Domain-Specific Language for Generating Administrative Process Applications

Antonio García-Domínguez, Ismael Jerez-Ibáñez and Inmaculada Medina-Bulo
Department of Computer Science, University of Cadiz, Av. Universidad de Cádiz 10, Puerto Real, Spain
antonio.garciadominguez@uca.es, ismael.jerezibanez@alum.uca.es, inmaculada.medina@uca.es

Keywords: Model-driven engineering, domain-specific languages, business modeling, code generation.

Abstract: Some organizations end up reimplementing the same class of business process over and over from scratch: an "administrative process", which consists of managing a structured document (usually a form) through several states and involving various roles in the organization. This results in wasted time that could be dedicated to better understanding the process or dealing with the fine details that are specific to the process. Existing virtual office solutions require specific training and infrastructure and may result in vendor lock-in. In this paper, we propose using a high-level domain-specific language to describe the administrative process and a separate code generator targeting a standard web framework. We have implemented the approach using Xtext, EGL and the Django web framework, and we illustrate it through a case study.

1 INTRODUCTION

In many organizations, there is a recurrent kind of business process which we call an "administrative process". These administrative processes involve managing a document with a certain structure and tracking it through different states. In each state, different parts of the document have to be viewable or editable by people with different roles in the organization. State transitions usually happen due to human decisions (possibly after a meeting, a review or some kind of negotiation), deadlines or a combination of both. The process usually concludes by reaching a "final" state (e.g. "accepted" or "rejected").

These processes are usually kept within a single organization and are not particularly complex by themselves, but their sheer number within some organizations can produce a high amount of repetitive work. Implementing each of these processes from scratch wastes precious time on writing and debugging the same basic features (form handling, state tracking, internal directory integration and so on) which should have been invested in obtaining a better understanding of the process desired by the users and fine tuning the process-specific business logic. In some cases, the developer tasked with implementing the process is not familiar with some of the best practices of the target technology, needing more time and producing less than ideal solutions.

Another problem is that even after the process is correctly implemented, the framework the implementation is based upon may become obsolete to the point of requiring a complete rewrite. This is compounded by the fact that since the processes may have needed urgent changes and tweaks, they may not be well-documented anymore and may require careful reverse engineering, which is time consuming and prone to mistakes. It would have been much better if most of the code had been produced from a process description: changing the target framework of several obsolete applications would only require writing a different code generator and adding some customizations.

Our organization has evaluated various generic "virtual office" solutions for implementing these administrative processes and having them run in a high-level process engine. While these solutions were acceptable for the simplest cases, adding process-specific UI and business logic and integrating them with in-house systems would have required learning yet another technology that may become obsolete or lose support in the long run. It would be much better if the resulting implementations were based upon a standard web framework chosen by IT, so it could be maintained by regular staff.

In this paper, we present the first version of a technology-agnostic domain-specific language dedicated to concisely describing these administrative processes. We show how it can describe an examination process and how we developed a code generator that followed the best practices of the Django

Garcia-Dominguez A., Jerez-IbÃąÃśez I. and Medina-Bulo I.
Domain-Specific Language for Generating Administrative Process Applications.
DOI: 10.5220/0005886801780183
In *Proceedings of the Fifth International Symposium on Business Modeling and Software Design (BMSD 2015)*, pages 178-183
ISBN: 978-989-758-111-3

web framework (Django Software Foundation, 2015) to implement the process as a website. The generated website is ready to be used and can be customized by any developer familiar with the Django framework.

2 RELATED WORK

There exist several business process management systems (BPMSs) that include some support for form-based steps within workflows, such as Bonita (Bonitasoft, 2015) or Intalio (Intalio, Inc., 2015). These engines are usually tightly integrated with a design tool which uses a graphical notation (usually BPMN-based) to describe the processes. This graphical notation is normally extended with engine-specific annotations to provide the full semantics of the process, and is persisted using XML-based formats.

While these systems can describe a much larger class of business processes than our DSL, the resulting process definitions are highly dependent on the underlying engine: migrating the same process to a new technology may require a rewrite. Version control is still possible with XML-based formats, but meaningful comparisons and merges require special-purpose tools. In addition, using a BPMS effectively requires a considerable amount of training, consulting and process analysis, which may not be feasible in smaller IT departments. Our approach focuses on a specific kind of business process (manage a complex document through multiple states, with different access rules in each state) and produces a standard web app that can be maintained as any other.

Scaling back from full-fledged BPMS engines, there have been many attempts to simplify the development of form-based applications. One recent initiative in this regard has been the EMF Forms (Eclipse Foundation, 2015b) Eclipse project. Using its tools, users can define the domain model and abstract layout once and then render it in various technologies, such as SWT or JavaFX for desktop apps or Tabris for mobile apps. On the one hand, EMF Forms allows developers to specify the concrete layout for the resulting forms, unlike our DSL, which delegates on the generator for the presentation aspects. However, a DSL could cover aspects that EMF Forms does not, such as access control and state transitions.

Another related topic is domain-driven design (DDD): an approach on software development that asserts that the primary focus in most software projects should be the creation of an adequate model that abstracts the problem domain, and the implementation of the relevant business logic around it (Evans, 2003). In this regard, several frameworks have been implemented to support DDD, enabling rapid iterations by generating a large portion of the application from a "pure" domain model (e.g. a set of Plain Old Java Objects or POJOs), such as Apache Isis (Apache Software Foundation, 2015) or OpenXava (OpenXava.org, 2015). In a way, a DSL conforms to the same view of producing software from a description of the problem domain: the only difference is that the DSL is focused on a specific kind of problem domain, rather than the generic approach of a DDD tool, making it more productive in that particular case.

In the context of e-government, several works have identified some differences in the way e-government processes should be modeled, in comparison to the business processes in the private sector. Klischewski and Lenk argued that in the public sector, many processes involve unstructured decision making (whether by a single official or after a meeting) and negotiations, so officials would need to be able to alter the course of the process (Klischewski and Lenk, 2002). Later in the same work, Klischenski and Lenk proposed a set of "Admin Points" as reusable process patterns that could be used as a shared vocabulary for e-government business processes. In a DSL, many of these admin points which involve negotiation and unstructured decision making could be modeled using decision-based transitions.

3 LANGUAGE DEFINITION

Summarizing the discussion in the previous sections, the design requirements for the language are:

- The description of a process should include the information to be stored within the managed document, the roles involved and the states that the document goes through.
- Each state should list the roles that can view or edit the different parts of the managed document.
- State transitions based on user decisions and dates should be supported by default, and custom business logic should be easy to integrate later on.

3.1 Abstract Syntax

Figure 1 shows a UML class diagram with the abstract syntax of our DSL. An APPLICATION is divided into ELEMENTS. There are five kinds of ELEMENTS. SITE is the simplest one: it only declares the name of the application (e.g. "Billing"). OPTIONS elements contain key/name pairs (PROPERTY instances) that may be useful to external generators.

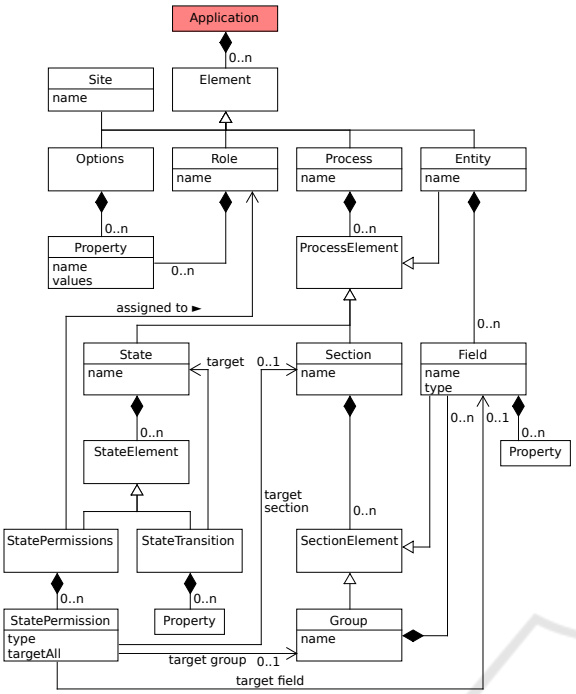

Figure 1: UML class diagram of the DSL's abstract syntax.

Next are ROLEs. These represent particular roles within the organization (such as "Accountant"). These elements provide a name and a set of PROPERTY instances which may be used by the generator to integrate the role with in-house user directories (such as an LDAP directory).

ENTITY elements represent data entities that must have been created before any documents can be filled in, such as "Country", "State" and so on. An ENTITY contains FIELD instances with the information to be stored about it. Every FIELD has a name and a domain-specific type (such as "currency" or "identity document") and zero or more PROPERTY instances providing additional information to generators.

Finally, the main and most complex kind of element is a PROCESS. These represent entire administrative processes (e.g. "Request for Leave"). They can contain three kinds of PROCESSELEMENTs:

- ENTITY instances, which will be specific to the process in this case. In the "Request for Leave" process, "LeaveReason" instances could be the various reasons for the leave.
- SECTION instances contain the FIELDs of the managed document, which can be optionally subdivided into GROUPs. A section could be "Billing information", and a group could be "bill item" (containing the "Price", "Quantity" and "Description" fields, for instance). This structure is useful for generating code and specifying access rules.

Listing 1: Simplified concrete syntax for our DSL.

```
1  site SiteName;
2  options { (Property;)* }
3  (role RoleName { (Property;)* } | role RoleName;)*
4  (entity EntityName {
5    (Type ((Property (, Property)*))? Name;)*
6  })*
7
8  (process ProcessName {
9    (entity EntityName {
10     (Type ((Property (, Property)*))? Name;)*
11   })*
12   (section SectionName {
13      (Type ((Property (, Property)*))? Name;
14      | group GroupName {
15        (Type ((Property (, Property)*))? Name;)+
16      } )*
17   })+
18   (state StateName {
19    (permissions RoleName {
20       ((editable|viewable) all;
21        | (editable|viewable) SectionName;
22        | (editable|viewable) SectionName.GroupName;
23        | (editable|viewable) SectionName.FieldName;
24        | (editable|viewable)
25            SectionName.GroupName.FieldName;
26       )*
27    })*
28    (transition (Property (, Property)*) StateName;
29 )* })+ })*
```

- STATE instances represent the states that the managed document can be in. It can contain a set of STATEPERMISSIONS for each role of interest, which in turn contain STATEPERMISSION instances that describe the kinds of actions that are available to the role. A role can receive all permissions at once, or it can receive the ability to edit or view a single section, group or field.

  A STATE can also contain STATETRANSITIONs to other states. A transition can activate when all the conditions (specified using PROPERTY instances) are met: these conditions could combine explicit decisions by users ("Accepted" or "Rejected"), dates ("Past deadline") or custom business logic. Alternative paths to the same state can be modeled with several STATETRANSITIONs.

3.2 Concrete Syntax

Since editing and version control should not require any special-purpose tools, we have picked a textual notation for the DSL which is mostly a one-to-one mapping to the model entities.

Listing 1 shows a simplified version of our original EBNF grammar for the concrete syntax. As usual, `(x)+` means "one or more x", `(x)*` means "zero or more x", `(x)?` means "zero or one x" and `x|y` means "x or y". Whitespace is ignored. Literal `(` and `)` and keywords are shown in bold, to avoid confusion. A *Property* is of the form `Name = Value (, Value)*`.

As the grammar can fit into less than 40 lines after some simplifications, we can conclude that it is a

rather simple DSL that should be easy to learn by the IT staff. However, it has a considerable number of cross-references in it, so it will require good tooling support to ensure that these references are not stale.

4 CASE STUDY

In this section, we will present a case study that uses the DSL to describe a simple examination process and then generate a web application that implements it. After outlining the process itself, we will describe the most important aspects of our implementation and evaluate the results obtained.

4.1 Description

The process in this case study is a test, involving two roles ("student" and "teacher") and these steps:

1. The student starts the test by introducing their personal information and answering the first part. Some of the questions are free-form, some have a predefined set of answers, and one of the questions pull answers from the database.

   The teacher can already see all the partially filled-in exams, but cannot enter any grades yet.

2. After a certain date, the second part of the test (with two numeric questions) becomes visible and the first part of the test is no longer editable by the student. Students can also fill in what they think about the test. Teachers can still see everything, but cannot enter any grades yet.

   The test can be turned in for examination before a certain deadline: after that deadline, it is turned in automatically and is no longer editable.

3. Once the test has been turned in, the teacher can grade it, but the student cannot see the grade yet.

4. After the teacher confirms the final grade, the examination is "closed" and the student can now see the grade. All fields are now read-only.

A simplified version of the DSL-based description of the process is shown in Listing 2. Some of the dates and field names have been shortened to save space.

Line 1 declares that the application to be generated has the name "School". Lines 2–5 include several options for the code generator that targets the Django web framework: in particular, they suggest using a certain base template that follows the organizational image, which is included in a Django app available at a certain URL. Line 6 declares the previously mentioned "student" and "teacher" roles.

Listing 2: DSL-based examination process.

```
site School;                                                          1
options {                                                             2
  django_base_template = "template/base.html";                        3
  django_extra_apps = "template = https://.../";                      4
}                                                                     5
role student; role teacher;                                           6
                                                                      7
process exam {                                                        8
 entity Answers3 { string answer; }                                   9
 section personal {                                                   10
  fullName studentname;                                               11
  identityDocument(label="National ID:") nid;                         12
  email(label="Email") mail;                                          13
 }                                                                    14
 section test {                                                       15
  group part1 {                                                       16
   string(blank="True") q1;                                           17
   choice(values="A1,A2,A3",blank="True") q2;                         18
   choice(table="Answers3",blank="True") q3;                          19
  }                                                                   20
  group part2 {                                                       21
   currency(label="Q4 (euros):",blank="True") q4;                     22
   integer(label="Q5 (integer):",blank="True") q5;                    23
  }                                                                   24
  choice(values="Good,OK,Bad",blank="True") opinion;                  25
 }                                                                    26
 section evaluation { float grade; }                                  27
                                                                      28
 state initial {                                                      29
  transition(decision_by="student",                                   30
   after_date="2015/03/01-14:00:00",                                  31
   before_date="2015/03/07-14:00:00") part1;                          32
 }                                                                    33
 state part1 {                                                        34
  permissions teacher { viewable all; }                               35
  permissions student { editable personal,                            36
                                 test.part1; }                        37
  transition(after_date="...") part2;                                 38
 }                                                                    39
 state part2 {                                                        40
  permissions teacher { viewable all; }                               41
  permissions student {                                               42
   viewable test.part1;                                               43
   editable personal, test.part2, test.opinion;                       44
  }                                                                   45
  transition(decision_by="student",                                   46
   before_date="...") evaluation;                                     47
  transition(after_date="...") evaluation;                            48
 }                                                                    49
 state evaluation {                                                   50
  permissions teacher { viewable all;                                 51
                        editable evaluation; }                        52
  permissions student { viewable personal, test; }                    53
  transition(decision_by="teacher") closed;                           54
 }                                                                    55
 state closed {                                                       56
  permissions teacher { viewable all; }                               57
  permissions student { viewable all; } }                             58
}                                                                     59
```

The rest of the listing from line 8 onwards is dedicated to the "exam" process. An entity "Answers3" is declared at line 9: its instances are used for the answers for question 3. In lines 10–27, the fields of the document are organized into three sections. The "test" section is divided into two groups and one additional field: using groups simplifies access control specifications later on. The optional fields have "blank" set to "True".

Lines 29–58 describe the 5 different states the process can be in. The "initial" state is a special case: it represents the state before the process starts, and its transitions describe who can start the process and when. The other 4 states match the four stages of the

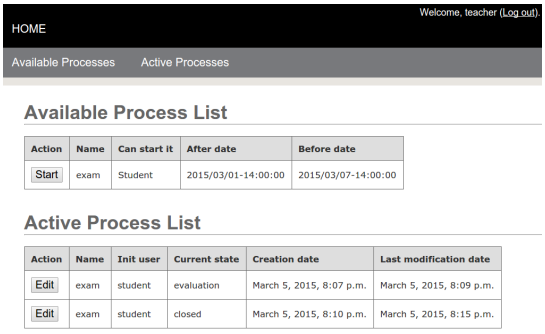

Figure 2: Generated web app: process list.

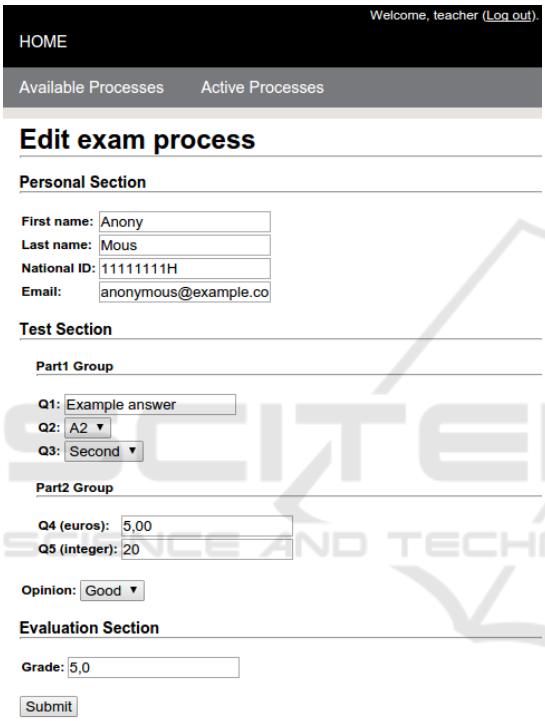

Figure 3: Generated web app: process form.

examination which were described before.

4.2 Implementation

The parser and editor for the language in Section 3 have been implemented using Xtext (Eclipse Foundation, 2014). From an EBNF grammar, Xtext generates a metamodel with the abstract syntax of the language and a set of Eclipse plugins which provide a parser and an advanced editor with live syntax checking and highlighting, autocompletion and an outline view.

We have also implemented a separate code generator that takes an APPLICATION described with our DSL and produces a web application in the Django framework. The generator is written in the Epsilon Generation Language (Eclipse Foundation, 2015a), which provides modularity and the ability to have "protected regions" that are preserved when overwriting an existing file. Our current version of the EGL source code for the Django generator has 3045 LOC.

The code generator produced from the example in Section 4.1 a ready-to-use Django site backed by a PostgreSQL relational database (shown in Figure 3). Thanks to the use of several advanced features in Django, the site only required around 1000 lines of Python code, 400 lines of HTML templates and 264 lines of documentation and support scripts.

4.3 Current Limitations

The DSL, its tooling and the evaluated generator currently present several limitations. One self-imposed limitation is that they do not aim to produce 100% of the required code: the generated code will always need to be customized in some way, due to special needs on the user interface, custom business logic that has to be added, or unexpected integrations with legacy systems. Following the accepted approach in the existing literature (Fowler, 2010), we have chosen to keep the DSL small and focused on describing administrative processes.

The DSL is focused on describing the current process, and does not have any provisions for migrating running processes to a new version with different states or very different information. Our current approach is to delegate on the target framework of the generator: for instance, since version 1.7 the Django framework has built-in data and schema migrations.

Since we transition to a new state as soon as its conditions are met and ignore all other transitions in the old state, we implicitly only allow one state to be active at a time. While this makes the DSL less general than a full-fledged business process modeling language such as BPMN (Object Management Group, 2014), largely based on Petri nets, it is on par with some of the "virtual office" platforms we evaluated and the legacy applications it is intended to replace.

States do not enforce preconditions, invariants or postconditions yet, beyond simply checking that the mandatory editable fields have been filled in. We intend to add support for the most common conditions to the DSL in the short term: the most advanced cases will be delegated to a protected region, which will have to be filled in by the developer.

5 CONCLUSIONS

A simpler class of business processes (administrative processes) are very common in many organizations today: these processes basically consist of managing a form through many states, involving various roles in the organization. Implementing the basic logic and infrastructure for them again and again wastes precious time that could be used on understanding better the process and implementing the fine details correctly.

This paper has presented an approach to improve the efficiency of implementing these solutions, while avoiding lock-in into a particular technology: using a high-level domain-specific language (DSL) for describing the process and writing a separate code generator for each target technology. The approach has been illustrated by describing an examination process, and has been implemented with Xtext (Eclipse Foundation, 2014) on the DSL side and EGL (Eclipse Foundation, 2015a) on the code generation side. The code generator produces a ready-to-use site that follows the best practices of the Django web framework (Django Software Foundation, 2015), accelerating the implementation of the process.

Our results have several limitations. The generated code is ready to be used, but will normally need to be customized to fully meet the needs of the users: we have preferred to keep the DSL and code generators small and focused. Additionally, the DSL is not intended to replace general-purpose workflow-based notations: it is specifically designed for the simpler administrative processes. Finally, we have only evaluated our approach through internal case studies.

We have several lines of work ahead. As we develop more advanced case studies, we will expand the DSL with concepts such as limits on number of processes per user or state pre/postconditions. In addition to refining the tooling, we will look into performing live validation of the structure of the process itself and providing graphical visualizations. We will also develop code generators for other web frameworks that are common in our organization (e.g. Symfony 2). Over the medium term, we will conduct studies with developers within and beyond our organization, evaluating the usability and productivity of our approach.

ACKNOWLEDGEMENTS

This work was funded by the research project “Mejora de la calidad de los datos y sistema de inteligencia empresarial para la toma de decisiones” (2013-031/PV/UCA-G/PR) of the University of Cádiz.

REFERENCES

Apache Software Foundation (2015). Apache Isis. http://isis.apache.org/. Last checked: March 3rd, 2015.

Bonitasoft (2015). Homepage of the Bonita BPM project. http://www.bonitasoft.com/. Last checked: March 3rd, 2015.

Django Software Foundation (2015). Home page of the Django web framework. https://djangoproject.com. Last checked: March 6th, 2015.

Eclipse Foundation (2014). Xtext project homepage. http://www.eclipse.org/Xtext/. Last checked: March 2nd, 2015.

Eclipse Foundation (2015a). Epsilon project homepage. https://eclipse.org/epsilon/. Last checked: March 5th, 2015.

Eclipse Foundation (2015b). Homepage of the EMF Forms project. https://www.eclipse.org/ecp/emfforms/. Last checked: March 3rd, 2015.

Evans, E. J. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, Boston, first edition.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, first edition.

Intalio, Inc. (2015). Homepage of the Intalio|BPMS project. http://www.intalio.com/products/bpms/overview/. Last checked: March 3rd, 2015.

Klischewski, R. and Lenk, K. (2002). Understanding and modelling flexibility in administrative processes. In *Electronic Government*, volume 2456 of *Lecture Notes in Computer Science*, pages 129–136. Springer.

Object Management Group (2014). Business Process Model and Notation 2.0.2. http://www.omg.org/spec/BPMN/2.0.2/. Last checked: March 2nd, 2015.

OpenXava.org (2015). OpenXava homepage. http://www.openxava.org/web/guest/home. Last checked: March 3rd, 2015.