Towards a Corpus of Use-Cases for Model-Driven Engineering Courses

Dimitrios S. Kolovos[1] and Jordi Cabot[2]

[1]University of York, UK – dimitris.kolovos@york.ac.uk
[2]ICREA UOC, Spain – jordi.cabot@icrea.cat

Abstract. Having taught Model-Driven Engineering courses for a number of years, in this paper we reflect on the importance of selecting appropriate use-cases for students to explore related principles and technologies. We discuss examples on both ends of the spectrum and we present guidelines for selecting use-cases that are pragmatic and motivating without being excessively complex.

1 Introduction

While the value of teaching software modelling and model-driven engineering at University level is increasingly recognised [1], effective teaching continues to be a challenging task. The authors of this paper have been somewhat unsuccessful in the past in convincing students about the potential of MDE. We believe a key reason for this is the poor choice of use-cases and examples we show in class. Based on our experience and on extensive discussions with colleagues, in this paper we wish to propose examples of alternative use-cases that have a potential to overturn the current situation.

2 Background

In our experience, to maintain the interest of students in a course they need to see practical value in the technologies and techniques they are taught. For software engineering courses this makes the selection of motivating use-cases a critical point. On top of the bad modelling practices discussed in [2], we have identified that use-cases we use often suffer from the following weaknesses that can end up demotivating students instead of sparking their interest.

Unrealistic Domains In an attempt not to overwhelm students with the complexity of developing fully-functional systems, educators often use unrealistic application domains. For example, a common use-case – variants of which we have used in the past – is to ask students to develop a DSL for modelling book libraries[1] and then write model-to-text transformations that generate HTML reports from library models or define additional validation constraints. This use-case has two advantages: the DSL contains concepts that students can relate to,

[1] help.eclipse.org/mars/topic/org.eclipse.emf.doc/tutorials/clibmod/clibmod.html

and it exercises most of the features of metamodelling technologies (inheritance, containment/non-containment/opposite references). On the other hand however, it is rather unlikely that one would implement a real-world library management system using models instead of e.g. a relational database as a data persistence format.

Artificial Development Processes Another common use-case to demonstrate model-to-model transformation is the (infamous) UML to RDBMS example. Here, students are typically asked to develop a transformation that produces a relational schema from a UML class diagram. This is a complex transformation that exercises many features of contemporary transformation languages, however, at the end of the process students end up with a model of a relational schema that is of little practical use. An additional model-to-text transformation can be used to produce SQL that can set up a database, however, this is not very helpful either as in practice students would then still need to interact with the database using low-level SQL commands. Extending the use-case and asking students to also generate code that can provide a high-level object oriented interface to the database from first principles is a rather complex task and the results are unlikely to be of comparable quality to object-relational frameworks like Hibernate. An alternative would be to ask students to produce e.g. Hibernate-based code, however this means that a complex framework Hibernate would have to be taught first – which is a major deviation from the aims of an MDE course.

In our experience, a significant proportion of highly-skilled students quickly realise the discrepancy between such use-cases and the practices and processes they would need to employ in a real world situation and gradually lose interest.

Non-Iterative Development Scenarios Students who – despite the poor selection of use-cases – can see a potential in the principles of MDE, often question its cost-effectiveness. In our view, this is largely because in most use-cases, students are shown how to develop an appropriate modelling language to model a system and then spend the bulk of their effort on developing model-to-model and model-to-text transformations that can transform their models to working code. When the use-case ends there, students can feel puzzled as they have spent a substantial amount of effort to develop and debug non-trivial transformations only to produce code – that they could have written manually with a fraction of the effort – once. Since it is well understood that developing MDE infrastructure takes a few iterations/product instances to pay off, it is important that this is highlighted to students by adding more than one change-adapt iterations to the development scenario.

3 Towards a Corpus of Use-Cases for MDE Courses

Although previous work (e.g. [2]) has identified the importance of selecting appropriate use-cases, there is a lack of concrete proposals in the literature. In this

section we attempt to outline a few use-cases that address some of the issues above, as a starting point for discussion towards building a community-wide body of concrete use-cases that can be reused in MDE courses worldwide.

3.1 Auto-Synchronised (Opposite) References in Java

Java lacks support for auto-synchronised (opposite) references. For example, consider the *Customer* and *Invoice* classes in Listing 1.1. While conceptually the *Customer.invoices* and *Invoice.customer* references are linked to each-other (i.e. setting customer *c1* as the customer of invoice *i1* should ideally automatically add *i1* to the invoices of *c1*), in the absence of built-in support for declaring this relationship, developers need to maintain the two references in sync manually as demonstrated in Listing 1.2. This is clearly tedious and error-prone. To achieve automated synchronisation, a developer would need to extend the implementation of *Invoice.setCustomer(...)* and also the behaviour of the *add()* and *remove()* methods of the list returned by *Customer.getInvoices()*. While this is certainly feasible, it is a mundane and repetitive task that would benefit from MDE-style automation.

Listing 1.1. Customer and Invoice

```
1  class Customer {
2    protected List<Invoice> invoices =
3      new ArrayList<Invoice>();
4
5    public List<Invoice> getInvoices() {
6      return invoices;
7    }
8  }
9
10 class Invoice {
11   protected Customer customer;
12   public Customer getCustomer() {
13     return customer;
14   }
15   public Customer setCustomer(Customer
         customer) {
16     this.customer = customer;
17   }
18 }
```

Listing 1.2. Maintaining references in sync

```
1  Customer c1 = new Customer();
2  Invoice i1 = new Invoice();
3  i1.setCustomer(c1);
4  c1.getInvoices().add(i1); // Sync the two references
```

To automate this task, the reference synchronisation code can be generated from a UML class diagram through a model-to-text transformation. To make this solution practically applicable, it should be implemented to accommodate hand-written code either through an appropriate inheritance scheme or by using protected regions [3] which the transformation preserves during re-generation. In the absence of such support, students are more likely to consider the use-case artificial and lose interest.

The main advantage of this use-case is that it addresses a real limitation of Java while not requiring knowledge of third-party libraries. A risk on the other hand is that students will need to develop a non-trivial model-to-text transformation to achieve this and that – as discussed above – they may consider that they could have written the reference synchronisation code manually faster. To mitigate this risk the use-case should involve more than one change cycles and/or large class diagrams that would reinforce the benefits of automation.

3.2 Using State Machines for Behaviour Comprehension and Code Generation

Moving away from class diagrams, in this use-case students can be presented with a small state-machine (5-7 states) and the equivalent code in Java, and can be asked to reason about the behaviour of the system e.g. how many distinct states the system can be in, from which other states the system can get to a particular state of interest, if there are any unreachable states etc. These should be straightforward to answer by inspecting the state machine but less obvious by reading through the Java code. The first aim of this use-case would be to demonstrate that models can help with understanding and reasoning about complex behaviour, which becomes much harder to grasp at the level of imperative code.

In a next step, a significantly larger state machine can be introduced which is not amenable to visual inspection, to demonstrate the need for automated model analysis capabilities (e.g. querying, validation, reachability analysis). In a final step, a model-to-text transformation can be used to produce an identical executable Java implementation of the state machine from the high-level model. Again, in each of the latter steps, multiple state machines should be involved to demonstrate how the initial effort spent to develop the queries and transformations pays off after a few iterations.

3.3 Wedding Organisation DSL

Moving away from UML, the aim of this use-case is to demonstrate the usefulness of constructing domain-specific languages (DSL) when existing modelling languages are not a good fit for the problem at hand. We also wish to steer away from generating executable code to demonstrate the breadth of applicability of MDE techniques. In line with our discussion so far, the DSL – and its supporting model management activities – should be relatively simple but genuinely practical for solving the problem at hand (unlike the library example discussed in Section 2).

In this use-case, from models conforming to a wedding event DSL such as the one displayed in Figure 1 students can be asked to generate (1) personalised invitation cards in HTML/LaTeX (if a guest is allocated to a table it means that they have been invited to the post-wedding dinner and another sentence needs to be added to the invitation card), (2) lists that will guide guests to their tables at the venue. Students can be asked to use a validation language to check models conforming to the DSL for the presence of conflicts (i.e. guests involved in a "conflict" should not be sitting on the same table), or even to employ a constraint solver to suggest an acceptable allocation of guests to tables.

Although at a first glance this use-case appears to be similar in nature to the library use-case discussed in Section 2 in our view it differs in a few key aspects. First, it represents a domain for which there is no existing widely-used software that students can compare against. By contrast, in the library management domain, students are likely to compare the produced MDE solution against existing library management systems (e.g. university library) that they are familiar with,

with an unfavourable outcome for the MDE solution. Second, and perhaps most important, in the absence of out-of-the-box user-friendly software, the proposed MDE solution is arguably a sensible way to support this activity in practice.

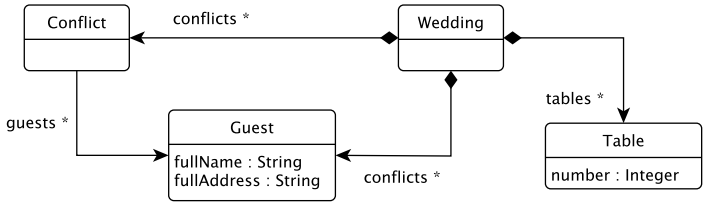

Fig. 1. Wedding DSL

Listing 1.3. Invitation card template

```
Dear [%=guest.fullName%],

Together with their families

Tom and Mary

invite you to celebrate their marriage
Sunday October 5th, 2015 at 14:00 at
    York Minster

[%if(Table.all.exists(t|t.guests.
    includes(guest)){%]
Dinner and dancing will follow at the
    Rosewood Inn,
12 Gillygate, York
[%}%]
```

4 Conclusions

In this paper we have highlighted the importance of selecting appropriate use-cases for MDE courses, and identified a number of common weaknesses that they can present. In an attempt to stimulate discussion towards a more convincing and inspiring MDE curriculum, we have outlined three concrete use-cases which, in our view, are pragmatic without being excessively complex. As further work, we plan to apply and validate these use-cases in our institutions and to start an initiative for building a corpus of examples with similar intentions on top of an appropriate technical infrastructure (e.g. GitHub organisation, Wiki).

References

1. Marian Petre. UML in Practice. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press.
2. Richard F. Paige, Fiona A. C. Polack, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Matragkas, and James R. Williams. Bad modelling teaching practices. In *ACM/IEEE MoDELS Educators Symposium (EduSymp)*, 2014.
3. Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. *The Epsilon Generation Language*, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.