Quality Assurance of Textual Models within Eclipse using OCL and Model Transformations

Thorsten Arendt, Gabriele Taentzer, Alexander Weber

Philipps-Universität Marburg, Germany
{arendt,taentzer,weber87}@informatik.uni-marburg.de

Abstract. Modern software development processes often use domain-specific modeling languages (DSMLs) combined with custom code generators and/or interpreters. Especially textual DSMLs as provided by Eclipse Xtext are becoming more and more popular. As a consequence, software quality assurance frequently leads back to quality assurance of the involved textual models. Here, various quality aspects have to be considered highly depending on the modeling purpose and domain. In this paper, we present a quality assurance tool set for textual models in Eclipse using several interrelated components like Xtext, EMF Refactor, Henshin and the OCL tools which are all based on the Eclipse Modeling Framework (EMF). The practicability and flexibility of this tool set are demonstrated by the design and implementation of a case study that is based on a textual modeling language for simple web applications named SWM (Simple Web Modeling Language).

Keywords: model-based development, textual modeling, quality assurance

1 Introduction

The use of models in modern software development processes is becoming more and more popular. Model-based software development (MBSD) lifts software models to be the primary artifacts in the software development process. This is especially true in model-driven software development (MDSD) where models are finally used for code generation purposes. Moreover, the use of (often textual) domain-specific modeling languages (DSMLs) is a promising trend in modern software development processes to overcome the drawbacks concerned with the universality and the broad scope of general-purpose languages like the Unified Modeling Language (UML) [18]. Such a DSML can help to bridge the gap between a domain experts view and the implementation.

Often, a DSML comes along with a code generator and/or interpreter to provide functionality that should be hidden from the domain expert. In the generator case, high code quality can be reached only if the quality of input models is already high. Typical quality assurance techniques considering the model syntax are model metrics, model smells, and model refactorings. In [2], we present the integration of these techniques in a predefined quality assurance process that can be adapted to specific project needs.

This paper contributes a **flexible quality assurance tool set for textual models** supporting quality assurance techniques like model metrics, smells, and refactorings integrated in textual model editors within the Eclipse IDE [7]. This set integrates the following tools which are all build atop the Eclipse Modeling Framework (EMF) [20, 8], a widely used open source technology used in MBSD: `Xtext` [22] for providing the language infrastructure, `EMF Refactor` [4, 9] for providing model quality assurance tooling, query languages like the `Object Constraint Language` (OCL/MDT) [17, 15, 21] for specifying quality assurance techniques, and the model transformation language `Henshin` [1, 13] for specifying refactorings. Here, EMF Refactor's code generation facilities provide the designer to concentrate on the essential specification parts only.

To demonstrate the practicability and flexibility of this tool set we present a case study for quality assurance of textual models. As example language we take a DSML called Simple Web Model (SWM) for defining a specific kind of web applications in a platform-independent way[1]. In the case study, we concentrate on quality aspect *completeness*, i.e., we analyze whether SWM models are ready for code generation and improve model parts using domain-specific refactorings. The case study shows that Eclipse is particularly valuable for implementing model quality assurance tools. On the one hand, the plugin technology provides a flexible choice of concrete specification languages. On the other hand, the use of the abstract syntax (provided by EMF) for implementing quality assurance techniques provides a flexible use for visual and textual models in Eclipse.

The paper is structured as follows: In Section 2, we reflect the used model quality assurance process. Section 3 presents the textual SWM language and an example SWM instance model being used in Section 4 to discuss and present techniques and tool support for quality assurance of SWM models. Finally, we conclude with related work in Section 5 and a summary in Section 6.

2 Model quality assurance

The increasing use of model-based or model-driven software development processes induces the need for high-quality software models. In [2], we propose a model quality assurance process that consists of two sub-processes: a process for the specification of project-specific model quality assurance techniques, and a process for applying them on concrete software models during a MBSD process (see right-hand side of Figure 1). For a rough model overview (for example, during a model review), a report on model metrics might be helpful. Furthermore, a model can be checked against the existence (respectively absence) of specified model smells. Each model smell found has to be interpreted in order to evaluate whether it should be eliminated by a suitable model modification (either by a manual model change or a refactoring). This check-improve cycle should be performed as long as needed to get a reasonable model quality.

In our approach, we define a process for specifying new quality assurance techniques as shown in the left-hand side of Figure 1. After having identified the

[1] Several variations of SWM are used in literature, for example in [6].

intended modeling purpose the most important quality goals are selected. Here, we have to consider several conditions influencing the selection of significant quality aspects being the most important ones for modeling in a specific software project. The selection of significant quality aspects depends on the modeling purpose. Since modeling purposes are quite different and vary in several software projects, a quality aspect that is very important in one software project might be less important in other ones.

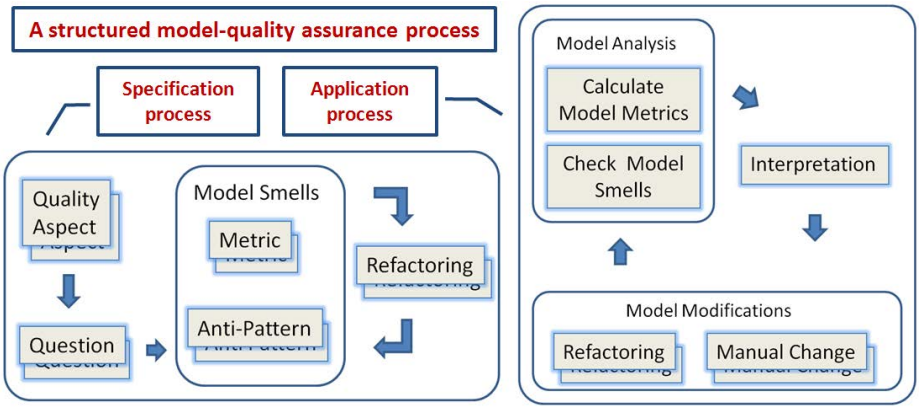

Fig. 1. A structured model-quality assurance process

In the next step, static syntax checks for these quality aspects are defined. This is done by formulating questions that should lead to so-called model smells hinting to model parts that might violate a specific model quality aspect. Some of these answers can be based on metrics. Other questions may be better answered by considering specific patterns which can be formulated on the abstract syntax of the model. A specified smell serves as precondition of at least one model refactoring that can be used to restructure models in order to improve model quality aspects but appreciably do not influence the semantics of the model.

3 SWM: simple web modeling language

In the case study, we assume the following scenario (taken from [6]): A software development company is repeatedly building simple web applications being mostly used to populate and manage persistent data in a database. Here, a typical three-layered architecture following the Model-View-Controller (MVC) pattern [12] is used. As implementation technologies, a relational database for persisting the data as well as plain Java classes for retrieving and modifying the data are employed for building the model layer. The company decided to develop its own textual DSML called Simple Web Modeling Language (SWM) for defining their specific kind of web applications in a platform-independent way. Based on SWM instances, platform-specific models following the MVC pattern

should be derived with model transformations from which the Java-based implementations are finally generated. Considering this transformation chain, the finally generated Java code defines the semantics of the SWM language.

The SWM language is defined as follows. A `WebModel` consists of two parts: a `DataLayer` for modeling entities which should be persisted in the database, and a `HypertextLayer` presenting the web pages of the application. An `Entity` owns several `Attribute`s (each having a `SimpleType`) and can be related to several other entities. A `Page` is either a `StaticPage` having a static content or a `DynamicPage` (`IndexPage` or `DataPage`) having a dynamic content depending on a referenced entity type. An `IndexPage` lists objects of this entity whereas a `DataPage` shows concrete information on a specific entity like its name, attributes, and references. Finally, pages are connected by `Link`s.

A prominent environment for textual modeling in Eclipse is Xtext [22] providing an exhaustive language infrastructure for the development of textual, grammar-based DSMLs. Listing 1 shows an excerpt of the SWM Xtext grammar[2]. This excerpt shows altogether three production rules, i.e. for `Entity`, `Attribute`, and `IndexPage` elements. The language terminals are defined using inverted commas (like *'index page'*). The additional information to derive a meta model is given by assignments. The assignments representing attributes and cross references are defined as single-valued assignments (= operator for mandatory attributes like *name*; additional ? operator for the optionally referenced *entity*) whereas the assignments representing containment references are all defined to be multi-valued (+ operator is used in addition to the = operator).

Listing 1. Part of the constructive Xtext grammar of the SWM language

```
...
Entity: 'entity' name=ID '{'
        attributes+=Attribute*
        references+=Reference*  '}' ;
Attribute:  'att' name=ID ':' type=SimpleType ;
...
IndexPage:
 'index page' name=ID ('shows entity' entity=[Entity])?'{'
 links+=Link* '}' ;
...
```

4 Quality assurance for SWM models

In this section, we use the structured model quality assurance process presented in Section 2 for textual models of the SWM language. First, we define concrete quality assurance techniques for SWM. Then, we demonstrate how these techniques are applied on a concrete SWM instance model. Finally, we show concrete specifications using different specification languages.

[2] The complete grammar can be found in Appendix A and is partially taken from [6].

4.1 Quality assurance techniques for SWM models

Since platform-specific models should be derived from SWM models and should be used to generate the Java-based implementations, the major quality aspect to be fulfilled on SWM models is *Completeness*. A model is complete if it contains all relevant information, and if it is detailed enough according to the modeling purpose [16]. This means for SWM models that (A) on the data layer each entity must contain all relevant attributes and references to other entities whereas (B) the hypertext layer must contain a complete set of (potentially linked) pages which should be depicted within the web application. Potential SWM model smells violating quality aspect *Completeness* are:

Empty Entity The entity does not have any attributes or references to other entities. (This violates completeness issues of type A.)

No Dynamic Page The entity is not referenced by a dynamic page to be depicted in the web application. (type B)

Unused Entity The entity is referenced neither by a dynamic page nor by another entity. (types A and B)

Missing Link The index page is not linked by the start page of the web application. (type B)

Furthermore, several metrics can be used to analyze completeness of SWM models. For example, metrics *Number of Entities in the Model (NEM)* and *Number of Dynamic Pages in the Model (NDPM)* can be used to get a first overview on the model structure. Here, a ratio between the values of these metrics less than 1 : 2 might be a hint for missing dynamic pages [3]. Similarly, metrics *Average number of Attributes (resp. References) in Entities of the Model (AvNAE resp. AvNRE)* are useful to detect missing information in the data layer.

After having specified appropriate model smells, suitable refactorings have to be defined in order to support the handling of smelly SWM models. Smells *No Dynamic Page* and *Unused Entity* can be eliminated by a refactoring which inserts both an index page and a data page referencing the corresponding entity to the hypertext model (refactoring `Insert Dynamic Pages`). For eliminating smell *Missing Link* an appropriate refactoring `Update Links to Index Pages` can be used that ensures that the start page owns links to all index pages of the model. Finally, there is no adequate refactoring to eliminate smell *Empty Entity*. Here, manual model changes should be performed.

4.2 Application of quality assurance techniques to SWM models

We now assume that the software company has to develop a web application for the rental system of a vehicle rental company. Listing 2 shows a first SWM model being developed in an early stage of the development process.

For the first overview on a model, a report on project-specific model metrics might be helpful. Calculated metric values are presented in a specific view within the Eclipse workbench. For reporting purposes, the results can be exported using

[3] I.e., one entity should be referenced by both an index page and a data page.

several output formats (like PDF, HTML, or widely used MS Office formats) and designs (like simple lists or tube diagrams). In our concrete example model, metrics NEM and NDPM (see Section 4.1) are calculated to `4` and `3`, respectively. This means that there are more entities in the web model than dynamic pages hinting to potentially missing dynamic pages.

Listing 2. Example SWM instance model before model review

```
webmodel VehicleRentalCompany {
    data {
        entity Customer {
            att name : String
            att email : Email
            ref address : Address }
        entity Address {
            att street : String
            att city : String }
        entity Car {
            att type : String }
        entity Agency {  }          }
    hypertext {
        index page carindex shows entity Car {
            link to page cardata }
        data page cardata shows entity Car {  }
        index page agencyindex shows entity Agency {     }
        static page indexpage {
            link to page carindex
            link to page agencyindex }
        start page is indexpage          }   }
```

To make this problems more explicit (and thus more obvious), EMF Refactor supports analysis functionality with respect to so-called model smells representing model parts to be improved. As for model metrics, our tool environment provides a configuration of specific model smells being relevant within the current project. Similarly to the metrics calculation process, a smell analysis can be triggered from an an element shown in the textual model editor. The results of a smell analysis are presented in a specific view within the Eclipse workbench. The bottom part of Figure 2 shows two entities being not referenced by a dynamic page (smell *No Dynamic Page* on entities *Customer* and *Address*) and one occurrence of smells *Empty Entity* and *Unused Entity* each. After selecting a concrete smell occurrence in the tree-based view the involved element is highlighted in the textual editor (see top part of Figure 2).

Besides manually changing the model, refactoring is the technique of choice to eliminate occurring smells. In our example, we can use refactoring *Insert Dynamic Pages* to eliminate smell *No Dynamic Page* on entity *Customer*[4].

[4] Note that we do not eliminate smell *No Dynamic Page* on entity *Address* since this entity is referenced by entity *Customer*, i.e. it is part of this entity.

The refactoring is triggered from the context menu of entity *Customer* (see highlighted part in Figure 2). Then, the tool set provides two previews: the first for visualizing model changes performed by the refactoring, the second for a concrete overview on smell occurrence changes when applying the refactoring. The result of refactoring *Insert Dynamic Pages* is shown in Figure 3. Two dynamic pages (an index page and a data page) referencing entity *Customer* are inserted into the hypertext layer of the model. Furthermore, the inserted data page is linked by the index page which is in turn linked by the static page named *indexpage* being the starting page of the hypertext layer (see Listing 2).

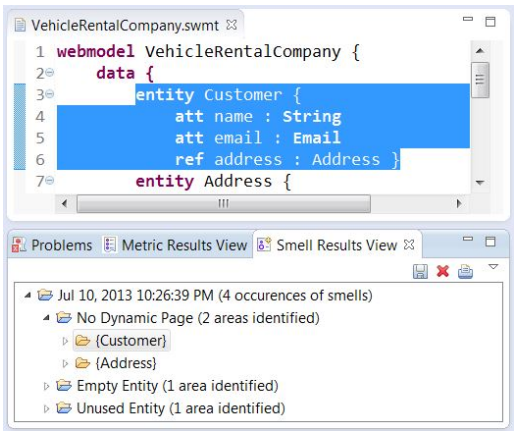

Fig. 2. Report of concrete smell occurrences in our example SWM model and highlighting of involved element in smell *No Dynamic Page* within the textual Xtext editor

```
18 static page indexpage {
19     link to page carindex
20     link to page agencyindex
21     link to page customerindex }
22 data page customerdata shows entity Customer { }
23 index page customerindex shows entity Customer {
24     link to page customerdata }
```

Fig. 3. Inserted and changed model elements after applying refactoring *Insert Dynamic Pages*

Due to space limitations, we have to skip further model analysis and refactoring steps here. However, we think that the application of the quality assurance process and the handling of the supporting tools are sufficiently and plausibly presented. For more detailed discussions on process and tooling we refer to [4]. An improved version of the example model concerning quality aspect *completeness* can be found in Appendix B of this paper.

4.3 Specification of quality assurance techniques for SWM models

EMF Refactor provides a wizard-based specification process for quality assurance techniques as well as basic code generation facilities. This has the advantage that the designer can concentrate on the essential specification part only.

Listing 3. OCL specification of SWM smell `No Dynamic Page`

```
context WebModel
def: noDynamicPages(): Set(Entity) =
 Entity.allInstances() -> excludesAll
  (DynamicPage.allInstances() -> collect(entity))
```

OCL has been proven to be well-suited to specify metrics. For example, metric *NEM* is simply defined using OCL expression `self.dataLayer -> size()` on context element `WebModel`. Listing 3 shows the OCL specification of SWM model smell *No Dynamic Pages*. OCL operation `noDynamicPages()` returns the set of entities being not referenced by any dynamic page (line 2): Starting from all instances of type `Entity` within the web model (line 3), we exclude those which are referenced by at least one dynamic page (line 4).

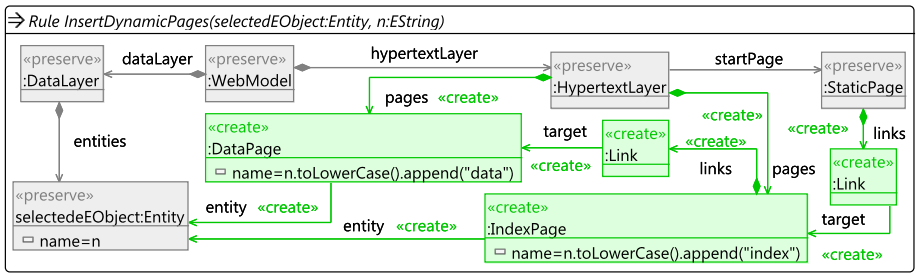

Fig. 4. Henshin rule specification for SWM refactoring `Insert Dynamic Pages`

For refactoring specification we use Java and the EMF model transformation language Henshin [1, 13] combined with OCL expressions being used for pre-condition checking (not shown here). Here, the use of a model transformation language like Henshin for refactoring specifications is a straightforward task. Figure 4 shows the Henshin rule specifying refactoring `Insert Dynamic Pages`. It uses elements of the SWM meta model which is generated by the Xtext framework. Nodes (and edges) tagged by $\langle\langle preserve \rangle\rangle$ represent unchanged model elements whereas those tagged by $\langle\langle create \rangle\rangle$ represent new ones. In our example, starting with the contextual element of type `Entity` (specified using node name *selectedEObject*), both a new index page and a new data page referencing this entity are created. Moreover, the inserted data page is linked by the index page which is in turn linked by the starting page of the hypertext layer.

All the sources of the case study (code, models, quality assurance techniques, etc.) can be found in the download section of the EMF Refactor web site [9].

5 Related Work

In this section, we give an overview on quality assurance tools within the EMF world, especially in the field of textual modeling using Xtext. For a comprehensive overview also considering quality assurance of UML models we refer to [4].

To the best of our knowledge, explicit tool support for metrics calculation on EMF models is not yet available (besides the EMF Refactor tooling). The EMF Query Framework [19] can be used to construct and execute query statements to compute metrics and to check constraints. The configuration of queries in suites as well as reports on query results in various forms are not provided. The EMF Validation Framework [10] supports the construction and assurance of

well-formedness constraints for EMF models. To the best of our knowledge, the functionalities of both frameworks are not integrated into textual Xtext editors.

The Epsilon language family [11] provides the Epsilon Validation Language (EVL) to validate EMF-based models with respect to constraints that are, in their simplest form, quite similar to OCL constraints. For reporting, EVL supports a specific validation view reporting the identified inconsistencies in a textual way. Suitable quick fixes are formulated in the Epsilon Object Language (EOL) being the core language of Epsilon. It is therefore not specifically dedicated to model refactoring. For this purpose, Epsilon provides the Epsilon Wizard Language (EWL) [14]. We compare our first refactoring prototype with EWL in [3]. Again, to the best of our knowledge, functionalities provided by Epsilon languages can not be used within textual Xtext editors in an integrated way.

In another approach, the authors propose the definition of EMF-based refactoring in a generic way [5]. However, they do not consider the comprehensive specification of preconditions. Our experiences in refactoring specification show that it is mainly the preconditions that cannot be defined generically.

Xtext has outstanding support for static model analysis and validation. Custom constraints and quick fixes can be defined to tackle errors and warnings instantaneously. However, the main purpose of these constraints and quick fixes is to address model consistency. They are not especially dedicated to quality assurance in a common sense. Moreover, there is no support for custom configurations of validation suites. Furthermore, Xtext provides basic refactoring functionality for generic renaming of arbitrary model elements. Support for custom refactorings (for example, for custom DSMLs like SWM) is not provided.

6 Conclusion and Future Work

In this paper, we present a flexible tool set for quality assurance of textual models within Eclipse. The tool set integrates a number of tools which are all built atop EMF (Xtext, EMF Refactor, OCL/MDT, and Henshin). In a case study, we use a DSML for defining a specific kind of web applications and concentrate on the quality aspect *completeness*. We use model metrics and model smells for static analysis and model refactoring for improving the structure of the models.

The implementation shows that (1) the structured model quality assurance process presented in [2] can be also adapted to textual models and that (2) the tool set presented in [4] is flexible enough to be integrated in textual model editors provided by Xtext. Here, EMF Refactor's code generation facilities provide the designer to concentrate on the essential specification parts only. Moreover, the case study shows that Eclipse is particularly valuable for implementing model quality assurance tools. On the one hand, the plugin technology provides a flexible choice of concrete specification languages. On the other hand, the use of the abstract syntax (provided by EMF) for implementing quality assurance techniques provides a flexible use for visual and textual models in Eclipse.

Future work on the topic presented in this paper is separated into two directions. On the one hand, we want to integrate further specification languages

into the EMF Refactor infrastructure. Here, we are currently working on the integration of EMF Query. On the other hand, it would be useful to provide a suite of predefined metrics, smells, and refactorings for specific Xtext grammars in order to analyze and improve the structure of the corresponding language.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and tools for In-Place EMF Model Transformation. In: MoDELS 2010. pp. 121–135. LNCS, Springer (2010)
2. Arendt, T., Kranz, S., Mantz, F., Regnat, N., Taentzer, G.: Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support. In: Software Engineering. LNI, vol. 183, pp. 63–74. GI (2011)
3. Arendt, T., Mantz, F., Schneider, L., Taentzer, G.: Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study. In: Model-Driven Software Evolution, Workshop Models and Evolution (2009)
4. Arendt, T., Taentzer, G.: A tool environment for quality assurance based on the Eclipse Modeling Framework. Journal on Automated Software Engineering 20, 141–184 (2013)
5. Aßmann, U., Bartho, A., Bürger, C., Cech, S., Demuth, B., Heidenreich, F., Johannes, J., Karol, S., Polowinski, J., Reimann, J., Schroeter, J., Seifert, M., Thiele, M., Wende, C., Wilke, C.: DropsBox: the Dresden Open Software Toolbox. Software & Systems Modeling pp. 1–37 (2012)
6. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, Morgan & Claypool (2012)
7. Eclipse. http://www.eclipse.org/
8. Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/
9. EMF Refactor. http://www.eclipse.org/emf-refactor/
10. EMF Validation Framework. http://www.eclipse.org/modeling/emf/?project=validation
11. Epsilon. http://www.eclipse.org/epsilon/
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software . Addison-Wesley, Boston, MA, USA (1995)
13. Henshin. http://www.eclipse.org/henshin/
14. Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update Transformations in the Small with the Epsilon Wizard Language. Journal of Object Technology 6(9), 53–69 (2007)
15. Model Development Tools (MDT). http://www.eclipse.org/modeling/mdt/
16. Mohagheghi, P., Dehlen, V., Neple, T.: Definitions and Approaches to Model Quality in Model-Based Software Development – A Review of Literature. Information and Software Technology 51(12), 1646–1669 (2009)
17. OMG: Object Constraint Language (OCL). http://www.omg.org/spec/OCL/2.3.1/
18. OMG: Unified Modeling Language (UML), http://www.uml.org
19. EMF Query. http://www.eclipse.org/projects/project.php?id=modeling.emf.query
20. Steinberg, D., Budinsky, F., Patenostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd Edition. Addison Wesley (2008)
21. Willink, E.D.: Modeling the OCL Standard Library. ECEASST 44 (2011)
22. Xtext. http://www.eclipse.org/Xtext

A Complete Xtext grammar of the SWM language

```
grammar org.eclipse.emf.refactor.examples.SimpleWebModel
                with org.eclipse.xtext.common.Terminals
generate simpleWebModel "http://www.eclipse.org/SWM/1.0"

WebModel: 'webmodel' name=ID '{'
        dataLayer=DataLayer
        hypertextLayer=HypertextLayer
    '}';
DataLayer: 'data {' {DataLayer}
        entities+=Entity*
    '}';
Entity: 'entity' name=ID '{'
        attributes+=Attribute*
        references+=Reference*
    '}';
Attribute: 'att' name=ID ':' type=SimpleType
    ;
enum SimpleType: Boolean | Email | Integer | String
    ;
Reference: 'ref' name=ID ':' type=[Entity]
    ;
HypertextLayer:     'hypertext {'
        pages+=Page+
        'start page is' startPage=[StaticPage]
    '}';
Page: StaticPage | DynamicPage
    ;
StaticPage: 'static page' name=ID '{'
    links+=Link*
    '}';
Link: 'link to page' target=[Page]
    ;
DynamicPage: IndexPage | DataPage
    ;
IndexPage:
 'index page' name=ID ('shows entity' entity=[Entity])?'{'
 links+=Link*
 '}';
DataPage:
 'data page' name=ID ('shows entity' entity=[Entity])? '{'
 links+=Link*
 '}';
```

Listing 4. Complete Xtext grammar of the SWM language

B Example SWM instance of model Vehicle Rental Company (after model review)

```
webmodel VehicleRentalCompany {
    data {
        entity Customer {
            att name : String
            att email : Email
            ref address : Address
            ref account : BankAccount  }
        entity Address {
            att street : String
            att postalCode : Integer
            att city : String }
        entity BankAccount {
            att number : Integer
            att bankCode : String
            att bankName : String  }
        entity Car {
            att manufacturer : String
            att type : String
            att power : Integer }
        entity Agency {
            ref address : Address }
    }
    hypertext {
        index page carindex shows entity Car {
            link to page cardata }
        data page cardata shows entity Car {  }
        index page agencyindex shows entity Agency {
            link to page agencydata }
        data page agencydata shows entity Agency {  }
        index page customerindex shows entity Customer {
            link to page customerdata }
        data page customerdata shows entity Customer { }
        static page indexpage {
            link to page agencyindex
            link to page carindex
            link to page customerindex }
        start page is indexpage
    }
}
```

Listing 5. Example SWM instance of model Vehicle Rental Company (after model review)