: Adding the `move()` message to the object diagram turns it into a story pattern, a *verbose* collaboration diagram that allows showing object attributes and messages at the same time. Story patterns also allow visualizing object and link creation and deletion, a visualization we will explore in more depth later.

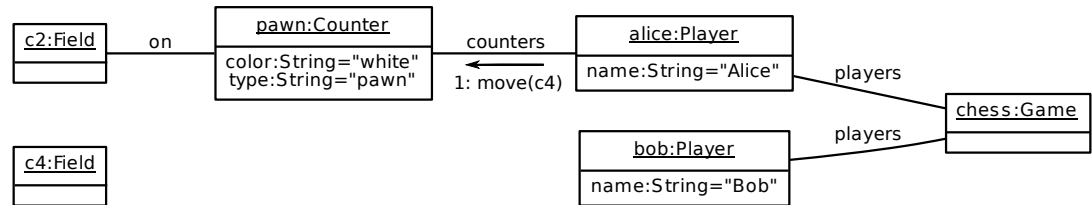
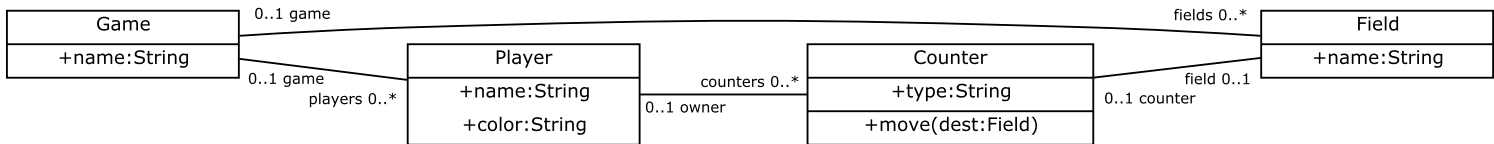


Figure 3: The derived class diagram captures modeling decisions regarding associations, their names and cardinalities. Note the subtle but very fundamental change from a *playing* link to a *players* association: *playing* is a verb, *players* is a plural noun.

After adding type information to objects and attributes the next step to further formalize the scenario is to derive a class diagram, the building plan for the identified objects. They offer a notation for the cardinality of associations between classes which allows developers to express the relationship between classes in more detail than object diagram links. [Figure 3](#) shows the derived class diagram for our example.



After deriving all these diagrams the developer can finally write an acceptance test that uses the formalized storyboard to verify the story patterns for each scenario step. The story pattern for the start scenario describes the object world the test has to set up before executing the method under test. After the execution, the test has to verify that the end situation as described by the corresponding story pattern has been reached. This way of formalizing scenario descriptions requires a special way of describing scenarios, namely the separation into steps and the identification of an event that can be represented as a method call in the story pattern. In other words: a *plot*.

The Fujaba Unified Process steps have been distilled by rigorously using an objects first approach in informatics lectures in school.³ Diethelm was in the rare position to teach a school class in informatics. She used the opportunity to show that starting with objects instead of classes improves the learning experience of students new to object oriented programming. Figure 4 shows a session of the Object Game. A role-playing variant of object oriented algorithms that teaches students the limited viewpoint of objects and the navigation through object structures via links. By continuously improving and evaluating the course over several years Diethelm has proven storyboarding to be a better approach to teaching object oriented software development in comparison to approaches starting with class diagrams.

³Ira Diethelm. "Strictly models and objects first: Unterrichtskonzept und -methodik für objektorientierte Modellierung im Informatikunterricht". <http://d-nb.info/98668760X>. PhD thesis. University of Kassel, 2007, pp. 1–223. ISBN: 978-3-86805-007-3. URL: <http://kobra.bibliothek.uni-kassel.de/bitstream/urn:nbn:de:hebis:34-2007101119340/1/DissIraDruckfassungA5.1.pdf>



Figure 4: *Object game*. Letting students play the role of an object in an object diagram teaches them the meaning of links and how to navigate through an object oriented data model.

⁴ Albert Zündorf. *Story Driven Modeling with Fujaba. Turning Scenarios into Automated Tests*. Google Tech Talks. June 4, 2008. URL: https://www.youtube.com/watch?v=nwcsj_Iz4ao (visited on 03/24/2014).

⁵ Luke Church, Chris Nash, and Alan F. Blackwell. "Liveness in notation use. From music to programming". In: *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group (PPIG 2010)*. 2010, pp. 2–11. URL: http://www.academia.edu/1124877/Liveness_in_Notation_Use_From_Music_to_Programming, p. 2

⁶ Team Kassel. *Fujaba4Eclipse update site*. 2012. URL: <http://www.se.eecs.uni-kassel.de/fileadmin/se/update/> (visited on 07/19/2013).

Going from scenarios to story patterns to class diagrams to acceptance tests makes storyboarding a very straight forward approach.⁴ Not only does it provide an easy learning curve for object oriented modeling, it also gradually evolves requirements specifications in the form of textual scenarios into formalized storyboards. Generating acceptance tests from storyboards creates a test suite that allows testing the requirements similar to a behavior driven development process. The small steps of storyboarding make it an ideal process for this thesis to improve the tool support for the task of deriving acceptance tests from textual scenarios.

Instant storyboarding

"Intuitively, liveness is an assessment of how 'responsive' a system is. When I perform an action, are my changes immediately apparent? Do I have to go through a number of auxiliary steps in order to understand the consequences of my actions? Liveness is a property both of the program notation, and also of its execution environment."

~ Church, Nash and Blackwell⁵

Although the storyboarding process is easy to follow, tool support for it is hard to come by. As of March 2014 the only tool that supports storyboarding is Fujaba. While the team at the University of Kassel provides an eclipse update site⁶ installation still requires downloading and running eclipse prior to even installing the Fujaba4eclipse plugin. Before a user can model storyboards he then needs to activate the XProm Plugin. After all this, Fujaba only manages the storyboards for a project and allows generating JUnit tests for them. It does not provide help for the storyboarding process itself. The user has to model the object diagrams based on the textual scenarios without any tool generated proposals. In essence, storyboarding with Fujaba requires a software engineer familiar with Fujaba and a machine powerful enough to run Eclipse with the Fujaba plugin.

With *instant storyboarding* I created a web application that completely removes these requirements. Accessing the tool is as easy as pointing a browser to <http://instant-storyboarding.de>. For inexperienced users it starts with the same simple chess example that I use in this thesis. Nevertheless, software engineers with in depth knowledge can still tweak

and alter the story patterns that derive a storyboard from textual scenario descriptions to their liking, on the couch, even with a tablet or mobile device.

I choose the term *instant* not only to reflect the accessibility of the web application but also to indicate the high level of liveness I was able to achieve with it. Church, Nash, and Blackwell⁷ gave the following descriptions for Tanimoto's⁸ four levels of *liveness* found in interactive programming environments:

Level 1 a visual representation is used as an aid to software design (Tanimoto was referring to a user document such as a flowchart, not a programming language). This provides a basic level of graphical representation, and can be made continuously visible, although mainly because of the fact that a paper document can be placed beside the screen, rather than on it.

Level 2 the visual representation specifies a program that can be manually executed, possibly after compilation. This provides a basic kind of physical action mapping, in that modification of the representation will eventually change the behavior of the program.

Level 3 the representation responds to an edit with immediate feedback, automatically executing or applying the changes. This allows users to make rapid actions, and often (after noting the system response) an opportunity to quickly reverse an incorrect action.

Level 4 the environment is continually active, showing the results of program execution as changes are made to the program. This provides high visibility of the effect of actions.

The *instant storyboarding* prototype updates the depending diagrams whenever the user presses enter in the textual scenario description or any of the story pattern rules implementing the recommendation algorithm. This provides users with a level 3 liveness experience. While triggering the update on every keystroke would theoretically lead to level 4 liveness, the natural language parsing of the textual scenario descriptions takes too long to leverage the effect.

⁷ Church, Nash, and Blackwell, "Liveness in notation use", pp. 2-3

⁸ Steven L. Tanimoto. "VIVA: A visual language for image processing". In: *Journal of Visual Languages & Computing* 1.2 (1990), pp. 127-139. ISSN: 1045-926X. DOI: 10.1016/S1045-926X(05)80012-6. URL: <http://www.sciencedirect.com/science/article/pii/S1045926X05800126>

Foundations of instant storyboarding

Implementing the *instant storyboarding* prototype would not have been possible without combining knowledge from several topics of interest. Not all of them are classical fields of research, but they show where I drew my inspiration from.

Immediate Visual Feedback

“Here’s something I’ve come to believe: creators need an immediate connection to what they are creating. That’s my principle. Creators need an immediate connection to what they create. And what I mean by that is, when you are making something, if you make a change or you make a decision, you need to see the effect of that immediately. There can’t be any delay and there can’t be anything hidden.”

~ Bret Victor, *Inventing on principle*⁹

Immediate visual feedback produces graphical representations that reflect the state or process of a system as soon as possible, ideally in near real time. For this thesis we will use a definition by Owen:¹⁰

visualize the formation of mental visual images, the act or process of interpreting in visual terms or of putting into visual form

⁹ Bret Victor. *Inventing on Principle*. 2012. URL: <http://vimeo.com/36579366> (visited on 06/30/2013).

¹⁰ G. Scott Owen. *Definitions and Rationale for Visualization*. Feb. 11, 1999. URL: <http://www.siggraph.org/education/materials/HyperVis/visgoals/visgoal2.htm> (visited on 06/30/2013).

¹¹ Leif Geiger. “Fehlersuche im Modell: modellbasiertes Testen und Debuggen.” PhD thesis. University of Kassel, 2011. URL: <http://d-nb.info/101373873X>.

¹² source: Geiger, “Fehlersuche im Modell: modellbasiertes Testen und Debuggen.”, p. 112.

¹³ Bret Victor. *Up and Down the Ladder of Abstraction*. Oct. 2011. URL: <http://www.ryrdream.com/LadderOfAbstraction/> (visited on 06/30/2013).

¹⁴ Chris Granger. *Light Table - a new IDE concept*. Blog. Apr. 12, 2012. URL: <http://www.chris-granger.com/2012/04/12/light-table---a-new-ide-concept/> (visited on 03/24/2014).

¹⁵ Chris Granger. *Light Table*. kickstarter. June 1, 2012. URL: <http://www.kickstarter.com/projects/306316578/light-table> (visited on 03/24/2014).

¹⁶ Chris Granger. *Light Table*. Home page. 2013. URL: <http://www.lighttable.com/> (visited on 03/24/2014).

In the case of software development this would match any tool that visualizes the current state of an application. One example of this is eDOBS,¹¹ a graphical version of a debugger that visualizes the current state of an application as a UML object diagram. Figure 5 shows a screenshot¹² of eDOBS visualizing a program state as a UML object diagram. It allows software developers to visualize the state of a Java program at execution time.

Bret Victor has already envisioned this kind of interactivity for development environments before giving the talk *Inventing on principle*. An early example running in the browser can be found at the top of his Article “Up and down the Ladder of Abstraction”.¹³ It shows a car that can be navigated around with the cursor keys. Jumping it up the ladder allows to play with various aspects of the steering behavior. Each chapter introduces and examines a new steering variable and interactivity allows changing the parameters of various examples by hovering over sliders. The new result is immediately visible to the user. In his talk he uses even more than one interactive example to show how immediate visual feedback makes developing algorithms easier.

His vision of a more interactive development environment has inspired so many developers, that it might become a reality. Granger¹⁴ has taken the ideas of Bret Victor and created the prototype for a new IDE concept called “Light Table”, raising USD 316,720 on kickstarter.¹⁵ Meanwhile the project has released an alpha version for the three major desktop platforms.¹⁶

In this thesis I will describe a prototype that encourages experimentation with storyboard-ing and graph transformations in the same interactive manner. Driven by the goal to give users instant feedback on storyboarding I improved the accessibility to a storyboarding tool by implementing it as a browser application. Whenever the user changes textual descriptions, structurization rules or formalization rules the prototype updates the corresponding diagrams. Combining storyboarding with this kind of immediate visual feedback is the main goal of this thesis. To be successful, several supporting research topics are needed.

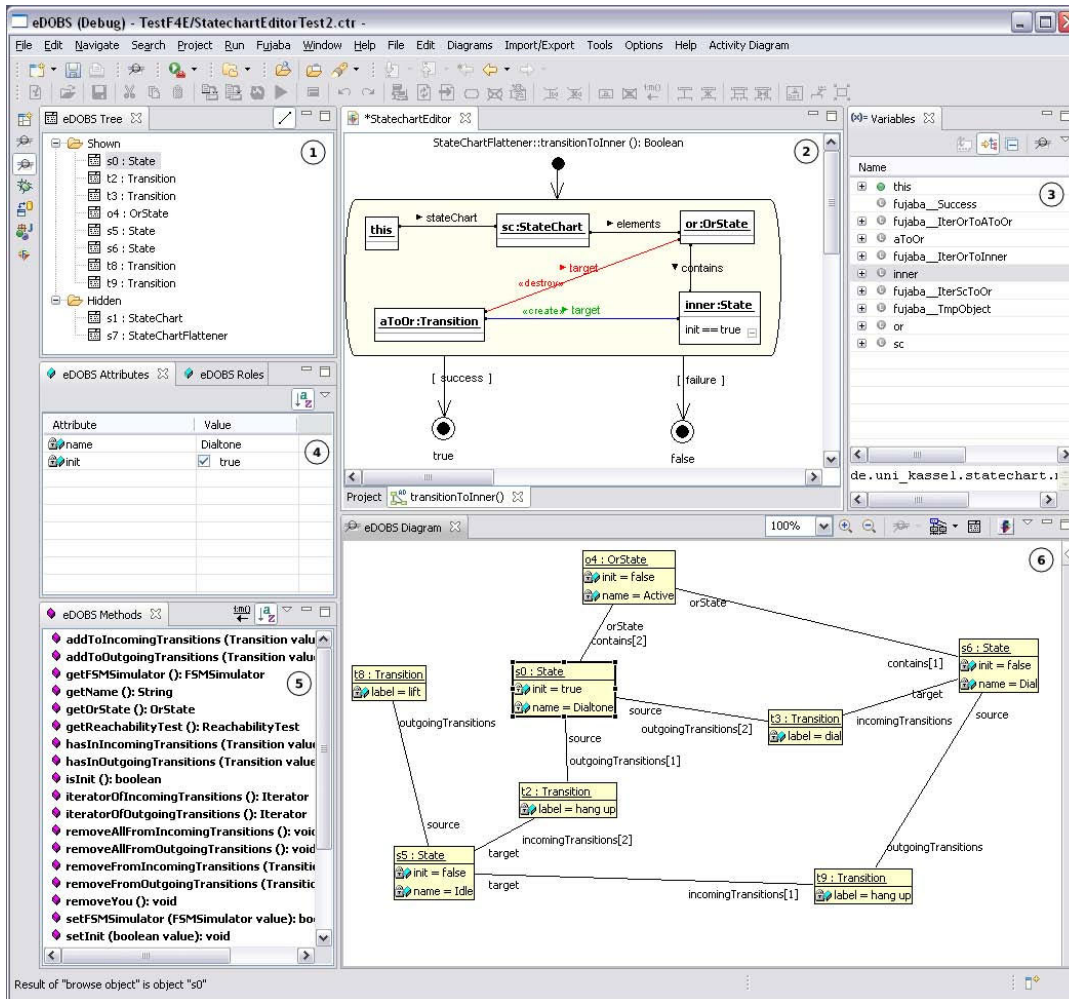


Figure 5: eDOBS, the eclipse Document Object Browsing System. The object diagram in view number 6 visualizes the application state during the execution of the story pattern in view number 2.

Ontology Learning

¹⁷ Philipp Cimiano. *Ontology learning and population from text - algorithms, evaluation and applications*. Springer, 2006, pp. I–XXVIII, 1–347. ISBN: 978-0-387-30632-2, p. 23

¹⁸ Philipp Cimiano and Johanna Völker. “Text2Onto - A Framework for Ontology Learning and Data-driven Change Discovery”. In: *Proceedings of the 10th International Conference on Applications of Natural Language to Information Systems (NLDB)*. ed. by Andres Montoyo, Rafael Munoz, and Elisabeth Metais. Vol. 3513. Lecture Notes in Computer Science. Alicante, Spain: Springer, June 2005, pp. 227–238. URL: http://www.aifb.uni-karlsruhe.de/WBS/jvo/publications/Text2Onto_nldb_2005.pdf

“The process of defining and instantiating a knowledge base is referred to as *knowledge markup* or *ontology population*, whereas (semi-)automatic support in ontology development is usually referred to as *ontology learning*.”
 ~ Philipp Cimiano, *Ontology Learning from Text: An Overview*¹⁷

Identifying terms and synonyms is the foundation for learning concepts and relations. Cimiano has depicted this hierarchy of ontology learning tasks in his “layer cake” shown in figure 6. When working with unstructured text the initial terms are identified with parsers from natural language processing. Traditional ontology learning parses large corpora of texts from the same domain and uses statistical methods and means to identify concepts and relations. Depending on the tool manual changes and multiple iterations may be used to refine the ontology.¹⁸ The resulting ontology describes the domain and can be used for reasoning about entities within that domain.

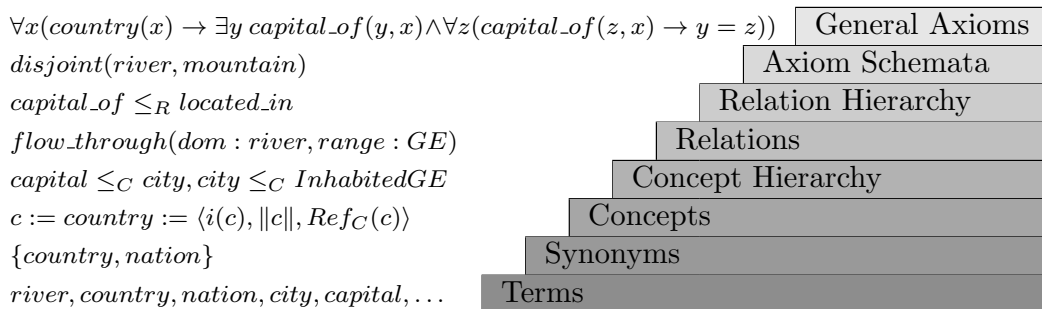


Figure 6: The hierarchical “layer cake” by Cimiano (2006) gives a summary of the different tasks in ontology learning.

A subtask of ontology learning is the population of the ontology with concept instances, called *instance learning*. When an ontology on chess contains the concept *queen* an instance would be the queen of the white player or the queen of the black player. From an ontology

learning perspective storyboarding starts with instance learning, as it starts with the identification of objects and links between them.

Similar to ontology learning, the first step in instant storyboarding is parsing natural language sentences and extracting grammatical relations from them. These grammatical relations make up the start graph for the graph transformations that will drive the rest of the storyboarding process. As graph transformations can be used to match patterns in a graph our storyboarding approach can be seen as a graphical version of the common pattern specification language¹⁹ (CPSL) used in information extraction tools like TextPro²⁰ and JAPE.²¹

Instead of learning an ontology by parsing longer text corpora the storyboarding process is limited to fewer examples of text. In fact, the parsed texts are pre-structured by separating them into individual steps, each describing a single scenario situation. With regard to immediate feedback, the developer wants to see the resulting story pattern for each step as soon as possible. Learning and visualizing as many concepts and relations as possible from each scenario step is the main goal of the *instant storyboarding* prototype before generating an acceptance test.

Behavior Driven Development

“The deeper I got into TDD, the more I felt that my own journey had been less of a wax-on, wax-off process of gradual mastery than a series of blind alleys. I remember thinking ‘If only someone had told me that!’ far more often than I thought ‘Wow, a door has opened.’ I decided it must be possible to present TDD in a way that gets straight to the good stuff and avoids all the pitfalls.”
 ~ Dan North, *Introducing BDD*²²

The original idea of Behavior Driven Development (BDD) was first introduced by Dan North in “Behavior Modification”²³, a response to Test Driven Development as proposed by Kent Beck.²⁴ Instead of writing unit tests for classes BDD shifts the focus towards writing textual scenarios that capture the desired behavior of a system. In our chess example that could be scenarios for the movement of the different chess pieces, eg. for the pawn always moving

¹⁹ Douglas E. Appelt and Boyan Onyshkevych. “The common pattern specification language”. In: *Proceedings of a workshop on held at Baltimore, Maryland: October 13-15, 1998*. TIPSTER ’98. Baltimore, Maryland: Association for Computational Linguistics, 1998, pp. 23–30. doi: 10.3115/1119089.1119095. URL: <http://acl.ldc.upenn.edu/X/X98/X98-1004.pdf>

²⁰ Douglas E. Appelt. *TextPro*. Oct. 10, 1999. URL: <http://www.ai.sri.com/~appelt/TextPro/> (visited on 07/19/2013).

²¹ H. Cunningham, D. Maynard, and V. Tablan. *JAPE: a Java Annotation Patterns Engine (Second Edition)*. Research Memorandum CS-00-10. Department of Computer Science, University of Sheffield, Nov. 2000. URL: <http://www.dcs.shef.ac.uk/~diana/Papers/jape.ps>.

²² Dan North. *Introducing BDD*. Mar. 2006. URL: <http://dannorth.net/introducing-bdd/> (visited on 07/19/2013). Repr. of “Behavior Modification”. In: *Better Software Magazine* (Mar. 2006). URL: http://www.stickyminds.com/s.asp?F=S10836_MAGAZINE

²³ Dan North. “Behavior Modification”. In: *Better Software Magazine* (Mar. 2006). URL: http://www.stickyminds.com/s.asp?F=S10836_MAGAZINE_2

²⁴ Kent Beck. *Extreme Programming Explained: Embrace Change*. First. Boston: Addison-Wesley Professional, 1999, p. 224. ISBN: 0201616416.

one or two fields forward in one move. BDD then formalizes these textual scenarios to derive an executable acceptance test suite, similar to the TDD test suite.

To make parsing textual scenarios easier they are expressed in a restricted language called Gherkin. A scenario for a pawns opening move could be written as in [listing 2](#).

Listing 2: The Gherkin scenario syntax consists of a scenario name followed by a sequence of Given-When-Then steps.

```

1 Scenario: opening move
2   Given Alice and Bob are playing chess
3   When Alice moves her pawn from field c2 to c4
4   Then Alice's counter should be at field c4.
```

Each Given-When-Then step is then mapped to executable code. Popular implementations like JBehave²⁵ or Cucumber²⁶ use regular expressions to resolve the method that implements a specific step. An exception or failure for any of the steps also marks the containing scenario as failed. This acceptance test suite should be executed like a traditional TDD unit test suite to allow developers to be confident that their changes do not break existing functionality.

On a second look at the Given-When-Then syntax you might recognize the core *steps* of a storyboard: start scenario, event and end scenario. While BDD implementations use regular expressions to find hand written code that implements the steps, storyboarding formalizes the textual scenario steps by modeling them as a series of story patterns and then uses Fubaja's code generation capabilities to obtain the acceptance test suite. In both approaches a developer formalizes a textual scenario or rather the inherent grammatical relations into a machine readable representation that can be used to derive an executable acceptance test.

A software developer has his world knowledge and experience to help him with the formalization. Our *instant storyboarding* prototype also needs a way to rewrite grammatical relations into a story pattern. The textual description of a storyboard is already split into at least a start step description and an end step description. While natural language processing can give us the grammatical relations for them we still need to model a story pattern for the start scenario as well as an end scenario to receive a complete storyboard. To close the gap between grammatical relations and a story pattern *instant storyboarding* uses graph transformations.

²⁵ Dan North. *What is JBehave?* 2003. URL: <http://jbehave.org/> (visited on 03/23/2014).

²⁶ Aslak Helleøy. *Cucumber - Making BDD fun.* 2008. URL: <http://cukes.info/> (visited on 03/23/2014).

Graph Transformations

Graph Transformations describe the way of rewriting one graph into another. The approach described by Ehrig, Pfender, and Schneider in 1973 as a generalization of textual grammars²⁷ was first applied to specialization and evolution in biology.²⁸

Andy Schürr and Albert Zündorf, at the time Ph.D. students of Manfred Nagl, first worked on PROGRES, an integrated environment and very high level language for *PRO*grammed Graph *RE*writing Systems and later mentored three master theses that led to Fujaba. Thorsten Fischer, Lars Torunski and Jörg Niere created Fujaba (*From UML to Java And Back Again*), a very powerful development environment that allows modeling structure and behavior of an application in story diagrams and UML. A code generation mechanism allows the generation of JUnit tests and Java code that implements the graphically modeled application. The story diagrams used to model behavior in Fujaba are based on a short hand form of the original PROGRES graph grammar rules.

The graphical representation of story diagrams is a mix of UML activity diagrams with UML 1.x collaboration diagrams / UML 2.x communication diagrams extended with a special notation for modifying elements. To illustrate the idea of how behavior is modeled in story diagrams [figure 7](#) shows a story pattern that moves a chess piece from its current field to a new field without checking any limitations. It could be part of the default implementation of the move method for pieces. Black elements need to be matched before red elements are deleted and green elements are created. Dashed elements represent optional matches. Objects without a class are already bound: `this` is the Piece whose move method has been called and `newpos` is the Field passed as an argument to `move(newpos:Field)`. So the diagram translates to: *if this piece is already on a field then delete the on link to oldpos. After that Create an on link between this piece and the field newpos.* In effect moving the piece to a new field.

The code for the behavior is generated with a template engine. It takes care of isomorphism checks, null checks and adds association implementations necessary for the graph transformation based behavior implementation. By generating the code with these extensions the use

²⁷ Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. "Graph Grammars: An Algebraic Approach". In: *SWAT (FOCS)*. IEEE Computer Society, 1973, pp. 167–180. URL: <http://dblp.uni-trier.de/db/conf/focs/focs73.html#EhrigPS73>.

²⁸ Hartmut Ehrig and Karl Wilhelm Tischer. "Graph Grammars and Applications to Specialization and Evolution in Biology." In: *J. Comput. Syst. Sci.* 11.2 (1975), pp. 212–236. URL: <http://dblp.uni-trier.de/db/journals/jcss/jcss11.html#EhrigT75>.

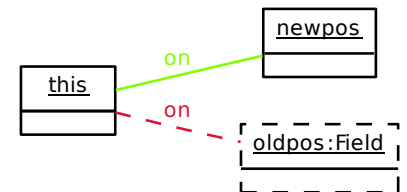


Figure 7: Story diagram for `piece.move(newpos:Field)`

of reflection becomes unnecessary which shows in the performance of an application generated with Fujaba.

Similar to the way behavior is specified with story diagrams, Fujaba also allows to specify functional tests with storyboards. A storyboard consists of at least two steps: a start scenario and an end scenario. Each step is formalized by a story diagram that with the exception of the end scenario must contain a collaboration statement. The acceptance test is then derived by creating the object world shown in the start scenario and then calling the method from the collaboration statement. To check if the implementation of the called method is correct, the existence of the object world described by the succeeding step is checked. If the story diagram cannot be matched the test fails. Otherwise the next collaboration statement is executed and the object world matching is repeated until the end step has been reached. Using the XProM Plugin Fujaba is capable of generating executable JUnit tests from the story board.²⁹

²⁹ Leif Geiger and Albert Zündorf. "Developing Tools with Fujaba XProM.". In: *GTTSE*. ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 4143. Lecture Notes in Computer Science. Springer, 2006, pp. 344–356. ISBN: 3-540-45778-X. URL: <http://dblp.uni-trier.de/db/conf/gttse/gttse2006.html#GeigerZ06>.

While Fujaba can generate code that can be cross compiled to JavaScript via GWT, the story diagrams are used to create a static version of the graph transformation implementation. Whenever the intended behavior is subject to change by altered requirements, the code has to be generated again. For a browser application that allows dynamically altering the graph transformations a story pattern interpreter is more suitable.

Browser Applications

In this thesis Browser Applications are complex client server applications that execute the application logic on the client side with JavaScript in a browser. A good example for this are the Google Drive Apps³⁰ which implement a word processor, spreadsheet calculator and more as a browser application. With gliffy³¹ there is an online diagram software that runs in any browser that supports the Adobe flash plugin. In contrast to desktop applications an installation is no longer required and users can instantly try out the application by pointing their browser to an URL.

³⁰ Google. *Google Drive Apps*. 2006. URL: <http://www.google.com/drive/apps.html> (visited on 07/19/2013).

³¹ Gliffy. *Online Diagram Software and Flowchart Software*. 2005. URL: <http://www.gliffy.com/> (visited on 07/19/2013).

Executing a lot of business logic on the client side other than Flash games has become more popular with the development of JavaScript frameworks that allow cross compiling of

Java to JavaScript. Although a lot of JavaScript frameworks like jQuery exist, their primary purpose is to ease DOM manipulation and navigation. This helps a lot with the development of AJAX in web pages but still feels like building web pages. With GWT Google has provided a tool that allows to implement client logic in Java and hide the dynamically typed nature of JavaScript which prevents a whole class of bugs. Using Java as the development language allows software engineers to reuse well known design pattern like Model View Presenter, test frameworks like JUnit and even classic IDEs like Eclipse to develop browser applications. As a result the development of more complex browser applications has become possible.

Another benefit of web applications is the more direct contact to users. Instead of tracking user interactions and receiving them in chunked uploads from a desktop application, a browser application allows us to monitor the users action in real-time. Most web server logs already provide a lot of information on the user, that can be extended by checking the browser environment with tools like Google Analytics³² or piwik.³³ We can automatically collect errors on the client and send them to the server, along with the state of the application. Basically, feedback reaches us faster because the user is already online.

As accessibility is an integral part to experimentation I decided to create a browser application instead of a desktop application. Being able to point your browser to <http://instant-storyboarding.de> and start toying around with the system is much easier than downloading and running an application or maybe even finding the correct repositories, checking out source code and trying to compile and run the beast.³⁴ We cannot scare users off with this kind of installation routine when talking about *instant* storyboarding.

³² Google. *Web Analytics & Reporting – Google Analytics*. 2005. URL: <http://www.google.com/analytics/> (visited on 03/23/2014).

³³ Piwik. *Free Web Analytics Software*. 2007. URL: <http://piwik.org/> (visited on 03/23/2014).

³⁴ Paul Graham. “Hackers & Painters: Big Ideas from the Computer Age”. In: O’Reilly Media, Inc., 2004. Chap. The Other Road Ahead, pp. 56–86. URL: <http://www.paulgraham.com/road.html> (visited on 03/14/2014).

Open Questions

Giving instant feedback on storyboarding will encourage developers to experiment with various parameters of the process and as a result improve the quality of textual scenario descriptions, derived storyboards and consequently acceptance tests. This hypothesis leads to the following research questions we are going to examine in this thesis:

1. *How can we give instant feedback on the storyboarding process?* Which steps should we visualize to lure the user into experimenting with storyboarding? How do we prevent information overload?
2. *How can we give instant feedback on the learning of instances, concepts and relations.* When do we visualize instances? When do we visualize concepts? How do we represent them in UML?
3. *How can we give instant feedback on the graph transformations?* How do we visualize the interpretation of our story pattern? How can the user get feedback when making changes to the story pattern?
4. *How can we provide instant acceptance tests for the visible storyboard?* Do we provide a Fujaba `.ctr`, an Eclipse project, a `.zip` or plain Java? Do we generate the code on the client or server side?
5. *Can this be achieved under the constraints of a browser environment?* Can we achieve the liveness required to engage the user without the interactive capabilities of desktop applications like Fujaba itself?

To answer these questions the rest of this text will describe the overall solution and the implementation decisions along the storyboarding process. Finally, when discussing the approach I will give an answer to these five questions.

Instant Storyboarding

Minimizing the feedback loop

**“In science if you know what you are doing you should not be doing it.
In engineering if you do not know what you are doing you should not be doing it.
Of course, you seldom, if ever, see the pure state.”**

~ Richard Hamming, The Art of Doing Science and Engineering, 1997

Tool support for storyboarding is rare. In fact, Fujaba is the only development environment that implements the process. Unfortunately, the research platform has stopped making regular releases and requires interested researchers to build it from source to follow latest research developments. The eclipse plugin update site maintained by the Software Engineering Research Group Kassel³⁵ provides a more stable version and still requires an Eclipse installation prior to starting the installation of Fujaba. This lack of a ready to run application already prevents most users from trying out and giving feedback on storyboarding.

The main motivation of this thesis is to simplify access to storyboarding. On the one hand, a browser application could be used in lectures without distracting students with any kind of installation procedure. On the other hand, papers on storyboarding could benefit by allowing reviewers to literally verify claims in their browser. We need to lower the barrier of entry for experimentation with storyboarding.

³⁵ Software Engineering Research Group Kassel, ed. *Fujaba4Eclipse Update Site*. 2013. URL: <http://www.se.eecs.uni-kassel.de/fileadmin/se/update> (visited on 01/04/2014).

Reexamining the storyboarding process

“An ontology is a fairly complex structure and it is often more practical to focus on the evaluation of different levels of the ontology separately rather than trying to directly evaluate the ontology as a whole.”

~ Brank et al., 2005

³⁶ Russell J. Abbott. “Program design by Informal English Descriptions.” In: *Commun.* ACM 26.11 (1983), pp. 882–894. URL: http://sunset.usc.edu/classes/cs577a_2003/coursenotes/ep/Program%20Design%20by%20Informal%20English%20Descriptions,%20Russell%20Abbott.pdf

³⁷ Ira Diethelm, Leif Geiger, and Albert Zündorf. “Systematic Story Driven Modeling”. In: *Technical Report* (Feb. 2004). URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/SDM04.pdf>.

The noun-verb analysis implicitly used in the storyboarding process I presented in “[Storyboarding by Example](#)” on page 13, has first been described by Abbott³⁶ in 1983. In the Fujaba community Diethelm, Geiger, and Zündorf³⁷ evolved the process to storyboarding and described it for human software developers. Developing an algorithm needs a more formal analysis of the process. During my research we could observe our programming methodology students to draw simple story patterns on paper, as suggested and demonstrated by the tutor, when trying to understand textual scenarios and specific situations in the object world of their application. We noticed that they initially left out nearly all technical details like type information or attributes and focused on the structure, discussing objects and relations. If something became unclear the technical details would be added on the fly.

In the design pattern lectures the story patterns became popular again, when our students were trying to understand the way a design pattern behaves at run time. While the types and attributes became nearly irrelevant in their drawings, they still helped them discuss and reproduce design pattern implementations. The observation here is that having a visual language for object oriented problems eases communication when the amount of information shown can be limited to the needed focus.

This sparked the idea to distinguish *informal story patterns* from *formal story patterns*. The informal patterns are basically a free form diagram consisting of lines, boxes and text. They can be used to incrementally add details to the diagram without requiring a strict notation. By adding technical details like type information, link names and sanitizing strings and numbers an informal diagram can become a formal story pattern. As soon as the amount of technical information can be used to generate code from it, a formal story pattern is reached. During storyboarding the informal story pattern slowly evolves into a formal story pattern as dis-

cussion moves from conceptual relations to more technical details like attributes and type information of individual objects.

Formalizing story patterns is closely related to deriving a class diagram for a story pattern. While Diethelm, Geiger, and Zündorf have considered this task as “usually very straight forward”³⁸ we will add it as a distinct task our new algorithm has to take into account. The “straight forward” may be true for seasoned software engineers, but as with everything done for the first time, there is a learning curve. On the one hand, discussing patterns and solutions with others, possibly guided by more experienced developers improves the results of this modeling process. On the other hand, discussing application behavior with the end user clarifies the requirements and heavily influences the modeling process. In this process of adding world knowledge the modeling decisions for the class diagram and storyboard are changed until they represent the requirement described by the textual scenario.

The final step of the storyboarding process is the generation of an acceptance test from a sound storyboard with formal story patterns for the individual steps, all adhering to the same class diagram. With a storyboard in place we can reuse the existing XProM Fujaba plugin to generate the code for executable JUnit tests as described in “[Instant acceptance tests](#)” on page 124. This last step marks the goal of providing executable JUnit tests for textual scenario descriptions as it completes the storyboarding process to meet the users expectations.

³⁸ Ira Diethelm, Leif Geiger, and Albert Zündorf. “Systematic Story Driven Modeling, a case study”. In: Edinburgh, Scotland, May 24 - 28, 2004. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/DGZ04.pdf>, p. 3.

These observations in our lectures lead to six individual tasks in instant storyboarding:

- Text analysis and extraction of at least: nouns, verbs and attributes.
- Modeling a graph of *objects*, *links* and *collaboration statements* that represents the given textual scenario description³⁹.
- Formalizing this informal story pattern by adding further attributes and type information.
- While formalizing the story pattern keep the classes consistent with other story patterns, in effect creating a class diagram in the background.

³⁹ With *object* and *link* we are using UML terminology. An ontology engineer would be more familiar with *concept instance* and *relation*.

- Derive a sound storyboard that contains all the necessary information to convert it to a Fujaba compatible format.
- Generate code for an acceptance test and provide a way to download the result from the server.

To give immediate feedback to the user, the intermediate results of these tasks and the algorithm that converted one result into the next need to be visualized.

This kind of transparency will increase the trust in the storyboarding approach. The faster the user receives feedback on his changes the easier it will be for him to iterate through changes in the textual description, trying to get better parser results or adding more context to improve the derived formal story patterns. So, how can we address the individual steps one at a time and engage users in experimenting with storyboarding?

The Masterplan

To give early feedback on textual scenarios we will examine how to automate the individual storyboarding steps and provide an immediate visualization of their intermediate results as well as any graph transformations in between. Starting with grammatical relations, continuing with informal and formal story patterns and ending with storyboards each intermediate result can be visualized as a graph. Story patterns already are a mix of object and collaboration diagrams which map very well to attributed nodes and labeled edges. As we will interpret graph transformations to modify these diagrams we will also visualize the transformation process to let users explore the execution of structurization and formalization rules. Well established graph rendering tools like Graphviz allows us to render the different diagram types for results and interpretation steps while keeping the development effort low.

Separating simple story activities from technical story activities will address the two different tasks of modeling a graph that represents the textual scenario and formalizing the identified objects and links. The informal story patterns will be created by applying graph transformations that restructure the grammatical relations into an informal story pattern. The set

of graph transformation rules for this first step can of course be changed by the user. Similar to the structurization rules of the first step a second set of rules is used to formalize the informal story pattern by adding technical details. The rules will be generated by a recommendation framework and can be extended with user defined rules. By reusing the same graph transformation interpreter to evolve the diagrams in each step we also reuse the same kind of visualization. Using a single graph transformation notation for structurization and formalization rules keeps the way the user interacts with the application consistent across all transformation steps.

The predecessor for the implementation of *Instant Storyboarding* was a prototype called “natural text to object diagram” created during my research at the University of Kassel.⁴⁰ It was still using Fujaba generated code to transform the Stanford Parser results into a formal object diagram. While the application was a successful demonstration for the use of graph transformations on parser results we had to rewrite large parts of it to adapt it to the workflow required for instant storyboarding.

Figure 8 gives an overview of the individual components that make up the instant storyboarding masterplan:

Web based storyboarding The interface to the user is a browser application showing a storyboard. Each scenario step starts with the textual scenario description provided by the user. Our application then automatically tries to derive an acceptance test by applying the graph structurization rules and graph formalization rules. To improve the result the user can examine visualizations of the intermediate graphs and the interpretation of graph transformations itself. We will examine the web application in more detail in “Web based storyboarding” on page 38.

Instant grammatical relations Linguists have already written several natural language parsers that identify parts of speech and grammatical relations. We will present the Stanford Parser and the grammatical relations it produces and how we wrap the parsing step as a web service in “Instant grammatical relations” on page 49.

⁴⁰ Jörn Dreyer et al. “NT2OD Online - Bringing Natural Text 2 Object Diagram to the web”. In: *ODiSE'10: Ontology-Driven Software Engineering Proceedings*. Ed. by Sergio de Cesare. Reno/Tahoe, Nevada, USA, Oct. 18, 2010. URL: http://dl.acm.org/ft_gateway.cfm?id=1937133&type=pdf.

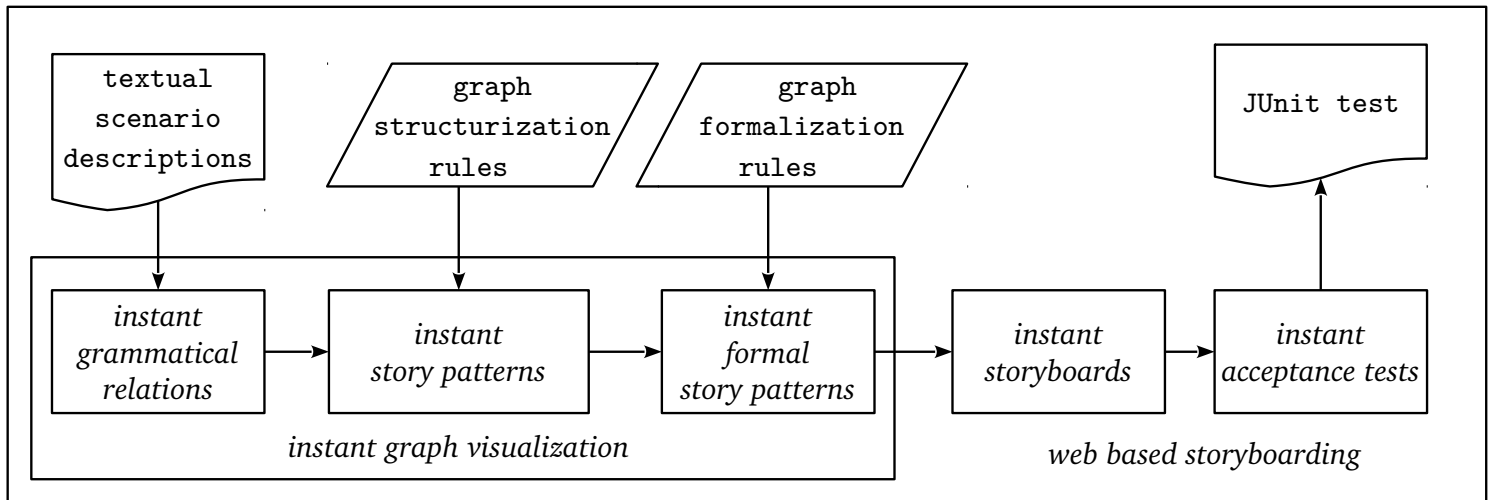


Figure 8: In the masterplan we can see that the main input for *instant storyboarding* are textual scenario descriptions. The bottom row shows the sequence of steps that are executed to generate the final JUnit tests. The process is implemented in customizable graph transformation rules that are interpreted by the web application whenever the user confirms an update by pressing enter in any of the text areas.

Instant graph visualization The parser result is the first graph we can visualize for the user. We will introduce the web service we implemented to render this and all the other graph variants that will be produced in the storyboarding process in “[Instant graph visualization](#)” on page 61.

Instant informal story pattern Taking the parser result as the input graph we can use the `graph_structurization` rules to transform it into an informal story pattern. The interpreter and an extendable ruleset for the grammatical relations of the Stanford parser will be presented in “[Instant informal story patterns](#)” on page 74.

Instant formal story pattern In “[Instant formal story patterns](#)” on page 98 we will examine the recommender framework used to formalize the story pattern. The main idea was to reuse graph transformation rules as a representation for the recommendation to enable reusing the graph transformation rule interpreter developed in the “[Instant informal story patterns](#)” step.

Instant storyboards Aggregating the derived story pattern into a sound storyboard is not a distinct step but requires the alignment of type information across all story patterns while they are created. “[Instant storyboards](#)” on page 110 describes how we are maintaining a class diagram in the background and use it as a source for the formalization rules of the recommendation framework.

Instant Acceptance Tests To finish the storyboarding process we will show how we use a headless version of Fujaba to generate acceptance tests for the storyboards in “[Instant acceptance tests](#)” on page 124. Wrapping Fujaba and the XProM plugin as a web service allows us to generate code for a JUnit test and make it available for download in the web browser.

The technical goal of this thesis is to develop a browser application⁴¹ that automates the individual storyboarding steps, makes the process visually transparent and allows the developer to intervene when necessary.

⁴¹ You are welcome to visit <http://instant-storyboarding.de> and try some of the examples now. The GUI is laid out to resemble the automation steps. Seriously, give it a try now!

Web based storyboarding

Introduction

Visualizing the individual steps of the storyboarding process needs a lot of screen space. On the one hand each step produces or changes a graph. On the other hand the developer may want to hide currently uninteresting graph visualizations. While users of web applications are used to scrolling on a web page, developers expect applications to use the whole available screen width. To address this problem we need a flexible and dynamic web application layout that by default hides technical details, visualizes the important steps and allows customizing the amount of visible information.

For our browser application we decided to provide two separate views on storyboards:

- Based on the Fujaba UI, a compact layout for the storyboard steps shows only the textual scenario and the formal story pattern derived by our browser application.
- An extended layout shows the details for the scenario step over the whole screen width, reveals additional text areas for graph transformation rules and also visualizes the intermediate results for parser results and informal story pattern.

With this separation we create a simple way for users to try the storyboarding process without being overwhelmed by the in depth information available to power users.

Graphs can be huge

Depending on the number of nodes and edges, graphs may rapidly outgrow the available screen space. While smaller graphs like in [figure 9](#) fit nicely next to a textual scenario step description, each sentence will add new elements to the graph as shown in [figure 10](#). It is possible to wrap them inside a scrollable area, but then users may miss information changing outside the currently visible area of the graph. In the end the available screen space is limited by the resolution available to the browser.

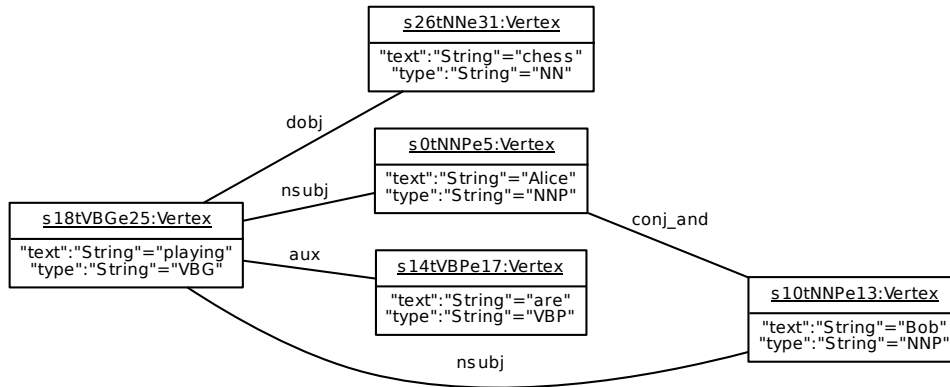


Figure 9: Grammatical relations for the first sentence of the chess example: "Alice and Bob are playing chess."

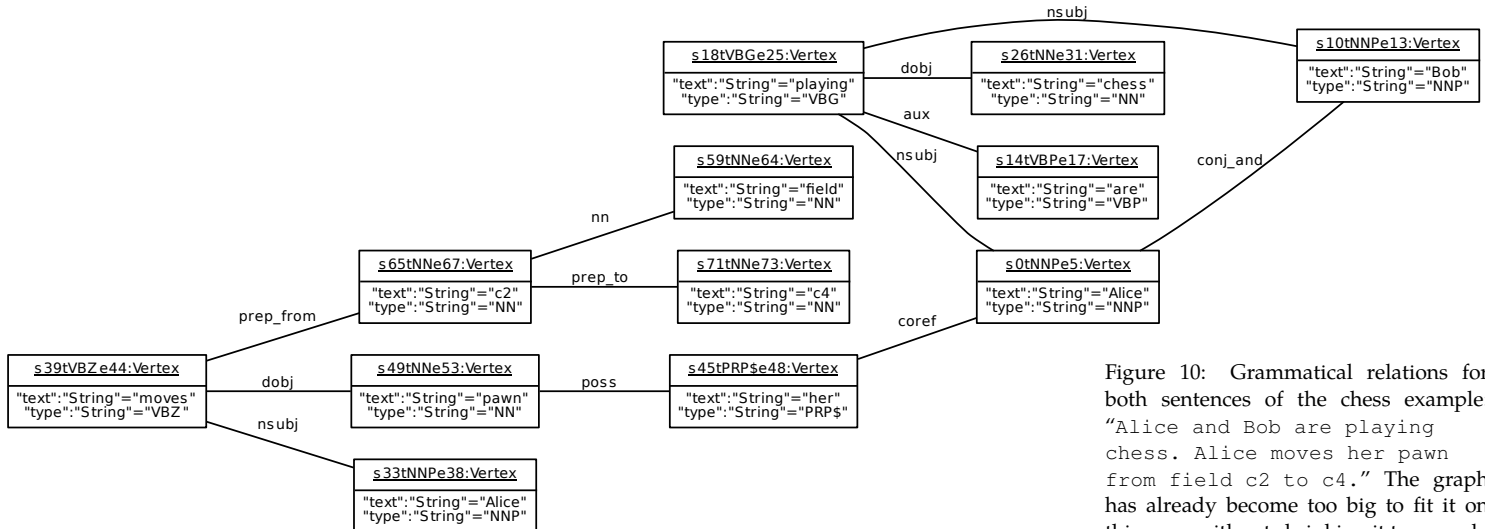


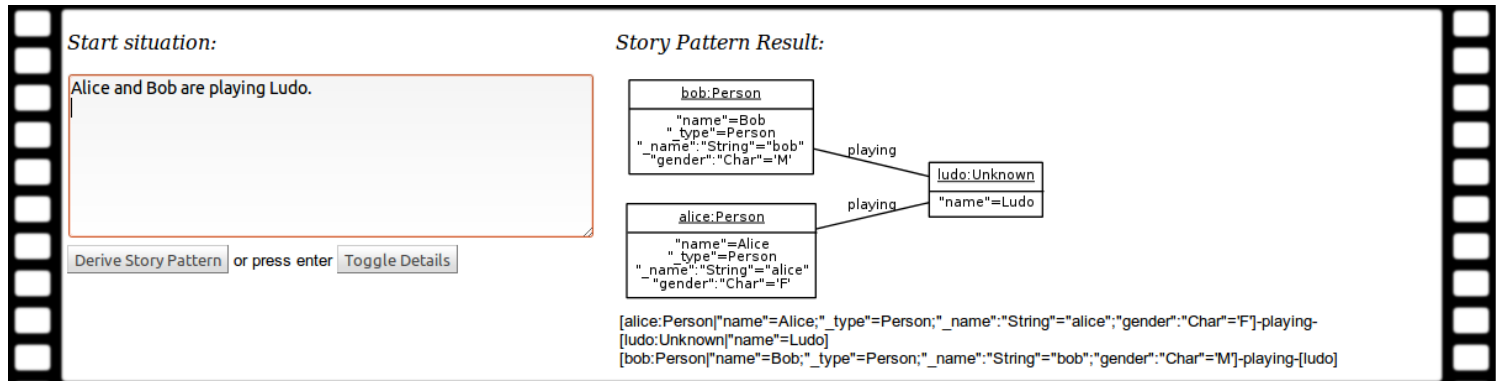
Figure 10: Grammatical relations for both sentences of the chess example: "Alice and Bob are playing chess. Alice moves her pawn from field c2 to c4." The graph has already become too big to fit it on this page without shrinking it to a nearly unreadable size.

Fortunately, all browsers provide scrolling and a zoom function to address that problem. Graph visualization in browsers should therefore layout the complete graph and use the browsers native zooming and scrolling capabilities. Nevertheless, showing the textual scenario description, the transformation rules and the generated graphs for at least one start and end situation all at once would almost always require the user to zoom out. A scenario needs a more intelligent layout that hides the intermediate transformation rules and graphs but allows the developer to explore them when necessary.

From overview to detail

The basic idea is, to first give the user an overview of the scenario, focusing on the flow of the storyboarding process. When first opening our browser application we will present a storyboard consisting of a start and end scenario. As you can see in [figure 11](#) each scenario initially only shows a text area for the input of the textual scenario description. As soon as the user presses enter the application will try to create a formal story pattern and render it next to the textual scenario description. This compact layout is inspired by the Fujaba UI and is intended to be used for the storyboarding process.

Figure 11: Screenshot of the compact scenario step layout



More experienced or curious users might want to examine and alter the details of the algorithm our browser application uses to derive the formal story pattern. Each scenario step can be extended to reveal visualizations of the parser result, an intermediate informal story pattern and the graph transformation rules that implement the storyboarding algorithm. The screenshot in [figure 12](#) shows the layout before any description has been added and parsed. It has been taken at a resolution of 1280x1024 pixels and already required using the zoom function to fit it on screen. After parsing, the size of the graphs would either require the user to zoom out even more to fit everything on screen at once or use the scroll bars to focus on the graph of interest. This extended layout has two purposes: it lets users examine the intermediate results of the storyboarding process on the one hand and provides a way to experiment and debug graph transformation rules for storyboarding on the other hand.

Preparing for liveness

For our purpose we decided to combine the Model View Presenter Pattern and UiBinder mechanism recommended by GWT with a classical Property Change Listener mechanism. At its heart, the core architecture of our implementation is based on the Model View Presenter Pattern. [Figure 13](#) shows the related classes and interfaces for the `StoryActivity`. Starting on the left side of the class diagram, our model – the `StoryActivity` – fires property changes which the presenter implementation `StoryActivityPresenter` listens for. To visualize a change the presenter updates the affected diagrams and text areas through the interface provided by the `StoryActivityView`. The `Presenter` interface defines the actions that can be triggered by the `StoryActivityViewImplementation`. The presenter implementation then takes care of fetching the new user input and updating the model, which in turn might trigger a property change event. As an example, when the users enters a textual scenario description, the `StoryActivityViewImpl` calls `onTextDescriptionChangedEvent()` on its `Presenter`. The `StoryActivityPresenter` fetches the new text and updates the model with it. This triggers a property change event which is again handled by the `StoryActivityPresenter`. It will make the call to the natural language parser and update the `StoryActivity` model with the result. This will fire a new property change for the

parser result which the presenter will handle again. First the result is visualized by updating the DiagramView for parser results then the presenter will apply the graph transformation rules to the parser result and update the model with the transformation result, proceeding in the chain of events until all subsequent steps of the storyboarding process have been processed. The approach allows us to concentrate the implementation logic in the presenters and cleanly separate it from the model and the view.

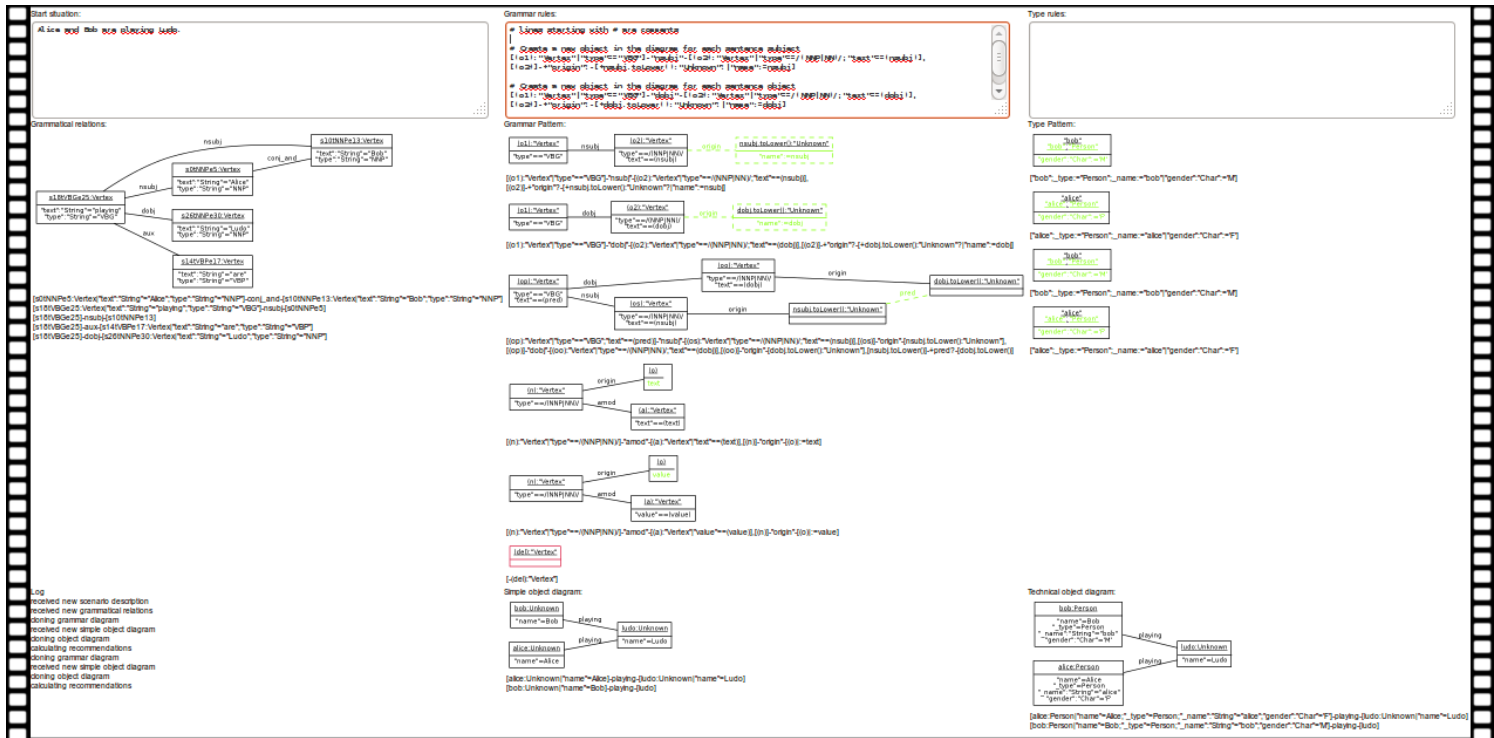


Figure 12: Screenshot of the extended scenario step layout

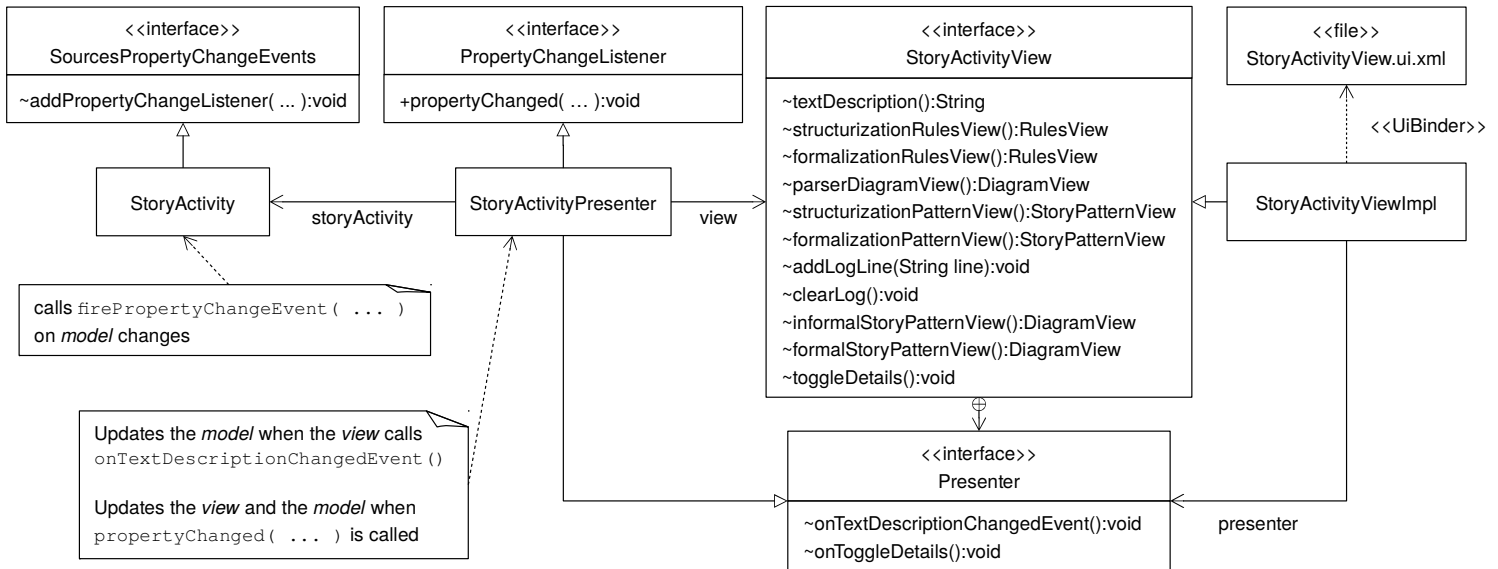


Figure 13: Class Diagram for the core architecture

Using the UiBinder mechanism introduced in Google Web Toolkit 2.0⁴² allows us to declaratively describe UI components of the view and reuse them in several places in our web application. To maximize the screenspace available we only added a very small navigation bar at the top. While the UiBinder components for this overall layout and the static pages used on some of the tabs are unique there are two main components that we reused several times. The more lightweight one is the `DiagramView` which we feed a URL and a description to render a graph. The more complex one is the `ScenarioStepView` which contains all UI elements needed for a scenario step. In addition to several `DiagramViews` it contains up to three text areas for user input and takes care of switching between the extended and the compact layout.

Another benefit of the model view presenter pattern is the independence of a browser when running tests. Usually, when trying to write behavior tests for web applications a tool

⁴² Google. *GWT Developer Guide - UI Binder*. 2009. URL: <http://www.gwtproject.org/doc/latest/DevGuideUiBinder.html> (visited on 07/19/2013).

⁴³ The selenium IDE is a Firefox plugin that can be used to record user interactions with a web page. By adding constraints and checks to the test the developer can create behavior tests that can be executed in a continuous integration environment. Selenium, ed. *Selenium. Web Browser Automation*. 2008. URL: <http://seleniumhq.org> (visited on 03/24/2014)

⁴⁴ see “[Interpreting structurization pattern](#)” on page 80

like selenium⁴³ would be used to emulate user interactions in a browser. With GWT applications it is possible to test the application logic with simple JUnit tests. Without the overhead of starting a browser instance to run tests the test suite can be executed much faster and thus more often. As a result many parts of the application logic for instant storyboarding could be developed against a JUnit test suite without the necessity of a readily available visualization.

Updating the visualization on the fly requires several remote procedure calls either by using the native GWT-RPC mechanism or by using JSON-P. Whenever the user updates the textual scenario or the transformation rules the depending graphs need to be updated. While story patterns can be interpreted in the browser⁴⁴ the text parsing and graph rendering currently rely on the execution of a web service. The results immediately update the visualization and trigger the depending steps. Chaining together the individual AJAX calls like this updates the graphs as fast as possible, which produces the feeling of liveness for the web application.

The extended layout

The easiest solution for laying out the different input areas and graphs was the 3x3 HTML table in [listing 3](#). The first row contains the text areas for user input: the textual scenario description, the graph transformation rules to restructure the parser result into an informal story pattern and the rules to formalize the story pattern. The second row contains a visualization of the parser result graph, a visualization of the structurization rules and a visualization of the formalization rules. In the third row of the extended layout we placed an area containing the application log, a visualization of the informal story pattern and finally the visualization of the formal story pattern. This implementation brought us through most of the development, but hiding cells to create a compact layout required rethinking the HTML markup.

The compact layout

As a web application we need to be able to toggle between complex and compact layout without a page refresh. Converting the table layout into a div based layout allowed us to hide parts of

```
1 <table width="100%" cellspacing="0" cellpadding="0">
2   <tr>
3     <td><!-- Step Description --></td>
4     <td><!-- Structurization Rules --></td>
5     <td><!-- Formalization Rules --></td>
6   </tr>
7   <tr>
8     <td><!-- Natural Language Parser Result --></td>
9     <td><!-- Structurization Pattern --></td>
10    <td><!-- Formalization Pattern --></td>
11  </tr>
12  <tr>
13    <td><!-- Log --></td>
14    <td><!-- Informal Pattern Result --></td>
15    <td><!-- Formal Story Pattern Result --></td>
16  </tr>
17 </table>
```

Listing 3: Table based layout

the table and rearrange the remaining cells into a compact layout. Simply hiding cells in the table by setting `CSS display:none` would hide them from the user but leave the table layout intact, making it impossible to arrange the top left cell with the textual scenario description and the bottom right cell with the final story pattern at the same height. Listing 4 shows the div based HTML that uses CSS table properties to emulate the extended layout and can be rearranged to create the compact layout by emulating a table with a single row and two cells, only by changing the CSS. The trick is to use an additional outer div that has no function in the extended layout but is used to emulate the table element in the compact layout. Using GWT and two different stylesheets to emulate the compact and extended layout with a div based layout allows us to instantly toggle between the two different visualizations.

Related work

Storyboards have been introduced with Fujaba but never left the domain of CASE tools or IDEs. With the Fujaba community being the only force behind the notation and the amount of work required to create graphical editors, to date, only eclipse has gained storyboarding support by reusing the existing codebase in the Fujaba for eclipse plugin. Rendering of storyboards in the browser makes its debut in the web application we developed.

As far as browser based ontology learning is concerned only few web applications are concerned with learning ontologies. With Wiki@nt Bao and Honavar have created a wiki specifically designed to model ontologies.⁴⁵ Far more applications try to learn ontologies by analyzing user generated content. Benz describes the ontology learning aspect behind bibsonomy⁴⁶ in his Dissertation.⁴⁷ While ontologies are not uncommon in web applications, explicit web applications to learn ontologies are rare.

Conclusion

The quick visual feedback of our web application rapidly draws testers into a state of experimentation where they try different sentences and scenario descriptions. Adding a compact layout to hide the underlying complexity keeps new users from being overwhelmed and al-

⁴⁵ Jie Bao and Vasant Honavar. "Collaborative Ontology Building with Wiki@nt - A multi-agent based ontology building environment". In: *Proceedings of the 3rd International Workshop on Evaluation of Ontology-based Tools (EON2004)*. Oct. 2004, pp. 1-10

⁴⁶ BibSonomy Project. *BibSonomy. The blue social bookmark and publication sharing system*. 2005. URL: <http://www.bibsonomy.org/> (visited on 03/24/2014).

⁴⁷ Dominik Benz. "Capturing Emergent Semantics from Social Annotation Systems". PhD thesis. University of Kassel, Feb. 26, 2013. URL: <http://nbn-resolving.org/urn:nbn:de:hebis:34-2013022642523>.

```

1 <div class='{compact.outer}' ui:field="outerDiv">
2   <div class='{compact.inner}' ui:field="innerDiv">
3     <div class='{compact.r1}' ui:field="r1Div">
4       <div class='{compact.r1c1}' ui:field="r1c1Div">
5         <!-- Step Description -->
6       </div>
7       <div class='{compact.r1c2}' ui:field="r1c2Div">
8         <!-- Structurization Rules -->
9       </div>
10      <div class='{compact.r1c3}' ui:field="r1c3Div">
11        <!-- Formalization Rules -->
12      </div>
13    </div>
14    <div class='{compact.r2}' ui:field="r2Div">
15      <div class='{compact.r2c1}' ui:field="r2c1Div">
16        <!-- Natural Language Parser Result -->
17      </div>
18      <div class='{compact.r2c2}' ui:field="r2c2Div">
19        <!-- Structurization Pattern -->
20      </div>
21      <div class='{compact.r2c3}' ui:field="r2c3Div">
22        <!-- Formalization Pattern -->
23      </div>
24    </div>
25    <div class='{compact.r3}' ui:field="r3Div">
26      <div class='{compact.r3c1}' ui:field="r3c1Div">
27        <!-- Log -->
28      </div>
29      <div class='{compact.r3c2}' ui:field="r3c2Div">
30        <!-- Informal Pattern Result -->
31      </div>
32      <div class='{compact.r3c3}' ui:field="r3c3Div">
33        <!-- Formal Story Pattern Result -->
34      </div>
35    </div>
36  </div>
37 </div>

```

Listing 4: Div based layout

allows them to expose and visually examine the internal algorithm on demand. In summary, choosing GWT as the target platform for a diagram savvy application was the right decision, especially with regard to accessibility for end users.

Outlook

Although the web application is already well suited to support this thesis claims, it can be improved to make it more suitable for everyday use. At the moment the application is designed to handle only one storyboard to demonstrate the storyboarding process. For everyday use a user and project component should be added. With HTML5 it would be possible to store the users textual scenario descriptions and transformation rules on the client without having to rely on a server for storage. In the far future it could be possible to extend the supported number of diagrams to match those of Fujaba, bringing the whole Story Driven Modeling process to the web.

Instant grammatical relations

While all natural language parsers take plain text as input, they vary greatly in implementation language, result format, precision and recall. To try out various parsers for the suitability in our web application a common, minimal, graph based result format, ready for further transformation is missing. By wrapping the individual parsers as JSON-P web services we can exchange the different parse results by switching to another URL. This approach will enable the instant storyboarding prototype to abstract from the parser implementation and work with any web service that takes a textual description and returns the result graph as JSON.

Introduction

Before we can think on visualizing textual scenarios we first need to enrich the plain text with more information. Without any annotations „Alice is playing chess.“ is just a series of characters to the computer. Like a child first learning about grammar in school we need the computer to identify sentences and words and determine subject, predicate and object. All this information needs to be stored in a suitable data model: a graph.

While the field of natural language processing (NLP) has produced dozens of parsers to annotate text, they all use their own data format. Basically, the parsers produce a graph which is converted to a bracketed textual treebank notation described by Marcus, Santorini, and Marcinkiewicz.⁴⁸ The output for our examplesentence could look like [listings 5](#) or [6](#).

OpenNLP and the Stanford Parser agree on the treebank, but the Stanford Parser wraps the sentence in an extra `ROOT` node and adds some line breaks for readability. For this thesis we are more interested in a graphical representation of the treebank. To get there, we need to find a generic way to transport the underlying graphs from the server to the browser.

```
(TOP (NP (NNP Alice)) ↔
↔ (VP (VBZ is) (VP ↔
↔ (VBG playing) (NP (NN ↔
↔ chess)))) (. .))
```

Listing 5: „Alice is playing Chess“ OpenNLP parse

```
(ROOT
(S
(NP (NNP Alice))
(VP (VBZ is)
(VP (VBG playing)
(NP (NN chess))))
(. .)))
```

Listing 6: „Alice is playing Chess“ Stanford parse

⁴⁸ Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. “Building a Large Annotated Corpus of English: The Penn Treebank.” In: *Computational Linguistics* 19.2 (1993), pp. 313–330. URL: <http://dblp.uni-trier.de/db/journals/coling/coling19.html#MarcusSM94>.

Unfortunately, none of the NLP parsers are ready for direct use in a web application. To access different parsers in that environment I

- chose a simple JSON data format suitable to transfer parser results from a web server to a web application via JSON-P,
- implemented a web service for the Stanford and OpenNLP parsers,
- provide a call mechanism for web applications that automatically reconstructs cyclic graphs from the JSON data, and finally
- compared POS-trees and grammatical relations for suitability in this thesis.

Together, this solution allows exchanging the NLP parser of our web application by pointing it to a new URL.

No common graph for parser results

Linguists and researchers in natural language processing have developed a wide variety of parsers. OpenNLP comes with a command line interface for their NLP parser and coreference annotation.⁴⁹ Text2Onto is an interactive application for semi automatic learning of ontologies.⁵⁰ GATE is an integrated development environment that allows you to compose and test an NLP toolchain for use in another application.⁵¹ And Stanford already provides an online demo for their CoreNLP tools.⁵² While these tools provide a different user interface, focusing on specific problems in NLP, at some point, they all internally work on a graph structure: an annotated corpus of text: a treebank.

Before we can use this treebank in graph transformations we have to represent the graph in a data format our transformation engine understands, much like NLP researchers use graphs in their papers to visualize sentence structures. So instead of whatever GUI comes with the parser, we need direct access to the data model that is used to store the annotations.

The parse results may be trees or even cyclic graphs that need to be transferred from the server to the browser before our web application can work with them. Unfortunately, JAXB,⁵³

⁴⁹ Apache. *OpenNLP*. 2010. URL: <http://opennlp.apache.org> (visited on 07/19/2013)

⁵⁰ Cimiano and Völker, “Text2Onto - A Framework for Ontology Learning and Data-driven Change Discovery”

⁵¹ H. Cunningham et al. “GATE: A framework and graphical development environment for robust NLP tools and applications”. In: *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*. 2002. URL: <http://gate.ac.uk/sale/acl02/acl-main.pdf>

⁵² Stanford. *CoreNLP*. 2011. URL: <http://nlp.stanford.edu:8080/corenlp/> (visited on 07/19/2013).

⁵³ Metro Project. *JAXB Reference Implementation*. 2003. URL: <https://jaxb.java.net/> (visited on 03/24/2014)

the Java EE 6 solution for this (un-)marshalling problem widely used by server applications by default only handles tree like structures.⁵⁴ We either need to find a more suitable framework or provide our own implementation to send cyclic graph structures from a web server to a browser with HTTP.

⁵⁴ Metro Project. *Mapping cyclic references to XML*. 2009. URL: https://jaxb.java.net/guide/Mapping_cyclic_references_to_XML.html (visited on 03/24/2014).

Wrapping parse results in a common graph

The graph drawing community, concerned with the geometric representation of graphs and networks, has specified GraphML⁵⁵ as a file and exchange format for graphs. We are going to wrap some of the NLP parsers as web services that return the treebank as a graph instead of plain text. Although XML is a widely used data exchange format, the better alternative for a web application would be JSON,⁵⁶ the native data representation of JavaScript running in browsers. Luckily, the graph drawing community has already created a JSON version of GraphML: GraphSON.⁵⁷

With the data structure in place we need to decide on an RPC mechanism that allows us to transparently use the NLP parser result graphs generated on the server side in our web application. The problem with browser applications is that they have to respect the same origin policy.⁵⁸ The browser will only allow HTTP requests to the same domain that served our web application. In the case of instant-storyboarding.de all parser web services would have to be deployed at URLs like instant-storyboarding.de/stanford or instant-storyboarding.de/opennlp. Instead of using a proxy server that the University of Kassel would have to change whenever a new parser is used, it is possible to work around the same origin policy with a mechanism called JSON-P.⁵⁹ GWT already provides an implementation for JSON-P calls that we will use to transfer GraphSON from the server to the client.

⁵⁵ GraphML Team. *The GraphML File Format*. 2002. URL: <http://graphml.graphdrawing.org/> (visited on 03/24/2014).

⁵⁶ Douglas Crockford. *Introducing JSON*. 2002. URL: <http://www.json.org/> (visited on 03/24/2014).

⁵⁷ Stephen Mallette and Marko A. Rodriguez. *GraphSON Reader and Writer Library*. 2012. URL: <https://github.com/tinkerpop/blueprints/wiki/GraphSON-Reader-and-Writer-Library> (visited on 06/11/2012).

⁵⁸ W3C. *Same Origin Policy*. 2009. URL: http://www.w3.org/Security/wiki/Same-Origin_Policy (visited on 03/24/2014).

⁵⁹ Kyle Simpson. *Defining Safer JSON-P*. 2010. URL: <http://www.json-p.org> (visited on 03/24/2014).

The Details

A common graph for parser results

Various XML based languages have been developed to represent Graphs. In 2000, The Graph eXchange Language (GXL) was developed as a standard exchange format for graphs to enable interoperability between software reengineering tools.⁶⁰ In 2002, the graph drawing community has standardized the Graph Markup Language (GraphML) as an exchange format for data transfer between graph drawing tools.⁶¹ And the Graphviz community has developed DotML as an XML variant of the Dot language.⁶² Of these three only the graph drawing community has developed⁶³ a JSON variant of their GraphML language shown in [listing 7](#). Instead of reinventing the wheel, we use the GraphSON variant of GraphML to represent and transfer the parser results in our web application.

⁶⁰ Richard C. Holt, Andreas Winter, and Andy Schürr. “GXL: Towards a Standard Exchange Format”. In: *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*. Limerick, June 2000. URL: <ftp://ftphost.uni-koblenz.de/ftp/outgoing/Reports/RR-1-2000/RR-1-2000.pdf>.

⁶¹ Ulrik Brandes et al. “GraphML Progress Report: Structural Layer Proposal”. In: *Proceedings of the 9th International Symposium Graph Drawing (GD '01) LNCS 2265*. Springer-Verlag, 2002, pp. 501–512. URL: <http://www.inf.uni-konstanz.de/algo/publications/behhm-gprsl-01.ps.gz>.

⁶² Martin Loetzsch. *The Dot Markup Language*. 2002. URL: <http://martin-loetzsch.de/DOTML/> (visited on 03/24/2014).

⁶³ Mallette and Rodriguez, *GraphSON Reader and Writer Library*.

```
{
  "graph": {
    "vertices": [
      {
        "name": "alice",
        "_id": "1", "_type": "vertex"
      }, {
        "name": "chess",
        "_id": "2", "_type": "vertex"
      }
    ],
    "edges": [
      {
        "_id": "3",
        "_type": "edge",
        "_outV": "1",
        "_inV": "2",
        "_label": "playing"
      }
    ]
  }
}
```

Listing 7: GraphSON example

During the work on this thesis it became necessary to mark edges as unidirectional. The example given on the GraphSON page⁶⁴ does not mention the optional edge attribute “`directed`” present in GraphML.⁶⁵ As reserved properties like “`_id`” and “`_type`” are prefixed with an underscore we will use “`_directed`” as an edge attribute. The change is consistent with the GraphSON notation and allows us to represent directed edges.

GWT, JSON-P and a parser web service API

With recent versions of GWT the support for JSON-P has been improved to a point where the developer just needs to implement a simple asynchronous remote procedure call as in [listing 8](#). In the background GWT injects a script element with the given source URL into the HTML DOM. The result is expected to be a valid JSON expression that is evaluated by the browser. By sending a “`callback`” or “`jsonp`” query parameter this allows the execution of an arbitrary JavaScript method that takes the actual result as an argument. A query parameter “`callback=handleResult`” and a result “`Hello JSON-P!`” becomes `handleResult("Hello JSON-P!");`. To free memory GWT also takes care of cleaning up old and injecting new script elements for every JSON-P call. In summary, calling web services outside the applications domain has become much easier with GWT natively supporting JSON-P.

⁶⁴ Mallette and Rodriguez, *GraphSON Reader and Writer Library*.

⁶⁵ Ulrik Brandes, Markus Eiglsperger, and Jürgen Lerner, eds. *GraphML Primer. Declaring an Edge*. 2012. URL: <http://graphml.graphdrawing.org/primer/graphml-primer.html#GraphEdge> (visited on 03/24/2014).

```

1 String url = getParserURL() + URL.encodeQueryString(description);
2 JsonRequestBuilder jsonp = new JsonRequestBuilder();
3 jsonp.requestObject(url, new AsyncCallback<JavaScriptObject>() {
4     public void onFailure(Throwable throwable) {
5         // handle error case
6     }
7     public void onSuccess(JavaScriptObject graph) {
8         // unparse GraphSON
9     }
10 });

```

Listing 8: JSON-P call with GWT

⁶⁶ Bauke Scholtz. *maximum length of HTTP GET request?* 2010. URL: <http://stackoverflow.com/a/2659995/828717> (visited on 03/24/2014).

⁶⁷ Point your browser to [http://instant-storyboarding.de/de.uks.nt2od.stanford/parser/Alice is playing Chess](http://instant-storyboarding.de/de.uks.nt2od.stanford/parser/Alice%20is%20playing%20Chess). to see the GraphSON result and try other sentences as well.

With the mechanism set, we still need to define the API our web application will use to execute a parser. A minimal service will take the corpus as a parameter of an HTTP GET request and return the result as GraphSON, so our web application can immediately work on it. Although there is no exact limit for the length of the URL in an HTTP GET request, today's browsers and servers usually allow for at least 2kb, Firefox even 8kb,⁶⁶ which is long enough for our purpose of sending textual scenario descriptions in the URL. As a technical requirement for JSON-P the URL must also contain query parameter for the callback the browser is meant to execute upon receiving the response. To keep the URL short, we decided to encode the corpus as a URL path segment instead of a parameter and used an abbreviated parameter "c" that takes the callback function. Listing 9 shows the resulting URL format our web application will call and gives the URLs to the OpenNLP and Stanford web services we implemented⁶⁷. Adding a the "c" parameter must wrap the GraphSON in a JavaScript function callback given by it. Having defined the web service we can now take a look at the implementation for OpenNLP and Stanford.

```

1 proto://host:port/path/to/service/<corpus>?c=<callback>
2
3 http://instant-storyboarding.de/de.uks.nt2od.opennlp/parser/<corpus>
4
5 http://instant-storyboarding.de/de.uks.nt2od.stanford/parser/<corpus>

```

Listing 9: Parser web service URL format

Wrapping OpenNLP as a web service

OpenNLP is distributed as a set of jars that we can wrap in a web service. The distributed package comes with a command line interface that provides easy access to the NLP parser implementation and can be used to try out the parser. Our web application expects the web service to take English text as an input and return GraphSON. We are going to use the jars as

a library and write a web service that directly uses the OpenNLP parser implementation to parse natural English language and convert the result into GraphSON.

To keep deployment cycles short we will not add the parsers trained model directly to our web service but expect it on the java classpath. Adding the trained model provided by the developers would add over 100MB that rarely change but would have to be copied to the servlet container every time we deploy the OpenNLP web service which happens quite often during development. As the model is provided as a jar we can add it to the servlet containers lib folder⁶⁸ so our web service can find it on the classpath.

Loading the model takes a few seconds, so we are going to implement the parser as a singleton. As we are using a servlet container we can use the `@Singleton` and `@PostConstruct` annotations to lazy initialize our parser when the class is first used. This way the sentence detector, tokenizer and parser model only need to be loaded once and allow reusing the initialized parser in subsequent parses.

After loading the model we need to chain three method calls to create a parse tree for a text corpus. Splitting the corpus into sentences is handled by calling `sentDetect(corpus:String)` of an instance of `SentenceDetectorME`. The loaded model is used to detect interpunctuation and leave numbers like “5.4” intact. Each sentence is tokenized by `tokenize(sentence:String)` of the initialized `TokenizerME`. Each array of tokens is merged back into a single line of text that can be parsed by using the static `ParserTool.parseLine()`. The result is an array of `Parse` that we add to a `List of Parse[]` for every sentence in the corpus. Furthermore, the elements create a parent / children tree as shown in figure 14.

In the last step of the OpenNLP server web service we need to represent the parser result as GraphSON. JAXB is the standard solution of the EJB stack to un/-marshal objects to and from XML or JSON. But it would require us to implement a GraphSON data model and in this case marshalling classes. For us, the code was much cleaner after dropping JAXB and implementing the marshalling with `JSON.simple`.⁶⁹

For the web service specific code we relied on the EJB infrastructure. The web service API is annotated with `@Get`, `@Path`, `@PathParam` and `@Produces`. This creates the necessary handler that extracts a part of the path in an HTTP GET request as the corpus, uses the parser

⁶⁸ Glassfish, Tomcat and Jetty all have a lib folder that can be used to provide jars to all servlets which is typically used for the database driver.

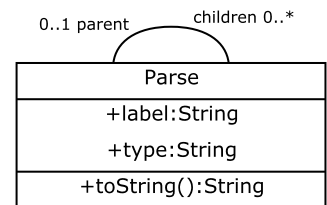


Figure 14: OpenNLP Class Diagram

⁶⁹ Yidong Fang. *JSON.simple. A simple Java toolkit for JSON*. 2008. URL: <https://code.google.com/p/json-simple/> (visited on 03/24/2014).

singleton to parse it and returns the GraphSON result. The whole implementation of the web service only takes two classes and about 250 lines of code.

The attentive reader familiar with JAX/RS may have noticed that in the previous section we have defined the API URL without a special path parameter although we use it in the web service implementation. We are using an Apache HTTP Server as a proxy to rewrite the URL and move the corpus from the path into a path parameter. Furthermore, the Apache server would allow us to host the different web services each on its own machine and even act as a load balancer.

Wrapping the Stanford Parser as a web service

The Stanford CoreNLP tools are distributed similar to OpenNLP. Just like OpenNLP, they are provided separately from a trained model and provide a command line interface for easy access to the implementation. As we have already learned how to write a wrapper for an NLP parser in the previous section the only new challenge is calling the parser and representing the result as GraphSON.

For Stanford an NLP pipeline with sentence detection, tokenization and part of speech tagging like in OpenNLP can be set up as well. The constructor for the `StanfordCoreNLP` class takes `Properties` as a parameter. Setting “`annotation`” to “`tokenize, ssplit, pos, lemma, ner, parse, dcoref`” even extends the pipeline with a coreference⁷⁰ annotation. The text is then parsed by wrapping it in an `Annotation` class and calling the `StanfordCoreNLP` objects `annotate(Annotation annotation)` method with it. As with OpenNLP, the CoreNLP tools expect to find a trained model on the classpath, which is best placed directly in the servlet containers shared extensions folder⁷¹.

After parsing, the `Annotation` object contains the result that we want to represent as GraphSON. In contrast to OpenNLP we can not only fetch a PoS tree but also grammatical relations. Passing `CorefChainAnnotation.class`, `SentencesAnnotation.class` and `TokensAnnotation.class` to the `Annotations.get(...)` method, we can iterate over coreferences, sentences and tokens to collect the nodes and edges we need to represent the result as GraphSON. With this the Stanford web service returns the grammatical relations

⁷⁰ As an example, in “*Alice moves her counter.*” a coreference for *Alice* is *her*.

⁷¹ Glassfish uses `<domain>/lib/ext`, Tomcat uses `$CATALINA_HOME/lib`

of the example sentence „Alice is playing chess.“ from the introduction as the GraphSON in listing 10.

```

1 {
2   "graph":{
3     "edges":[
4       {
5         "_id":"e0", "_type":"edge",
6         "_outV":"s0tNNPe5",
7         "_label":"nsubj",
8         "_inV":"s9tVBGe16"
9       }, {
10        "_id":"e1", "_type":"edge",
11        "_outV":"s6tVBZe8",
12        "_label":"aux",
13        "_inV":"s9tVBGe16"
14      }, {
15        "_id":"e2", "_type":"edge",
16        "_outV":"s17tNNPe22",
17        "_label":"dobj",
18        "_inV":"s9tVBGe16"
19      }
20    ],
21    "vertices":[
22      {
23        "_id":"s0tNNPe5", "_type":"vertex",
24        "text":"Alice", "type":"NNP"
25      }, {
26        "_id":"s9tVBGe16", "_type":"vertex",
27        "text":"playing", "type":"VBG"
28      }, {
29        "_id":"s17tNNPe22", "_type":"vertex",
30        "text":"Chess", "type":"NNP"
31      }, {
32        "_id":"s6tVBZe8", "_type":"vertex",
33        "text":"is", "type":"VBZ"
34      }
35    ]
36  }
37 }

```

Listing 10: „Alice is playing Chess“ graphson

Cyclic graphs and JSON-P

On the technical side, grammatical relations introduce a (un-)marshaling problem, because they may contain cycles. When choosing a framework to use for creating a JSON string for the result graph, we first created a Java data model for a graph and then used JAXB to marshal it. Marshaling tree like structures as with the parse result from OpenNLP worked out of the box. But after mapping the grammatical relations from Stanford to our initial graph data model JAXB would fail to marshal cyclic graphs because we had not used the required annotations.⁷² The automatic conversion of java objects to JSON / XML with JAXB is nice, but when trying to find out how to return the result in JSON-P with JAXB we decided that directly controlling a `String` instead of indirectly controlling a given marshaling mechanism was more straight forward. Replacing JAXB with json-simple and writing the result as GraphSON even reduced the code complexity as the extra data model and marshaling annotations were now unnecessary.

⁷² `@XmlID` and `@XmlIDREF` are your friends, see: Metro Project, *Mapping cyclic references to XML*.

Furthermore, using a standard like GraphSON allowed us to write a generic unmarshaling mechanism that would reconstruct any parser result as a graph based data model in the browser. After making the JSON-P call with GWT, our `GraphSONReader.unparse()` returns the returned `JavaScriptObject`, creating the vertices as `UMLObject` objects and reattaching the edges as `UMLLink` objects. The UML prefixed classes had originally been inspired by the Fujaba data model and are used to represent nodes and edges in the subsequent graph transformations. As a result our web application can run graph transformations on any URL that returns GraphSON via JSON-P.

PoS trees vs. grammatical relations

On the conceptual side, grammatical relations are a better starting point to derive collaboration diagrams than part of speech trees. Where PoS trees are used to show the structure of a sentence grammatical relations shift the focus to dependencies between words. POS trees give you an answer to “What are the nouns in this sentence?” Grammatical relations also give you an answer to “What is the role of this noun in the sentence?” In collaboration diagrams

the question becomes “Who does what?” which is a lot easier to answer with grammatical relations being able to point out subject, predicate and object of a sentence.

Another advantage of grammatical relations is the ability to represent coreferences, even across sentence borders. When trying to answer “What is related to *Alice*?” in the chess example from [listing 1](#) coreferences will point out “...*her* pawn ...” or that “*Alice* moves ...”. For our goal of representing textual scenario description as a collaboration diagram, the higher level representation of grammatical relations is the most suitable starting point.

Related Work

While a lot of natural language parsers have been implemented only few can be accessed with a REST/JSON API. The most notable one being S.Pr.A.W., a Stanford Parser API for the Web⁷³ which is a JRuby on Rails wrapper around the Stanford Parser that even supports JSON-P. Unfortunately, the service is very unreliable and does not return GraphSON, so we deployed our own wrapper.

⁷³ Andrew LeBlanc. *S.Pr.A.W.. Stanford Parser API for the Web*. 2010. URL: <https://github.com/LeBlanc/SPRAW> (visited on 03/24/2014).

Conclusion

Using a JSON-P based web service that returns the parse result in GraphSON makes exchanging the parser very easy. On the server side, implementing wrappers for OpenNLP and the Stanford Parser showed that the json-simple library is a straight forward and controllable way to produce GraphSON. On the client side, our web application reconstructs the graph described by the GraphSON result. As a result, the current implementation can be used to execute graph transformations not just on parser results, but any web service returning GraphSON via JSON-P.

None of the natural language parsers that I tried were directly usable in a web application. Neither are any of them implemented in JavaScript, nor can they be compiled from Java to JavaScript using GWT. For the latter they would have to manually be split into client and server parts, as they all use standard Java file IO that GWT cannot convert to JavaScript. Last but not least, the memory requirements for eg. the Stanford Parser with 6GB RAM cannot yet be fulfilled by every client. Leaving the parsing step on the server is the better choice for now.

Outlook

Meanwhile, the World Wide Web Consortium has created a working draft for Cross-Origin Resource Sharing (CORS) that uses a `Access-Control-Allow-Origin: *` header to make cross domain HTTP requests.⁷⁴ CORS makes workarounds like JSON-P superfluous and, as of August 2012, all modern browser engines support it.⁷⁵ With this, JSON-P can be considered deprecated and our web application should adopt to the new standard.

A much more interesting task would be to use GWT to compile a Java based NLP parser to JavaScript. The biggest problem here is that you cannot use classes from the `java.io` package which most parsers use to read files. Accessing the model files is an even bigger challenge as they come at around 100MB to 200MB. Nevertheless, using HTML5 client storage could be used to cache trained models. Executing the parser on the client would relieve the server from the CPU intensive task and remove this possible performance bottleneck.

Of course, OpenNLP and the Stanford Parser are not the only NLP parsers. Using JSON-P and GraphSON makes it easy to examine other parsers and experiment with graph transformations on their results. Considering the “*instant*” part of this thesis title a good starting point would be the MaltParser⁷⁶ as examined by Cer et al.⁷⁷

While adding other parsers the initialization should be done upon startup of the servlet container, not upon the first request. The current web service wrappers for OpenNLP and the Stanford Parser both use lazy loading, causing the first request to take up to 30 seconds when initializing the parser with the learned model. The `@Singleton` annotation then keeps the initialized parser in memory, speeding up subsequent requests. A better way would be to configure the servlet container to load and initialize the wrapper servlets with the `<load-on-startup>` element.⁷⁸ Using the `<load-on-startup>` will however cause the initialization whenever the servlet is deployed which might be undesired for development purposes.

⁷⁴ Anne van Kesteren. *Cross-Origin Resource Sharing*. Jan. 29, 2013. URL: <http://www.w3.org/TR/cors/> (visited on 06/29/2013).

⁷⁵ Todd Anglin has written a nice Article with examples on how to use CORS: Todd Anglin. *Using CORS with All (Modern) Browsers*. Oct. 3, 2011. URL: http://blogs.telerik.com/kendoui/posts/11-10-03/using_cors_with_all_modern_browsers (visited on 03/24/2014).

⁷⁶ Johan Hall, Jens Nilsson, and Joakim Nivre. *MaltParser*. 2006. URL: <http://www.maltparser.org/> (visited on 03/24/2014)

⁷⁷ Daniel M. Cer et al. “Parsing to Stanford Dependencies: Trade-offs between Speed and Accuracy.” In: *LREC*. ed. by Nicoletta Calzolari et al. European Language Resources Association, 2010. ISBN: 2-9517408-6-7. URL: <http://dblp.uni-trier.de/db/conf/lrec/lrec2010.html#CerMJM10>.

⁷⁸ Rajiv Mordani. *JSR-000315 Java™ Servlet 3.0. Maintenance Release*. Feb. 6, 2011. URL: <https://jcp.org/aboutJava/communityprocess/mrel/jsr315/index.html> (visited on 03/24/2014)

Instant graph visualization

Introduction

The parser web services from chapter “[Instant grammatical relations](#)” on page 49 provide their results as GraphSON⁷⁹ and we already reconstruct the graph with JavaScript objects in the browser. A visualization of the graph however is still missing. Fortunately, there is an excellent tool to plot graphs called Graphviz. As it is widely used in scientific papers to render graphs we will reuse it for our web application.

⁷⁹ As an example [listing 10](#) can be found on page 57.

Like many parsers Graphviz is a command line tool which lacks a ready to use web service suitable for a web application. In order to visualize graphs in our web application, I

- created a domain specific language to encode graphs in URLs,
- implemented a web service that parses these URLs and
- returns an image of the graph rendered with Graphviz.

Similar to the parser wrappers, this web service is a wrapper for a well known tool that will allow us to reuse the code already developed by others.

The Problem

Storyboarding is a highly graphical activity where immediate visual feedback allows the user to be more productive. Every change to the textual scenario description will change the graph returned by the parser. And every storyboarding step after that will also change the working graph. To visualize the graph transformations that implement the storyboarding process as well as any intermediate graphs we need a flexible and fast graph rendering engine. The sooner the user can see an updated graph the earlier can he try another textual formulation or graph transformation.

My Idea

There are two ways to render graphs or images in web applications. The traditional approach is to use a web service that takes care of the image creation process and provides a URL where

⁸⁰ Tobin Harris. *yUML*. 2009. URL: <http://yuml.me/> (visited on 03/24/2014).

⁸¹ If their service is up, following [http://yuml.me/diagram/plain/class/edit/\[Alice\]-playing-\[Chess\]](http://yuml.me/diagram/plain/class/edit/[Alice]-playing-[Chess]) will bring you to their online editor for the diagram encoded in the URL.

⁸² Tobin Harris. *yUML pipeline*. July 30, 2010. URL: https://groups.google.com/forum/#!msg/yuml/tLO8P188c_0/XsvFV43r25sJ (visited on 07/19/2013).

the web application can fetch the result for inclusion in a web page. With HTML5 it has also become possible to use JavaScript and styled div elements to render the graph directly in the browser. For our web application I decided to stick to the traditional approach and wrap Graphviz as a web service. The implementation was heavily inspired by the yuml.me⁸⁰ service that is capable of rendering class diagrams encoded in the URL⁸¹. I initially used their service, but over time it became evident that they were having availability issues and were unable to render real object diagrams or, as is necessary for our web application, story pattern, where colored nodes and edges have semantic relevance. Thankfully, the yuml.me developers gave a little insight into their implementation, so I decided to rewrite and adapt it to my requirements.⁸²

The Details

The yuml.me service parses the URL, converts the graph to the .dot language for Graphviz, renders it as SVG and then uses XSLT transformations to add fancy decorations. For the purpose of our web application varying the decorations is unnecessary, so we will omit the XSLT transformations. We still need to create a DSL to describe graphs in a URL, write a .dot script and use Graphviz to render it, before we can return an image to the browser.

A DSL for storyboarding

The yuml.me notation for class diagrams is very readable for humans. [Listing 11](#) shows the verbose version of a simple class diagram. Each row is used to specify a relation between two classes. The unique idea of yuml.me is to visually describe the kind of relation and class by using an ASCII art like notation. Classes are enclosed by square brackets, an association is represented as a dash. If you want to add a name to a relation put it between two dashes. If you want to add attributes or methods split the class with a pipe and separate attributes with semi-colons. To link to the diagram with a URL you concatenate the rows with a comma and prepend it with the yuml.me URL to the online editor⁸³. While more variations are available⁸⁴ yuml.me is not a modeling but a drawing tool. You can for example use more than two pipes in a class which does not make sense in a UML class diagram. The expressiveness of the

⁸³ The yuml.me sample becomes [http://yuml.me/diagram/plain/class/edit///CoolClassDiagram,\[Customer\]<>-orders*>\[Order\],\[Order\]++-0..*>\[LineItem\],\[Order\]-\[note:Aggregate root.\]](http://yuml.me/diagram/plain/class/edit///CoolClassDiagram,[Customer]<>-orders*>[Order],[Order]++-0..*>[LineItem],[Order]-[note:Aggregate root.])

⁸⁴ Examples for inheritance, notes, cardinalities, directed associations and even colors can be found at <http://yuml.me/diagram/class/samples>

[yuml.me](#) notation gives users a lot of flexibility but still lacks features needed to model the story patterns used in storyboarding.

```
1 // Cool Class Diagram
2 [Customer] <>-orders* > [Order]
3 [Order] ++-0..* > [LineItem]
4 [Order] - [note:Aggregate root.]
```

Listing 11: [yuml.me](#) notation example

Inspired by the yuml notation for class diagrams I gradually evolved the notation to describe the different kinds of diagrams needed in our web application. The [yuml.me](#) service concentrates on class, activity and use case diagrams, so I first added the ability to render object diagrams which most resemble class diagrams, but have their identifiers underlined. Adding a method call notation then enabled us to render collaboration diagrams. Story Pattern became possible after adding modifiers to create, delete and optionally match elements. [Table 1](#) gives an overview of the FUMML notation and the resulting visualization.

At this point we can express a subset of Story Pattern using literal Strings or numbers in attribute assignments. To assign and reference variables we will revisit the DSL in “[A DSL for Storyboarding revisited](#)” on page 77. For now let us continue with this more simple subset.

Extending the [yuml.me](#) notation has led us to a domain specific language for storyboarding. The ASCII art inspired DSL is very readable, yet flexible enough to encode object diagrams, class diagrams, collaboration diagrams and story patterns. With FUMML, the GraphSON we received⁸⁵ for our example sentence “*Alice is playing Chess.*” can be encoded into a single line for use in a URL. Due to the page width I manually added line breaks to the result in [listing 12](#) represented by ↵ and ↴. Now all we need is a parser for this notation.

⁸⁵ See [listing 10](#) on page 57.

Parsing URLs with ANTLR

Once we have a parser grammar we can use a parser generator to generate the necessary parser code. In the previous subsection we described some examples which we now use to deduce a parser grammar. [Figure 15](#) gives an overview of the FUMML Grammar containing the core elements.

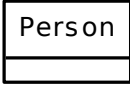
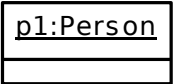
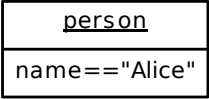
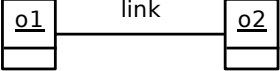
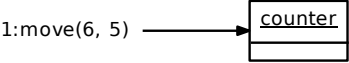
FUML Notation	visual representation
To render a class in yUML it is enclosed in square brackets: <code>[Person]</code> (<i>click the notation to open it in the browser</i>)	
The most fundamental change for FUML is the shift from classes and associations to objects and links. To produce object diagrams we need a way to underline the identifier:	
Enclose name and class with an underscore: <code>[_p1:Person_]</code>	
An object can have attributes: <code>[_person_ name=="Alice"]</code>	
And objects can be connected with links: <code>[_o1_-link-_o2_]</code>	
Using underlining as a convention is all the difference it needs to distinguish class diagrams from object diagrams. But we also want to be able to render collaboration diagrams:	
We will use an object followed by an arrow to send a message: <code>[_counter_] <-1:move (6, 5)</code>	

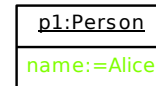
Table 1: FUML notations and their visual representation can be accessed by pointing a browser to [http://instant-storyboarding.de/fuml/diagram/story/\[FUMLNotation\].png](http://instant-storyboarding.de/fuml/diagram/story/[FUMLNotation].png).

FUML Notation
visual representation

Story Pattern introduce create, delete and optional modifiers to objects and links. Fujaba colors elements **green** or **red** to represent their creation respective deletion. We decided to reuse well known characters like +, - and ? to represent them in FUML:

Assigning an attribute is done with the := operator:

```
[_p1:Person_|name:=Alice]
```



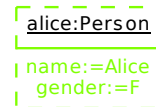
Create an object by prepending it with +:

```
+[_alice:Person_|name:=Alice;gender:=F]
```



Make creation optional by appending a ?:

```
+[_alice:Person_|name:=Alice;gender:=F]?
```



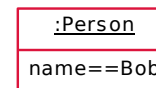
Create a link by prepending the name with +:

```
[_o1_]++link-[_o2_]++
```



Delete an object by prepending it with -:

```
-[_:Person_|name==Bob]
```



Listing 12: FUML: "Alice is playing Chess."

```

1 [s9tVBGe16:Vertex|↵
2   ↵"text":"String"="playing";↵
3   ↵"type":"String"="VBG"↵
4 ↵]↵
5 ↵-nsubj-↵
6 ↵[s0tNNPe5:Vertex|↵
7   ↵"text":"String"="Alice";↵
8   ↵"type":"String"="NNP"↵
9 ↵]↵
10 ↵,↵
11 ↵[s9tVBGe16]↵
12 ↵-aux-↵
13 ↵[s6tVBZe8:Vertex|↵
14   ↵"text":"String"="is";↵
15   ↵"type":"String"="VBZ"↵
16 ↵]↵
17 ↵,↵
18 ↵[s9tVBGe16]↵
19 ↵-dobj-↵
20 ↵[s17tNNPe22:Vertex|↵
21   ↵"text":"String"="Chess";↵
22   ↵"type":"String"="NNP"↵
23 ↵]

```

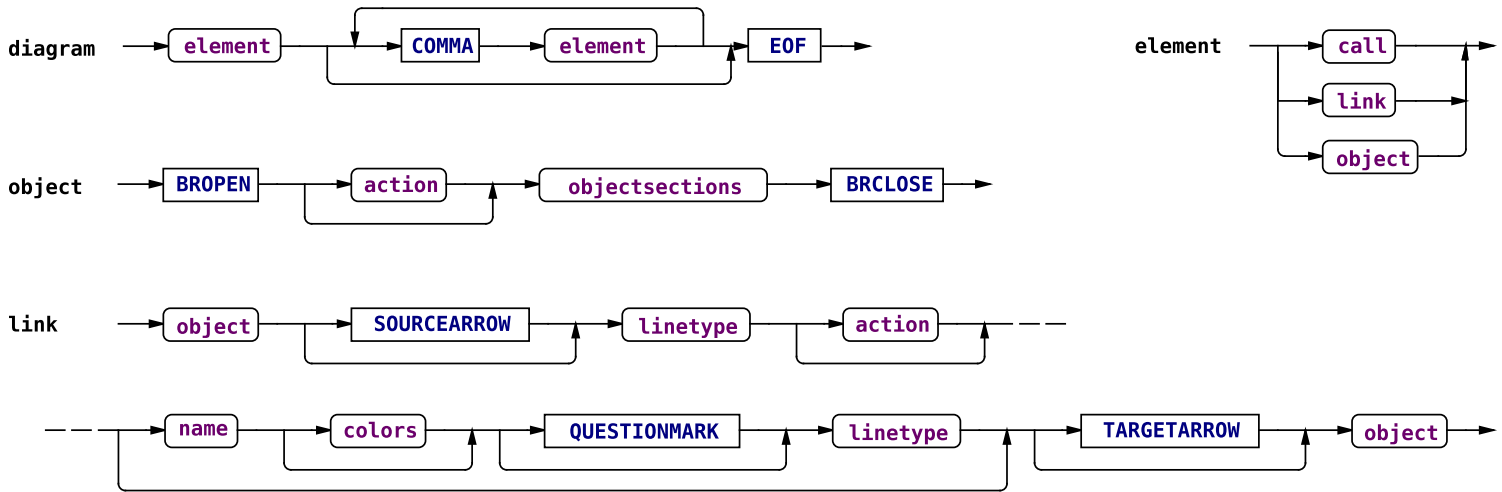


Figure 15: FUML: Grammar Overview

This excerpt shows that a *diagram* is made up of a series of comma separated *elements* which can be an *object*, a *link* or a *call*. Objects are surrounded by square brackets, links connect two objects and calls attach a *method* to an object. For now we will concentrate on parsing the *object* and *link* elements to keep things simple. While adding the *call* element to FUML is straight forward, adding it to the interpreter is a different matter and will be examined in “[Structuring the data model](#)” on page 117.

Using a parser generator for this step saved us the effort to write the parser on our own, but we will revisit that task in subsection “[A DSL for Storyboarding revisited](#)” on page 77.

On the technical side, the implementation relies on the Apache HTTP Server to correctly handle the URL and feed it to Glassfish without mangling any characters. With the introduction of UTF-8 the problem should have been a non-issue. A common Java EE stack uses Apache and `mod_proxy` in front of the servlet container that will in the end handle the HTTP requests. To feed [yuml.me](#) like URLs to our JAX-RS web service we were using a simple rewrite rule that reattaches the URL part representing the diagram in `http://instant-storyboarding.de/fuml/diagram/story/[alice]-[bob].png` as a URL parame-

ter: `http://instant-storyboarding.de/uml/diagram/story/?fuml=[alice]-[bob].png`, injecting a simple `?fuml=`. Unfortunately, UTF-8 characters get mangled in the process because Apache tries to unescape the regex match result into ISO-8859-1. The first google search suggested using an Apache RewriteMap as shown in Lines 1-4 of [listing 13](#) to provide our own escaping which solves the problem.⁸⁶ As it turns out we can alternatively use the RewriteRule flags `noescape|NE` and `B` to prevent Apache from mangling the URL parameter. While the initial workaround served the purpose we finally settled on the officially documented solution shown in line five of [listing 13](#).

⁸⁶ The original page is gone but can still be found in the [internet archive](#): Xiao Jianfeng. *JAVA UTF-8*. 2010. URL: <http://92jssp.com/blog/default/2010/10/27/JAVA-UTF-8> (visited on 01/14/2012).

```

1 #RewriteMap escape int:escape
2 #RewriteRule ^/fuml/diagram/story/(.*)$↵
3 ↵ http://localhost:8080/fuml/diagram/story/?fuml=${escape:$1} [QSA,L,P]
4 RewriteRule ^/fuml/diagram/story/(.*)$↵
5 ↵ http://localhost:8080/fuml/diagram/story/?fuml=$1 [QSA,L,P,NE,B]

```

Listing 13: Apache Rewrite Rule

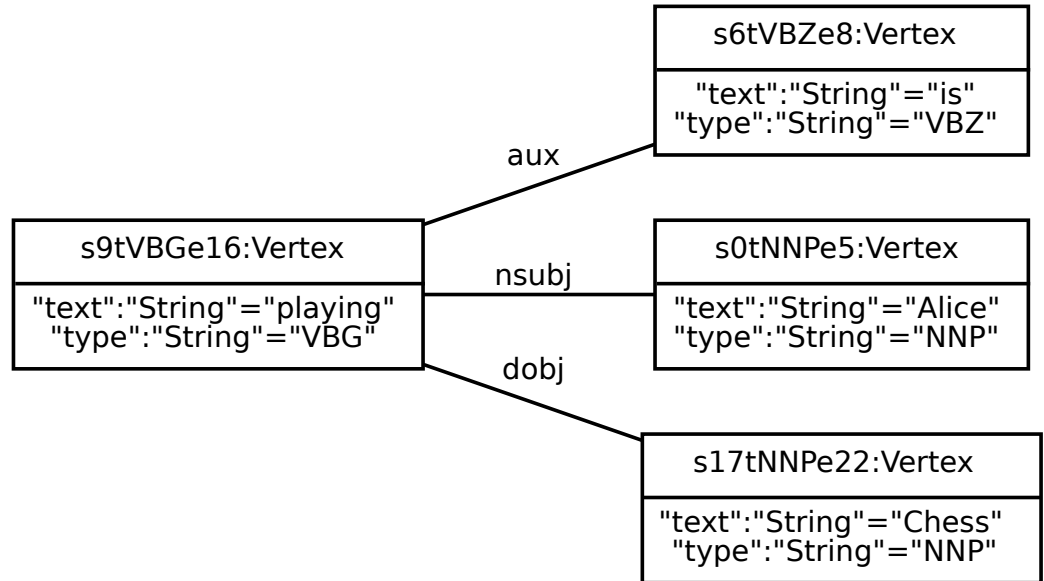
After solving these implementation details ANTLR produces a parse tree that we can use to reconstruct the Storyboard model. Actually, this is another graph transformation step. But since the reconstruction is generic we hard coded the mapping from parse tree nodes to storyboard elements.

Visualizing collaboration diagrams

⁸⁷ Mike Houston. *jGraphViz*. 2008. URL: <http://jgraphviz.sourceforge.net/> (visited on 03/24/2014).

Although there are low level Java bindings for Graphviz⁸⁷ we followed the approach of a simple Graphviz wrapper that is used to create a dot file and then call the Graphviz executable to render the graph. For our short example sentence “*Alice is playing Chess.*” the Stanford parser produced grammatical relations that have been returned to the browser as GraphSON. Our web application reconstructs the graph with JavaScript objects in the browser and creates the FUML notation shown in [listing 12](#). The FUML web service parses the URL with ANTLR and creates the .dot file shown in [listing 14](#). The result of the Graphviz rendering shown in [figure 16](#) is returned as a png and displayed by the browser.

Figure 16: Grammatical relations in "Alice is playing Chess."



```

1 digraph G {
2 fontname = "Bitstream Vera Sans"; fontsize = 8; rankdir="LR";
3 node [ fontname = "Bitstream Vera Sans", fontsize = 8, shape = box, margin = 0, width = 0.5, height = 0.1 ]
4 edge [ fontname = "Bitstream Vera Sans", fontsize = 8, arrowhead = "none" ]
5 node_654206246 [ color="black",
6 label = <<table valign="top" border="0" cellpadding="4" cellspacing="0" width="10" height="10">
7 <tr><td><u>s9tVBGe16:Vertex</u></td></tr>
8 <hr/>
9 <tr><td>text:"String"="playing"<br/>"type:"String"="VBG"</td></tr>
10 </table>>
11 ]
12 node_814338642 [ color="black",
13 label = <<table valign="top" border="0" cellpadding="4" cellspacing="0" width="10" height="10">
14 <tr><td><u>s17tNNPe22:Vertex</u></td></tr>
15 <hr/>
16 <tr><td>text:"String"="Chess"<br/>"type:"String"="NNP"</td></tr>
17 </table>>
18 ]
19 node_624559471 [ color="black",
20 label = <<table valign="top" border="0" cellpadding="4" cellspacing="0" width="10" height="10">
21 <tr><td><u>s6tVBZe8:Vertex</u></td></tr>
22 <hr/>
23 <tr><td>text:"String"="is"<br/>"type:"String"="VBZ"</td></tr>
24 </table>>
25 ]
26 node_128103882 [ color="black",
27 label = <<table valign="top" border="0" cellpadding="4" cellspacing="0" width="10" height="10">
28 <tr><td><u>s0tNNPe5:Vertex</u></td></tr>
29 <hr/>
30 <tr><td>text:"String"="Alice"<br/>"type:"String"="NNP"</td></tr>
31 </table>>
32 ]
33 edge [label = dobj, color = black, fontcolor = black, style=solid, dir=none]
34 node_654206246 -> node_814338642
35 edge [label = nsubj, color = black, fontcolor = black, style=solid, dir=none]
36 node_654206246 -> node_128103882
37 edge [label = aux, color = black, fontcolor = black, style=solid, dir=none]
38 node_654206246 -> node_624559471
39 }

```

Listing 14: “Alice is playing Chess.” in .dot notation. Used to render [figure 16](#)

```

1 fontname = "Bitstream Vera Sans"; fontsize = 8; rankdir = "LR";
2 node [ fontname = "Bitstream Vera Sans", fontsize = 8, shape = box, margin = 0, width = 0.1, height = 0.1 ]
3 edge [ fontname = "Bitstream Vera Sans", fontsize = 8, arrowhead = "none" ]
4
5 ${foreach nodes node}
6   ${node.id} [ color="${node.color}", ${if node.fillcolor}fillcolor="${node.fillcolor}", ${end}
7     ${if node.style}style="${node.style}", ${end}
8     label=<<table valign="top" border="0" cellborder="0" cellspacing="0" cellpadding="4" width="10" height="10"
9       color="${node.color}">
10      <tr><td><u>${node.name}</u>${:,node.type},
11      ${if node.newNameOrType}
12        <br/>
13        ${if node.newName}<font color="chartreuse">${node.newName}</font>
14        ${else}<u>${node.name}</u>${end}
15        ${if node.newType}<font color="chartreuse">${node.newType}</font>
16        ${else}<u>${:,node.type}</u>${end}
17      </td></u></td></tr>
18      <hr/>
19      <tr><td>
20        ${foreach node.attributes a}
21          ${if a.comparator="ASSIGN"}<font color="chartreuse">${a.name}<u>${:,a.type}</u>:=<u>${a.value}</u></font><br/>
22          ${else}<u>${a.name}</u>${:,a.type}</u>${a.comparator}<u>${a.value}</u><br/>${end}
23        ${end}
24      </td></tr>
25    </table>>
26   ]
27 ${end}
28
29 ${foreach edges edge}
30   edge [
31     label=${if edge.bgcolor}<<table border="0"><tr><td bgcolor="${edge.bgcolor}">${edge.name}</td></tr></table>>
32     ${else}<u>${edge.name}</u>${end},
33     ${if edge.fgcolor}color="${edge.fgcolor}", fontcolor="${edge.fgcolor}",${end}
34     ${if edge.style}style="${edge.style}",${end}
35     ${if edge.directed}dir=forward, arrowhead=normal, arrowsize=0.5
36     ${else}dir=none${end}
37   ]
38   ${edge.sourceid} -> ${edge.targetid}
39 ${end}

```

Listing 15: jmte .dot template. Used to generate listing 14

If you haven't noticed by now, this is another graph transformation that creates yet another textual notation. As you can see in lines 9-17, 20-28, 31-39 and 42-50 of [listing 14](#) we make use of Graphviz pseudo HTML notation for nodes to render the objects. Initially, we hard coded this step, adding line by line to the `.dot` file. To remove the Java compilation step we meanwhile moved to the Java Minimal Template Engine⁸⁸ (jmte) which uses the template shown in [listing 15](#). Being able to change the rendering template on the fly allows us to experiment with different `.dot` features and Graphviz renderings without having to redeploy the FUMML web service.

⁸⁸ Oliver Zeigermann and Daniel Florey. *jmte. Java Minimal Template Engine*. 2010. URL: <https://code.google.com/p/jmte/> (visited on 03/24/2014).

Related Work

The FUMML web service was written as a drop in replacement for [yumml.me](#). The [instant-storyboarding.de](#) prototype initially used [yumml.me](#) to render graphs which allowed us to focus on the underlying graph transformations and fix graphical glitches after we had tested our ideas. Thankfully, they explained how they implemented the service, making it a lot easier to just extend their ideas and adapt the web service to the instant storyboarding process.

Conclusion

Storyboarding starts with natural English text and our goal was to produce a graphical representation of it early in the automated process. It allows us to easily evaluate the output of the parser result by visualizing it as a graph. Rendering the parser result graph in our web application gives the developer an immediate visual feedback and allows him to interactively explore the natural language parsers capabilities. We hope that automated feedback educates the user to use simple language and short sentences. Evolving the [yumml.me](#) notation to cover all diagrams necessary for storyboarding and implementing a web service for them allows us to use the textual DSL not only for our web application but also other web pages and even eases the creation of example diagrams for research papers. With the FUMML web service we created an easy to use visualization tool for the storyboarding process that follows the spirit of immediate feedback.

Outlook

If the client is powerful enough it would make sense to offload the rendering of the graphs to a JavaScript rendering engine that runs directly in the browser. The first step could be using the Canviz⁸⁹ JavaScript library to render xdot files directly in the browser canvas. Furthermore, it would be interesting to see how force or gravity based rendering engines as implemented by Mike Bostock⁹⁰ in D3.js⁹¹ behave when the graphs change a lot. Depending on the rendering engine it might even be possible to avoid the placement issues with Graphviz where objects seem to jump around if Graphviz decides to place the same object at a new location every other rendering run.

When the graph is rendered on the client side using JavaScript we could also allow the user to change the jmt template used for the .dot generation. Currently, the template is stored on the server and web application users cannot change it. Allowing them to submit new versions would raise several security issues which we decided to avoid by not adding an 'upload your own template' feature. Moving the graph rendering engine to the client allows us to add this feature without having to worry about introducing security issues on the server.

The FUMML web service itself could also profit from yumml.me like decorations. Especially yumml.me's "scruffy" renderings draw attention and resemble the *work in progress* or *draft* look of hand drawn UML diagrams during prototyping. Applying XSLT transformations to the SVG produced by Graphviz is a nice task for a bachelor thesis.

⁸⁹ Ryan Schmidt. *canviz. JavaScript library for drawing Graphviz graphs to a web browser canvas*. 2006. URL: <http://code.google.com/p/canviz/> (visited on 03/25/2014)

⁹⁰ A lot of interesting visualization can be found on his personal blog: Mike Bostock. *Mike Bostock*. 2011. URL: <http://bost.ocks.org/mike/> (visited on 03/25/2014).

⁹¹ Mike Bostock. *Force Layout*. 2012. URL: <https://github.com/mbostock/d3/wiki/Force-Layout> (visited on 03/25/2014).

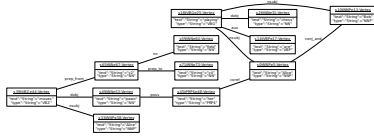


Figure 17: Visualizing the grammatical relations for “Alice and Bob are playing chess. Alice moves her pawn from field c2 to c4.” takes a lot of space. Here, for comparison, the full size [figure 10](#) from page 39 has been scaled to take roughly the same space as the two sentences.

Instant informal story patterns

Introduction

As [figure 17](#) shows, graphs describing grammatical relations can become quite verbose. Furthermore, their point of view is focused on language processing.

We need to transform the parser result into an informal story pattern to reach the next step in the Storyboarding process. As we have defined and implemented a web service for natural language parsers we can rely on the result being a graph structure as described in “[A common graph for parser results](#)” on page 52 provided as GraphSON. We still need to find a way for developers to work with the graph and graph transformations in a visually accessible way.

To restructure the parsers result graph into an informal story pattern representing the textual scenario I

- created a textual DSL for graph transformation rules,
- wrote an interpreter to execute them in the browser,
- extended the collaboration diagram renderer to also render story patterns, and finally
- created a set of structurization rules to transform grammatical relations of the Stanford Parser into an informal story pattern.

This allows us to automatically interpret grammatical relations from a software developers point of view.

Clarification of the term “story pattern”

In “[Instant graph visualization](#)” on page 61 we created a textual DSL for simple story pattern, the visual representation of a textual scenario step description. Now we will be using graph transformations in the form of story patterns to restructure grammatical relations into an informal story pattern. We are using the following terms to distinguish these story patterns:

Story pattern is a diagram mix of elements from UML object diagrams and UML collaboration diagrams, enriched with graph transformation concepts such as element matching, creating and deleting. A visual representation of a graph transformation rule.

Informal story pattern is the storyboard element we are trying to derive. The result of applying structurization rules to grammatical relations.

Structurization rule is a graph transformation rule used to restructure grammatical relations into an informal story pattern.

Structurization pattern is a visual representation of a structurization rule as a story pattern.

Formalization rule is a graph transformation rule used to formalize an informal story pattern.

Formalization pattern is a visual representation of a formalization rule as a story pattern.

Interpreting story pattern is the process of executing the structurization and formalization rules. Initially the working graph consists of grammatical relations that become an informal story pattern that becomes the final formal story pattern. We not use the term *intepreting graph transformations* because the visualization always uses story pattern.

With this in mind we can rephrase the second sentence of this paragraph to be clearer: we will be using structurization rules to derive informal story pattern from grammatical relations. Examination of formalization rules will be deferred until “[Instant formal story patterns](#)” on page 98.

A linguists view on scenario descriptions

Instant graphs of grammatical relations are a nice visual feedback for linguists, but the focus on grammatical relations is still far away from being helpful to a software engineer. Take our example sentence “*Alice is playing chess.*” While a linguist will be able to read the grammatical relations from the object diagram in [figure 16](#) on page 69 a software developer would

Tag	part-of-speech
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NP	Proper noun, singular
NPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PP	Personal pronoun
PP\$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP\$	Possessive wh-pronoun
WRB	Wh-adverb

Table 2: Part of speech tags as described by Beatrice Santorini. *Part-of-speech tagging guidelines for the Penn Treebank Project*. Tech. rep. MS-CIS-90-47. Department of Computer and Information Science, University of Pennsylvania, 1990. URL: <ftp://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz>, pp. 6-7

recognize the object diagram but might need to refresh his NLP knowledge before understanding the terminology of *NNP*, *VBG* and *aux*. Nevertheless, natural language processing or grammatical relations are the foundation for every software engineers work.

As already described in the “Storyboarding by Example” section on page 13, the first task in Storyboarding requires the software developer to identify nouns and verbs in the textual scenario description. Actually, when looking at the available part of speech tags in table 2 and grammatical relations for natural English language the problem is a lot more complex than just identifying nouns and verbs. NLP parser results are very detailed and need to be aggregated into a more concise UML object diagram to be expedient in instant storyboarding.

A software engineers view on scenario descriptions

Instead of manually deriving a story pattern from a textual scenario description we will use structurization rules to restructure the grammatical relations. Identifying nouns and verbs - or rather matching various patterns in grammatical relations and creating objects, links and messages will allow us to construct an informal story pattern, something software engineers are more familiar with than part-of-speech tags or grammatical relations.

While we are experimenting with different parsers and new graph transformations the structurization rules are subject to change. To minimize the feedback loop we will implement an interpreter that can execute graph transformations in the browser. Specifying and executing graph transformation rules in a web application saves the compilation step and allows rapidly iterating through changed rule sets.

The technical part of this challenge is implementing an interpreter that can be executed in the native browser language JavaScript. As we will see in the following sections, GWT allows us to reuse state of the art software engineering tools to implement the interpreter in Java and cross compile it to JavaScript. Not having to install an Adobe Flash plugin or the Java browser plugin greatly lowers the barrier of entry for interested developers.

The Details

As already mentioned in “A DSL for storyboarding” on page 62 we will now revisit our FUMML language to introduce variable assignments and references. Then we can implement an interpreter for these graph transformation rules and add visualizations. Not only for the graph transformation rules and their results but also for intermediate steps of the execution for debugging purposes. Finally, we can experiment with the structurization rules to restructure grammatical relations into an informal story pattern.

A DSL for Storyboarding revisited

To write down graph transformations that assign values found in other elements of the working graph we need to distinguish literal string identifiers from path expressions. Referencing the name attribute of another element requires two additions to our notation: assigning objects to a variable and referencing their properties. As an example listing 16 shows a structurization rule that creates an object for every proper noun in the grammatical relations and in a second rule deletes the “Vertex” nodes to clean up the diagram.

```

1 # lines starting with # are comments
2 # Create a new object in the diagram for each sentence subject
3 [(o):"Vertex"|"type"=="NNP";"text"==(sub)], [+sub.toLower():"Unknown"?|"name":=sub]
4 # Clean up object diagram by removing parser elements
5 [-(del):"Vertex"]

```

As a preview, the graphical version of these rules is shown in figures 18 and 19 which we will examine later in “Visualizing Story Pattern execution” on page 89.

If more than one rule is specified they are executed in order of appearance. They will however all use the same working graph, which allows to collect nodes with a marker object and iterate over the marked objects in a subsequent rule. We will see another example graph transformation rule where this is used later.

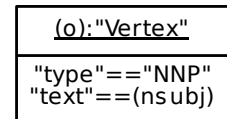
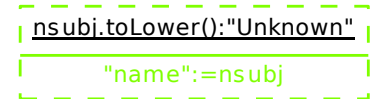


Figure 18: Structurization Pattern: Create an object for every proper noun

Listing 16: Structurization Rules: Create an object for every proper noun

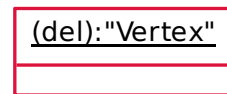


Figure 19: Structurization Pattern: Delete vertexes

For our new graph transformation rules string literals now must be surrounded by single (') or double ticks ("). The first square bracketed part in Line 3 thus matches any nodes in the graph of type "Vertex" that have an attribute "type" with a value of "NNP". Any non literals now describe new actions for the interpreter.

Assigning values to a variable is done by putting the variable name in braces. When matching the first rule the interpreter will assign the name of the current "Vertex" node to "o" and the value of the "text" attribute to "nsubj". We can now reference these variables in other parts of the rule.

The second square bracketed part of the rule references the "nsubj" value twice. It will assign the value, as is, to the "name" attribute and use it lowercased as the object name. The variable references can use methods like "toLowerCase()" to manipulate the assigned value. Since "o" is not referenced we use it as a wildcard to match all "Vertex" nodes regardless of their name.

Instead of literals we can also use regular expressions to match strings. Like in JavaScript they are surrounded by forward slashes: `/NN.*/` will match "NN" and "NNP". This not only reduces the number of rules when dealing with similar rules but also saves the interpretation of rules that only differ in a literal string.

The last two primitive types that can be assigned are also taken from JavaScript. By using numbers or the two reserved strings `true` and `false` for boolean values we can match and assign all primitive datatypes used in JavaScript. They are however not necessary when rewriting grammatical relations into an informal story pattern and are only mentioned here for completeness.

Figure 20 shows the new rules of the FUMML grammar that are necessary to generate a parser for the above changes. We refined the original `valueExpression` by now distinguishing between quoted strings, regular expressions and the assignment values. While the differentiation has no effect on the FUMML rendered graphs, it requires a lot of thought when adding it to the interpreter.

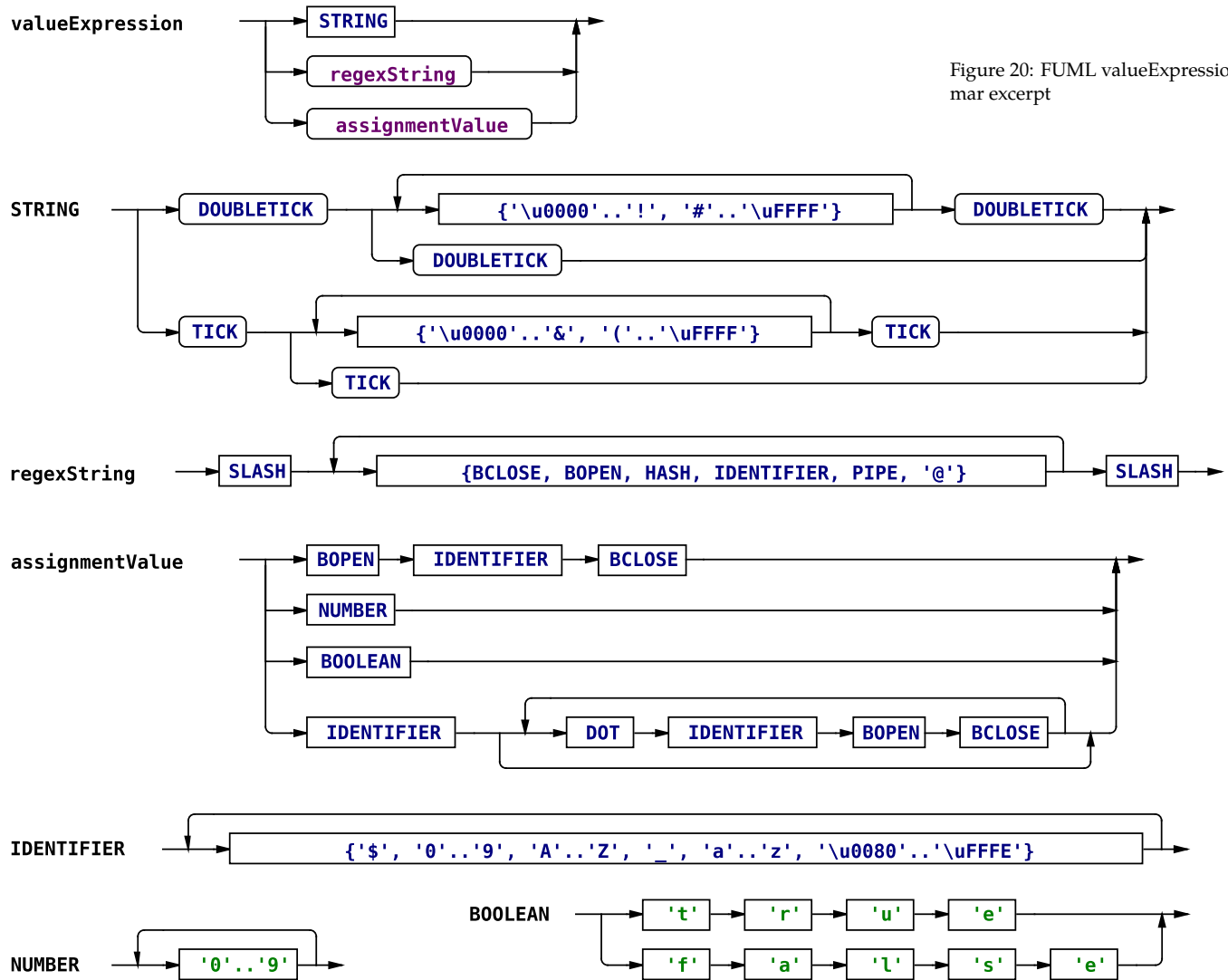


Figure 20: FUML valueExpression grammar excerpt

Interpreting structurization pattern

The interpreter now used in our web application was developed in two major iterations. Similar to the DSL the first iteration was able to match and manipulate only literal string values. In the second iteration I added the capability to assign and reference values as well as regular expression based matching. By using a straight test driven development approach I was able to add feature after feature to the code in short cycles without breaking existing functionality.

In the previous chapter we developed the DSL parser to create the graph transformations from a textual string notation and the `GraphSONReader` described on page 58 to create the working graph from a textual GraphSON representation. As good software engineers the first thought we should have is reusing them to set up our unit tests. Unfortunately, that would create an extra compile step and a significant delay for test execution because the DSL parser uses native JavaScript functions and thus needs to be executed in a `GWTUnitTest`. On my machine a `GWTUnitTest` takes roughly 10 seconds to fire up a jetty server, compile the Java to JavaScript if necessary and run the hosted mode browser before finally executing the unit test. Instead of using the parsers I decided to directly create the small working graph and story pattern models in the unit test. This not only allows me to skip the `GWTTestCase` setup time and execute the unit tests as plain JUnit tests but also keeps the interpreter from depending on the two parser packages.

When adding the interpreter to our web application I created integration tests that use the two parser implementations to set up more complex scenarios in a `GWTTestCase`. This of course revealed several bugs in the parsers, the interpreter and the web application. Creating unit tests where possible and fixing the remaining issues has produced a test suite that reliably shows when code changes break existing and expected behavior.

The interpreter itself uses a stack based approach to keep track of the graph matching steps. Executing a story pattern can be divided in two phases: searching the working graph for the next subgraph match and applying the changes described in the story pattern to that match. A. Zündorf has described the algorithm for the code generation used in Fujaba in his habilitation thesis.⁷² For an interpreter he best described the algorithm in a recorded lecture.⁹² Lacking the visual capabilities of a video, the figures 21 to 47 will use a step by step example to demonstrate how the interpreter executes a story pattern:

⁷² Albert Zündorf. *Rigorous Object Oriented Software Development with Fujaba*. Draft Version 0.3. 2002. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/Zuen02.pdf>

⁹² Albert Zündorf. *Rule Matching*. 2010. URL: <http://seblog.cs.uni-kassel.de/fileadmin/se/courses/MDESS10/MDE04RuleMatching/MDE04RuleMatching.html> (visited on 07/19/2013)

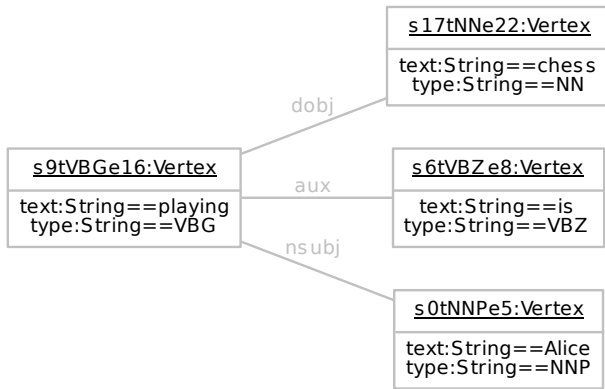


Figure 21: *The working graph.* For our “Alice is playing chess.” example the parser returns this graph which we use as the working graph for the interpreter. The grey borders of the nodes and edges indicate that none of the elements have yet been matched to an element in the structurization pattern.

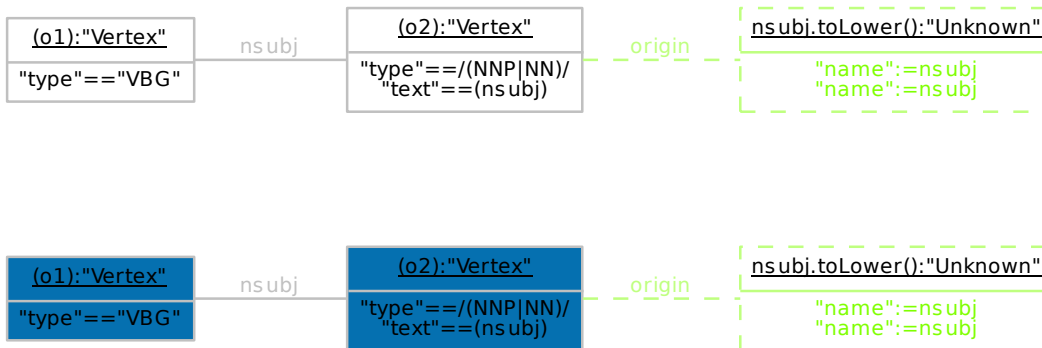


Figure 22: *The story pattern.* The rule executed in this example basically says to search the working graph for two “Vertex” nodes that are linked by an “nsubj” edge. If they exist try to match an “origin” edge to an “Unknown” node and create the missing parts. The grey border represents mandatory matches, the dotted border optional matches and the green border elements to create on every match.

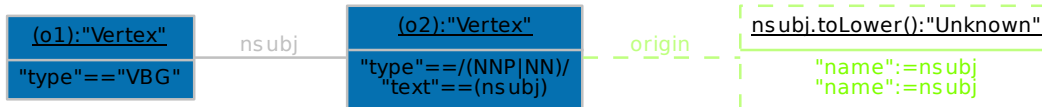


Figure 23: *Identifying by type options in the pattern.* The interpreter starts the matching phase by collecting available node types in the story pattern. In this case, the only type the interpreter can search for is “Vertex”. This expansive operation is visualized by a dark blue background.

Figure 24: *Choosing an initial search option.* Since no other search options are available the interpreter chooses one of the “by type” options. The currently chosen search option is always rendered in cyan in the story pattern. At this point a “Vertex” node that has a “type” attribute with the literal value “VBG”.

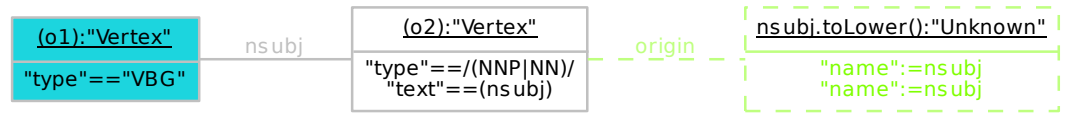


Figure 25: *Collecting possible candidates in the working graph.* Searching by type is expensive because the interpreter has to inspect every Node in the working graph. If the attributes match the description in the story pattern the interpreter will mark them as valid candidates for a match, indicated by a blue background.

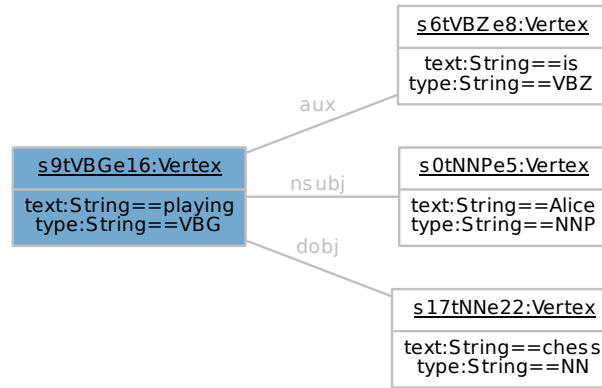
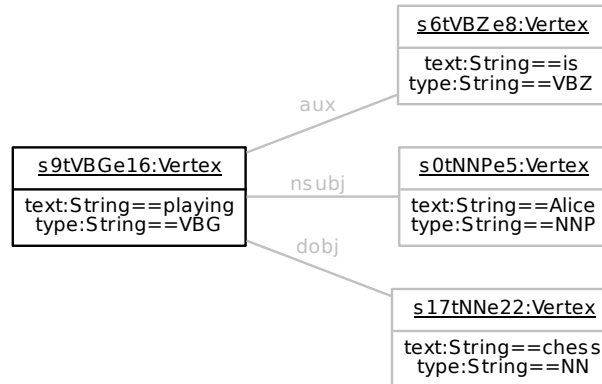


Figure 26: *Choosing a candidate in the working graph.* Again, the interpreter chooses one of the collected candidates in the working graph. It is marked as a matched element, indicated by the black border.



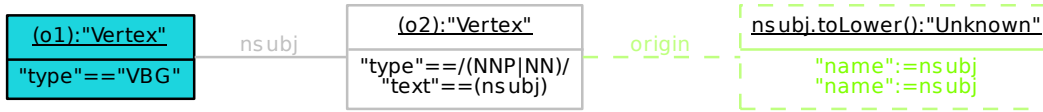


Figure 27: *Marking the match in the pattern.* The by type search was successful and the previously chosen search option is marked as a matched element. In the story pattern as in working graphs matched elements are indicated by a black border.

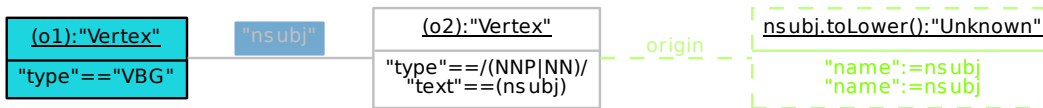


Figure 28: *Identifying to many options from current match.* Starting from the "Vertex" node the interpreter identifies the new available search options. Traversing the "ns subj" link is cheaper than searching "by type", indicated by a medium blue background.

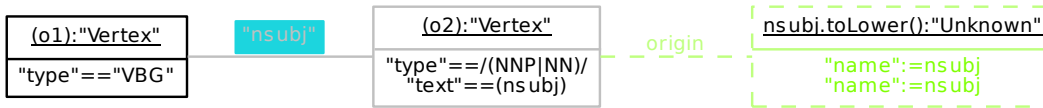


Figure 29: *Choosing a search option.* Traversing the link is currently the cheapest (and only search option) for the interpreter so he chooses to search for it next, again indicated by the cyan background.

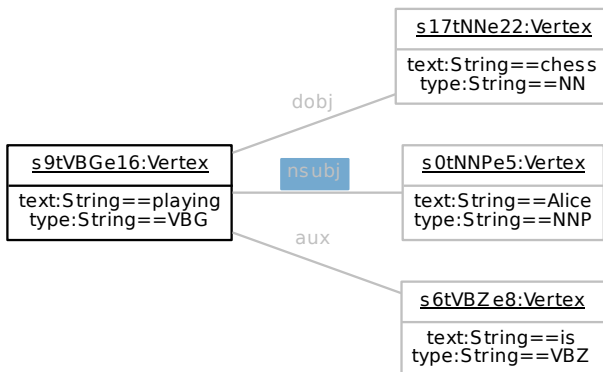


Figure 30: *Collecting possible candidates in the working graph.* Only one "ns subj" link starting from already matched "Vertex" node currently exists in the working graph, so the interpreter marks it as a possible candidate.

Figure 31: *Choosing a candidate in the working graph.* The sole candidate is marked as a match in the working graph.

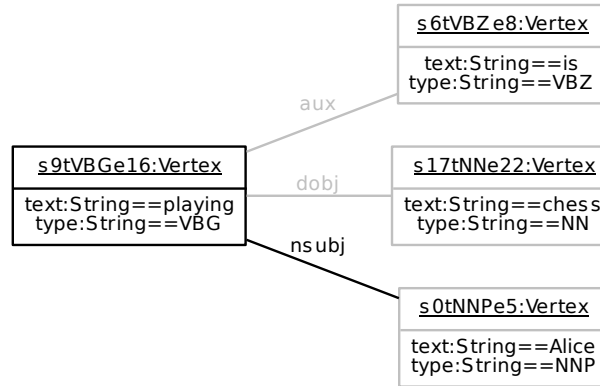


Figure 32: *Marking the match in the pattern.* With the next search option matched the interpreter can now mark the "nsubj" link in the pattern as matched, too.

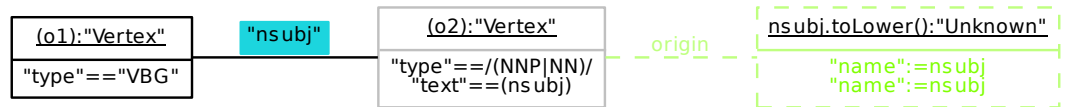


Figure 33: *Identifying to one options from current match.* Starting from the "nsubj" link the interpreter identifies the new available search options. At the end of the link in the working graph the interpreter can expect to find an object that matches the description in the story pattern. This cheap "to one" search option is indicated by a light blue background.

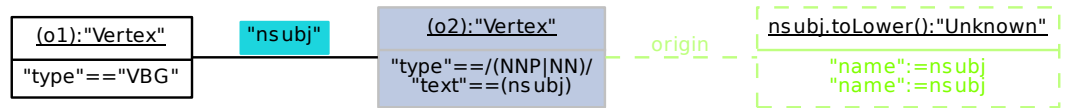
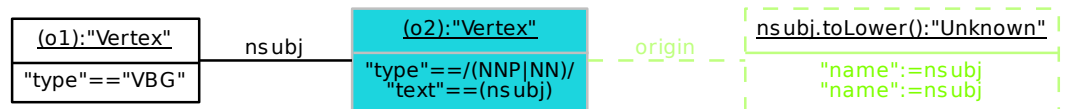


Figure 34: *Choosing a search option.* Checking the correct object is at the end of the "nsubj" link is currently the cheapest (and only search option) for the interpreter so he chooses to search for it next, again indicated by the cyan background.



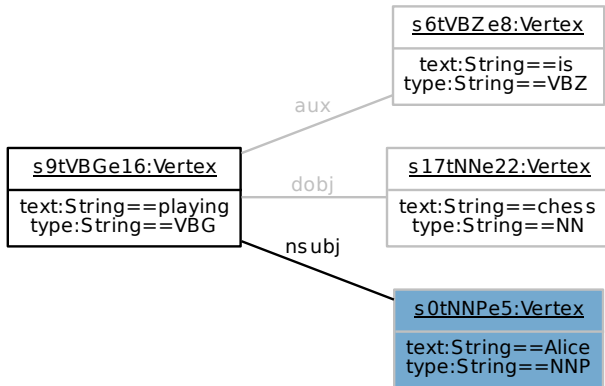


Figure 35: *Collecting possible candidates in the working graph.* Collecting the possible candidates already checks the attributes of the element in question. A mismatch would disqualify the current “to one” search and backtrack the steps the interpreter has taken to find other search options. In this case however the attributes match and the interpreter can mark the second “Vertex” node as a possible candidate.

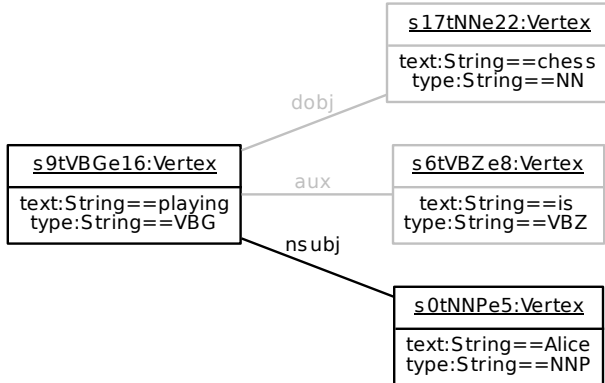


Figure 36: *Choosing a match in the working graph.* The interpreter can mark the second “Vertex” node in the working graph as a match.

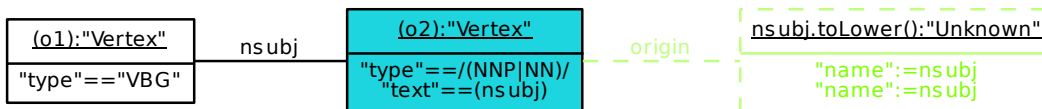


Figure 37: *Marking the match in the pattern.* By marking the second vertex node in the story pattern as matched no mandatory matches are left over for the interpreter to match.

Figure 38: *Identifying optional to many options from current match.* The dashed borders represent optional matches, so the interpreter will now try to find a match for the "origin" link in the working graph.

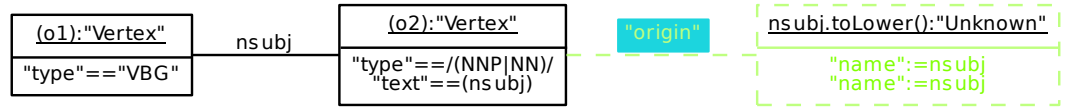


Figure 39: *Creating an object.* All mandatory matches of the story pattern have been found in the working graph so the interpreter can now apply the story pattern changes. First he will create objects. In this case an object named "alice" of type "Unknown". The path expression "nsubj.toLowerCase()" references the assigned "(nsubj)" value of the "text" attribute in the second matched "Vertex" node and lowercases it, before assigning it as the name for the new object.

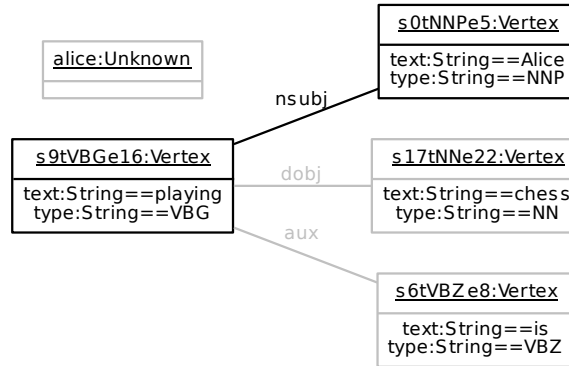
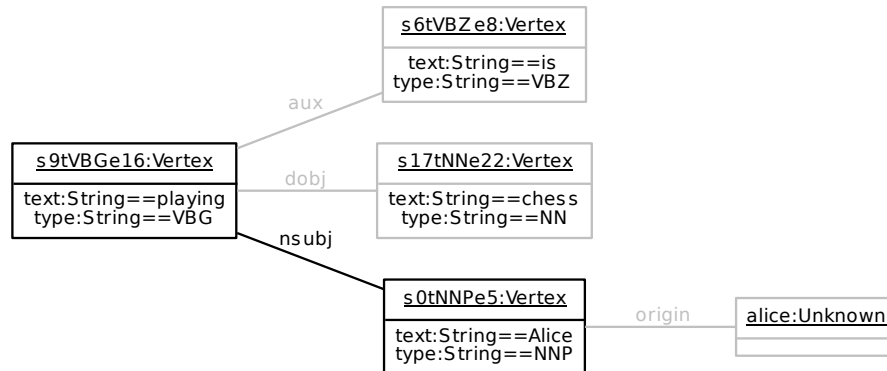


Figure 40: *Creating origin link.* The new "alice" object is now linked to the "origin" of its name. In the web application example rules this "origin" link is used as a marker to identify subject and object in a later rule.



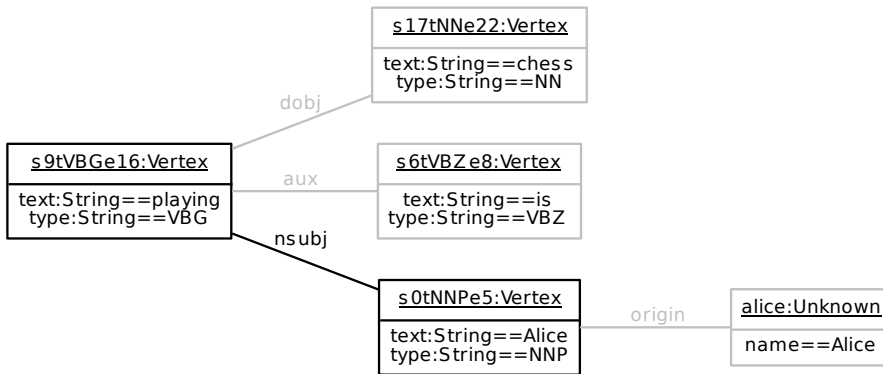


Figure 41: *Assign attributes.* After creating objects and links the interpreter creates any attributes for the new elements in the working graph as specified in the story pattern. Since no elements are to be deleted the rule application ends here.

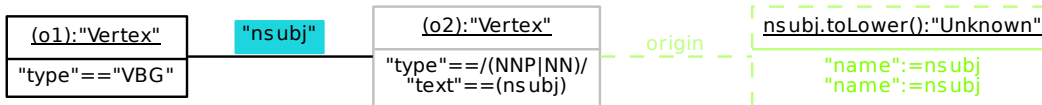


Figure 42: *Backtrack third search option in story pattern.* After the pattern has been applied the interpreter starts searching for the next full pattern match by backtracking the search options. Before the last "Vertex" node search option the interpreter was searching for the "nsubj" link.

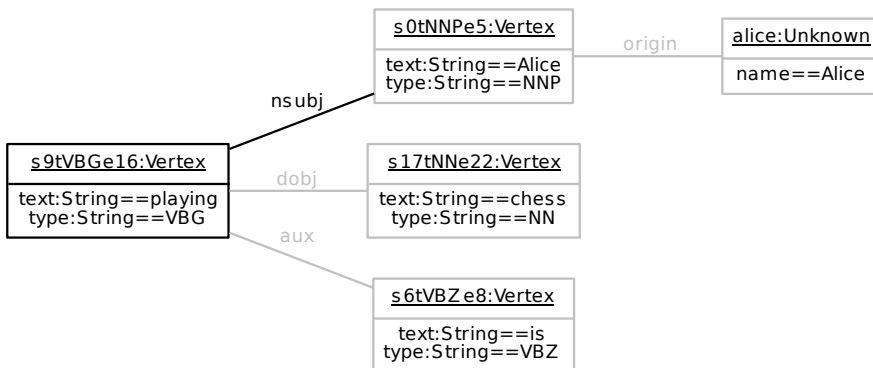


Figure 43: *Backtrack third search option in working graph.* The working graph contains the new "alice:Unknown" object and "origin" link, but the last search option is no longer marked as a match. Only the first "Vertex" node and the "nsubj" link are matched to an element in the story pattern, indicated by the black border. Since no other "nsubj" link exists the interpreter cannot choose another candidate as a match for the search option and has to continue backtracking.

Figure 44: Backtrack second search option in story pattern. Before searching for the "nsubj" link the interpreter was searching for the first "Vertex" node.

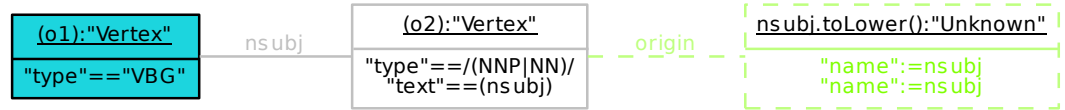


Figure 45: Backtrack second search option in working graph. One "Vertex" node has already been matched and no other "Vertex" nodes qualifies as a candidate (see figure 25). Again, the interpreter continues to backtrack the search options.

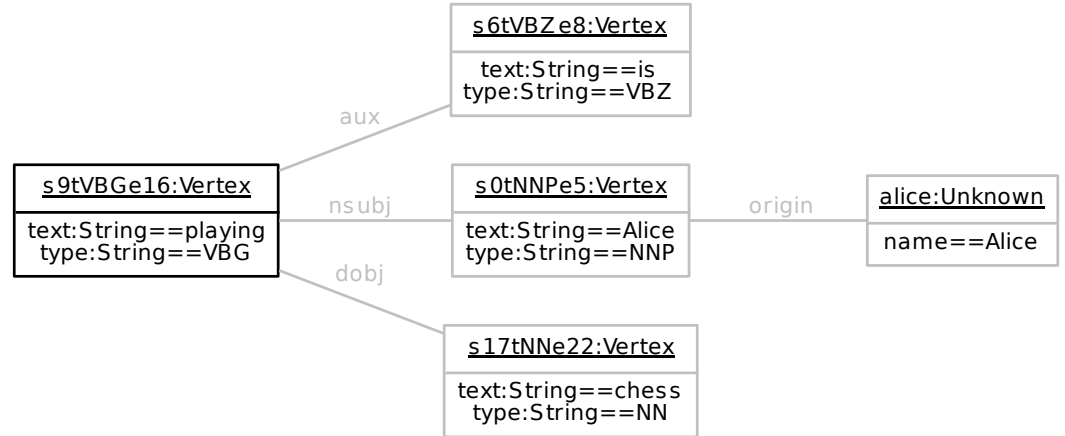
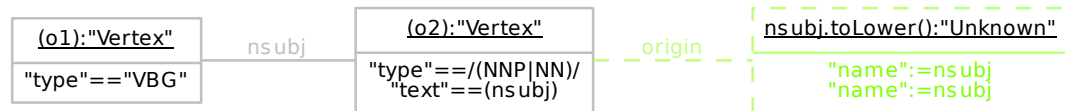
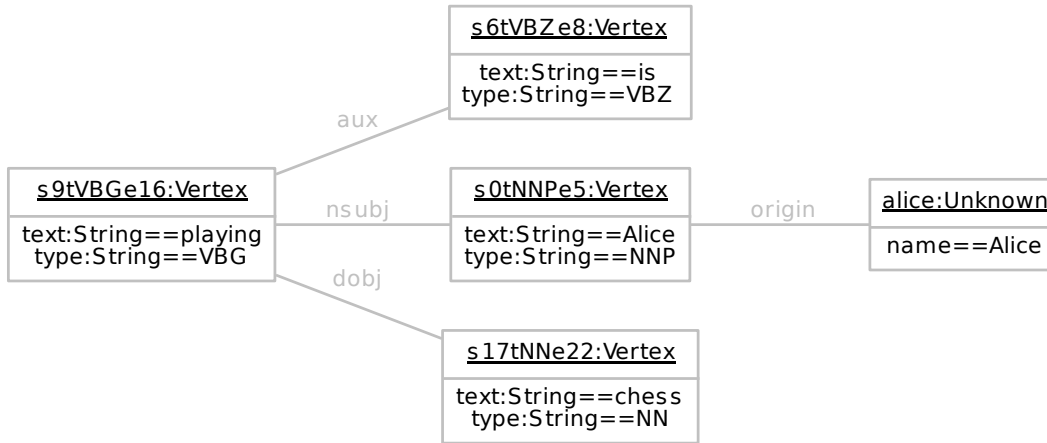


Figure 46: Backtrack first search option in story pattern. Having reached the root of the pattern matching steps the interpreter has no further search options available. He already searched for "Vertex" nodes and no other mandatory matches have a different type so the execution ends here.





Figures 21 to 47 have been taken directly from the debug visualization of our web application. In case a user wants to examine the interpreters rule execution for a given structuration or formalization rule he can click the story pattern visualization to display the individual steps below it. This in depth visualization of how the web application works allows software engineers and researchers to get an immediate feedback on how rule changes affect the storyboarding process.

Visualizing Story Pattern execution

In “A DSL for Storyboarding revisited” we added the dynamic aspects of story pattern to the FUMML grammar. To render debug graphs as shown in figures 21 to 47 we added foreground and background colors to the grammar. As you can see in the three figures 48 to 50 the foreground (`fg`) and background (`bg`) color can be given either as a SVG color name or hexadecimal color code. Figure 51 shows the relevant rules of the ANTLR grammar.

Figure 47: Backtrack first search option in working graph. The working graph now contains the result of the story pattern execution. An “alice:Unknown” object with a “name” attribute derived from the grammatical subject of our example sentence “Alice is playing chess.”

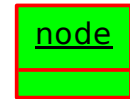


Figure 48: FUMML for a red foreground and a lime background:
`[node{fg:red,bg:lime}]` or
`[node{fg:#ff0000,bg:#00ff00}]`

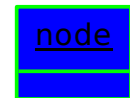


Figure 49: FUMML for a lime foreground and a blue background:
`[node{fg:lime,bg:blue}]` or
`[node{fg:#00ff00,bg:#0000ff}]`

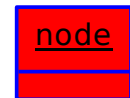


Figure 50: FUMML for a blue foreground and a red background:
`[node{fg:blue,bg:red}]` or
`[node{fg:#0000ff,bg:#ff0000}]`

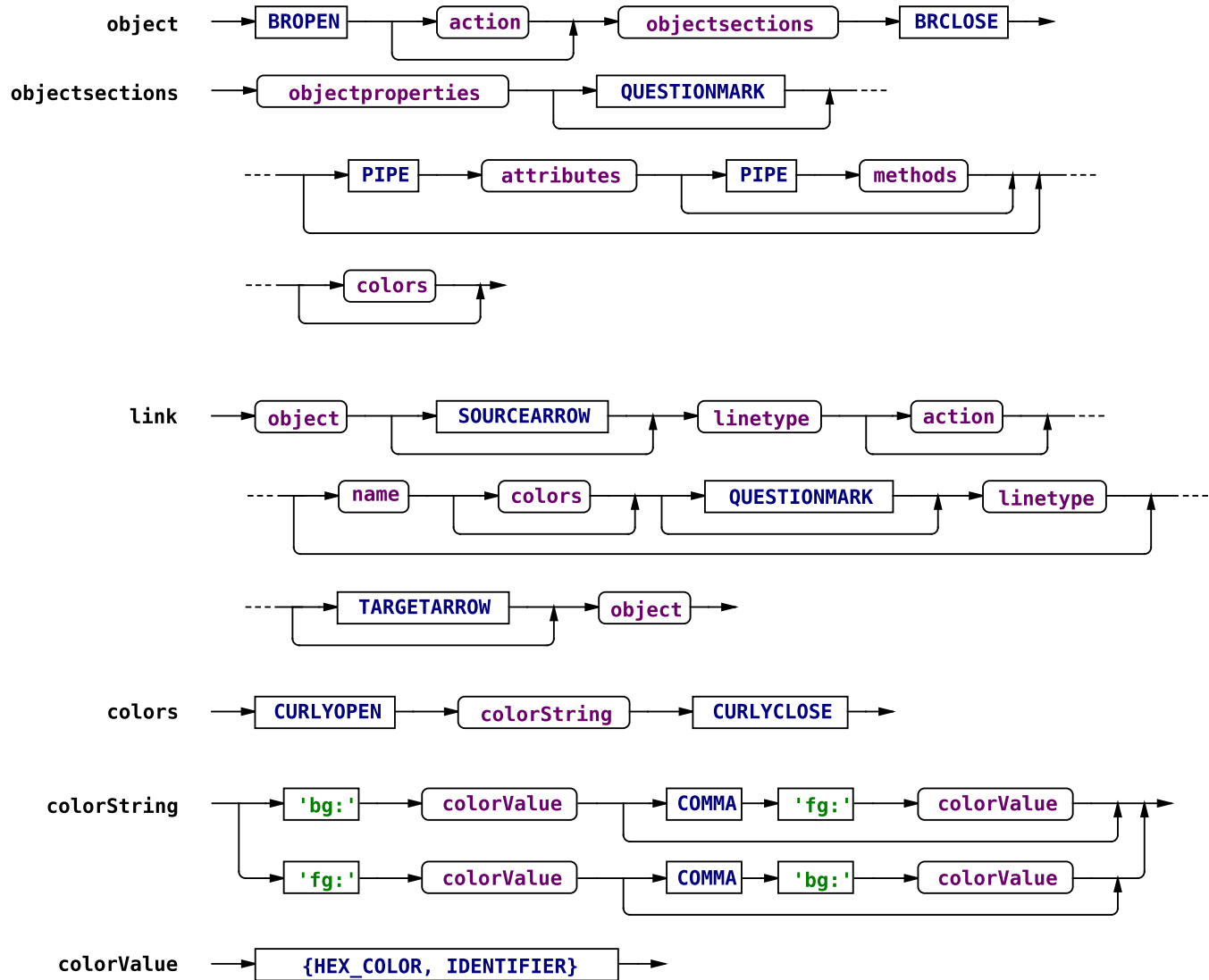


Figure 51: FUML colors excerpt

By default, the FUMML service renders elements with a black foreground on a white background. Elements to create are rendered in chartreuse, elements to delete in crimson. The debug output of the interpreter overwrites these defaults by explicitly specifying the colors. Unmatched elements are rendered in grey or a lighter version of chartreuse and crimson respectively. This part of the FUMML grammar concludes the series of changes that we introduced to instantly visualize story pattern.

From grammatical relations to object diagram

With the parse and render pipeline in place we can finally start working on the actual problem: finding graph transformation rules that restructure grammatical relations into a collaboration diagram. Let us start with [figure 52](#) showing the relations the Stanford parser gives us for "Alice and Bob are playing chess."

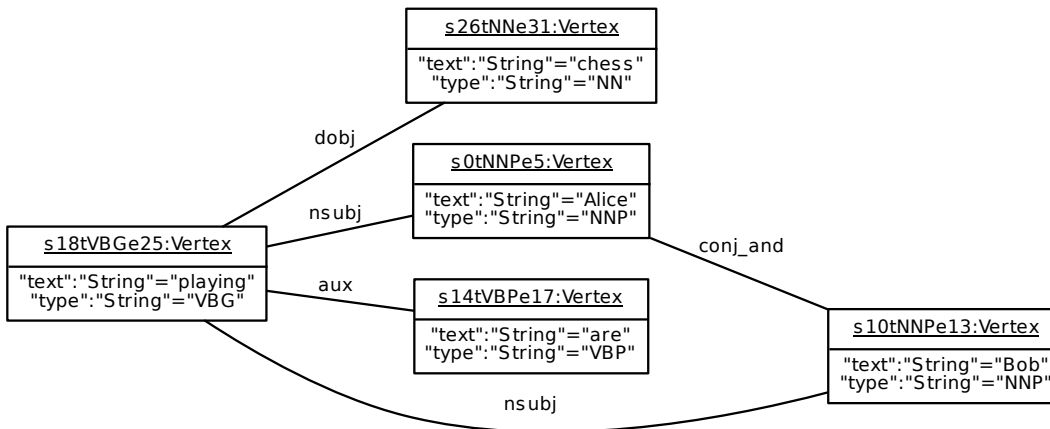


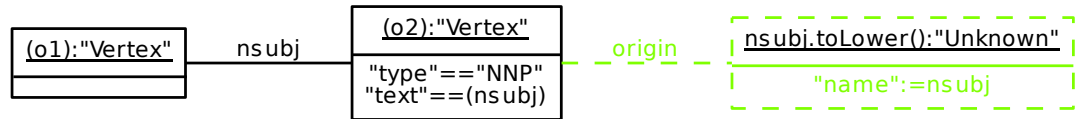
Figure 52: Grammatical relations for "Alice and Bob are playing chess."

To apply the formalization steps described in "Storyboarding by Example" on page 13 we first need to identify the nouns in the sentence. In grammatical relations we can find the

subject and object of a sentence at the end of "nsubj" or "dobj" edges. As a first step let us create a rule that creates an object for every proper noun (NNP) at the end of a "nsubj" edge.

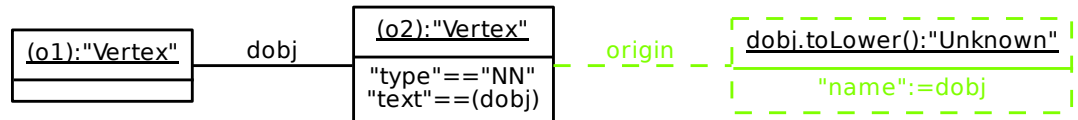
The rule in figure 53 will match Alice and Bob and create a new object for each of them, assigning the text value of the matched node to the (nsubj) variable and referencing it as the value of the name attribute for the new object.

Figure 53: Create a new object for every nsubj and add an *origin* link between them.



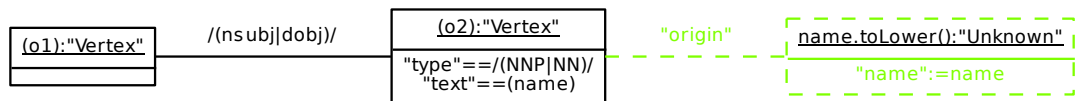
To create an object for "chess" as well, we need to match nouns (NN) at the end of "dobj" edges. The rule in figure 54 looks very similar to the first rule and they both are very specific to our example.

Figure 54: Create a new object for every dobj and add an *origin* link between them.



The only difference between the two rules is the name of the edge and the type of the node. Using regular expressions we can combine the two rules into one as shown in figure 55

Figure 55: Create a new object for every nsubj or dobj and add an *origin* link between them.



After executing the rule our working graph looks like figure 56. At this point we have created an object for every noun and linked it to the origin node so we can reference them when implementing the next formalization rule.

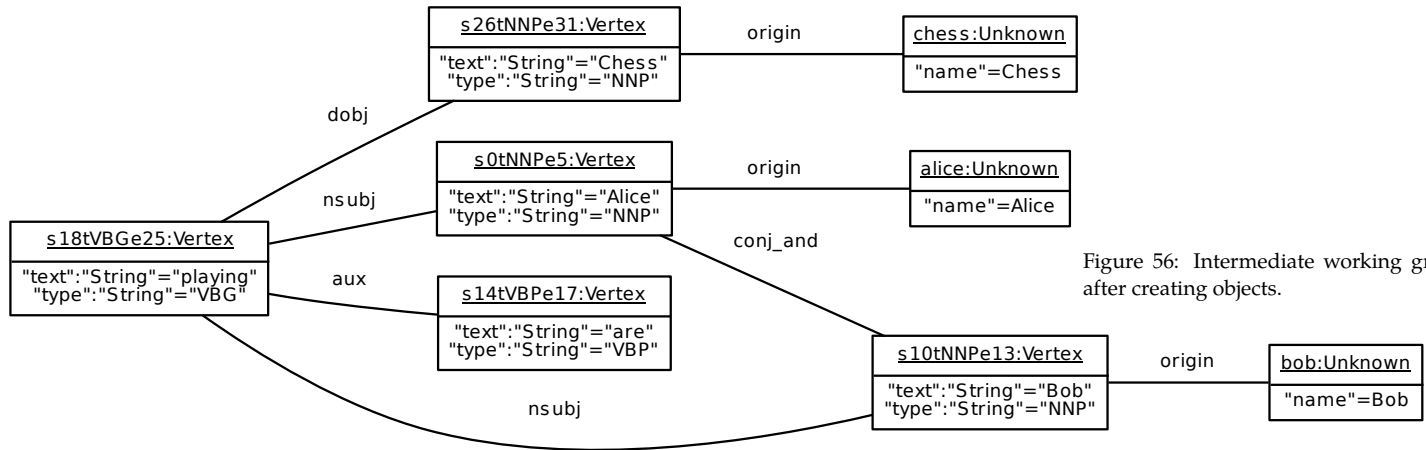


Figure 56: Intermediate working graph after creating objects.

To map the second storyboarding formalization step to a graph transformation rule we have to identify verbs and use them as edges between the corresponding subject and predicate. Figure 57 shows where we use the origin nodes we created earlier to find the predicate between object and subject.

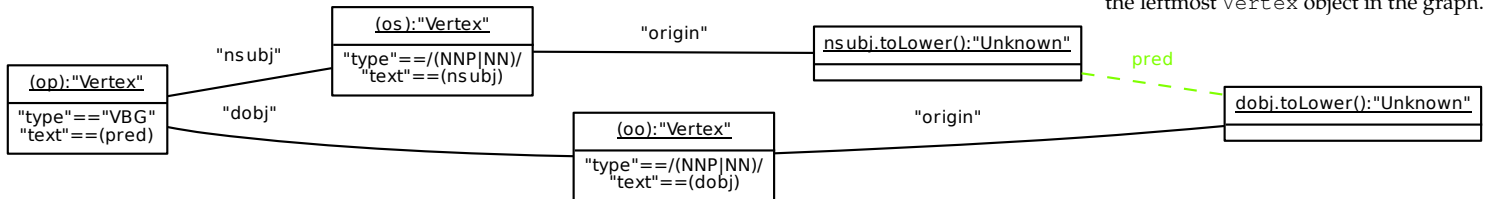


Figure 57: Create a link between subject and object that is named after the predicate `pred`. `pred` is a variable that is created with the `(pred)` notation and assigned the value of the text attribute of the leftmost `Vertex` object in the graph.

Executing this rule leaves us with the working graph shown in [figure 58](#). The objects we created for “Alice”, “Bob” and “chess” are now connected by two “playing” links.

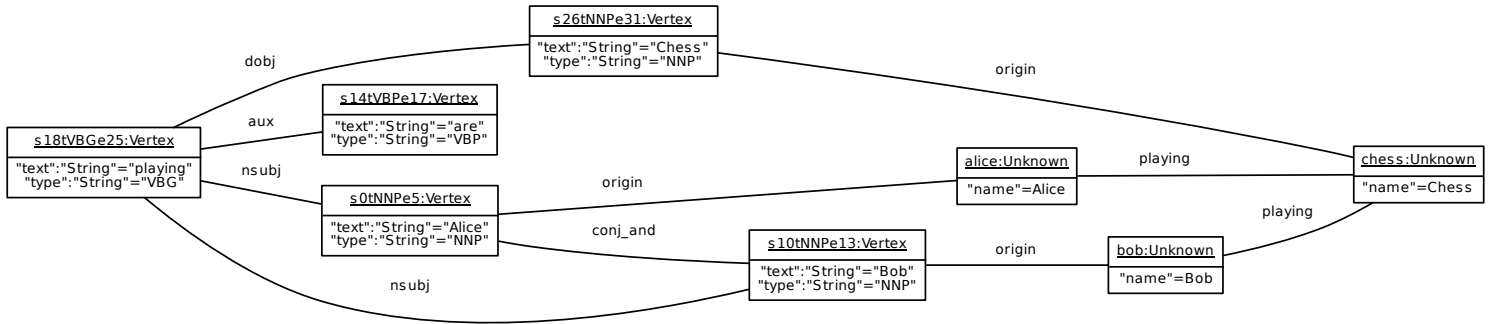


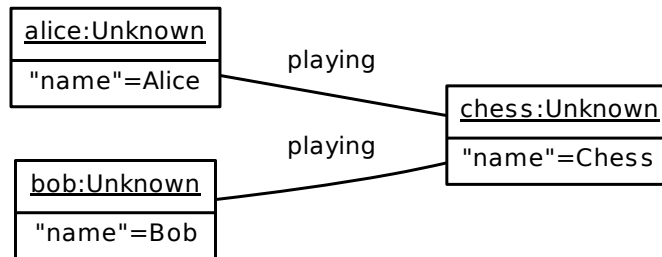
Figure 58: Intermediate working graph after creating links.

`(del):"Vertex"`

Figure 59: Delete all vertex nodes

To clean up the working graph we can remove all “Vertex” nodes in a very simple rule shown in [figure 59](#). In the end the working graph contains an informal object diagram as shown in [figure 60](#).

Figure 60: Final working graph



Until now, we merely rearranged information that already existed in the grammatical relations. The structure of the resulting object diagram already looks good, but on a closer look you will see that we made all objects instances of an “Unknown” class we invented along the way. This, together with the fact that none of the attributes we use has a type is the reason for the word “informal” in the title of this chapter. Determining more sensible classes for objects as well as attribute types will be the topic of the next chapter.

Related Work

In the Fujaba community Giese, Hildebrandt, and Seibel⁹³ implemented an interpreter for Fujaba story diagrams based on EMF models and Eclipse in 2009. They showed that using an interpreter for story diagram execution improves the flexibility in research projects by streamlining the workflow and omitting the code generation step. While our interpreter is executed in the browser we experienced the same flexibility and workflow benefits when experimenting with rules in the previous section “[From grammatical relations to object diagram](#)”.

Natural language engineers use regular expressions to match semantic relations. King and Satuluri⁹⁴ examined the use of dependency paths for this task. While the work was never published it shows that there is interest in increasing the capturing power of pattern representations indicating a semantic relation. Dependency paths, as described by Lin and Pantel,⁹⁵ already are an improvement over regular expressions. While these works are focused on Natural Language Engineering, they have one problem in common: matching pattern in graphs. Our approach uses graph transformations to match structure and regular expressions to match strings in combination with an immediate visualization of the rules as well as rule execution. The decision, which notation is more easy to grasp is left to the reader.

Conclusion

Based on the two parser wrappers we developed in “[Instant grammatical relations](#)”, developers can now execute graph transformations on part of speech trees or grammatical relations in the browser. First, we extended the story pattern DSL from “[A DSL for storyboarding](#)” on

⁹³ Holger Giese, Stephan Hildebrandt, and Andreas Seibel. “Improved Flexibility and Scalability by Interpreting Story Diagrams.” In: *ECEASST 18* (2009). URL: <http://dblp.uni-trier.de/db/journals/eceasst/eceasst18.html#GieseHS09>.

⁹⁴ Josh King and Venu Satuluri. “Extracting Semantic Relations Using Dependency Paths”. unpublished. URL: <http://www.cse.ohio-state.edu/~satuluri/final788.pdf>.

⁹⁵ Dekang Lin and Patrick Pantel. “Discovery of inference rules for question-answering.” In: *Natural Language Engineering 7.4* (2001), pp. 343–360. URL: <http://dblp.uni-trier.de/db/journals/nle/nle7.html#LinP01>.

page 62 with variable assignments and references to have a notation for graph transformation rules in our web application. The main contribution of this chapter is the Story Pattern Interpreter that can execute these rules directly in the browser. To give instant feedback to the user, we added custom color handling to the diagram renderer from chapter “Instant graph visualization” on page 61. It allows our web application to instantly visualize graph transformation rules as well as the rule execution steps.

An early prototype of our web application used hard coded graph transformations to extract objects and relations from part of speech trees. The main goal for implementing an interpreter was to replace the hard coded implementation with a more flexible solution that allows experimenting with rule changes more quickly. Being able to omit the story pattern compilation from Fujaba to Java and the Java to JavaScript compilation with GWT already shortens the deployment cycle of new code. But switching from executing hard coded rules to interpreting rules on the fly completely removes the involvement of a developer familiar with the code of our web application. Changing the parser can be done by switching to another URL and changing the graph transformations is a matter of editing them in the browser.

Outlook

Since my focus was on creating a web application that allows experimenting with story pattern for a wider audience than the Fujaba community, I concentrated on completing the technology stack. Due to time constraints, I had to move to the next step in the storyboarding process, which left room for improvement regarding the user experience. The most pressing matter I think is having to specify story pattern rules in a textual DSL. Meanwhile, JavaScript has gained a lot of attention and frameworks like AngularJS⁹⁶ bring more and more software engineering principles like dependency injection and testability to browser applications. With Canviz⁹⁷ there is an xdot renderer that runs in most modern web browsers. With Draw2D⁹⁸ there even exists a JavaScript drawing framework for Visio like drawings. The more simple task may be to exchange the currently used FUMML web service with Canviz to render diagrams directly in the browser. Replacing the text based input of graph transformation rules with an in-place editor based on Draw2D should be the goal to allow users to edit structuring rules without a media break.

⁹⁶ Google. *AngularJS. Superheroic JavaScript MVW Framework*. 2010. URL: <http://angularjs.org> (visited on 03/25/2014)

⁹⁷ Schmidt, *canviz*

⁹⁸ Andreas Herz. *Draw2D touch*. 2007. URL: <http://draw2d.org/> (visited on 03/25/2014).

Currently, there are two implementations for the storyboarding DSL. The FUMML renderer uses a parser generated with ANTLR but the web application uses a hand written parser. While there was already an ANTLR parser generator for JavaScript at the time of writing the web application it had a few bugs which stopped me from investigating the possibility further. I needed something that worked and accepted the cost of maintaining two implementations. The handwritten parser should be replaced with a proper generated JavaScript version of the FUMML renderer.

Personally, I wonder why all NLP papers use some kind of Graphviz based rendering to visualize parse trees, dependency paths or grammatical relations but fail to recognize the underlying graph nature. Especially with dependency paths, they seem to reinvent part of graph pattern matching over and over again. Maybe a group of software engineers familiar with graph theory should write a paper on the topic of NLP with graph transformations to examine the strengths and weaknesses of this approach.

Instant formal story patterns

Introduction

Following the storyboarding process from “Storyboarding by Example” on page 13 an informal story pattern is the starting point for discussion among developers to derive a class diagram. To automate this step in our web application we need to find a replacement for the discussion among developers. In alignment to the storyboarding process we need to formalize the story pattern by determining classes of objects and assigning types to all their attributes. Borrowing from Artificial Intelligence⁹⁹ and Information Retrieval communities¹⁰⁰ we will replace the discussion among developers with a simple recommendation framework.

To evolve the informal story pattern into a formal story pattern I

- created a recommender framework that fetches type information by accessing online resources and
- produces formalization rules that can be executed by the interpreter described in “Interpreting structurization pattern” on page 80 with implementations for
- a Vorname.com recommender that identifies male and female given names and
- a fallback recommender.

More sources are certainly possible, but I concentrated on these example recommenders to obtain a set of formalization rules.

Missing Type information

Our web application produces the informal story pattern shown in figure 61 for our example sentence “Alice and Bob are playing chess.” No instance specifications and no attribute types are given. This is the starting point for developers to discuss the class diagram for the story pattern. Most developers will recognize that *Alice* and *Bob* are given names and can easily infer a *Person* class for these objects. Developers knowing that chess is a game might

⁹⁹ Pattie Maes. “Agents that Reduce Work and Information Overload”. In: *Communications of the ACM* 37.7 (1994), pp. 30–40. URL: <http://www.cs.brandeis.edu/~cs125a/content/agentsmaes.doc>.

¹⁰⁰ P. Resnick and H. R. Varian. “Recommender systems”. In: *Communications of the ACM* 40.3 (1997), pp. 56–58. ISSN: 0001-0782. DOI: <http://doi.acm.org/10.1145/245108.245121>. URL: https://wiki.cc.gatech.edu/scq/qualifier/images/c/c6/Resnick-Recommendation_Systems.pdf.

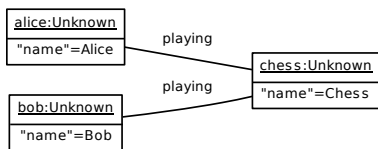


Figure 61: Informal story pattern

prefer *Player*. Since this discussion requires an opinion on how to model world knowledge that is not included in the textual scenario description, we need to find a way to add external knowledge resources to our storyboarding process. On the one hand our web application should be able to access specialized resources, e.g given names or other named entities. On the other hand we need to be able to fallback to a generic solution in case we are dealing with unknown concepts. Just assuming class “*Object*” for all instances and type “*string*” for all types will fail when trying to generate code for any type safe language.

My Idea

The internet not only contains unstructured information and cat memes but also machine readable world knowledge. As an example Google provides an API to freebase¹⁰¹ and the DBpedia, a knowledge base sourced from the Wikipedia, has a public SPARQL endpoint.¹⁰² In addition to these web services it is always possible to use information extraction on web sites that only provide unstructured data.

In the previous chapters we already developed an interpreter that executes story patterns and a set of structurization rules. Now we will create a recommender framework that uses the same graph transformation rules to provide world knowledge to our web application and create a set of formalization rules. Each recommender takes the informal story pattern as an input and calculates recommendations based on a web service like DBpedia, freebase or any other knowledge resource. The result is returned as a list of recommendations, each represented by a formalization rule that can be executed by the interpreter.

In addition to web services we will implement two special recommenders. To keep track of classes already present in other formal story pattern we can implement an existing classes recommender that has access to all derived formal story pattern of our web application¹⁰³. To handle unknown concepts we can use a fallback recommender that invents new class names as soon as they become necessary and tries to identify basic types. On their own, these two recommenders already provide the class and type information necessary to formalize the informal story pattern and generate a data model for it.

¹⁰¹ Google. *Freebase. A community-curated database of well-known people, places, and things.* 2007. URL: <http://www.freebase.com/> (visited on 03/25/2014).

¹⁰² OpenLink Software. *Virtuoso SPARQL Query Editor.* 2009. URL: <http://dbpedia.org/sparql> (visited on 03/24/2014).

¹⁰³ At this time we only handle one storyboard at a time, but the approach can easily be extended to a project or workspace wide recommender when they are added to our application.

For the initial version we will query and execute the recommenders in order. While weighing the recommendations based on a confidence level might increase the quality of the resulting story pattern formalization, we will start with a more simple approach. For now, this will make the results and as a consequence the web application more predictable.

The Details

As described in “[Preparing for liveness](#)” on page 41 we will create a property change listener for informal story pattern in our web application. It will be triggered whenever the interpreter has transformed the parser result into an informal story pattern. This new pattern will be used by the recommendation framework to calculate class and type recommendations in the form of new formalization rules.

A recommender framework

For the first version of the recommender framework I wanted to examine the suitability of story pattern as a form of representation for recommendations. As a result, the naive implementation of the `RecommenderManager` interface queries each recommender with the informal story pattern and applies the resulting formalization pattern recommendations to the story activity in the order they are received. This might cause one result to overwrite another, and is subject to HTTP timing issues when the recommenders use web services as a knowledge source. A more predictable approach is implemented in the `OrderedRecommenderManagerImpl`. It will wait for a recommender result before querying the next recommender in the queue.

The vorname.com recommender

Continuing our “Alice and Bob are playing chess.” example let us write a recommender that identifies the given names in the sentence and determines their gender. First we need to find proper nouns in the sentence. For each of them we need to determine if it is a

given name and the most likely gender of it. Finally, we want to return a formalization pattern for each given name we found that applies the information to the working graph.

Identifying proper nouns is the easiest part, because we already have a working graph with named objects. We just need to iterate over all objects and check if their name happens to be a given name. The first website where we could do that was <http://www.vorname.com/> where you can query given names for their gender, language, name day and meaning. Unfortunately, the website does not provide semantically annotated content so we have to manually extract the results.

A first review of the website reveals that querying for a name always redirects the browser to a specifically constructed URL. Alice redirects to <http://www.vorname.com/name,Alice.html> and Bob redirects to <http://www.vorname.com/name,Bob.html>. Querying a specific gender version of Kim can be done by appending `_w` or `_m` to the name.

Since the web site heavily tries to optimize their google page rank we can already find the gender in the HTML title tag as shown in [listing 17](#). Our recommender can thus just check if the title contains “Mädchennamen” or “Jungennamen” to determine the gender. In case we fetch the url for an unknown name the title seems to be broken as you can see in the third title tag in the listing. Normal visitors will be redirected to <http://www.vorname.com/index.php?keyword=<unknown name>&cms=suche&loadpage=suche> when the search for a name does not return a result. As a result, identifying an unknown name can be implemented by searching the title tag for “en namen”.

```

1 <title>Alice Vorname Herkunft und Bedeutung des Althochdeutschen Mädchennamen Namenstag Namensbedeutung</title>
2
3 <title>Bob Vorname Herkunft und Bedeutung des Althochdeutschen Jungennamen Namenstag Namensbedeutung</title>
4
5 <title>unknown Vorname Herkunft und Bedeutung des en namen Namenstag Namensbedeutung</title>

```

Listing 17: HTML title tag for Alice, Bob and an unknown name at vorname.com

Locating the title tag in the vorname.com HTML and checking if it contains any of the three strings is just a string matching problem and can even be done without a full HTML parser. If we have identified a given name we need to create a formalization pattern that represents

```

"alice"
"alice":Person"
"gender":Char':='F'

```

Figure 62: Given name recommendation

a recommendation for the working graph. When we match “Mädchennamen” for an object named “Alice” we want to tell the web application to change the type to “Person”, add a “name” attribute of type “String” with a value of “Alice” and add a “gender” attribute of type “Char” with a value of “f”. The formalization pattern representation we generate for this can be found in [figure 62](#). The list of formalization patterns is returned to our web application and executed by the interpreter.

In this section I demonstrated a very specific recommender that adapts an existing web page, creating a resource for our web application. To our surprise the web site never introduced any changes that would require us to change the way we parse the content since we started experimenting with it. We were expecting a moving target where website redesigns or adaptations in the search engine relevant parts of the HTML would require us to constantly update the parsing process. The only change was a validation of the User Agent string to prevent robots to extract content in the same way google prevents fetching web pages by stupid scripts. In contrast to vorname.com google provides an API, which was meant to provide machine readable content.

A generic JSON-P recommender

The vorname.com recommender is implemented in GWT and requires a redeployment of our web application when the code was updated. Similar to the parser I wanted to allow users to include their own web services to provide world knowledge to the storyboarding process. For this I created a generic recommender that implements the necessary browser parts of a recommender and uses FUMML to exchange data with JSON-P web services.

The necessary parts in the browser start with converting the current working graph into FUMML notation, include making the JSON-P call and finish with handling the list of FUMML encoded recommendations. At this point we can already reuse existing code to implement each step. The `FUMMLWriter` developed in “[Visualizing collaboration diagrams](#)” on page 68 already converts a storyboard to FUMML. We learned how to make JSON-P calls in “[GWT, JSON-P and a parser web service API](#)” on page 53. And parsing FUMML has been implemented in the `DSLParser` as described in “[A DSL for Storyboarding revisited](#)” on page 77. As with

the `vorname.com` recommender, handling the recommended rules is then the responsibility of the `RecommenderManager`.

A Freebase Recommender in PHP

Having a generic implementation for the browser part of arbitrary recommenders still leaves the task of writing actual recommender services. To demonstrate that they are language-independent we will use PHP to query googles freebase API and determine the class for any unidentified objects in the working graph.

After registering for an API key freebase can be queried up to 100.000 times in 24h. The content can be used under the Creative Commons Attribution (CC-BY) license¹⁰⁴ which allows us to share and remix the data as long as we credit the Freebase community appropriately.¹⁰⁵ With the legal requirements out of the way we can query the webservice using the Search API. We could also query Freebase using the Topic API to fetch all known facts for a given topic or using the Metaweb query language (MQL), but since we are trying to identify the class of an object the search API is the best fit.

Based on the PHP version of the search API example¹⁰⁶ our freebase recommender implementation fits in under 100 lines of code of which we will show a few excerpts to give an idea of how the class recommendation for an object is calculated. The first thing we had to add was a way to extract the names from the FURL encoded working graph. Instead of writing a full fledged FURL parser in PHP I took a shortcut and just used a regular expression to match object names as shown in [listing 18](#).

```

9 $fuml = $_GET['fuml']; // read url parameter
10 preg_match_all('/\[("[^"]+)\]/', $fuml, $matches);
11 $names = array_unique($matches[1]);

```

¹⁰⁴ Creative Commons. *Attribution 3.0 Unported*. 2007. URL: <http://creativecommons.org/licenses/by/3.0/> (visited on 03/25/2014).

¹⁰⁵ Google. *How to Attribute Freebase on Your Site*. 2014. URL: <http://www.freebase.com/policies/index> (visited on 03/25/2014).

¹⁰⁶ Google. *Search Overview. Freebase API*. 2014. URL: <https://developers.google.com/freebase/v1/search-overview> (visited on 03/25/2014).

Listing 18: PHP: extract names from FURL. The regular expression in line 10 matches names enclosed by double quotes following the opening square bracket: matching `["alice":"Person"]-playing-["chess":null]` will fill the `$names` array with `alice` and `chess`.

¹⁰⁷ Google. *Search Cookbook. Freebase API*. 2014. URL: <https://developers.google.com/freebase/v1/search-cookbook> (visited on 03/25/2014).

As you can see in [listing 19](#) we make an individual query to the freebase search API for each unique name. According to the Search API Cookbook¹⁰⁷ we filter all things known to freebase that have a name property matching our object name. Since we are trying to determine a class for the object we specify the output to contain type information. This will give us a list of types for every name, so we will aggregate the types of three results to calculate a more general result.

```

22 $service_url = 'https://www.googleapis.com/freebase/v1/search';
23 $rules = array(); // initialize empty array for rules
24
25 foreach ($names as $name) {
26     $params = array(
27         'filter' => '(all name:"'.$name.'")', // all things that match our name
28         'output' => '(type)', // give us extra type information
29         'limit' => 3, // we will aggregate up to three results later
30         'key' => $API_KEY ); // works without for debug purposes
31     $url = $service_url . '?' . http_build_query($params); // build query URL

```

Listing 19: PHP: build freebase query

Each result is a JSON object containing a score as well as a list of types along with their id and name. On the one hand we have to find the name of each type deep in the tree, on the other hand we can use the score to calculate a more general score. Results are ordered by score and types are ordered by notability as well, so we can weigh earlier results higher. With this in mind the two foreach loops in [listing 20](#) calculate a score for every type by weighing them accordingly.

```
40 //aggregate multiple results
41 $types = array();
42 $resultno = 0;
43
44 foreach($response['result'] as $result) {
45     $typeno = 0;
46     foreach ($result['output']['type']['/type/object/type'] as $type) {
47         if ($type['id']=='/common/topic') { // ignore common topic type
48             continue;
49         }
50         if (!isset($types[$type['id']])) {
51             $types[$type['id']] = array('name' => $type['name'], 'score' => 0);
52         }
53         $weight = (1/++$typeno)*(1/++$resultno);
54         $types[$type['id']]['score'] += $result['score']*$weight;
55     }
56 }
57
58 // sort by score
59 uasort($types, "sort_by_score");
```

Listing 20: PHP: calculate score for types

With the best type determined by our calculated score we can recommend a Class for each object name in the working graph. Listing 21 shows that we simply pick the type with the highest score, create a CamelCase version of the name and then express our recommendation in FUML.

```

64 $type = reset($types); // get first element of array
65
66 // to create CamelCase upcase first letter of every word
67 $words = ucwords(strtolower($type['name']));
68 // and remove whitespace
69 $class = str_replace(' ', '', $words);
70
71 $score = $type['score'];
72
73 $rule = '['.$name.':null;"_type":="'.$class.'"'];
74 $rules[] = array('fuml'=>$rule, 'score'=>$score); // collect results

```

Listing 21: PHP: encode recommendation in FUML. To update only objects without a class the formalization rule created in line 73 only changes the class name if none is present. In line 74 we add the generated rule to our `$rules` array.

To return a JSON encoded list of recommendations that our generic JSON-P recommender understands we have to wrap them in the JSON-P callback, which is a matter of three lines of code in PHP as shown in listing 22.

```

82 // send as JSON-P
83 header('content-type: application/json; charset=utf-8');
84 $json = json_encode($rules);
85 echo $_GET['callback'] . '('. $json . ')';

```

Listing 22: PHP: send JSON-P encoded list of recommendations

For our *Alice and Bob are playing chess* example the freebase recommender will recommend the class `Game` for *chess*. Our web application will render this recommendation as shown in figure 63.

While the freebase recommender works for our simple example it is sometimes too specific for proper nouns and too generic for nouns. On the one hand freebase will identify *Alice* as *Alice Cooper* and *Bob* as *Bob Marley*. Their most notable type is `MusicalArtist`, which might be correct but is too specific for our use case: identifying actors in a textual scenario. On the other hand the recommender will return `AutomotiveClass` for *car* and `MassTransportationSystem` for *bus*. Both too generic classes, when in these cases the CamelCase version of the noun would have been sufficient. The best way of identifying the class for a noun might need more thought but we will defer this discussion until the conclusion of this chapter.

A fallback recommender

Instead of examining yet another recommender that can only add knowledge from a specific domain we will end the details of this chapter with a fallback recommender. In contrast to the other recommenders it is always executed as the last recommender. It will complete any missing type and class information to ensure a formally correct story pattern.

A straight forward implementation just iterates over all yet unclassified objects and invents a class for each of them: `UnknownClass1`, `UnknownClass2` and so on. If we do not clean up the working graph after applying the structurization ruleset and leave the NLP parsers Vertex nodes in place we can improve the fallback class name based on the part of speech tag. All non proper nouns, singular (NN) or plural (NNS), can be CamelCased to derive a meaningful classname: *chess*, *car* and *lecture* become `Chess`, `Car` and `Lecture` respectively. This leaves proper nouns that we have to invent a class for. Since the vorname.com recommender already identifies named actors in the textual scenario this should however seldom be necessary.

Attribute types can be guessed by examining their values. We implemented the identification of basic types like `null`, boolean `true` and `false`, integers, doubles and strings. Since our app currently only has Java as the target language we also try to match colors and, if

```

"chess":null
"chess": "Game"
  
```

Figure 63: SP: chess is a game

successful, create a `color` attribute of type `java.awt.Color` with the correct color value. Using class names from the Java Class Library is a target platform specific assumption and we will present ideas on how to improve this in the outlook section of this chapter.

Related Work

Using online resources to look up named entities on the fly already became popular with linguists before I started working on the recommender framework used in our web application. Named entity disambiguation with online resources has been described and implemented in the AIDA online tool¹⁰⁸ by Yosef et al.¹⁰⁹ With freebase, Yago and DBpedia AIDA uses an impressive set of resources and even tries to correlate the individual entities. The accurate results come at a performance cost that we are trying to avoid for instant storyboarding.

¹⁰⁸ D5: Databases and Information Systems. *AIDA Web interface (aida)*. Max-Planck-Institut Informatik. 2011. URL: <https://gate.d5.mpi-inf.mpg.de/webaida/> (visited on 03/25/2014).

¹⁰⁹ Mohamed Amir Yosef et al. "AIDA: An Online Tool for Accurate Disambiguation of Named Entities in Text and Tables". In: *PVLDB* 4.12 (2011), pp. 1450–1453. URL: <http://dblp.uni-trier.de/db/journals/pvladb/pvladb4.html#YosefHBSW11>.

Conclusion

In contrast to other named entity recognition solutions our recommender framework is very simple but it quickly produces results that can be used to add world knowledge to our working graph. The vorname.com recommender together with the fallback recommender already produce formalization rules to create an acceptable formal story pattern. As already mentioned in "[A Freebase Recommender in PHP](#)" adding online resources can improve the results even further. I did not invest a lot of time to write the freebase recommender because it should just demonstrate the usage of online resources. There is room for improvement.

Outlook

The whole recommender framework was written with Java as the target platform in mind. As a first step the fallback recommender could be split into a platform agnostic fallback recommender and a Java recommender that rewrites type and class names. This would allow writing other target platform recommenders such as Python, C#, PHP or JavaScript.

The freebase recommender is just a quick hack that was developed in two evenings. It might give better results when adding more search properties or limiting the results to more specific types. Before investing time, interested developers should go through the list of academic papers that has been collected at the freebase wiki.¹¹⁰

Each recommender adds complexity the current `RecommenderManager` was not meant to handle. Experimenting with various recommenders requires a few improvements to the UI. A list of recommenders could visualize the order of recommenders and allow the user to change it and even deactivate individual recommenders. At the latest a new target platform recommender will make it necessary to add this kind of UI.

The minimal freebase recommender has already shown that it is possible to calculate a score for every recommended rule. In a similar way that natural language parsers provide more than on parse the recommender manager could be extended to weight the different recommendations and only apply rules above a user defined threshold. Multiple results with different scores would enable the manager to calculate alternative formal storyboards: A second or third best solution.

Another thing that came to my mind when comparing ontology learning with the object diagram rewriting I presented in the last two chapters is the suitability of graph transformations for open world systems. Ontologies can be used for automated reasoning to infer new rules from an existing set of rules. Instead of using description logic graph transformations could be used to work on ontologies. The approach might be more visually accessible than formal notations.

¹¹⁰ Google. *Research. Papers about Freebase.* 2010. URL: <http://wiki.freebase.com/wiki/Research> (visited on 03/25/2014).

Instant storyboards

Introduction

In the previous chapters we put together the basic building blocks of a storyboard. Our web application can create formal story patterns for textual scenario descriptions of storyboard steps. But a storyboard contains at least two steps: a start situation and an end situation. We need to make sure the object classes we derived in the first story pattern will be reused in subsequent pattern or else *Alice* might become a `Player` in the start scenario and a `Human` in the end scenario. Furthermore, we need to identify an event in the start scenario to complete our storyboard with a collaboration statement.

To derive sound storyboards in our web application I

- implemented a `KnownClassesRecommender` that recommends Class names based on the existing formal story pattern of every storyboard step,
- extended the existing structurization rule set to cover new grammatical relations,
- added collaboration statements to the FUML notation and renderer,
- updated structurization rules that rewrite a Stanford parse into an informal story pattern to differentiate gerunds from other verb forms¹¹¹ and use them as collaboration statements.

Together, these changes will ensure the derived storyboard can be used to generate a unit test.

Two story patterns do not yet make a storyboard

To demonstrate the problem arising from an object class changing between storyboard steps we can extend our example sentence as shown in [listing 23](#).

For the start scenario the Stanford parser produces the grammatical relations shown in [figure 64](#). With our current structurization rules the informal story pattern will omit the second sentence completely because we do not yet match `VBG` vertices or `prep_to` relations. A closer look at the `prep_to` relation also reveals two nouns with a `nn` (noun compound modifier¹¹²)

¹¹¹ Wikipedia. *Uses of English verb forms*. 2012. URL: http://en.wikipedia.org/wiki/Uses_of_English_verb_forms (visited on 03/25/2014).

¹¹² The complete list of grammatical relations can be found in the manual. Marie-Catherine de Marneffe and Christopher D. Manning. *Stanford typed dependencies manual*. Sept. 2008. URL: http://nlp.stanford.edu/software/dependencies_manual.pdf (visited on 03/25/2014)

```

1 Start scenario:
2 Alice and Bob are playing chess.
3 Alice moves her pawn to field e4.
4
5 End scenario:
6 Alice's pawn is on e4.

```

Listing 23: Extended chess scenario

relation: *e4* and *field*. In this case the Stanford parser considers *e4* to be the head noun of the noun phrase and *field* to modify the head noun. To add the facts expressed in the second sentence we need to add new structurization rules to represent the facts in an informal story pattern accordingly.

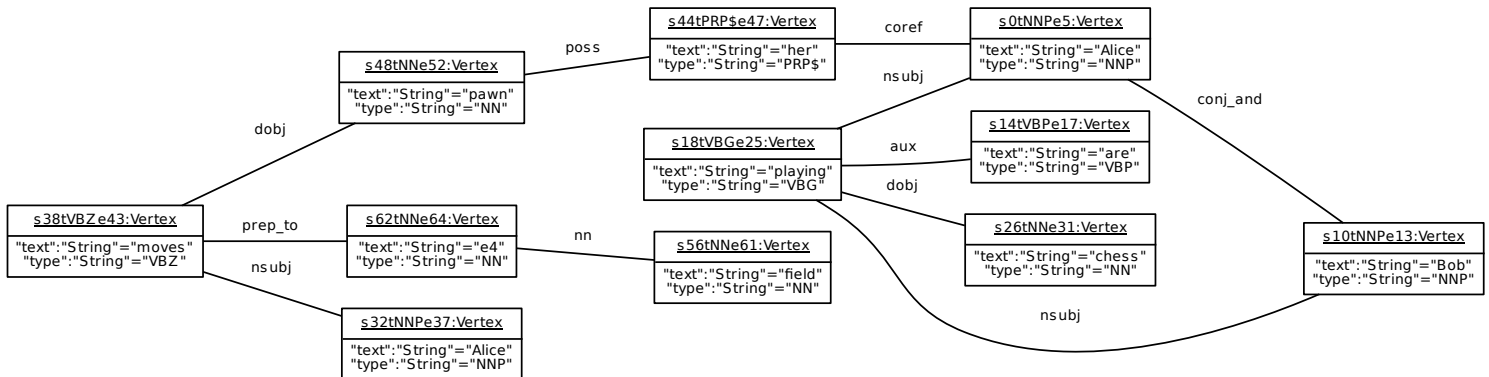
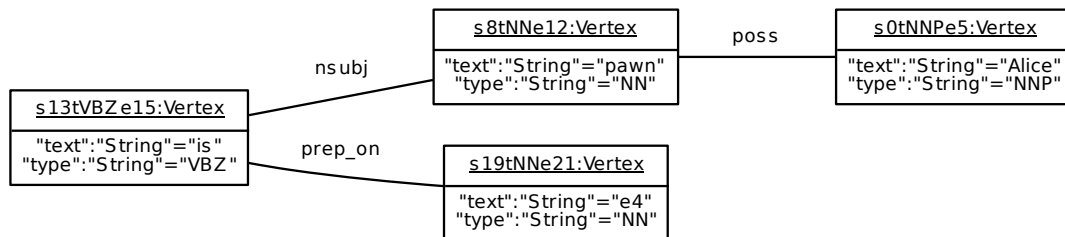


Figure 64: Grammatical relations for the start scenario

In the end scenario the noun modifier *field* has been omitted, which causes the fallback recommender to recommend `E4` as the class for *e4*. In the start scenario it derives the class from the noun phrase *field e4*, using CamelCase to recommend `FieldE4`. Our current rules also only use the head noun of a noun phrase to derive an object identifier. This leads to a

formal inconsistency: in the start and end pattern the object will be named `e4` but it will be an instance of `FieldE4` in the start scenario and of `E4` in the end scenario.

Figure 65: Grammatical relations for the end scenario



While missing objects in the start pattern do not harm the consistency of the storyboard, deriving different classes for multiple occurrences of an object does. We need to find a way to handle these inconsistencies and add structurization rules that will capture the facts described in the second sentence of the start scenario.

My Idea

In our web application all scenario steps and their story patterns can be accessed in the client. Furthermore, the order of recommenders is currently predetermined because it has been hardcoded in the `RecommenderManagerImpl`. To stop the `FallbackRecommender` from recommending conflicting classes I will introduce a `KnownClassesRecommender` that uses all existing formal story pattern of a storyboard to recommend an already identified class name for an object instance. It will be queried before the `FallbackRecommender` has a chance to recommend any undetermined classes.

Capturing more facts from the grammatical relations in an informal story pattern can be done by going through the list of possible grammatical relations and creating graph transformation rules for each of them. Since most of them do not occur in our example sentences I took the less time consuming route and created structurization rules on the fly whenever I

came upon a new grammatical relation in the parse result. For the extended example scenario I created rules for the new grammatical relations `nn`, `prep_to`, `prep_on`, `poss` and `coref`.

The Details

A known classes recommender

Our web application uses a property change mechanism to trigger the different steps of the storyboarding process. When the informal story pattern of a scenario step changes the `RecommenderManager` queries any available `Recommenders` and applies the recommendations to create a new formal story pattern. To reuse classes already determined for an object our application needs to identify the same object across story pattern and assign the same class before any other recommender comes up with a different class.

At the time of writing, the `RecommenderManager` implementation already queried the fallback recommender last. To ensure the necessary order of recommenders I only needed to update `RecommenderManager` to always query the `KnownClassesRecommender` first. Deriving recommendations for existing classes is then a matter of iterating over all objects in the working graph, the new informal story pattern, and searching all formal story pattern of the storyboard for objects with the same name. If we find a matching object we can recommend the same class. The class will not be overridden by subsequent recommendations because they only match objects not having a class, yet.

Extending the structurization ruleset

Adding a second sentence to our example produced a lot of new relations in the Stanford parse. Readers familiar with the Stanford parser will recognize the `coref` relation linking together the grammatical relations for the two sentences. The coreference resolution links `Alice` and `her` and allows us to recognize that they are actually the same object.

The next grammatical relation we will examine is also related to nouns. The noun compound modifier occurs when a noun phrase consists of more than one noun. We will use the

head noun to derive the object name for the informal object diagram. In the formalization step the recommenders can then take into account any origin links that make up this object to recommend a class name.

We still do not map any prepositions to the informal story pattern. In *Alice moves her pawn to field e4*, the Stanford parser identifies a `prep_to` proposition which we should match to capture the existence of *e4* in the informal story pattern. Similar to the `dobj` and `nsubj` relations this preposition also is linked to the verb in the sentence. We can thus extend the regular expression in our first structurization rule to also match some prepositions as shown in [listing 24](#). The corresponding visual representation is given in [figure 66](#)

Listing 24: Structurization Rule: also match prepositions

```

1 # Create a new object in the diagram for each sentence subject
2 [(o1):"Vertex"|"type"==/(VBG|VBZ)/]↔
3 ↔/(nsubj|dobj|prep_to|prep_on)/↔
4 ↔[(o2):"Vertex"|"type"==/(NNP|NN)/;"text"==(noun)]↔
5 ↔,↔
6 ↔[(o2)]↔
7 ↔-+"origin"?↔
8 ↔[+noun.toLowerCase():"Unknown"?|"name":=noun]

```

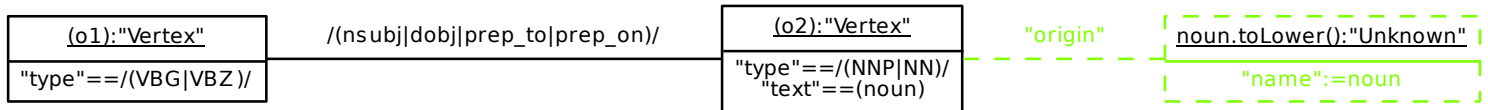


Figure 66: Structurization Pattern: also match prepositions

Why did we not add more prepositions like `prep_over`, `prep_under` or `prep_beside`? These three already show that there are prepositions that carry a special meaning we might want to capture for the story pattern. For these three a *location* link would be suitable. But I stopped after adding `prep_to` and `prep_on` to our story pattern rule to continue with the other yet unmatched grammatical relations.

The last simple addition to our structurization ruleset concerns the `poss` relation between `pawn` and `her`. As an example for capturing special meaning I added a new rule to map this relation to an `owner` link as shown in [listing 25](#) and [figure 67](#) respectively.

```

1 # ownership
2 [ (v1 : "Vertex") - "poss" - [ (v2 : "Vertex") ], ↔
3 ↔ [ (v2 : "Vertex") - "coref" - [ (v3 : "Vertex") ], ↔
4 ↔ [ (v1) - "origin" - [ (o1) ], ↔
5 ↔ [ (v3) - "origin" - [ (o2) ], ↔
6 ↔ [ (o1) ] -+ "owner" ?- [ (o2) ]

```

Listing 25: Structurization Rule: ownership

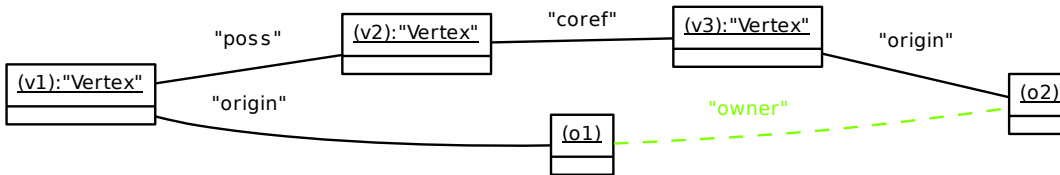


Figure 67: Structurization Pattern: ownership

You might have noticed that the rule tries to match a `coref` relation in combination with a `poss` relation. To create a link we need a source and a target object but the `poss` relation only links the noun to a possessive pronoun. To resolve the pronoun to an actual noun we can follow the `coref` relation. In our example this results in *Alice* being the owner of the *pawn*.

Adding collaboration statements to FUMML

After adding rules for the new grammatical relations we now only need to identify an event in the textual scenario that we can map as a collaboration statement. Before we can do that we need to extend our FUMML renderer to parse and render collaboration diagrams.

Extending the parser for the FUML service is done with the *call* element that links a *method* to an *object* as shown in the syntax diagram in figure 68.

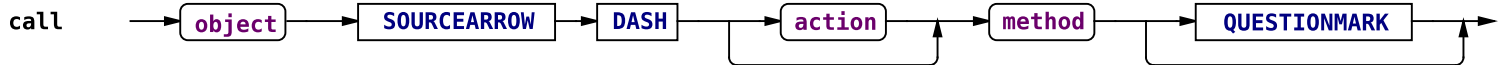


Figure 68: Call syntax for FUML notation

With the notation in place we can move forward to rendering a collaboration statement with Graphviz. Since Graphviz only knows *node*, *edge* and *graph* properties we will create a special collaboration statement node that has no border, shows the collaboration statement and uses a directed edge to the object receiving the message (see listing 26).

Listing 26: JMTE template: collaboration statement

```

1  ${foreach messages node}
2    ${node.id} [ color="transparent",
3      ${if node.fillcolor}fillcolor="${node.fillcolor}",${end}
4      ${if node.style}style="${node.style}",${end}
5      label=
6        <<table valign="top" color="transparent" border="0" cellpadding="0"
7          cellspacing="0" cellpadding="5" width="10" height="10">
8          <tr><td>${node.name}${: ,node.type,}
9            ${if node.newNameOrType}
10             <br/>
11             ${if node.newName}<font color="chartreuse">${node.newName}</font>
12             ${else}${node.name}${end}
13             ${if node.newType}:<font color="chartreuse">${node.newType}</font>
14             ${else}${: ,node.type,}${end}
15           ${end}
16         </td></tr>
17       </table>>
18     ]
19   ${end}

```

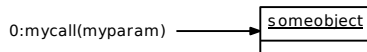


Figure 69: Example collaboration statement

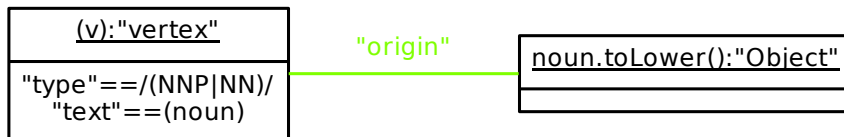
With the template in place the FUML service now creates the diagram shown in figure 69 for `[_someobject_] <-0:mycall (myparam) .`

Structuring the data model

With the extension of the renderer we can use a declarative notation to visualize collaboration statements in story patterns. Now, we also need to make the interpreter aware of collaboration statements. While we could use ANTLR to generate a parser for the extended version of the FURL notation an interpreter generator has yet to be written. But we can add collaboration statements to our working graphs without even writing a single line of code for the story pattern interpreter. Instead of the interpreter we are going to change the structurization rules.

Until now, all the structurization rules we presented were designed to work on the same hierarchical level as the working graph. A link is represented as a link, an object as an object and a collaboration statement as a collaboration statement. As the interpreter only works with links and objects we cannot work with story pattern directly, but we can work with the data model of a story pattern, because it is made only of links and objects: a link becomes an object of type `Link`, an object becomes an object of type `Object` and a collaboration statement become an object of type `CollabStmt`. This shift of focus from story pattern to story pattern data model also introduces objects of type `Attribute`. Figures 70 to 80 show the updated structurization rules.

Similar to the old structurization ruleset we start searching for nouns in the grammatical relations and create a data model element representing an object for each occurrence. This time we model the step with the three rules in figures 70 to 72 to differentiate between nouns and proper nouns.



After creating the objects for nouns we are going to aggregate compound nouns. As shown in figures 73 to 74 they can be identified with the `nn` grammatical relation. To aggregate the compound nouns in the right order we also match a third vertex in the grammatical relations that indicates which of the nouns is the first. Both rules delete the individual objects for the compound noun, create a new object with the aggregated name and reattach the `origin` links to allow lookup of the underlying grammatical relations by subsequent rules.

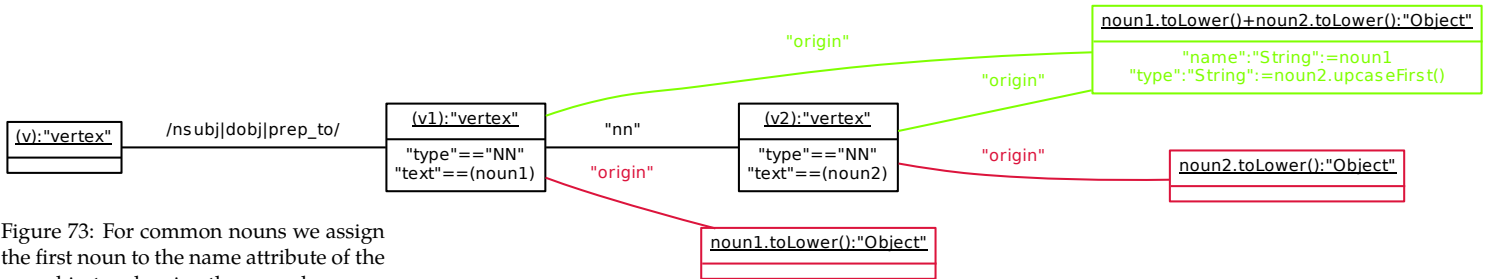


Figure 73: For common nouns we assign the first noun to the name attribute of the new object and assign the second noun as the type attribute of the new object.

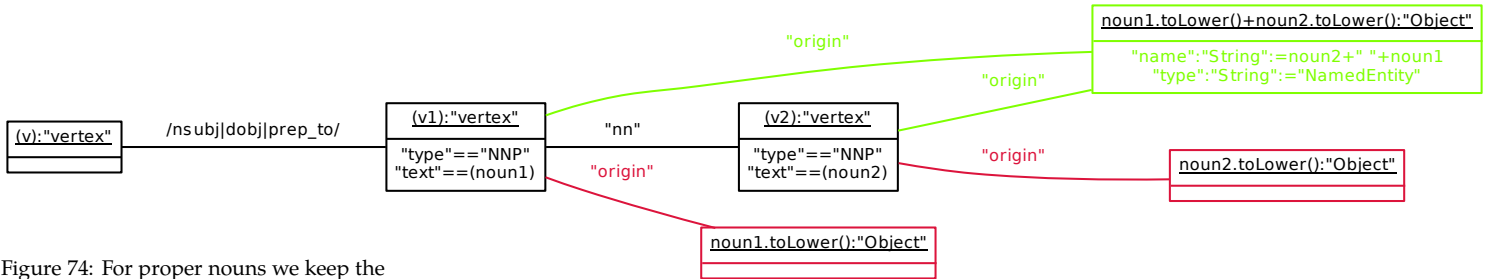


Figure 74: For proper nouns we keep the `NamedEntity` type attribute. We reverse the order of the noun vertices as they are matched in the grammatical relations to create a readable name attribute.

With the aggregated objects for compound nouns in place we will continue with the three rules in figures 75 to 77. The first adds attributes to the objects that were created. The second creates an `origin` link to the preposition of a coreference relation. The third creates an `owner` link for the possession modifier in the grammatical relations. In combination, they allow identifying and modeling ownership relations across sentence borders.

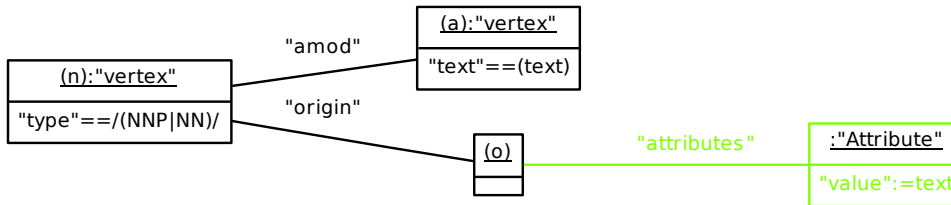


Figure 75: Create an `Attribute` element in the data model for attribute modifiers in the grammatical relations.

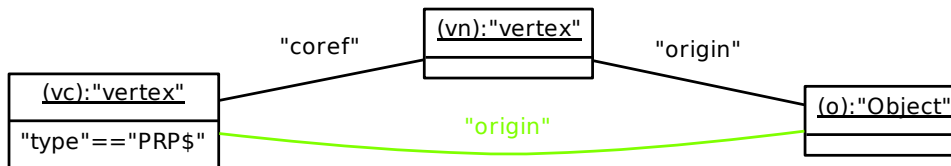


Figure 76: Creates an `origin` link for coreference relations.

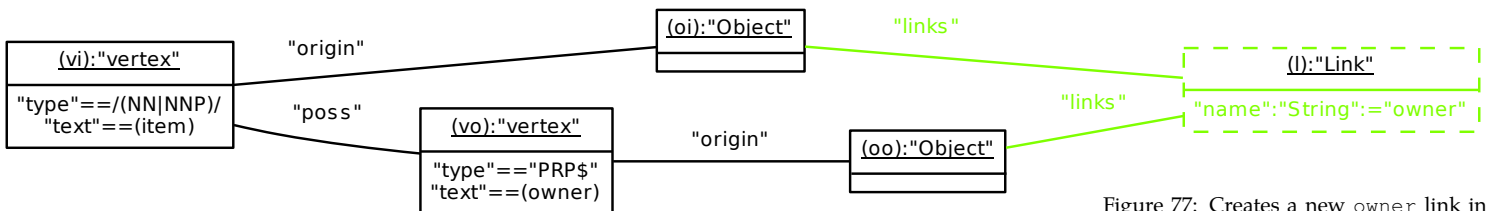


Figure 77: Creates a new `owner` link in the data model to represent the ownership identified by a `poss` relation in the grammatical relations.

The new structurization rule in [figure 78](#) for the second storyboarding formalization step is very similar to the old rule in [figure 57](#) on page 93: it matches the same mandatory elements but now creates a `Link` object in the data model. Furthermore, [figure 79](#) shows the new rule to create collaboration statements in the data model. It is followed by the rule in [figure 80](#) which adds parameters to the collaboration statement before we are finished and just remove the vertexes from the working graph with the old but still valid rule in [figure 59](#) on page 94.

Figure 78: We need to match a subject, predicate, object triple and the objects that have been created for the nouns. Then we create a new `Link` object in the data model.

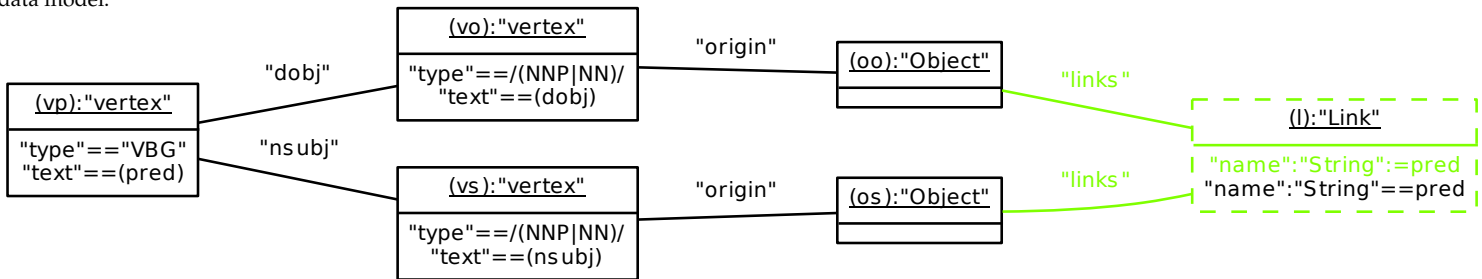
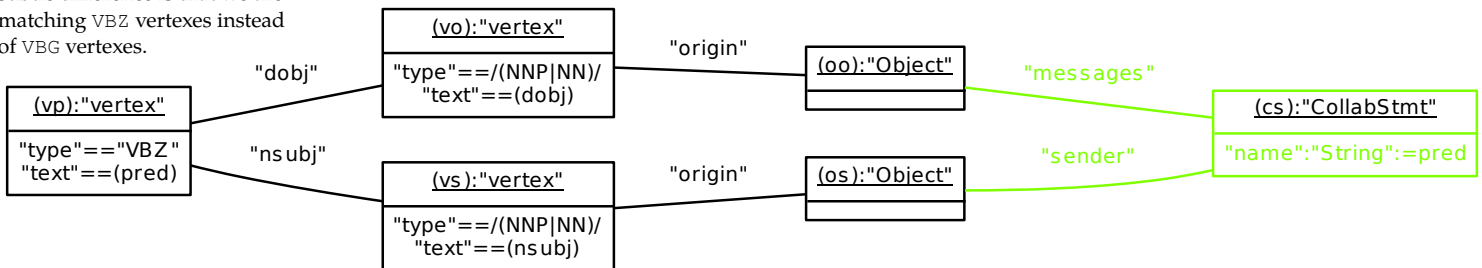


Figure 79: The rule to create collaboration statements looks nearly identical to the rule that creates `Link` objects in the data model. The major difference is the creation of a `CollabStmnt` object. The more subtle difference is that we are matching `VBZ` vertexes instead of `VBG` vertexes.



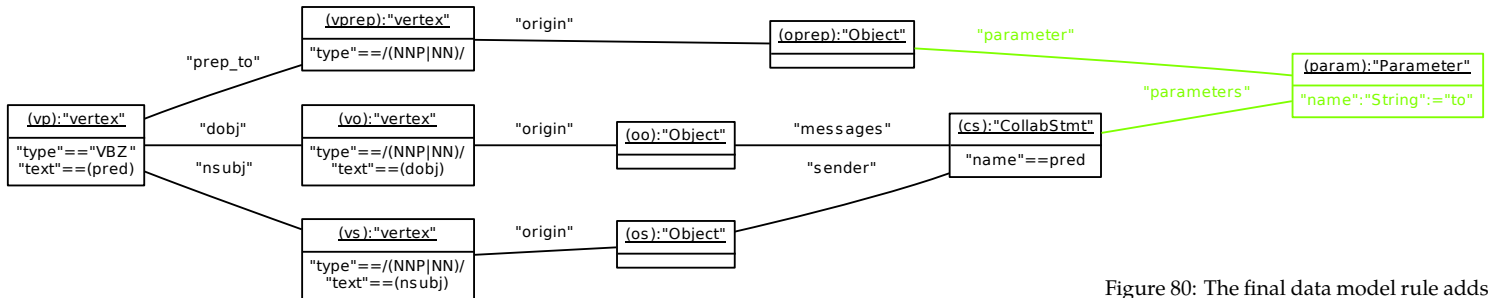
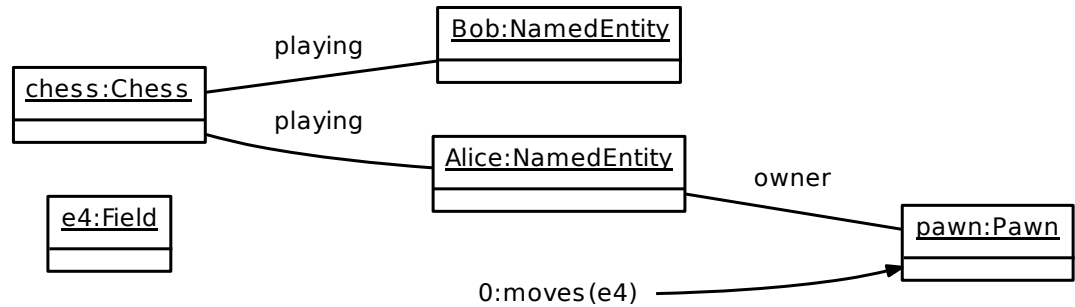


Figure 80: The final data model rule adds parameters to the collaboration statement. It not only needs to match the same elements as the previous rule but also tries to match a preposition that will then be used to model a `Parameter` object for the collaboration statement.

The attentive reader will have recognized that we are using VBZ (Verb, 3rd person singular present) vertex nodes to identify an event. We are assuming that a scenario mostly describes what is happening in present participle. An event then uses the present tense to indicate who does what. This is of course an artificial assumption but we found that for the purpose of identifying an event in a textual scenario it either works out of the box or makes the writer of the scenario description rephrase his words to make it work. Which in turn educates him to adhere to a more simple and structured writing style, which not only makes the text easier to understand for the natural language parser, but also for other users.

The resulting data model can now be used to generate a story pattern in FUMML notation that also uses the new collaboration statement notation. For that we created a new `FUMML-Writer` that expects the working graph to contain the object types we introduced in the data model rules above: `Object`, `Attribute`, `Link`, `CollabStmt` and `Parameter`. Depending on these types the writer produces the corresponding FUMML notation for a complete story pattern that we then render with the FUMML service. Together with the above rules and our example sentence this results in the informal object diagram shown in [figure 81](#).

Figure 81: The informal diagram now contains a collaboration statement where *Alice* sends the *pawn* a message to *move* to *field e4*. The collaboration statement currently uses *moves*, the wrong verb form, which we will address with a stemming formalizer later.



Conclusion

To close the gap between formal but unrelated story pattern and a consistent storyboard we made the individual story pattern aware of each other and refined the structurization rules we developed earlier. The `KnownClassesRecommender` is an essential improvement for the storyboarding process as it ensures that objects keep their class throughout all steps. By extending the rule set to cover new grammatical relations I was able to demonstrate how to capture more concepts from the grammatical relations. Especially a rule to identify a concept that can be used as the collaboration statement in the start step is necessary to complete the storyboard. In combination our web application can now derive sound storyboards that can be used as the basis for a unit test.

Outlook

While the `KnownClassesRecommender` uses a story pattern to match existing classes the access to formal story pattern of other storyboard steps has been hardcoded. Replacing this hardcoded bridge with an API that can be accessed from within a formalization pattern would allow all recommenders access to the whole storyboard. This gives the formalization recom-

menders a much bigger working graph to work with and might further improve their results.

Although we extended the structurization rule set to cover some prepositions and added a rule that creates the essential collaboration statement we still only capture a fraction of the English language. As mentioned earlier in “[Extending the structurization ruleset](#)” on page 113 `prep_to` and `prep_on` are not the only prepositions. The results of our web application could be gradually improved by composing structurization rules that capture the meaning of additional grammatical relations.

Adding additional rules however might require a more sophisticated handling for them. While the current plain text field can be used to quickly comment a specific rule or use traditional copy and paste to enhance an existing rule a recommender approach similar to the formalization step could be used to dynamically decide which set of structurization rules to apply to grammatical relations.

I also noticed that some formalization recommenders benefit or even require the grammatical relations to still be present in the working graph to create useful recommendations. As an example the `FallbackRecommender` can concatenate nodes with a `nn` relation to recommend a more specific CamelCased class name. Not removing the parser result from the working graph however will lead to parser specific formalization recommenders. The current separation of informal and formal story pattern is a result of trying to solve one problem at a time. Again, the solution might be to merge the two steps and combine a set of recommenders with a yet to develop `ChainedRecommenderManager` that dynamically decides when to execute a specific recommender. An initial approach could separate the recommenders into structurization and formalization recommenders and wait for all structurization recommenders to complete before querying formalization recommenders. This would simulate the current two step approach and could evolve into a more diverse strategy in combination with recommendation scores.

Distinguishing a collaboration statement from a link is currently based on verb tense. This indirectly represents a key requirement for writing textual scenario descriptions. While this will not be a revelation to most software engineers a missing collaboration statement also prevents the generation of a unit test. In these cases adding a fallback collaboration statement would improve the user experience.

Instant acceptance tests

Introduction

The XProM2 plugin for Fujaba generates executable JUnit tests from storyboards. Unfortunately, the functionality is hidden in the Fujaba standalone application and lacks a web service. Forcing a user to run Fujaba, somehow import the storyboard and finally generate the acceptance test breaks the workflow and completely nullifies the instant storyboarding experience we try to demonstrate with our web application. Still, the final step in the storyboarding process needs to deliver an executable acceptance test.

To wrap the XProM2 plugin and Fujaba as a web service I

- created a headless version of Fujaba with the necessary XProM2 and CodeGen2 plugin,
- created webservice wrapper around it with GWT and
- added the UI to access the Fujaba project repository and source code.

Integrating this service into our web application allows users to download an executable JUnit test for the storyboard derived from the textual scenario descriptions.

The Problem

When a software developer identifies a new Use Case he can use Fujaba to model a storyboard for it. At the end of the storyboarding process he can use the XProM2 plugin to derive an acceptance test for it by generating a JUnit test case. While our web application focuses on instant storyboarding by automating and visualizing the process in the browser we still need to deliver this acceptance test to be on par with the XProM2 functionality.

When a developer chooses to use our web application to model a storyboard he should be able to continue working with the result in the tool he prefers. Fujaba is the only other tool capable of modeling storyboards so we need to provide an easy import of our result. While the most generic form of the result is an executable Java jar containing the sourcecode

generated by the XProM2 plugin for the classes as well as the JUnit test we will provide the source code as well as a Fujaba project repository in `ctr` format.¹¹³

The code to generate a JUnit test from a Fujaba storyboard is hidden in the XProM2 plugin. Our web application now needs a way to access it. To achieve this we need a way to use a headless¹¹⁴ version of the Fujaba tool suite as a service, similar to the natural language parsers in section “GWT, JSON-P and a parser web service API” on page 53.

My Idea

Deploying Fujaba and the XProM2 plugin as a web service requires a headless version of the code generation pipeline that does not require a graphical user interface. We basically want to send a UML model containing the storyboard to the server, run the code generation for the JUnit test there and return the resulting source code to the browser. Variations of this might return a Fujaba `ctr` or an executable jar.

The Fujaba maven plugin already contains code to generate code from Fujaba projects in a maven step. We can reuse the approach taken there and extend it with the XProM2 plugin to create a headless version of the necessary code generation.

Using GWT we can seamlessly transport the storyboard model from the browser to a web service that wraps the headless code generation. As a result we will return a timestamp that is used to construct a URL under which the generated sources and Fujaba project repository can be downloaded.

Running Fujaba headless

Fujaba 5 has been developed with a Desktop application in mind. Unfortunately, this has led to the dependency on a graphical user interface for some tasks. Especially displaying error logging and creating classes is, in early 2013, still tightly integrated with the UI.

With the Maven2 Fujaba Plugin¹¹⁵ developers can add a Fujaba code generation step to their headless maven builds. Similar to that plugin we need to set up the Fujaba preferences, initialize the factories used to create the UML model and initialize the code generation. In

¹¹³ Christian Schneider. “CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten”. PhD thesis. 2007. URL: <http://kobra.bibliothek.uni-kassel.de/handle/urn:nbn:de:hebis:34-2007121319874>, p. 142.

¹¹⁴ There currently is no command line interface to Fujaba that could be used to integrate it into shell scripts.

¹¹⁵ Manuel Bork. *Maven2 Fujaba Plugin 2*. Software Engineering Group Kassel. 2007. URL: <http://www.se.eecs.uni-kassel.de/~maven/sites/mvnFujabaPlugin/dependencies.html> (visited on 03/25/2014)

addition we also initialize the XProM2 plugin and create a wrapper that takes a Fujaba UML-Project and a destination path and generates the files with CodeGen2.

The hard part is not writing a few lines of code, but bundling a web application with all the dependencies in the correct places. CodeGen2¹¹⁶ uses velocity as a template engine and needs the templates somewhere on the classpath. Without changes to CodeGen2 or the classloader used by it velocity will not be able to access the templates when they are hidden in jars. As a quick solution I created a new source directory called `tplsrc` and copied the templates there. This way they will be deployed like other Java class files, ready to be used by velocity. If necessary a developer with access to the server can even change the templates without having to redeploy the web application. Which is one of the main reasons for using templates.

¹¹⁶ Leif Geiger, Christian Schneider, and Carsten Reckord. "Template- and modelbased code generation for MDA-Tools". In: *3rd International Fujaba Days*. Paderborn, Germany, Sept. 2005. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/CodeGen2.pdf>.

Exchanging storyboards with GWT-RPC

While the storyboard UI of the web application uses a property change listener pattern to update subsequent steps in the storyboarding process I decided to only trigger the code generation when the Generated Code tab is selected. Instant Storyboarding is meant to be visual and the code generation only serves to prove the formal correctness of the storyboard and create a Fujaba project repository that can be imported to Fujaba.

Some of the recommenders already use GWT-RPC to transfer the storyboard to a recommender implemented as a web service. We will use the same approach to transfer the final storyboard from our web application to the Fujaba web service. GWT allows us to reuse the data model on the server side, where we can then create a Fujaba project from the storyboard.

Wrapping Fujaba as a Webservice

With the storyboard on the server side we now need to instantiate a data model expected by the Fujaba code generation. Looking at the XProM2 implementation we can see that it is responsible for the code generation of storyboards. It iterates over all Story Activities and creates the necessary story pattern to set up the object world in the start scenario, execute the collaboration statement and check the object world according to the next scenario step.¹¹⁷

¹¹⁷ Details of the implementation have been described by Geiger in his PhD thesis Leif Geiger. "Fehlersuche im Modell: modellbasiertes Testen und Debuggen." PhD thesis. University of Kassel, 2011. URL: <http://d-nb.info/101373873X>

CodeGen2 can generate code for partial elements of the UML data model used by Fujaba. For our purpose we need a complete project, as XProM2 will add the JUnit test classes and methods to it on the fly. This `UMLProject` needs to contain a `UMLPackage` that marks the root of all the contained classes. After this basic project setup we can continue to create the actual storyboard.

With some helper methods to create `UMLStoryActivity`, `UMLStoryPattern`, `UMLObject` and `UMLLink` instances the instantiation of a data model representing the storyboard is straight forward. First, we create a `UMLStoryActivity` and add a `UMLStartActivity`. For each `StoryActivity` in the storyboard we create a `UMLStoryActivity` and set its description to the textual scenario description. The description will be added to the code as a comment, so developers will still see what the generated code is meant to test. Then we add `UMLObject` instances for each object. These objects must be collected in a `HashMap` so we can correctly create `UMLLinks` between them when iterating over the links in the `StoryActivity`. Care must be taken to create `UMLObject` and `UMLLink` instances from the correct package because Fujaba and our web application are using the same class name for objects and links. Finally, we end the `UMLStoryActivity` with a `UMLStopActivity`.

Now we can tweak a few details in the model. To trigger the XProM2 code generation for the storyboard we also need to mark the `UMLStoryActivity` as a test scenario by adding a `XProM2Plugin.TEST_SCENARIO` stereotype to it. Furthermore, CodeGen2 by default uses the Fujaba codestyle which we will keep to make importing and reusing the generated Fujaba project repository easier.

After this we can start the code generation by handing over the complete `UMLProject` and the destination path to the code generation wrapper. We use a timestamp based path that is accessible by the web server to allow direct browsing of the generated files. The code generation web service then returns the timestamp the code was generated at which the web UI uses to construct and add several download links.

Accessing the generated acceptance tests

The asynchronous GWT-RPC call will receive a timestamp as the result of the code generation. Since the generated files are accessible at a public URL containing the same timestamp I can generate HTML links to the generated code, the `Fujaba project.ctr` and for a quick check the `StoryboardTest.java` containing the JUnit test. The generated code also contains any classes that have been identified by the Fujaba code generation. An excerpt of the code generation run at timestamp 1395613372706 is shown in [listing 27](#).

¹¹⁸ Leif Geiger, Christian Schneider, and Carsten Reckord. "Template- and model-based code generation for MDA-Tools". In: *3rd International Fujaba Days*. Paderborn, Germany, Sept. 2005. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/Cod eGen2.pdf>

¹¹⁹ Yatta Solutions. *Yatta*. 2012. URL: <http://yatta.de> (visited on 03/25/2014)

¹²⁰ Yatta Solutions. *UML Lab*. 2012. URL: <http://www.uml-lab.com> (visited on 03/25/2014)

¹²¹ Leif Geiger and Albert Zündorf. "Transforming Graph Based Scenarios into Graph Transformation Based JUnit Tests." In: *AGTIVE*. ed. by John L. Pfaltz, Manfred Nagl, and Boris Böhlen. Vol. 3062. Lecture Notes in Computer Science. Springer, 2003, pp. 61–74. ISBN: 3-540-22120-4. URL: <http://dblp.uni-trier.de/db/conf/agtive/agtive2003.html#GeigerZ03>

¹²² Leif Geiger and Albert Zündorf. "Story driven testing - SDT". In: *ACM SIGSOFT Software Engineering Notes* 30.4 (2005), pp. 1–6. URL: <http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft30.html#GeigerZ05>

¹²³ Geiger and Zündorf, "Developing Tools with Fujaba XProM."

¹²⁴ Geiger, "Fehlersuche im Modell: modellbasiertes Testen und Debuggen."

Related Work

Without the work that had happened in the Fujaba community at the university of Kassel this part of the storyboarding process would have been a lot harder to support in our web application. The foundations of CodeGen2 were laid by Geiger, Schneider, and Reckord¹¹⁸ which has meanwhile outgrown Fujaba and even led to the foundation of Yatta Solutions¹¹⁹, the company behind the commercial successor of Fujaba: UML Lab¹²⁰.

Geiger and Zündorf¹²¹ described the first version of test code generation in 2003 and extended the idea in subsequent¹²² papers.¹²³ A detailed German description of the test case generation has been written by Geiger¹²⁴ as part of his PhD thesis.

Conclusion

With XProM2 available as a web service our web application can complete the last step in the storyboarding process. It can send the existing storyboard to the server with an asynchronous GWT-RPC call and creates download links when the result arrives. The web service uses a headless version of Fujaba with the necessary XProM2 and CodeGen2 plugins for the code generation of JUnit tests. To continue working with the storyboard derived from a textual scenario description developers can import a Fujaba project repository saved next to the generated sources.

```

1 // Alice and Bob are playing chess. Alice
2 // moves her white pawn from field c2 to c4.
3
4 // story pattern
5 try
6 {
7     fujaba__Success = false;
8
9     // create object c2
10    c2 = new Field ( );
11
12    // create object pawn
13    pawn = new Pawn ( );
14
15    // create object Alice
16    Alice = new Person ( );
17
18    // create object c4
19    c4 = new C4 ( );
20
21    // create object chess
22    chess = new Chess ( );
23
24    // create object Bob
25    Bob = new Person ( );
26
27    // create link object_c2 from this to c2
28    this.setC2 (c2);
29
30    // create link owner from Alice to pawn
31    Alice.setPawn (pawn);
32
33    // create link object_pawn from this to pawn
34    this.setPawn (pawn);
35
36    // create link object_c4 from this to c4
37    this.setC4 (c4);
38
39    // create link players from Bob to chess
40    Bob.setPerson (chess);
41
42    // create link players from Alice to chess
43    Alice.addToPlayers (chess);
44
45    // create link object_chess from this to chess
46    this.setChess (chess);
47
48    // create link object_Bob from this to Bob
49    this.setBob (Bob);
50
51    // create link object_Alice from this to Alice
52    this.setAlice (Alice);
53
54    // collabStat call
55    pawn.moves();
56    fujaba__Success = true;
57 }

```

Listing 27: Excerpt from the generated code for the JUnit `setUp()` method for the start scenario. As of May 23rd 2014 the full source for this storyboarding run is available online at <http://instant-storyboarding.de/~jfd/nt2od/fujabaWorkspace1395613372706/>. Other code generation runs can currently be accessed by pointing your browser to <http://instant-storyboarding.de/~jfd/nt2od/> and navigating to one of the subdirectories.

Outlook

¹²⁵ Nina Aschenbrenner et al. “Fujaba goes Web 2.0”. In: *6th International Fujaba Days*. Ed. by Uwe Aßman, Jendrik Johannes, and Albert Zündorf. Dresden, Germany, 2008, pp. 10–14. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/ADJZ08.pdf>

¹²⁶ Password protected outside the network of the University of Kassel. Marcel Hahn and Ruben Jubeh. *Fujaba Web Runtime*. Software Engineering Group Kassel. 2010. URL: <https://gforge.cs.uni-kassel.de/projects/fujabawebbrt/> (visited on 03/25/2014)

¹²⁷ Google, *AngularJS*

¹²⁸ W3C. *HTML Components. Componentizing Web Applications*. 1998. URL: <http://www.w3.org/TR/NOTE-HTMLComponents> (visited on 03/25/2014)

¹²⁹ W3C. *File API. 8. The FileReader Interface*. 2013. URL: <http://www.w3.org/TR/FileAPI/#FileReader-interface> (visited on 03/25/2014)

¹³⁰ A simple text editor based on this can be found at [thiscouldbeter](http://thiscouldbeter.com/2012/12/18/loading-editing-and-saving-a-text-file-in-html5-using-javascript/). *Loading, Editing, and Saving a Text File in HTML5 Using Javascript*. Dec. 18, 2012. URL: <http://thiscouldbeter.wordpress.com/2012/12/18/loading-editing-and-saving-a-text-file-in-html5-using-javascript/> (visited on 03/25/2014)

Fujaba sometimes uses magic guesses hidden in some of the helper methods used to create the UML data model. One of them is used to guess the cardinality of associations when creating them on the fly. The current guess is based on the plural s of the link name: `players` becomes a `0..*` cardinality `player` a `0..1` cardinality. This works ok for links named by humans. For the structurization rules a link will be named after the verb and not the role of the object or subject in the sentence. One solution would be to change the structurization rules to use the subjects class lowercased as the link name. If that is an improvement is left to be determined by future research or asking a linguist. He might even suggest to use stemming to find the root of the verb: eg. `play` and use that to determine the cardinality. A software engineer might write a recommender that rewrites link names based on number of links with the same name. Best ask a linguist first.

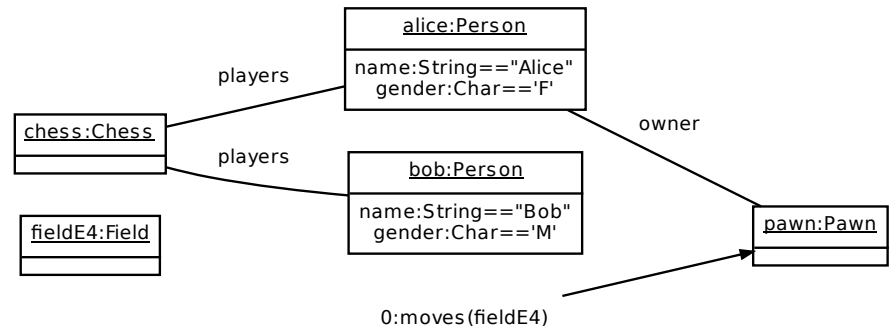
Another question I did not try to answer was if it is possible to use GWT to cross compile Fujaba with CodeGen2 and XProM2 to JavaScript. Since I started working on this Aschenbrenner et al.¹²⁵ added GWT support for the association implementations generated by the code generation with `fujaba-web-runtime`¹²⁶. That might be a starting point to make the logic implemented in Fujaba compatible with GWT but it still leaves the task of developing a web based user interface. A native JS solution such as `AngularJS`¹²⁷ could be used to get databinding with HTML templates that can be used to create graphical HTML components. Maybe the W3C HTML Components¹²⁸ standard is a solution if more expressive HTML is needed.

While our web application is now capable of exporting the generated storyboard as a Fujaba project the other direction is not yet possible. The idea is to use a Fujaba project to recommend type information in the formalization step of our web application. An initial implementation could load a user specified Fujaba project in `ctr` format and collect the used class names. With the `FileReader` API specified by the W3C¹²⁹ this can meanwhile be done without even sending the file to the server.¹³⁰

Instant Examples

After describing the technical details of Instant Storyboarding in full detail it is now time to demonstrate how extendable the whole framework has become. Let me briefly repeat the textual example scenario in [listing 23](#) from page 111 and the excerpt I used to explain nearly all storyboarding steps in this thesis:

```
1 Start scenario:
2 Alice and Bob are playing chess.
3 Alice moves her pawn to field e4.
4
5 End scenario:
6 Alice's pawn is on e4.
```



The attentive reader will have noticed the subtle changes from the initial example in the introduction in [listing 1](#) on page 13. In the following sections I will describe example by example how the textual description as well as the structurization and formalization rules can be changed to improve the result.

Change link verbs to nouns

When you look at the diagram in [figure 1](#) on page 13 you may be dissatisfied by the verb *playing* being the link caption. A *player* makes more sense and is what UML designers are used to. To change the name we will use a short hack to the structurization rules as described in [listing 28](#).

Listing 28: Change link verbs to nouns

```
1 # rule to replace playing with players,
2 # makes the codegeneration create 0..n links:
3 [_(1):"Link"_"|"name":"String"=="playing";"name":"String":="players"]
```

The rule matches all *playing* links and changes their caption to *play*. This is of course a workaround and a more general, challenging, yet time consuming solution would be to write a WordNet based recommender. In this case the goal is to demonstrate how the user can instantly change the story pattern with his domain knowledge.

Adding more context to scenarios

Another way to change the storyboard is adding more context to the textual scenario. In our chess example we can extend the scenario to also capture the concept of a turn as in [listing 29](#). The resulting story pattern for start and end situation are shown in [figure 82](#).

Listing 29: Add turn example

```
1 Start scenario:
2 Alice and Bob are playing chess.
3 Alice moves her pawn to field e4.
4 It is Alice's turn.
5
6 End scenario:
7 Alice's pawn is on e4.
8 It is now Bob's turn.
```

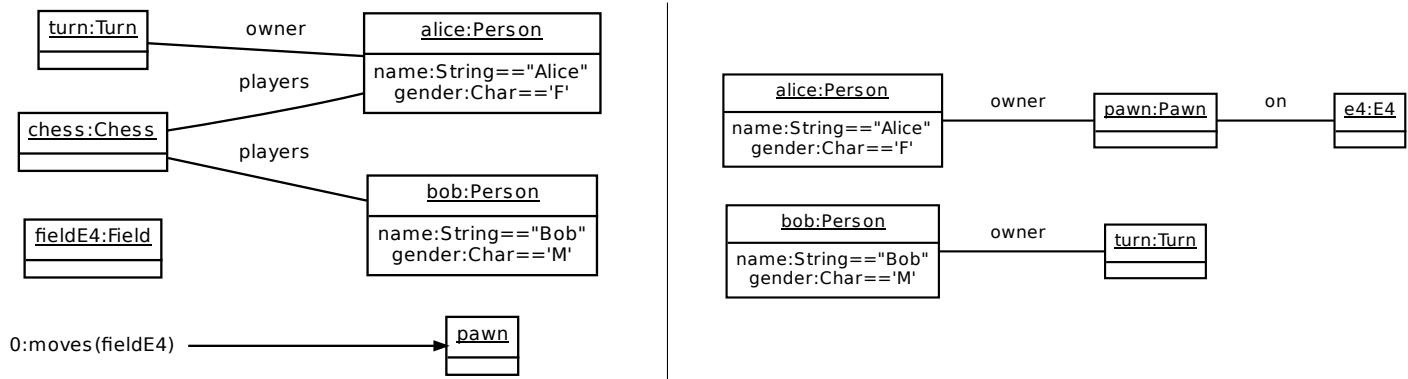


Figure 82: Start and end Story pattern now also contain the turn. After Alice has made her move Bob becomes the owner of the turn.

Again, looking at the two diagrams above may be dissatisfying because instead of a *Turn* object and an *owner* link a more familiar modeling decision might have been the introduction of a *Game* object and a *turn* link. It is left as an exercise for the reader to adapt the example from the previous section accordingly.

Capturing more context from grammatical relations

When adding context to the textual scenarios it might become necessary to take new grammatical relations into account. To demonstrate that, we will extend the chess scenario to also contain the source field of Alice's move as in [listing 30](#).

```

1 Start scenario:
2 Alice and Bob are playing chess.
3 Alice moves her pawn from field e2 to e4.
4
5 End scenario:
6 Alice's pawn is on e4.

```

Listing 30: Add source example

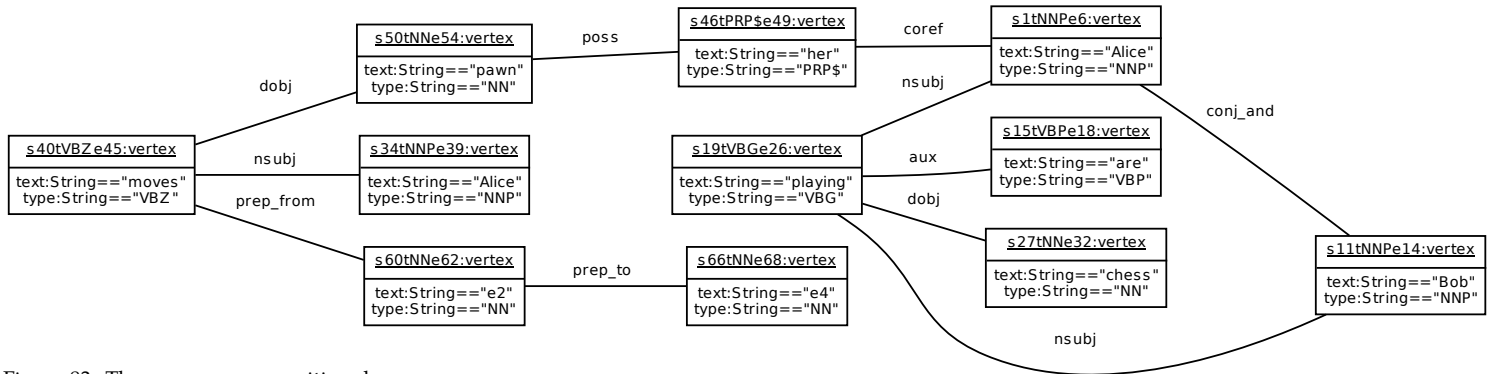


Figure 83: The `prep_to` preposition depends on `prep_from`.

The resulting grammatical relations in [figure 83](#) above reveal that the `prep_to` preposition depends on `prep_from`. Our current structurization rules only take into account a `prep_to` in combination with a `dobj` and `nsbj` relation to the same verb. To capture the `prep_from` as a second parameter we need to add two rules. The first one in [listing 31](#) and [figure 84](#) is a duplication of the existing `prep_to` based rule that adds a `to` parameter to the collaboration statement. We just adjust it to match `prep_from` and add a `from` parameter.

Listing 31: Capture the `prep_from` prepositions of the verb

```

1 #add parameter for prep_from as 'from' parameter
2 [_(vp) : "vertex" _ | "type"=="VBZ"; "text"==(pred)] - "dobj" - [_(vo) : "vertex" _ | "type"==/NNP|NN/; "text"==(dobj)],
3 [_(vp)_] - "nsbj" - [_(vs) : "vertex" _ | "type"==/NNP|NN/; "text"==(nsbj)],
4 [_(vo)_] - "origin" - [_(oo) : "Object" _],
5 [_(vs)_] - "origin" - [_(os) : "Object" _],
6 [_(oo)_] - "messages" - [_(cs) : "CollabStmt" _ | "name"==pred],
7 [_(os)_] - "sender" - [_(cs)_],
8 [_(vp)_] - "prep_from" - [_(vprep) : "vertex" _ | "type"==/NNP|NN/],
9 [_(vprep)_] - "origin" - [_(oprep) : "Object" _],
10 [_(oprep)_] - "parameter" -+ [_(param) : "Parameter" _ | "name": "String" := "from"],
11 [_(cs)_] - "parameters" -+ [_(param)_]

```

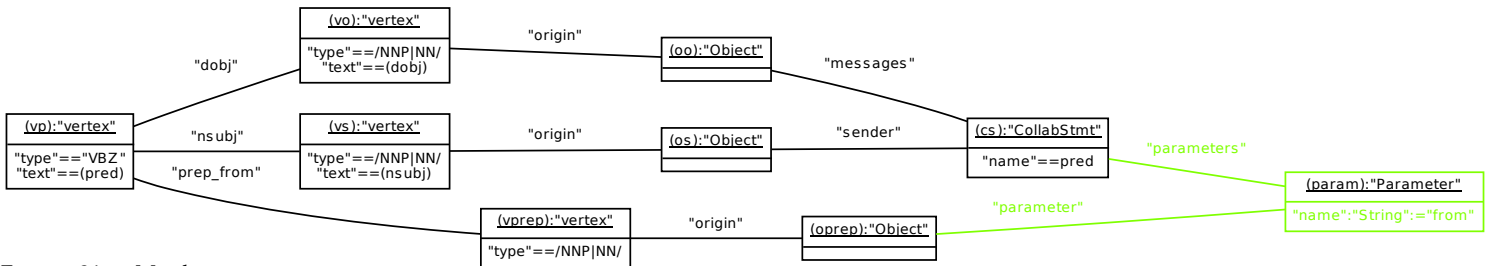


Figure 84: Matching a `prep_from` preposition and add a *from* parameter to the collaboration statement.

The second rule in [listing 32](#) and [figure 85](#) extends the first new rule with a match for the depending `prep_to` relation. If we find a match, we can add an additional *to* parameter to the collaboration statement. After the formalization rules have been applied that will give us the formal story pattern in [figure 86](#).

Together, these two rules demonstrate how the meaning of a sentence can be captured by matching patterns in the grammatical relations and transforming them into story patterns.

Listing 32: Capture the *from* and *to* prepositions of the verb

```

1 #add parameter for prep_to as 'to' parameter if it is related to prep_from
2 [_ (vp) : "vertex" _ | "type"=="VBZ"; "text"==(pred) ] -"dobj"- [_ (vo) : "vertex" _ | "type"==/NNP|NN/; "text"==(dobj) ],
3 [_ (vp) _] -"nsbj"- [_ (vs) : "vertex" _ | "type"==/NNP|NN/; "text"==(nsbj) ],
4 [_ (vo) _] -"origin"- [_ (oo) : "Object" _ ],
5 [_ (vs) _] -"origin"- [_ (os) : "Object" _ ],
6 [_ (vp) _] -"prep_from"- [_ (vprep) : "vertex" _ | "type"==/NNP|NN/ ],
7 [_ (vprep) _] -"origin"- [_ (oprep) : "Object" _ ],
8 [_ (vprep) _] -"prep_to"- [_ (vprepto) : "vertex" _ | "type"==/NNP|NN/; "text"==(prep_to) ],
9 [_ (vprepto) _] -"origin"- [_ (oprepto) : "Object" _ ],
10 [_ (oo) _] -"messages"- [_ (cs) : "CollabStmt" _ | "name"==pred ],
11 [_ (os) _] -"sender"- [_ (cs) _ ],
12 [_ (oprepto) _] -+"parameter"-+ [_ (param) : "Parameter" _ | "name": "String":="to" ],
13 [_ (cs) _] -+"parameters"-+ [_ (param) _ ]

```

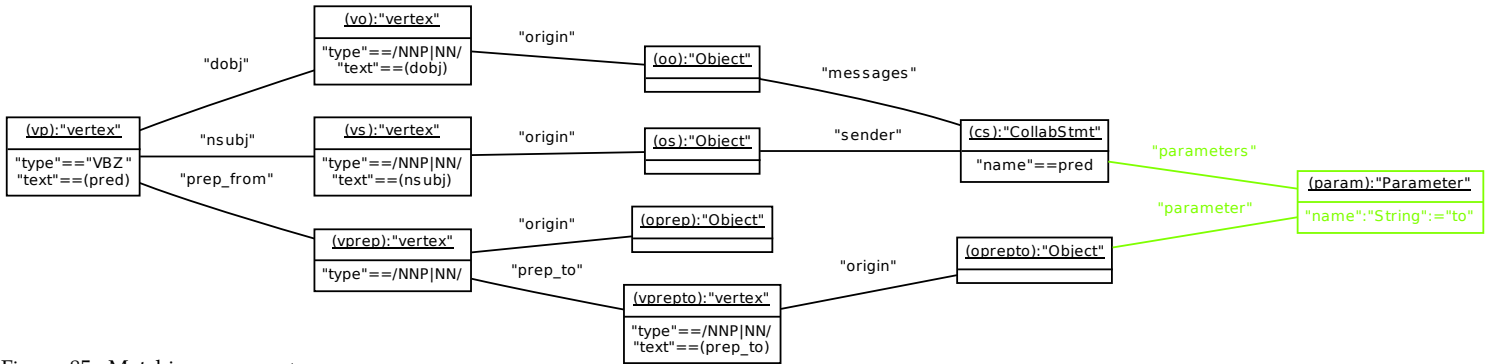
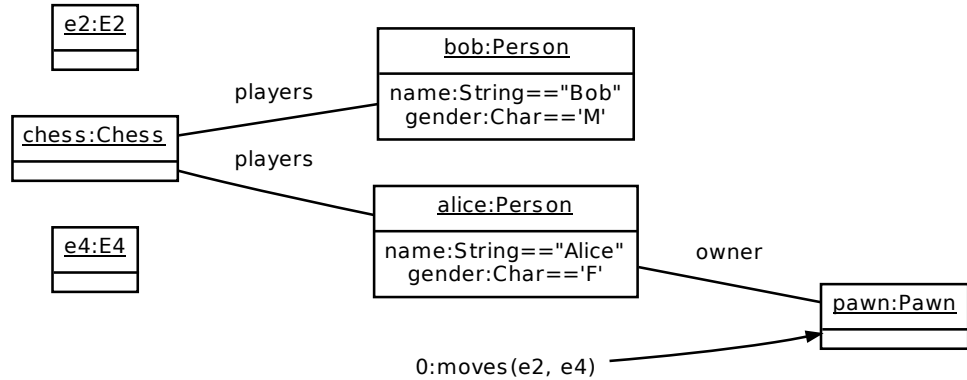



Figure 85: Matching a `prep_to` preposition depending on `prep_from` and adding a second `to` parameter to the collaboration statement.

Figure 86: Formal story pattern for a collaboration statement with two parameters.



A web shop example

Until now, we only modified the chess example. What happens if we describe a completely different scenario? [Listing 33](#) and [figure 87](#) show the textual scenario for a web shop and the derived storyboard.

```

1 Start scenario:
2 Cindy adds a notebook to her shopping cart.
3
4 End scenario:
5 The notebook is listed in the cart.

```

Listing 33: Web shop example

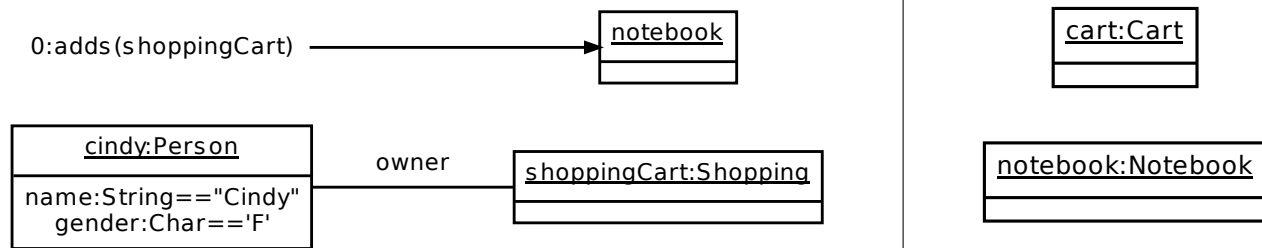


Figure 87: Start and end story pattern for the web shop example.

Both story patterns can be improved. Take a moment and examine the start pattern. Do you see something you would model differently? Take a closer look at the collaboration statement. It roughly translates back to “Cindy adds her shopping cart to a notebook”. To fix this, we need to swap the nodes we match and change the structurization rules as shown in [listing 34](#).

The second flaw is easier to spot as the end pattern lacks a relation between *notebook* and *cart*. Examining the grammatical relations of the textual scenario in [figure 88](#) helps us find the reason: it uses the passive. [Listing 35](#) shows a structurization rule to capture the in relation in a passive sentence and create a link for it. Now let us polish the start diagram by aligning the shopping cart class with the end scenario using the rule in [listing 36](#). The final storyboard then looks like [figure 89](#).

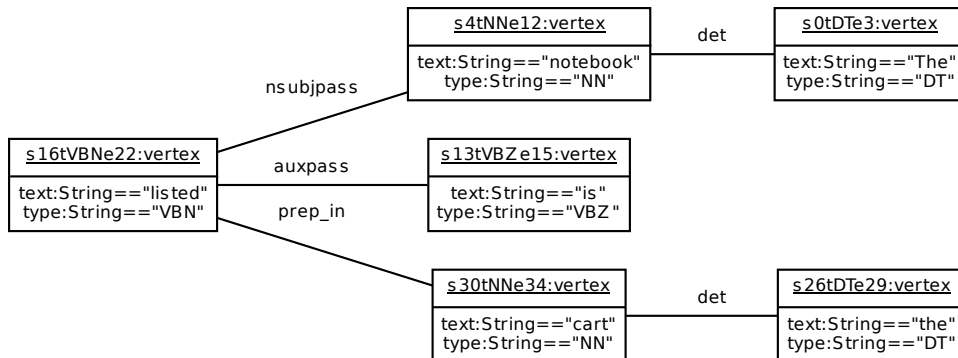
```

1 # Create Collaboration Statements
2 [_(vp) : "vertex_" | "type"=="VBZ"; "text"==(pred)] - "prep_to" - [_(vo) : "vertex_" | "type"==/NNP|NN/; "text"==(prep_to)],
3 [_(vp)_] - "nsubj" - [_(vs) : "vertex_" | "type"==/NNP|NN/; "text"==(nsubj)],
4 [_(vo)_] - "origin" - [_(oo) : "Object_" ],
5 [_(vs)_] - "origin" - [_(os) : "Object_" ],
6 [_(oo)_] - "messages" - [_(cs) : "CollabStmt_" | "name" : "String" :==pred],
7 [_(os)_] - "sender" - [_(cs)_]
8
9 #add parameter
10 [_(vp) : "vertex_" | "type"=="VBZ"; "text"==(pred)] - "prep_to" - [_(vo) : "vertex_" | "type"==/NNP|NN/; "text"==(prep_to)],
11 [_(vp)_] - "nsubj" - [_(vs) : "vertex_" | "type"==/NNP|NN/; "text"==(nsubj)],
12 [_(vo)_] - "origin" - [_(oo) : "Object_" ],
13 [_(vs)_] - "origin" - [_(os) : "Object_" ],
14 [_(oo)_] - "messages" - [_(cs) : "CollabStmt_" | "name"==pred],
15 [_(os)_] - "sender" - [_(cs)_],
16 [_(vp)_] - "dobj" - [_(vdobj) : "vertex_" | "type"==/NNP|NN/],
17 [_(vdobj)_] - "origin" - [_(oprep) : "Object_" ],
18 [_(oprep)_] - "parameter" - [_(param) : "Parameter_" | "name" : "String" :="to"],
19 [_(cs)_] - "parameters" - [_(param)_]

```

Listing 34: Structurization Rule: swap the order of nodes to capture the *add to* construct in the right order.

Figure 88: Grammatical relations for a passive sentence.



```

1 # create in link for prep_in relation in a passive sentence
2 [_ (vp) : "vertex" _ | "type" == "VBN" ] - "prep_in" - [_ (vs) : "vertex" _ | "type" == /NNP | NN / ],
3 [_ (vp) _ ] - /nsubjpass / - [_ (vo) : "vertex" _ | "type" == /NNP | NN / ],
4 [_ (vs) _ ] - "origin" - [_ (os) : "Object" _ ],
5 [_ (vo) _ ] - "origin" - [_ (oo) : "Object" _ ],
6 [_ (os) _ ] -+ "links" -+ [_ (l) : "Link" _ | "name" : "String" : "= in" ] ? ,
7 [_ (oo) _ ] -+ "links" -+ [_ (l) : "Link" _ ] ?

```

Listing 35: Structurization Rule: add *in* link for a passive sentence.

```

1 # change type of the shopping cart to Cart
2 [_ (o) : "Object" _ | "name" : "String" == "shoppingCart" ; "type" : "String" : "= Cart" ]

```

Listing 36: Structurization Rule: change type of the shopping cart to *Cart*.

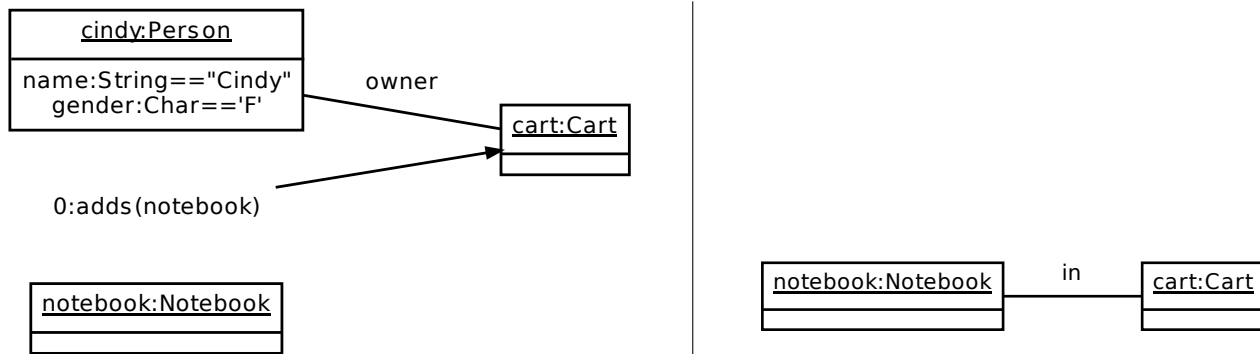


Figure 89: Fixed start and end story pattern for the web shop example.

Gherkin as parsed english

This last example will revisit all kinds of changes that we already learned. In Behavior Driven Development features of a software system are often collected in textual scenarios that follow the Gherkin language. We are going to examine the example in [listing 37](#). A Gherkin feature starts with an In order to ..., As an ..., I want to ... schema to provide context and then describes scenarios following a Given ... when ... then ... schema.

Listing 37: Gherkin *Buy last coffee* scenario for using a coffee machine. An example used in the Behat 3.0.12 documentation, see Konstantin Kudryashov et al. *Writing Features*. 2014. URL: <http://docs.behat.org/en/latest/guides/1.gherkin.html#features> (visited on 02/18/2015)

```

1 Feature: Serve coffee
2   In order to earn money
3   Customers should be able to
4   buy coffee at all times
5
6 Scenario: Buy last coffee
7   Given there are 1 coffees left in the machine
8   And I have deposited 1 dollar
9   When I press the coffee button
10  Then I should be served a coffee

```

dollar:Dollar

coffeeButton:Coffee

machine:Machine

Figure 90: Story pattern for the *Buy last coffee* start

coffee:Coffee

Figure 91: Story pattern for the *Buy last coffee* end

Listing 38: Gherkin *Buy last coffee* start and end scenario

```

1 Start scenario:
2 Given there are 1 coffees left in the machine
3 And I have deposited 1 dollar
4 When I press the coffee button
5
6 End scenario:
7 Then I should be served a coffee

```

On the one hand, this minimal structure makes most of the structurization rules we have seen so far useless, because they were crafted with a different sentence structure in mind. On the other hand, Instant Storyboarding is flexible enough to be adopted to Gherkin on the fly.

First, we are only interested in the textual scenario, which we will split into the start and end scenario used in storyboarding. As [listing 38](#) shows, we use the same lack of interpunction and, without any changes to our rules, obtain the story pattern in [figures 90](#) and [91](#).

The reason for the missing objects and relations is that the parser has to identify sentence boundaries before he can try to identify grammatical relations. Figure 92 shows the grammatical relations, which links the grammars of the three individual sentences to each other by four *dep* relations. This can be corrected as the following sequence of figures will explain:

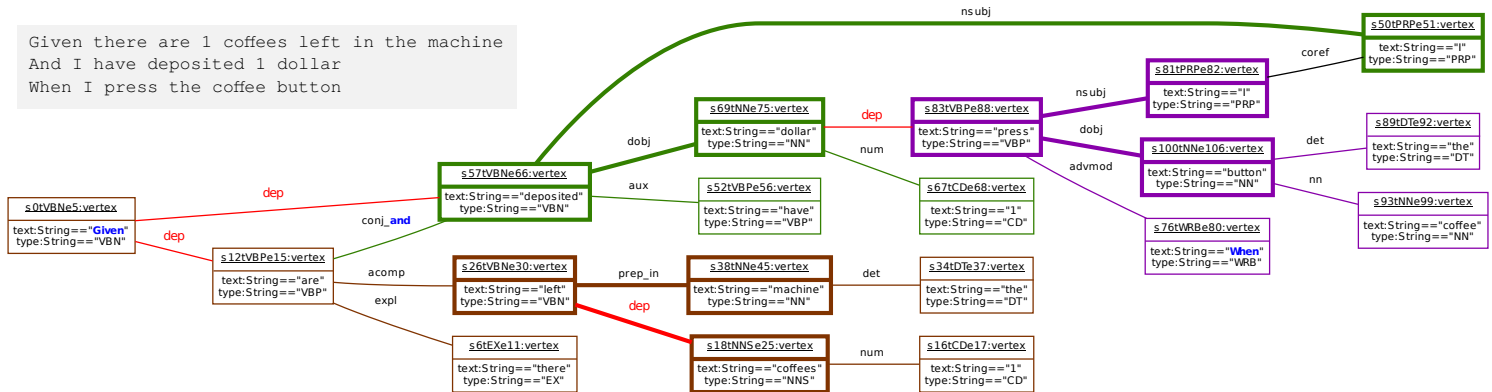


Figure 92: We start with the grammar for the original scenario that we just copied and pasted. I colored the grammar parts to hint which parts belong to the same sentence. The significant subject-predicate-object relations our current structurization rules partly fail to match are highlighted in bold, the sentence start words are blue, and the *dep* relations are red.

Given there are 1 coffees left
and I have deposited 1 dollar
when I press the coffee button

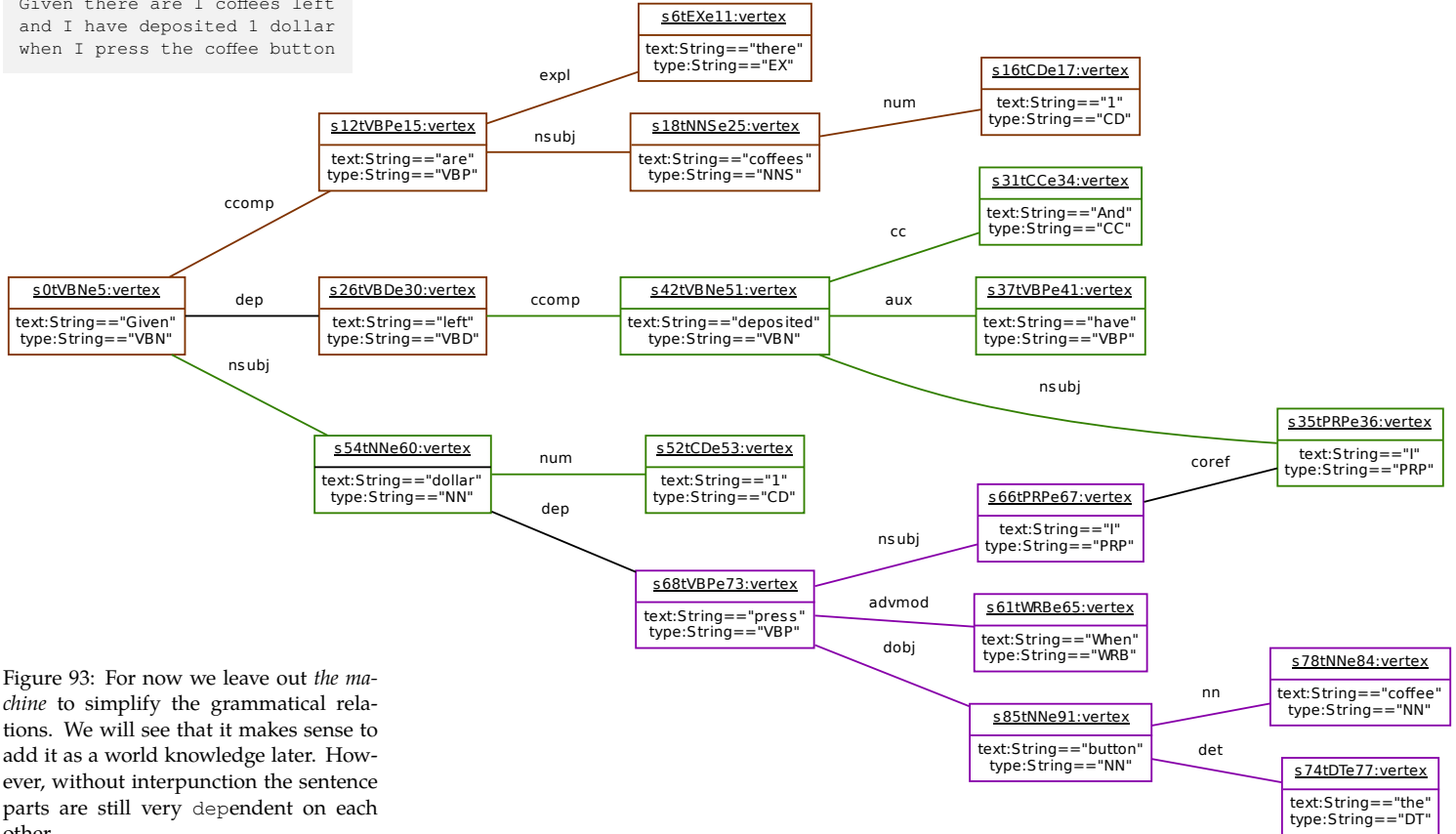


Figure 93: For now we leave out *the machine* to simplify the grammatical relations. We will see that it makes sense to add it as a world knowledge later. However, without interpunction the sentence parts are still very dependent on each other.

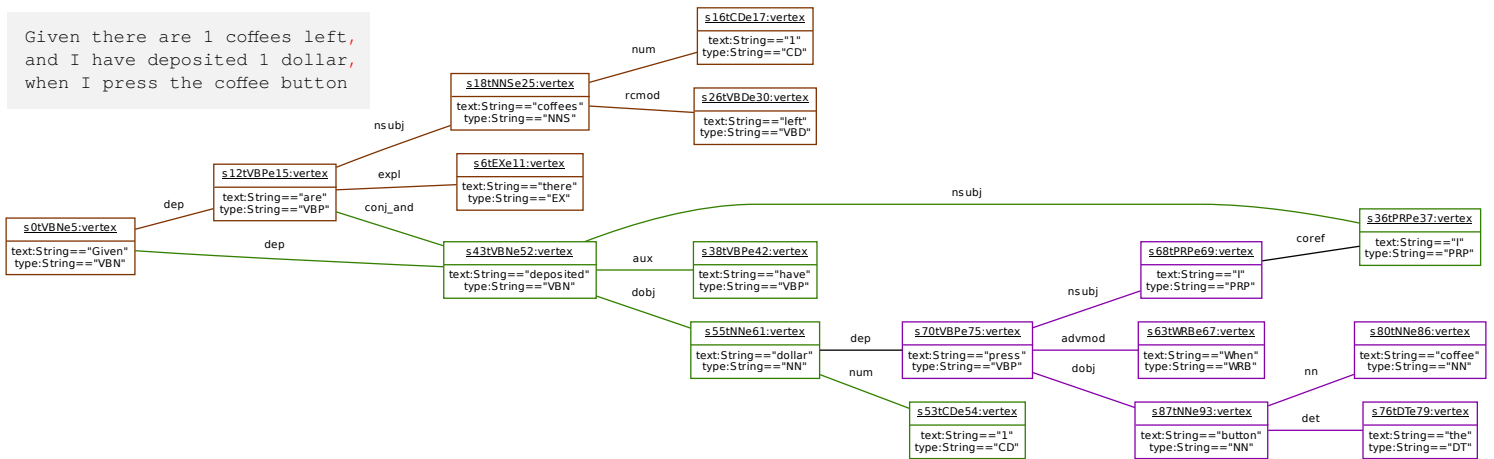


Figure 94: Indicating sentence boundaries with commas allows the parser to more cleanly identify where *left* belongs to in the grammar. Nevertheless, the dep relations remain.

Given there are 1 coffees left,
and I have deposited 1 dollar.
When I press the coffee button

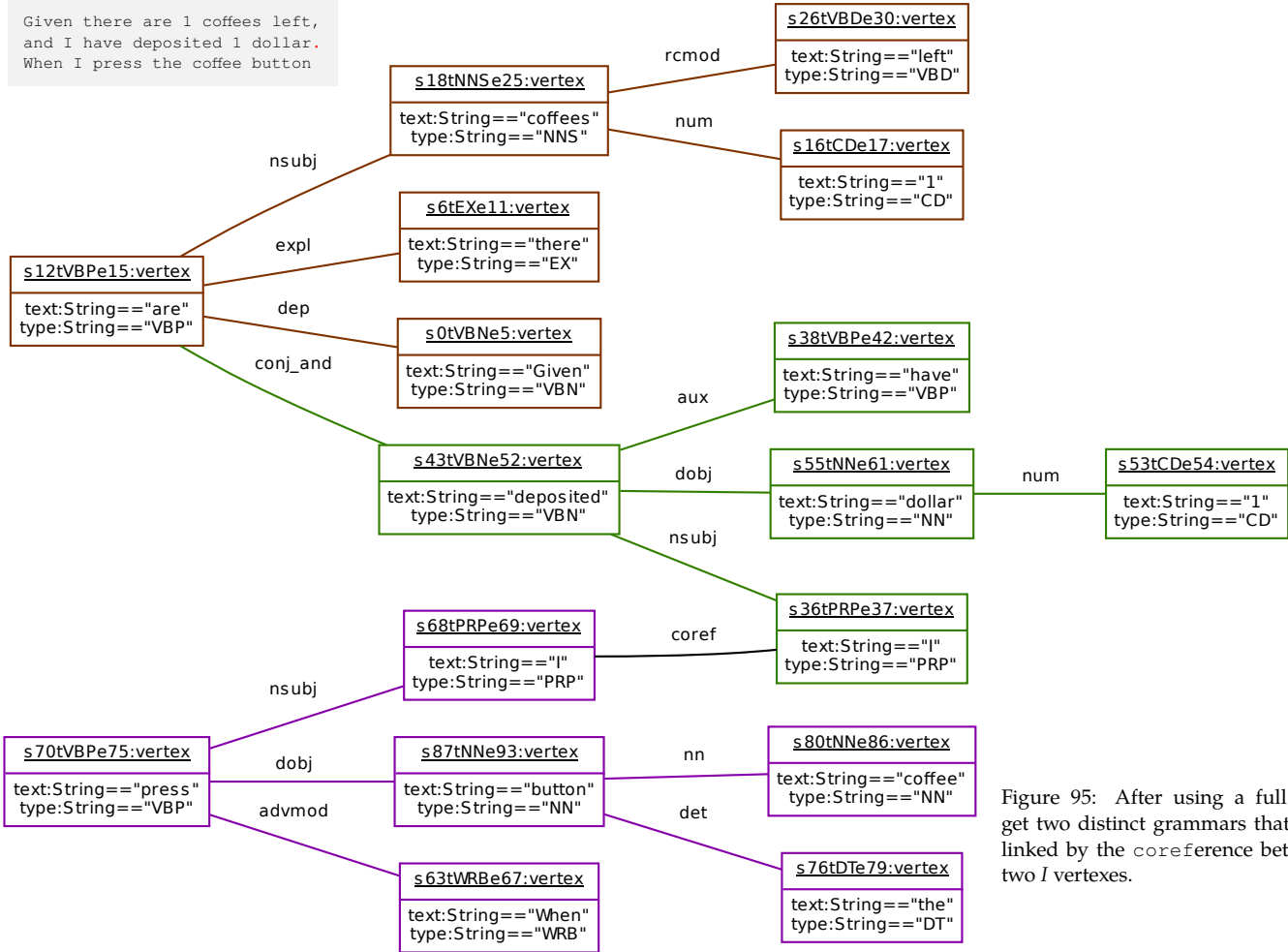


Figure 95: After using a full stop we get two distinct grammars that are only linked by the coreference between the two *I* vertexes.

With the grammar from [figure 95](#) our structurization rules should be able to identify the two subject-predicate-object relations. But the derived informal story pattern still looks like [figure 90](#) without *the machine*, which we removed on purpose. Why is that? The structurization rules we developed for the *Ludo* example had to distinguish between verb tenses so we could separate links from collaboration statements. With gherkin we can identify the collaboration statement by looking for the *When* adverbial modifier (*advmod*). This also allows us to include any verb tenses when trying to match grammatical relations. Another change in our structurization rules that is now used by default is the addition of plural nouns (NNS) to any noun matching rules. Finally, we will add little hacks to represent *I* as the *actor* and fix the object name derived from *coffees* from plural to singular. These rule changes are shown in [listing 39](#) which is only an excerpt from the full ruleset.

Listing 39: Updated structurization rules for gherkin.

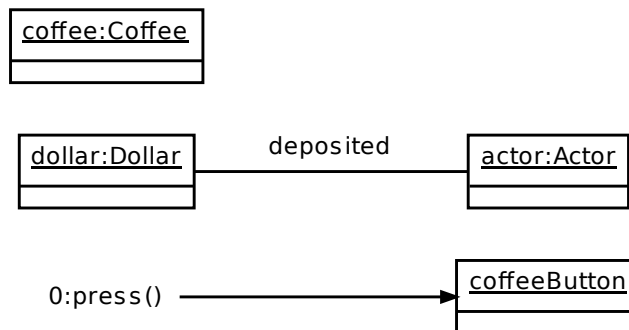
```

1 # replace I PRP with actor NN
2 [_ (v) : "vertex" _ | "text" == "I"; "type" == "PRP"; "text" : "String" := "actor"; "type" : "String" := "NN" ]
3
4 # first create objects for all plural nouns
5 [_ (v) : "vertex" _ | "type" == "NNS"; "text" == (noun) ],
6 + [_ noun.toLower() : "Object" _ | "name" : "String" := noun.toLower(); "type" : "String" := noun.upcaseFirst () ] ?
7
8 # Create a Link in the diagram for each matching subject, predicate, object triple
9 [_ (vp) : "vertex" _ | "type" == /VBG|VBP|VBZ|VBN/; "text" == (pred) ] - "nsubj" - [_ (vs) : "vertex" _ | "type" == /NNP|NN/; "text" == (nsubj) ],
10 [_ (vp) _ ] - "dobj" - [_ (vo) : "vertex" _ | "type" == /NNP|NN/; "text" == (dobj) ],
11 [_ (vs) _ ] - "origin" - [_ (os) : "Object" _ ],
12 [_ (vo) _ ] - "origin" - [_ (oo) : "Object" _ ],
13 [_ (os) _ ] -+ "links" -+ [_ (l) : "Link" _ | "name" : "String" := pred; "name" : "String" == pred ] ?,
14 [_ (oo) _ ] -+ "links" -+ [_ (l) : "Link" _ ] ?
15
16 # Create Collaboration Statements
17 [_ (vp) : "vertex" _ | "type" == /VBG|VBP|VBZ|VBN/; "text" == (pred) ] - "dobj" - [_ (vo) : "vertex" _ | "type" == /NNP|NN|NNS/; "text" == (dobj) ],
18 [_ (vp) _ ] - "nsubj" - [_ (vs) : "vertex" _ | "type" == /NNP|NN|NNS/; "text" == (nsubj) ],
19 [_ (vp) _ ] - "advmod" - [_ (wrb) : "vertex" _ | "type" == "WRB"; "text" == "When" ],
20 [_ (vo) _ ] - "origin" - [_ (oo) : "Object" _ ],
21 [_ (vs) _ ] - "origin" - [_ (os) : "Object" _ ],
22 [_ (os) _ ] -- "links" -- [_ (l) : "Link" _ | "name" : "String" == pred ],
23 [_ (oo) _ ] -- "links" -- [_ (l) : "Link" _ ],
24 [_ (oo) _ ] -+ "messages" -+ [_ (cs) : "CollabStmt" _ | "name" : "String" := pred ],
25 [_ (os) _ ] -+ "sender" -+ [_ (cs) _ ]
26
27 #fix plural -> singular
28 [_ (o) : "Object" _ | "name" : "String" == "coffees"; "type" : "String" == "Coffees"; "name" : "String" := "coffee"; "type" : "String" := "Coffee" ]

```

With these rules our informal story pattern in [figure 96](#) becomes much more complete compared to the pattern we started with in [figure 90](#). All verbs have been mapped to links or collaboration statements.

Figure 96: The story pattern after transforming the grammar with the gherkin specific structurization rules.



Before we add back *the machine* to the textual start scenario let us examine what kind of context it adds to the scenario. By adding *the machine* only to the first sentence we leave out that actually all objects are linked to *the machine*. To give our structurization rules a chance to capture this meaning we would have to add *the machine* to all aspects of the scenario, which makes it awkward to read as seen in [listing 40](#).

```

1 Start scenario:
2 Given there are 1 coffees left in the machine,
3 and I have deposited 1 dollar in the machine.
4 When I press the coffee button on the machine
5
6 End scenario:
7 Then I should be served a coffee by the machine
  
```

Listing 40: Gherkin *Buy last coffee* start and end scenario with *the machine* added to every part of the sentence.

Instead of repeating ourselves in the scenario we could add this kind of context to the end of the structurization rules. That also allows us to directly model the link names. [Listing 41](#) and the visual representation in [figure 97](#) show how this can be done with a single rule.

```

1 # add machine as context directly as context
2 [_ (o1): "Object" _ | "name"=="dollar" ]?--+"links"?--+[_ (l1): "Link" _ | "name": "String" :="payment" ]?,
3 +[_ (l1) _ ]?--+"links"?--+[_ (m): "Object" _ | "name" :="machine"; "type":="Machine" ],
4 [_ (o2): "Object" _ | "type"=="Button" ]?--+"links"?--+[_ (l2): "Link" _ | "name": "String" :="controls" ]?,
5 +[_ (l2) _ ]?--+"links"?--+[_ (m) _ ],
6 [_ (o3): "Object" _ | "name"=="coffee" ]?--+"links"?--+[_ (l3): "Link" _ | "name": "String" :="products" ]?,
7 +[_ (l3) _ ]?--+"links"?--+[_ (m) _ ]

```

Listing 41: Structurization rules to add *the machine* context in FUMML notation.

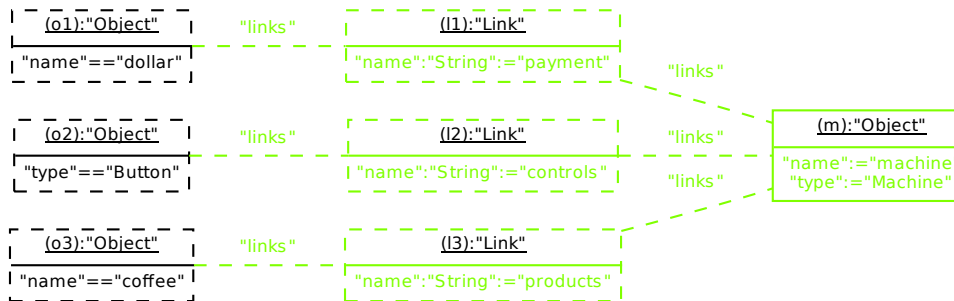


Figure 97: Structurization rules to add *the machine* context as a story pattern.

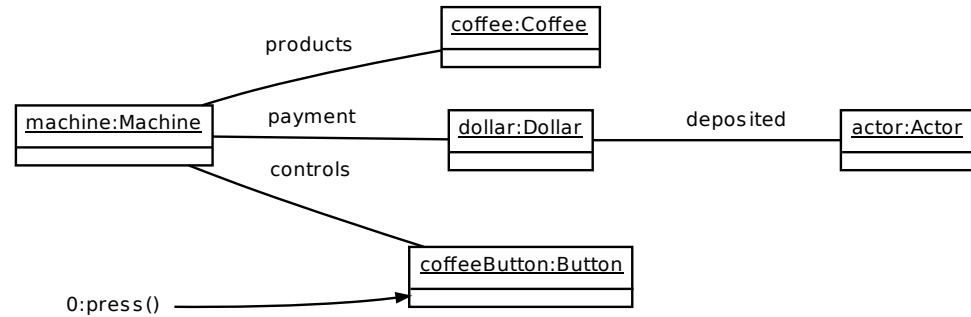
The resulting informal story pattern in [figure 98](#) captures every aspect of the gherkin scenario and can be used to generate a JUnit test as it can be done for our chess example.

Lessons learned and further research opportunities

With the exception of the last gherkin example, the above examples were all developed in front of an audience while discussion was possible. Reproducing them here revealed some more glitches that were overlooked, like the wrong direction of a collaboration statement of the web shop example or a bug in the interpreter that would prevent the execution of *the machine* rule.

Nevertheless gherkin was an interesting challenge, as it allows a different approach for distinguishing links from collaboration statements. A future task may be to also parse the gherkin feature description for context and allow parsing scenarios without having to split them in start and end steps.

Figure 98: The story pattern after adding *the machine* via structurization rules. Using a structurization rule allows us to specify more concrete link names than *in* or *on* which would have been derived had we polluted the textual scenario notation with *in/on the machine*.



The big difference to classical behavior driven development implementations is that we do not implement the meaning of a line of text as code but use graph transformations to derive an object diagram that is the foundation for the data model. The scenarios are not only a way to discuss functionality with end users, but also yield the basic entities for the modeling discussion. Whether that is a good thing because it shrinks the gap between end users and implementation or a bad thing because it frequently leads to changes of entities and the data model, making it more dynamic could be subject of further research. A start would be to use a NOSQL database to store the storyboard.

Conclusion

Before answering the main research question of this thesis “Can we improve tool support for storyboarding?” let us answer the partial questions from the “[Open Questions](#)” section on page 29.

1. *How can we give instant feedback on the storyboarding process?* First of all, getting tool support for storyboarding is now a matter of pointing a browser to our web application instead of running Fujaba. No installation of Java or any other Tool is required anymore since all desktop operating systems already provide a web browser. We significantly lowered the barrier of entry for interested developers or researchers. Our application itself resembles a storyboard which helps the user identify start, intermediate and end scenarios without explicit explanation. Whenever he presses enter in the textual scenario description the derived storyboard visualization next to it will get updated immediately. Intermediate steps of the process are hidden by default to prevent information overload.
2. *How can we give instant feedback on the learning of instances, concepts and relations.* Expanding the details of a scenario reveals the individual storyboarding steps automatically performed by the web application as described in “[The Masterplan](#)” on page 34. The first column shows the natural language parser result below the textual scenario description. The second column contains a textual version of the structurization rules that can be changed by the user. Any changes to the rules immediately trigger an update of the structurization

pattern and the resulting informal story pattern. The third column contains a textual and graphical version of the formalization rules that produce the final formal story patten.

The distinction between formal and informal story pattern is an important separation of concerns. While the informal pattern focuses on visualizing learned instances and relations using objects and links the formal pattern visualizes the learned concepts by showing updated type information. Giving feedback early by updating the rule visualizations and the intermediate results allows users to experiment with the rules and interactively explore the storyboarding process.

3. *How can we give instant feedback on the graph transformations?* Even more detailed information on the execution of a graph transformation is available by clicking the corresponding story pattern visualization. It will expand to a debug view with step by step visualizations of matching the story pattern to the working graph.
4. *How can we provide instant acceptance tests for the visible storyboard?* Providing the necessary functionality with a web service requires packaging Fujaba, CodeGen2 and XProM2 in a headless version. After getting the classpath right and writing a few lines of glue code our application currently provides a Fujaba `.ctr` as well as plain Java source code for the derived storyboard.
5. *Can this be achieved under the constraints of a browser environment?* Although some of the automated steps remain on the server side we showed that a browser is not an obstacle when it comes to executing graph transformations. Our story pattern interpreter can even use the rich markup capabilities of HTML to instantly visualize the rule execution. The most resource hungry tasks like natural language parsing and code generation remain on the server side and allow our web application to deliver *instant storyboarding* even on mobile and tablet devices.

Can we make storyboarding more accessible?

Instant Storyboarding as provided by our web application lures developers into an iterative learning process. The consequences of changes to a rule can be observed as soon as possible to answer the users question “What if I change this?” Interactively exploring the capabilities of the current rules used in the application will teach him to keep sentences simple and short. This will not only allow our application to derive a meaningful storyboard but also helps other humans to understand the scenario description.

If rephrasing a textual scenario does not give the desired result our web application allows users to influence each step in the storyboarding process. The Stanford Parser can be exchanged with OpenNLP or any other web service that provides a result in GraphSON. The structurization rules can be changed directly in the browser without the need of starting Fujaba and compiling the pattern to make them executeable. If necessary even the recommenders can be customized. Actually, it would be possible to use the web application to take the result of any JSON-P web service returning GraphSON as input and execute a completely different set of rules on it.

With the web application I developed to make storyboarding easily accessible I also created a framework that can be used to experiment with graph transformations, interpreters, recommenders, visualizations and natural language parsers. None of them is perfect, but that only poses a chance for other researchers to continue my work in an area of their interest.

Outlook

In the Outlook sections of the previous chapters I already pointed out minor changes that will improve the current web application. There are however more fundamental challenges that I can think of.

The recommenders used with the current framework are more or less dumb wrappers around other web resources. Their results are based on the working graph and the web resource that an algorithm turns into story pattern. Currently the algorithm is hard coded in each recommender. To me, the following interesting research topics arise: Can we learn the recommendations using genetic algorithms, neural networks or other machine learning approaches? Users could rate the results and draw the informal and formal story pattern they expect. This will lead to the question if learned recommendations can improve the formal story pattern compared to the current set of recommenders.

The second fundamental challenge I see is adding a project concept to the web application and allowing users to authenticate themselves to permanently save and share these projects. Adding explicit editors for activity and class diagrams to the web interface together with a code generation web service for complete projects could bring the whole story driven development process supported by Fujaba to the web.

The last challenge I see may seem blasphemous to some of my fellow researchers, but I have come to believe that the whole web application should be reimplemented in native JavaScript. One of the most time consuming tasks in software development is user interface design. Java

has recently started to allow declarative user interface modeling with JavaFX. While there is a framework for everything in Java the most elegant solution for web applications I came to know is AngularJS that allows data binding between HTML templates and JavaScript objects. Together with JavaScript drawing libraries like Draw2D this gives unparalleled possibilities to create browser based graphical editors. All without sacrificing testability when following the dependency injection philosophy enforced by AngularJS.

Bibliography

- Abbott, Russell J. "Program design by Informal English Descriptions." In: *Commun. ACM* 26.11 (1983), pp. 882–894. URL: http://sunset.usc.edu/classes/cs577a_2003/courses/ep/Program%20Design%20by%20Informal%20English%20Descriptions,%20Russell%20Abbott.pdf.
- Appelt, Douglas E. and Boyan Onyshkevych. "The common pattern specification language". In: *Proceedings of a workshop on held at Baltimore, Maryland: October 13-15, 1998*. TIPSTER '98. Baltimore, Maryland: Association for Computational Linguistics, 1998, pp. 23–30. DOI: 10.3115/1119089.1119095. URL: <http://acl.ldc.upenn.edu/X/X98/X98-1004.pdf>.
- Aschenbrenner, Nina et al. "Fujaba goes Web 2.0". In: *6th International Fujaba Days*. Ed. by Uwe Aßman, Jendrik Johannes, and Albert Zündorf. Dresden, Germany, 2008, pp. 10–14. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/ADJZ08.pdf>.
- Bao, Jie and Vasant Honavar. "Collaborative Ontology Building with Wiki@nt - A multi-agent based ontology building environment". In: *Proceedings of the 3rd International Workshop on Evaluation of Ontology-based Tools (EON2004)*. Oct. 2004, pp. 1–10.
- Beck, Kent. *Extreme Programming Explained: Embrace Change*. First. Boston: Addison-Wesley Professional, 1999, p. 224. ISBN: 0201616416.

- Benz, Dominik. "Capturing Emergent Semantics from Social Annotation Systems". PhD thesis. University of Kassel, Feb. 26, 2013. URL: <http://nbn-resolving.de/urn:nbn:de:hebis:34-2013022642523>.
- Brandes, Ulrik et al. "GraphML Progress Report: Structural Layer Proposal". In: *Proceedings of the 9th International Symposium Graph Drawing (GD '01) LNCS 2265*. Springer-Verlag, 2002, pp. 501–512. URL: <http://www.inf.uni-konstanz.de/algo/publications/beh-hm-gprsl-01.ps.gz>.
- Cer, Daniel M. et al. "Parsing to Stanford Dependencies: Trade-offs between Speed and Accuracy." In: *LREC*. Ed. by Nicoletta Calzolari et al. European Language Resources Association, 2010. ISBN: 2-9517408-6-7. URL: <http://dblp.uni-trier.de/db/conf/lrec/lrec2010.html#CerMJM10>.
- Church, Luke, Chris Nash, and Alan F. Blackwell. "Liveness in notation use. From music to programming". In: *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group (PPIG 2010)*. 2010, pp. 2–11. URL: http://www.academia.edu/1124877/Liveness_in_Notation_Use_From_Music_to_Programming.
- Cimiano, Philipp. *Ontology learning and population from text - algorithms, evaluation and applications*. Springer, 2006, pp. I–XXVIII, 1–347. ISBN: 978-0-387-30632-2.
- Cimiano, Philipp and Johanna Völker. "Text2Onto - A Framework for Ontology Learning and Data-driven Change Discovery". In: *Proceedings of the 10th International Conference on Applications of Natural Language to Information Systems (NLDB)*. Ed. by Andres Montoyo, Rafael Munoz, and Elisabeth Metais. Vol. 3513. Lecture Notes in Computer Science. Alicante, Spain: Springer, June 2005, pp. 227–238. URL: http://www.aifb.uni-karlsruhe.de/WBS/jvo/publications/Text2Onto_nldb_2005.pdf.
- Cunningham, H., D. Maynard, and V. Tablan. *JAPE: a Java Annotation Patterns Engine (Second Edition)*. Research Memorandum CS-00-10. Department of Computer Science, University of Sheffield, Nov. 2000. URL: <http://www.dcs.shef.ac.uk/~diana/Papers/jape.ps>.
- Cunningham, H. et al. "GATE: A framework and graphical development environment for robust NLP tools and applications". In: *Proceedings of the 40th Anniversary Meeting of the*

- Association for Computational Linguistics*. 2002. URL: <http://gate.ac.uk/sale/acl02/acl-main.pdf>.
- Diethelm, Ira. "Strictly models and objects first: Unterrichtskonzept und -methodik für objektorientierte Modellierung im Informatikunterricht". <http://d-nb.info/98668760X>. PhD thesis. University of Kassel, 2007, pp. 1–223. ISBN: 978-3-86805-007-3. URL: <http://kobra.bibliothek.uni-kassel.de/bitstream/urn:nbn:de:hebis:34-2007101119340/1/DissIraDruckfassungA5.1.pdf>.
- Diethelm, Ira, Leif Geiger, and Albert Zündorf. "Systematic Story Driven Modeling". In: *Technical Report* (Feb. 2004). URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/SDM04.pdf>.
- "Systematic Story Driven Modeling, a case study". In: Edinburgh, Scotland, May 24 - 28, 2004. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/DGZ04.pdf>.
- Dreyer, Jörn et al. "NT2OD Online - Bringing Natural Text 2 Object Diagram to the web". In: *ODiSE'10: Ontology-Driven Software Engineering Proceedings*. Ed. by Sergio de Cesare. Reno/Tahoe, Nevada, USA, Oct. 18, 2010. URL: http://dl.acm.org/ft_gateway.cfm?id=1937133&type=pdf.
- Ehrig, Hartmut, Michael Pfender, and Hans Jürgen Schneider. "Graph-Grammars: An Algebraic Approach". In: *SWAT (FOCS)*. IEEE Computer Society, 1973, pp. 167–180. URL: <http://dblp.uni-trier.de/db/conf/focs/focs73.html#EhrigPS73>.
- Ehrig, Hartmut and Karl Wilhelm Tischer. "Graph Grammars and Applications to Specialization and Evolution in Biology." In: *J. Comput. Syst. Sci.* 11.2 (1975), pp. 212–236. URL: <http://dblp.uni-trier.de/db/journals/jcss/jcss11.html#EhrigT75>.
- Geiger, Leif. "Fehlersuche im Modell: modellbasiertes Testen und Debuggen." PhD thesis. University of Kassel, 2011. URL: <http://d-nb.info/101373873X>.
- Geiger, Leif, Christian Schneider, and Carsten Reckord. "Template- and modelbased code generation for MDA-Tools". In: *3rd International Fujaba Days*. Paderborn, Germany, Sept. 2005. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/CodeGen2.pdf>.

- Geiger, Leif and Albert Zündorf. "Developing Tools with Fujaba XProM." In: *GTTSE*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Vol. 4143. Lecture Notes in Computer Science. Springer, 2006, pp. 344–356. ISBN: 3-540-45778-X. URL: <http://dblp.uni-trier.de/db/conf/gttse/gttse2006.html#GeigerZ06>.
- "Story driven testing - SDT." In: *ACM SIGSOFT Software Engineering Notes* 30.4 (2005), pp. 1–6. URL: <http://dblp.uni-trier.de/db/journals/sigsoft/sigsoft30.html#GeigerZ05>.
 - "Transforming Graph Based Scenarios into Graph Transformation Based JUnit Tests." In: *ACTIVE*. Ed. by John L. Pfaltz, Manfred Nagl, and Boris Böhlen. Vol. 3062. Lecture Notes in Computer Science. Springer, 2003, pp. 61–74. ISBN: 3-540-22120-4. URL: <http://dblp.uni-trier.de/db/conf/active/active2003.html#GeigerZ03>.
- Giese, Holger, Stephan Hildebrandt, and Andreas Seibel. "Improved Flexibility and Scalability by Interpreting Story Diagrams." In: *ECEASST 18* (2009). URL: <http://dblp.uni-trier.de/db/journals/eceasst/eceasst18.html#GieseHS09>.
- Graham, Paul. "Hackers & Painters: Big Ideas from the Computer Age". In: O'Reilly Media, Inc., 2004. Chap. The Other Road Ahead, pp. 56–86. URL: <http://www.paulgraham.com/road.html> (visited on 03/14/2014).
- Holt, Richard C., Andreas Winter, and Andy Schürr. "GXL: Towards a Standard Exchange Format". In: *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*. Limerick, June 2000. URL: <ftp://ftphost.uni-koblenz.de/ftp/outgoing/Reports/RR-1-2000/RR-1-2000.pdf>.
- King, Josh and Venu Satuluri. "Extracting Semantic Relations Using Dependency Paths". unpublished. URL: <http://www.cse.ohio-state.edu/~satuluri/final788.pdf>.
- Lin, Dekang and Patrick Pantel. "Discovery of inference rules for question-answering." In: *Natural Language Engineering* 7.4 (2001), pp. 343–360. URL: <http://dblp.uni-trier.de/db/journals/nle/nle7.html#LinP01>.
- Maes, Pattie. "Agents that Reduce Work and Information Overload". In: *Communications of the ACM* 37.7 (1994), pp. 30–40. URL: <http://www.cs.brandeis.edu/~cs125a/content/agentsmaes.doc>.

- Marcus, Mitchell P., Beatrice Santorini, and Mary Ann Marcinkiewicz. "Building a Large Annotated Corpus of English: The Penn Treebank." In: *Computational Linguistics* 19.2 (1993), pp. 313–330. URL: <http://dblp.uni-trier.de/db/journals/coling/coling19.html#MarcusSM94>.
- North, Dan. "Behavior Modification". In: *Better Software Magazine* (Mar. 2006). URL: http://www.stickyminds.com/s.asp?F=S10836_MAGAZINE_2.
- Resnick, P. and H. R. Varian. "Recommender systems". In: *Communications of the ACM* 40.3 (1997), pp. 56–58. ISSN: 0001-0782. DOI: <http://doi.acm.org/10.1145/245108.245121>. URL: https://wiki.cc.gatech.edu/scqualifier/images/c/c6/Resnick-Recommender_systems.pdf.
- Rosson, Mary Beth and John M. Carroll. *Usability Engineering: Scenario-Based Development of Human-Computer Interaction*. San Diego, CA: Academic Press, 2002. ISBN: 1-55860-712-9.
- Santorini, Beatrice. *Part-of-speech tagging guidelines for the Penn Treebank Project*. Tech. rep. MS-CIS-90-47. Department of Computer and Information Science, University of Pennsylvania, 1990. URL: <ftp://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz>.
- Schneider, Christian. "CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten". PhD thesis. 2007. URL: <http://kobra.bibliothek.uni-kassel.de/handle/urn:nbn:de:hebis:34-2007121319874>.
- Tanimoto, Steven L. "VIVA: A visual language for image processing". In: *Journal of Visual Languages & Computing* 1.2 (1990), pp. 127–139. ISSN: 1045-926X. DOI: 10.1016/S1045-926X(05)80012-6. URL: <http://www.sciencedirect.com/science/article/pii/S1045926X05800126>.
- Yosef, Mohamed Amir et al. "AIDA: An Online Tool for Accurate Disambiguation of Named Entities in Text and Tables". In: *PVLDB* 4.12 (2011), pp. 1450–1453. URL: <http://dblp.uni-trier.de/db/journals/pvladb/pvladb4.html#YosefHBSW11>.
- Zündorf, Albert. *Rigorous Object Oriented Software Development with Fujaba*. Draft Version 0.3. 2002. URL: <http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/Zuen02.pdf>.

Online references

- Anglin, Todd. *Using CORS with All (Modern) Browsers*. Oct. 3, 2011. URL: http://blogs.telerik.com/kendoui/posts/11-10-03/using_cors_with_all_modern_browsers (visited on 03/24/2014).
- Apache. *OpenNLP*. 2010. URL: <http://opennlp.apache.org> (visited on 07/19/2013).
- Appelt, Douglas E. *TextPro*. Oct. 10, 1999. URL: <http://www.ai.sri.com/~appelt/TextPro/> (visited on 07/19/2013).
- Bork, Manuel. *Maven2 Fujaba Plugin 2*. Software Engineering Group Kassel. 2007. URL: <http://www.se.eecs.uni-kassel.de/~maven/sites/mvnFujabaPlugin/dependencies.html> (visited on 03/25/2014).
- Bostock, Mike. *Force Layout*. 2012. URL: <https://github.com/mbostock/d3/wiki/Force-Layout> (visited on 03/25/2014).
- *Mike Bostock*. 2011. URL: <http://bost.ocks.org/mike/> (visited on 03/25/2014).
- Brandes, Ulrik, Markus Eiglsperger, and Jürgen Lerner, eds. *GraphML Primer. Declaring an Edge*. 2012. URL: <http://graphml.graphdrawing.org/primer/graphml-primer.html#GraphEdge> (visited on 03/24/2014).
- Commons, Creative. *Attribution 3.0 Unported*. 2007. URL: <http://creativecommons.org/licenses/by/3.0/> (visited on 03/25/2014).
- Crockford, Douglas. *Introducing JSON*. 2002. URL: <http://www.json.org/> (visited on 03/24/2014).

- Databases, D5: and Information Systems. *AIDA Web interface (aida)*. Max-Planck-Institut Informatik. 2011. URL: <https://gate.d5.mpi-inf.mpg.de/webaida/> (visited on 03/25/2014).
- Fang, Yidong. *JSON.simple. A simple Java toolkit for JSON*. 2008. URL: <https://code.google.com/p/json-simple/> (visited on 03/24/2014).
- Gliffy. *Online Diagram Software and Flowchart Software*. 2005. URL: <http://www.gliffy.com/> (visited on 07/19/2013).
- Google. *AngularJS. Superheroic JavaScript MVW Framework*. 2010. URL: <http://angularjs.org> (visited on 03/25/2014).
- *Freebase. A community-curated database of well-known people, places, and things*. 2007. URL: <http://www.freebase.com/> (visited on 03/25/2014).
 - *Google Drive Apps*. 2006. URL: <http://www.google.com/drive/apps.html> (visited on 07/19/2013).
 - *GWT Developer Guide - UI Binder*. 2009. URL: <http://www.gwtproject.org/doc/latest/DevGuideUiBinder.html> (visited on 07/19/2013).
 - *How to Attribute Freebase on Your Site*. 2014. URL: <http://www.freebase.com/policies/index> (visited on 03/25/2014).
 - *Research. Papers about Freebase*. 2010. URL: <http://wiki.freebase.com/wiki/Research> (visited on 03/25/2014).
 - *Search Cookbook. Freebase API*. 2014. URL: <https://developers.google.com/freebase/v1/search-cookbook> (visited on 03/25/2014).
 - *Search Overview. Freebase API*. 2014. URL: <https://developers.google.com/freebase/v1/search-overview> (visited on 03/25/2014).
 - *Web Analytics & Reporting – Google Analytics*. 2005. URL: <http://www.google.com/analytics/> (visited on 03/23/2014).
- Granger, Chris. *Light Table*. kickstarter. June 1, 2012. URL: <http://www.kickstarter.com/projects/306316578/light-table> (visited on 03/24/2014).
- *Light Table*. Home page. 2013. URL: <http://www.lighttable.com/> (visited on 03/24/2014).

- *Light Table - a new IDE concept*. Blog. Apr. 12, 2012. URL: <http://www.chris-granger.com/2012/04/12/light-table---a-new-ide-concept/> (visited on 03/24/2014).
- Hahn, Marcel and Ruben Jubeh. *Fujaba Web Runtime*. Software Engineering Group Kassel. 2010. URL: <https://gforge.cs.uni-kassel.de/projects/fujabaweb/rt/> (visited on 03/25/2014).
- Hall, Johan, Jens Nilsson, and Joakim Nivre. *MaltParser*. 2006. URL: <http://www.maltparser.org/> (visited on 03/24/2014).
- Harris, Tobin. *yUML*. 2009. URL: <http://yuml.me/> (visited on 03/24/2014).
- *yUML pipeline*. July 30, 2010. URL: https://groups.google.com/forum/#!msg/yuml/tL08P188c_0/XsvFV43r25sJ (visited on 07/19/2013).
- Hellesøy, Aslak. *Cucumber - Making BDD fun*. 2008. URL: <http://cukes.info/> (visited on 03/23/2014).
- Herz, Andreas. *Draw2D touch*. 2007. URL: <http://draw2d.org/> (visited on 03/25/2014).
- Houston, Mike. *jGraphViz*. 2008. URL: <http://jgraphviz.sourceforge.net/> (visited on 03/24/2014).
- Jianfeng, Xiao. *JAVA UTF-8*. 2010. URL: <http://92jsp.com/blog/default/2010/10/27/JAVA-UTF-8> (visited on 01/14/2012).
- Kassel, Software Engineering Research Group, ed. *Fujaba4Eclipse Update Site*. 2013. URL: <http://www.se.eecs.uni-kassel.de/fileadmin/se/update> (visited on 01/04/2014).
- Kassel, Team. *Fujaba4Eclipse update site*. 2012. URL: <http://www.se.eecs.uni-kassel.de/fileadmin/se/update/> (visited on 07/19/2013).
- Kesteren, Anne van. *Cross-Origin Resource Sharing*. Jan. 29, 2013. URL: <http://www.w3.org/TR/cors/> (visited on 06/29/2013).
- Kudryashov, Konstantin et al. *Writing Features*. 2014. URL: <http://docs.behat.org/en/latest/guides/1.gherkin.html#features> (visited on 02/18/2015).
- LeBlanc, Andrew. *S.Pr.A.W. Stanford Parser API for the Web*. 2010. URL: <https://github.com/LeBlanc/SPRAW> (visited on 03/24/2014).

- Loetzsch, Martin. *The Dot Markup Language*. 2002. URL: <http://martin-loetzsch.de/DOTML/> (visited on 03/24/2014).
- Mallette, Stephen and Marko A. Rodriguez. *GraphSON Reader and Writer Library*. 2012. URL: <https://github.com/tinkerpop/blueprints/wiki/GraphSON-Reader-and-Writer-Library> (visited on 06/11/2012).
- Marneffe, Marie-Catherine de and Christopher D. Manning. *Stanford typed dependencies manual*. Sept. 2008. URL: http://nlp.stanford.edu/software/dependencies_manual.pdf (visited on 03/25/2014).
- Mordani, Rajiv. *JSR-000315 Java™ Servlet 3.0. Maintenance Release*. Feb. 6, 2011. URL: <https://jcp.org/aboutJava/communityprocess/mrel/jsr315/index.html> (visited on 03/24/2014).
- North, Dan. *Introducing BDD*. Mar. 2006. URL: <http://dannorth.net/introducing-bdd/> (visited on 07/19/2013). Repr. of “Behavior Modification”. In: *Better Software Magazine* (Mar. 2006). URL: http://www.stickyminds.com/s.asp?F=S10836_MAGAZINE_2.
- *What is JBehave?* 2003. URL: <http://jbehave.org/> (visited on 03/23/2014).
- Owen, G. Scott. *Definitions and Rationale for Visualization*. Feb. 11, 1999. URL: <http://www.siggraph.org/education/materials/HyperVis/visgoals/visgoal2.htm> (visited on 06/30/2013).
- Piwik. *Free Web Analytics Software*. 2007. URL: <http://piwik.org/> (visited on 03/23/2014).
- Project, BibSonomy. *BibSonomy. The blue social bookmark and publication sharing system*. 2005. URL: <http://www.bibsonomy.org/> (visited on 03/24/2014).
- Project, Metro. *JAXB Reference Implementation*. 2003. URL: <https://jaxb.java.net/> (visited on 03/24/2014).
- *Mapping cyclic references to XML*. 2009. URL: https://jaxb.java.net/guide/Mapping_cyclic_references_to_XML.html (visited on 03/24/2014).
- Schmidt, Ryan. *canviz. JavaScript library for drawing Graphviz graphs to a web browser canvas*. 2006. URL: <http://code.google.com/p/canviz/> (visited on 03/25/2014).
- Scholtz, Bauke. *maximum length of HTTP GET request?* 2010. URL: <http://stackoverflow.com/a/2659995/828717> (visited on 03/24/2014).

- Selenium, ed. *Selenium. Web Browser Automation*. 2008. URL: <http://seleniumhq.org> (visited on 03/24/2014).
- Simpson, Kyle. *Defining Safer JSON-P*. 2010. URL: <http://www.json-p.org> (visited on 03/24/2014).
- Software, OpenLink. *Virtuoso SPARQL Query Editor*. 2009. URL: <http://dbpedia.org/sparql> (visited on 03/24/2014).
- Solutions, Yatta. *UML Lab*. 2012. URL: <http://www.uml-lab.com> (visited on 03/25/2014).
- Yatta. 2012. URL: <http://yatta.de> (visited on 03/25/2014).
- Stanford. *CoreNLP*. 2011. URL: <http://nlp.stanford.edu:8080/corenlp/> (visited on 07/19/2013).
- Team, GraphML. *The GraphML File Format*. 2002. URL: <http://graphml.graphdrawing.org/> (visited on 03/24/2014).
- thiscouldbeter. *Loading, Editing, and Saving a Text File in HTML5 Using Javascript*. Dec. 18, 2012. URL: <http://thiscouldbeter.wordpress.com/2012/12/18/loading-editing-and-saving-a-text-file-in-html5-using-javascript/> (visited on 03/25/2014).
- Victor, Bret. *Inventing on Principle*. 2012. URL: <http://vimeo.com/36579366> (visited on 06/30/2013).
- *Up and Down the Ladder of Abstraction*. Oct. 2011. URL: <http://worrydream.com/LadderOfAbstraction/> (visited on 06/30/2013).
- W3C. *File API. 8. The FileReader Interface*. 2013. URL: <http://www.w3.org/TR/FileAPI/#FileReader-interface> (visited on 03/25/2014).
- *HTML Components. Componentizing Web Applications*. 1998. URL: <http://www.w3.org/TR/NOTE-HTMLComponents> (visited on 03/25/2014).
- *Same Origin Policy*. 2009. URL: http://www.w3.org/Security/wiki/Same_Origin_Policy (visited on 03/24/2014).
- Wikipedia. *Chess*. 2001. URL: <http://en.wikipedia.org/wiki/Chess> (visited on 03/24/2014).

- Wikipedia. *Uses of English verb forms*. 2012. URL: http://en.wikipedia.org/wiki/Uses_of_English_verb_forms (visited on 03/25/2014).
- Zeigermann, Oliver and Daniel Florey. *jmtc. Java Minimal Template Engine*. 2010. URL: <https://code.google.com/p/jmtc/> (visited on 03/24/2014).
- Zündorf, Albert. *Rule Matching*. 2010. URL: <http://seblog.cs.uni-kassel.de/fileadmin/se/courses/MDESS10/MDE04RuleMatching/MDE04RuleMatching.html> (visited on 07/19/2013).
- *Story Driven Modeling with Fujaba. Turning Scenarios into Automated Tests*. Google Tech Talks. June 4, 2008. URL: https://www.youtube.com/watch?v=nwcsj_Iz4ao (visited on 03/24/2014).

Figures

Prefixes have the following meaning:

GR = Grammatical Relations, ISP = Informal Story Pattern, FSP = Formal Story Pattern

WG = Working Graph, SP = Story Pattern, SR = Structurization Rule

<i>An introduction to Storyboarding</i>	11
1 Informal Object Diagram	13
2 Verbose Collaboration Diagram	14
3 Derived Class Diagram	14
4 Object game	15
<i>Foundations of instant storyboarding</i>	19
5 eDOBS, the eclipse Document Object Browsing System	21
6 The hierarchical "layer cake"	22
7 Story diagram for <code>piece.move(newpos:Field)</code>	25

<i>Instant Storyboarding</i>	31	
8 The Masterplan for instant storyboarding	36	
9 Grammatical relations for the first sentence	39	
10 Grammatical relations for both sentences	39	
11 Screenshot: compact layout	40	
12 Screenshot: extended layout	42	
13 Class Diagram: core architecture	43	
14 OpenNLP Class Diagram	55	
15 FUML: Grammar Overview	67	
16 GR: "Alice is playing Chess."	69	
17 Shrunk Grammatical relations	74	
18 SP: Create an object for every proper noun	77	
19 SP: Delete vertexes	77	
20 FUML valueExpression grammar excerpt	79	
21 WG: Initial working graph	81	
22 SP: Initial Structurization Pattern	81	
23 SP: Identifying by type options	81	
24 SP: Choosing an initial search option	82	
25 WG: Collecting possible candidates	82	
26 WG: Choosing a candidate	82	
27 SP: Marking the match	83	
28 SP: Identifying to many options from current match	83	
29 SP: Choosing a search option	83	
30 WG: Collecting possible candidates	83	
31 WG: Choosing a candidate	84	
32 SP: Marking the match	84	
33 SP: Identifying to one options from current match	84	
34 SP: Choosing a search option	84	
35 WG: Collecting possible candidates	85	

36	WG: Choosing a match	85	
37	SP: Marking the match	85	
38	SP: Identifying optional to many options from current match	86	86
39	WG: Creating an object	86	
40	WG: Creating origin link	86	
41	WG: Assign attributes	87	
42	SP: Backtrack third search option	87	
43	WG: Backtrack third search option	87	
44	SP: Backtrack second search option	88	
45	WG: Backtrack third search option	88	
46	SP: Backtrack first search option	88	
47	WG: Backtrack first search option	89	
48	FUML: red / lime node	89	
49	FUML: lime / blue node	89	
50	FUML: blue / red node	89	
51	FUML: colors excerpt	90	
52	GR: "Alice and Bob are playing chess."	91	
53	SP: From grammar to object diagram: new object for nsubj	92	
54	SP: From grammar to object diagram: new object for dobj	92	
55	SP: From grammar to object diagram: new object for nsubj or dobj	92	92
56	WG: From grammar to object diagram: intermediate result 1	93	
57	SP: From grammar to object diagram: create predicate link	93	
58	WG: From grammar to object diagram: intermediate result 2	94	
59	SP: From grammar to object diagram: cleanup	94	
60	WG: From grammar to object diagram: final result	94	
61	ISP: Alice and Bob are playing Chess	98	
62	SP: given name recommendation	102	
63	SP: chess is a game	107	
64	GR: start scenario	111	

65	GR: end scenario	112
66	SP: match prepositions	114
67	SP: ownership	115
68	FUML: call syntax	116
69	WG: example collaboration statement	116
70	SP: Data Model, create object for proper noun	117
71	SP: Data Model, create object for noun	117
72	SP: Data Model, link noun and object	117
73	SP: Data Model, aggregate compound noun	118
74	SP: Data Model, aggregate compound proper noun	118
75	SP: Data Model, create attribute	119
76	SP: Data Model, link coreferences to noun	119
77	SP: Data Model, create owner link for poss relation	119
78	SP: Data Model, create link for predicates	120
79	SP: Data Model, create collaboration statement for VBZ	120
80	SP: Data Model, create collaboration statement parameter	121
81	WG: Data Model, informal object diagram	122

Instant Examples 131

82	SP: Alice's or Bob's turn	133
83	GR: Moving from e2 to e4	134
84	SR: capture prep_from	135
85	SR: capture prep_to depending on prep_from	136
86	FSP: prep_to depending on prep_from	136
87	SP: Web shop example	137
88	GR: The passive	138
89	SP: Web shop example (fixed)	139

90	SP: <i>Buy last coffee</i> start	140	
91	SP: <i>Buy last coffee</i> end	140	
92	GR: coffee machine example	141	
93	GR: coffee example without machine	142	
94	GR: coffee example without machine, grammar change I	143	
95	GR: coffee example without machine, grammar change II	144	
96	SP: coffee example after gherkin structurization	146	
97	SR: add <i>the machine</i>	147	
98	SP: coffee example with <i>the machine</i>	148	

Listings

Prefixes have the following meaning:

TS = Textual Scenario, SR = Structurization Rule

An introduction to Storyboarding 11

1 TS: A simple chess scenario 13

Foundations of instant storyboarding 19

2 Gherkin chess opening move scenario 24

Instant Storyboarding 31

3 Table based layout 45

4 Div based layout 47

5 „Alice is playing Chess“ OpenNLP parse 49

6 „Alice is playing Chess“ Stanford parse 49

7	GraphSON example	52
8	JSON-P call with GWT	53
9	Parser web service URL format	54
10	„Alice is playing Chess“ graphson	57
11	yuml.me notation example	63
12	FUML: “Alice is playing Chess.”	66
13	Apache Rewrite Rule	68
14	.dot “Alice is playing Chess.”	70
15	jmte .dot template	71
16	SR: Create an object for every proper noun	77
17	HTML title tags at vorname.com	101
18	PHP: extract names from FUML	103
19	PHP: build freebase query	104
20	PHP: calculate score for types	105
21	PHP: encode recommendation in FUML	106
22	PHP: send JSON-P encoded list of recommendations	106
23	Extended chess scenario	111
24	SR: match prepositions	114
25	SR: ownership	115
26	JMTE: collaboration statement	116
27	Java: start scenario set up	129

Instant Examples 131

28	SR: Change link verbs to nouns	132
29	TS: Add turn example	132
30	TS: Add source example	133
31	SR: capture prep_from	134

32	SR: capture <code>prep_to</code> depending on <code>prep_from</code>	135
33	TS: Web shop example	137
34	SR: swap order for <i>add to</i>	138
35	SR: add <i>in</i> link for passive sentence	139
36	SR: change type of the shopping cart to <code>Cart</code>	139
37	Gherkin <i>Buy last coffee</i> scenario	140
38	Gherkin <i>Buy last coffee</i> start and end scenario	140
39	SR: gherkin structurization	145
40	Gherkin <i>Buy last coffee</i> start and end scenario with <i>the machine</i>	146
41	SR: add <i>the machine</i>	147

Tables

<i>Instant Storyboardg</i>	31	
1 FUML notations and their visual representation		64
2 Part of speech tags	76	

This thesis aims at empowering software customers with a tool to build software tests them selves, based on a gradual refinement of natural language scenarios into executable visual test models. The process is divided in five steps:

1. First, a natural language parser is used to extract a graph of grammatical relations from the textual scenario descriptions.
2. The resulting graph is transformed into an informal story pattern by interpreting structurization rules based on Fujaba Story Diagrams.
3. While the informal story pattern can already be used by humans the diagram still lacks technical details, especially type information. To add them, a recommender based framework uses web sites and other resources to generate formalization rules.
4. As a preparation for the code generation the classes derived for formal story patterns are aligned across all story steps, substituting a class diagram.
5. Finally, a headless version of Fujaba is used to generate an executable JUnit test.

The graph transformations used in the browser application are specified in a textual domain specific language and visualized as story pattern. Last but not least, only the heavyweight parsing (step 1) and code generation (step 5) are executed on the server side. All graph transformation steps (2, 3 and 4) are executed in the browser by an interpreter written in JavaScript/GWT.

This result paves the way for online collaboration between global teams of software customers, IT business analysts and software developers.



Jörn Friedrich Dreyer learned in a bank, studied business informatics, took the red pill and went down the rabbit hole of software engineering. After co-founding Yatta Solutions he finished his Ph.D. and went on to write code for ownCloud. He recently found his way back to business informatics as a solutions architect, filling the gap between ownCloud and customer requirements.

To contact him visit his blog:
IT Business Tonic
tools and hacks that glow in the dark
<http://www.butonic.de>