The SDMLib solution to the FIXML case for TTC2014

Christoph Eickhoff, Tobias George, Stefan Lindel, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

cei|tge|slin|zuendorf@cs.uni-kassel.de

This paper describes the SDMLib solution to the FIXML case for the TTC2014 [9]. SDMLib provides Java code generation for class models / class diagrams. In addition, SDMLib provides a mechanism for learning class models from generic example object structures. Thus, for the FIXML case we just added an XML reader that reads an example file and creates a generic object structure reflecting its content.

1 Introduction

Our team at Kassel University found this case particularly interesting as we give a course on CASE tool construction where one assignment to the students is to learn a class diagram from an XML file containing object descriptions but without an explicit XML schema. Thus, the case looked quite familiar to us.

In addition, our team has developed a software development process called *Story Driven Modeling* [3, 1]. Story Driven Modeling starts with textual scenarios that describe example situations and how they evolve through the execution of a certain user action. Next, the textual scenarios are extended with informal object diagrams modeling how the desired program might represent the described situations as object structures at runtime. Initially, the informal object diagrams may omit object types. The types are added in another design step that formalizes the object diagrams until a class diagram can be derived. This initial class diagram may be extended several times in order to support additional scenarios and in order to e.g. add support for certain design patterns like composite pattern or visitor pattern or strategies. Then, an implementation of the modeled classes may be generated using e.g. Fujaba [2] or UMLLab [8] or SDMLib [7].

To support Story Driven Modeling, SDMLib provides *Generic Object Diagrams* [5]. Generic Object Diagrams are able to represent untyped object structures, they allow to add type information at runtime and SDMLib is able to learn a class diagram from Generic Object Diagrams and to generate a Java implementation from it. This is discussed in Section 2.

To address the FIXML case, we just used the standard Java XML parser and wrote a small transformation that translate the read XML data into a Generic Object Diagram. Then, we used the SDMLib mechanism to learn a class diagram and to generate a Java implementation, cf. Section 3.

2 SDMLib support for Story Driven Modeling

SDMLib provides classes for a generic graph. This allows users to create generic object structures, e.g., in a JUnit test as shown in Listing 1.

```
1 ...
2 GenericGraph graph = new GenericGraph();
3
4 GenericObject building = graph.createObjects("WilliAllee", "Building")
```

Submitted to:
TTC 2014

© Albert Zündorf
This work is licensed under the
Creative Commons Attribution License.

```
5  .with("name", "WA73");
6
7  GenericObject wa13 = graph.createObjects("seFloor", "Floor")
8  .with("name", "WA13").with("level", "1");
9
10 graph.createLinks().withSrc(building).withTgt(wa13).withTgtLabel("has");
11
12 GenericObject wa03 = graph.createObjects("digitalFloor", "Floor")
13 .with("name", "WA03").with("level", "0").with("guest","Ulrich");
14
15 graph.createLinks().withSrc(building).withTgt(wa03).withTgtLabel("has");
16 ...
```

Listing 1: Creating a Generic Object Model via Java API

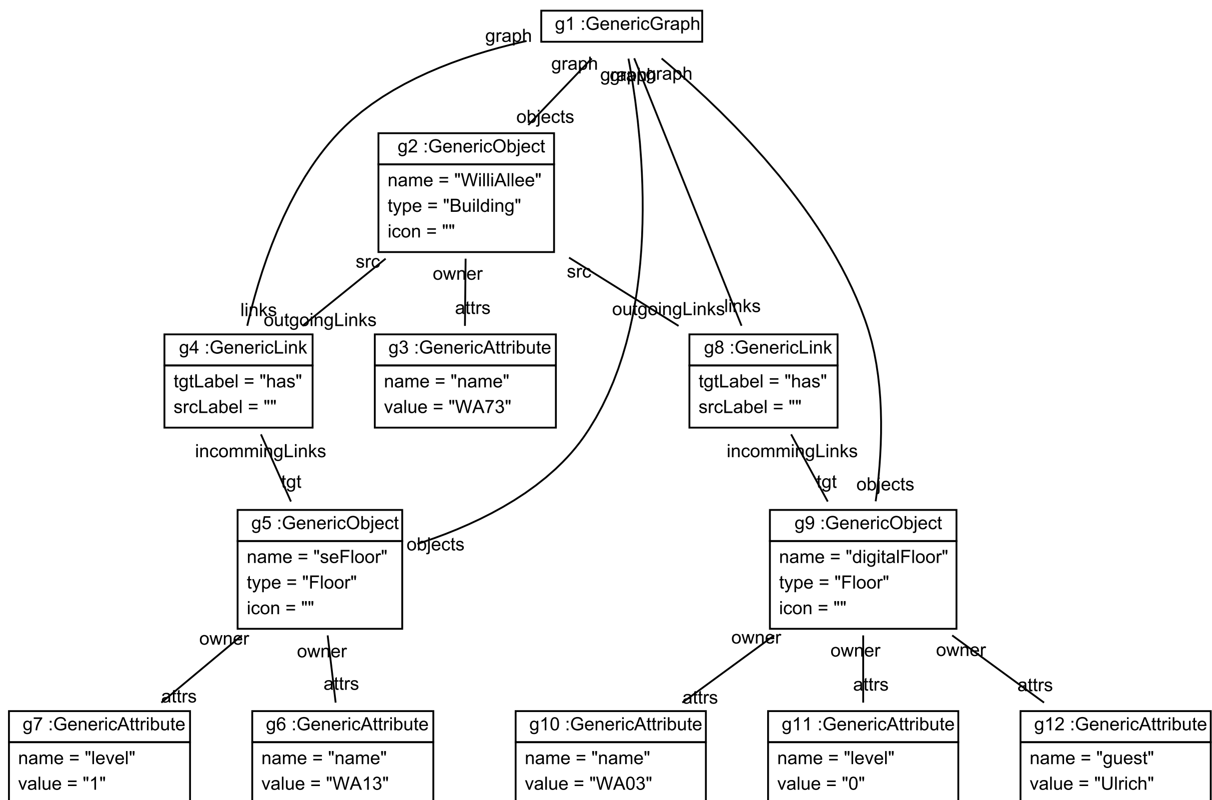

Figure 1: Example Informal Generic Object Model

Using GraphViz [4], SDMLib is able to render object models as object diagrams, cf. figure 1.

Listing 2 shows the SDMLib algorithm for learning a class model from a generic object structure. First, line 4 loops through all generic objects and line 6 queries the class model for a class with a name corresponding to the type of the current generic object. Method `getOrCreateClazz` creates a new class, if

the object type shows up for the first time. Then, line 8 loops through the generic attributes attached to the current generic object. For each attribute method `getOrCreateAttribute` retrieves an attribute declaration in the current class, cf. line 9.

```
1  public ClassModel learnFromGenericObjects(String pName, GenericGraph gg){
2     this.setPackageName(pName);
3     // derive classes from object types
4     for (GenericObject gObj : gg.getObjects()) {
5        if (gObj.getType() != null) {
6           Clazz clazz = this.getOrCreateClazz(gObj.getType());
7           // add attribute declarations
8           for (GenericAttribute attr : gObj.getAttrs()) {
9              Attribute attr = clazz.getOrCreateAttribute(attr.getName());
10             learnAttrType(attr, attr);
11    } } }
12    LinkedHashSet<String> alreadyUsedLabels = new LinkedHashSet<String>();
13    // now derive assocs from links
14    for (GenericLink link : gg.getLinks()) {
15       String sourceType = link.getSrc().getType();
16       if (sourceType == null) continue; //<============
17       String targetType = link.getTgt().getType();
18       if (targetType == null) continue; //<============
19       String sourceLabel = link.getSrcLabel();
20       if (sourceLabel == null) {
21          sourceLabel = StrUtil.downFirstChar(sourceType) + "s";
22       }
23       String targetLabel = link.getTgtLabel();
24       if (targetLabel == null) {
25          targetLabel = StrUtil.downFirstChar(sourceType) + "s";
26       }
27       Association assoc = getOrCreateAssoc(sourceType, sourceLabel,
28                                            targetType, targetLabel);
29       if (alreadyUsedLabels.contains(
30                link.getSrc().hashCode() + ":" + targetLabel)) {
31          assoc.getTarget().setCard(R.MANY);
32       }
33       if (alreadyUsedLabels.contains(
34                link.getTgt().hashCode() + ":" + sourceLabel)) {
35          assoc.getSource().setCard(R.MANY);
36       }
37       alreadyUsedLabels.add(link.getSrc().hashCode()+":"+targetLabel);
38       alreadyUsedLabels.add(link.getTgt().hashCode()+":"+sourceLabel);
39    }
40    return this;
41 }
```

Listing 2: Learning a Class Model from a Generic Object Model

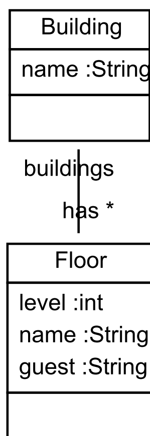

Figure 2: Class Model learned from Generic Object Model

Learning the type of an attribute is done by method `learnAttrType` called in line 10 of listing 2. Basically, we retrieve the value of the current generic attribute. For our generic object model, attribute values are just strings. To learn more specific attribute types, we just try to parse the value string into an `int`, a `double` or a `java.util.Date` value. On success, we store the detected type in variable `attrType`. On different attribute values belonging to the same attribute declaration, this parsing step may compute different results. For example one generic object may have a `num` attribute with value `42` while the next generic object may have a `num` attribute with value `23.5`. The first case results in an attribute type `int` while the second produce an attribunte od type `double`. To resolve this, we compare the type computed for the current value with the type of the attribute declaration that has been computed previously. If the new type is *more general* than the old type, we switch to the new type.

Next, the loop in line 14 of listing 2 is used to learn associations from generic links. For each link we retrieve the types of the connected objects and the role labels for the link ends. Then, method `getOrCreateAssoc` searches the class model for a matching association or creates one, otherwise. Note, that this step is sensible to the direction of links, two similar links with swapped source and target roles might result in two associations with swapped roles instead of a single one. This is easy to fix but results in a more complicated learning algorithm and is thus omitted for lack of space.

Finally, we have to deal with association cardinalities. The most general approach is to use to-many cardinality for all association roles. To-many associations are able to store to-one relations, too, and thus to-many association would work in all cases. However, in many cases a to-one cardinality would suffice and might be more natural to the user. Thus, SDMLib starts with a to-one cardinality for all new associations and roles and we change the role cardinality as soon as there is an object with two similar links attached to it.

Once a class model has been learned, we use Graphviz to render it to the user as a class diagram, cf. figure 2. In addition, the user has the possibility to refactor the learned class diagram e.g. via the SDMLib API. From the resulting class model, SDMLib generates a Java implementation with one plain Java class per model class with private attributes and public get and set methods for each attribute and with private attributes with public access methods for each association role (the access methods of the two roles that build an association call each other to achieve referential integrity of the pairs of pointers that represent a link, to-many associations use special container to hold multiple pointers). For each model class like `Building` we generate a `BuildingSet` class. These classes are used for to-many roles. In addition, these set classes provide the same methods as the original model classes, e.g. `FloorSet::getName()`. In a set class, methods like `getName()` are applied to each contained element, the results are collected and then returned. Thus, for a variable `mainBuilding` of type `Building` the call `mainBuilding.getHas()` delivers the set of floors of that building and `mainBuilding.getHas().getName()` delivers a list of names of these floors. We also generate model specific classes like `BuildingPO` that are used to represent pattern objects in model transformations. For more details see [6]. Finally, we generate factory classes that facilitate the creation of model objects and that provide a reflective access layer for the the model. This means, you may ask these factories for the names of all attributes and association roles of a model class and you may read and write attribute values using their names as simple strings. This reflective layer is also used to provide generic serialization mechanisms to load and store model object structures from / in JSON or XML format.

3 Solving the FIXML case with SDMLib

For the FIXML case, we just developed an XML reader that turns the example input data into generic object structures. Then, the SDMLib techniques are used to learn a class model and to generate Java code for it. Our solution to the FIXML case first learns one separate class model for each sample XML file. Then, we use all sample files to learn one common class model that covers all cases. As the class model learning algorithm for each generic object, attribute, and link first looks whether it has already an appropriate declaration, you can also start with a class model learned from other cases and add more examples later.

Once we have learned a class model and we have generated its Java implementation, SDMLib also allows to convert generic object structures into model specific object structures. This is done using the reflective access layer generated for the model. Once the generic object structure has been transformed into a model specific object structure, you may program model specific algorithms based on the generated Java implementation leveraging static type checking and compile time consistency checks. Then you may load XML files, convert them to model specific objects and run your algorithm. Your algorithm may also utilize the set based model layer generated by SDMLib or even the model transformation layer. As simple example for the set based layer you may write `currentOrder.getOrdqty().getQty().sum()`. This looks up the set of `OrdQty` objects attached to the current order. Then we look up the `qty` attribute of these objects. Generally, attribute values are collected in list to allow multiple occurences of the same value. For lists of numbers, SDMLib provides some special operations like `min`, `max`, and `sum`. The latter computes the sum of the numbers of the list.

Thus, SDMLib does not only generate a Java implementation for the example XML files. It also provides a mechanism to load XML files to a model specific object structure and SDMLib allows to run complex algorithms and model transformations on that data. Overall, the FIXML case was made for us. SDMLib provides a lot of functionality for generic object structures, learning class models, and generating Java code.

References

[1] I. Diethelm, L. Geiger, and A. Zündorf. Systematic story driven modeling. Technical Report, Universität Kassel, 2002.

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *TAGT*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.

[3] U. Norbisrath, R. Jubeh, and A. Zündorf. *Story Driven Modeling.* CreateSpace Publishing Platform, 2013.

[4] A. Research. Graphviz - graph visualization software, 2008.

[5] SDMLib Generic Object Diagrams. https://rawgit.com/fujaba/SDMLib/master/doc/index.html, 2014.

[6] SDMLib Model Navigation and Model Transformations Example. https://rawgit.com/azuendorf/SDMLib/master/SDMLib.net/doc/StudyRightObjectModelNavigationAndQueries.html, 2014.

[7] Story Driven Modeling Library. http://sdmlib.org/, 2014.

[8] UML LAB from Yatta Solutions. http://www.uml-lab.com/de/uml-lab/, 2014.

[9] FIXML Case for the TTC 2014. https://github.com/transformationtoolcontest/ttc2014-fixml, 2014.