Bottom Up Creation of a DSL Using Templates and JSON

Claude Petitpierre
EPFL
Station 14, 1015 Lausanne, Switzerland
claude.petitpierre@epfl.ch

ABSTRACT

This paper proposes a concept and a workbench that allow developers to devise various kinds of applications with the help of templates. The workbench supports the development from the creation of the templates up to the (automatic) generation of a DSL (Domain Specific Language), providing developers with a bottom up support that mirrors the top down MDE (Model Driven Engineering) attempt to bolster application development.
Unlike usual RAD (Rapid Application Development) tools or wizards embedded in IDEs, the tool we propose is generic and independent of any platform (but of the IDE). By conception, it automatically takes into account the modifications and the new components into the subsequent phases of the development. These components are instantiated under the control of a specification structure (JSON objects, Javascript Standard Object Notation) from which the DSL and the corresponding compiler can be generated. The DSL can be used to extend the application, as well as to develop other applications that require similar presentations and operations.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*

General Terms

Design|vspace-1mm

Keywords

Templates, JSON, DSL

1. INTRODUCTION

This paper proposes a concept and a workbench [1] that allow developers to devise applications with the help of templates linked to JSON (Javascript Standard Object Notation [2]). It spans the development of applications from the creation of the templates up to the automatic generation of a DSL (Domain Specific Language [3]).

Broadly said, software design is performed under two umbrellas: the planned and the evolutionary designs. The first category is mainly built around UML [3, 4], which provides software engineers with graphical notations to describe application designs, and tools to automatically derive implementations or at least skeletons thereof. These approaches are very complex and can only be envisaged to create generators that are heavily reused.

Evolutionary designs rely on the XP (extreme programming) or agile development concepts [5]. The latter provides rules to organize the development of an application, promoting incremental development with short term planning, frequent testing and refactoring to maintain the coherence of the application. This approach is very successful, but it does not provide specific development tools nor modeling support and lets developers on their own to produce the repetitive parts of an application.

In industry, engineers use mainly RAD (Rapid Application Development) tools[12] or wizards (command sequencers)[13] available in most IDEs (Integrated Development Environment) that guide them through the creation of the components of their applications, such as the folder hierarchy, the GUIs, the HTML pages, the database tables and the interfaces to these tables, as well as the data objects used to transfer the data handled by the users from/to the database. They also frequently start from a tutorial, extract the parts they can reuse and build their application around that kernel. The rest of the application is often done by cut-pasting the previous components and adapting them to fit the new tasks, a boring, error-prone and maintenance ill-suited phase.

According to our approach, the developer of an application first creates the simplest prototype that is pertinent in the frame of the whole application: typically, a single page that contains a single field to enter a string in a database table, which can be done with the help of one of the aforementioned techniques. From this skeleton, the developer then selects components containing parts that can be repeated (an entry field) or that could themselves be repeated (a page) and converts these elementary components into a set of templates and a description structure. Of course, this phase can be skipped if ancient templates are available.

In addition to the templates and the parameters, we have defined a list that describes which files must be created from which templates, in such a way that the whole application can be completely reconstructed at any time, using the latest templates.

Our tool has provision to support the edition and the management of the templates and the file description. When the application grows and gets more stable, the workbench can be used to generate a compiler for a DSL, automatically derived from the specification object. From then on, the parameters of the application can be entered through this DSL, which verifies that the subsequent specifications conform to the development underway when the DSL was generated.

Note that this approach is not limited to Web application. We have used it to generate GUIs, tutorials and even the aforementioned compiler.

Content

The next section presents the problem solved by our approach and the contributions to its solution. Section 3 introduces the JSON templates and the way we use them. Section 4 describes the editor we have developed to manage the templates related to a whole application. Section 5 details how to reference the files to create. Section 6 explains the way we can build the DSL (Domain Specific Language) from the specification object and Section 7 presents some experiments we did with the workbench. Section 8 introduces some related work.

2. USE OF TEMPLATES

Using a RAD tool, a developer can create the first prototype of an application in minutes. But what does then her/his team do for the subsequent 6 months ? Rails motto is "favoring convention over configuration", a very interesting and broadly used concept, but in order to avoid ending up to a ... conventional application, it should be possible to escape the conventions when necessary.

During development, engineers try different possibilities, creating new skeletons that they then remove to settle to other solutions, which requires frequent refactoring. Application stakeholders also want to evaluate several representations of their applications to find the best ones. On the other hand, end-users require applications to have homogeneous presentations and behavior, and thus the new components are all de facto similar to the first one generated. These considerations make the use of templates particularly pertinent.

In order to position our approach, let us look at the following source:

```
@Entity public class Customer {
    @Id private Long id = null;
    public String name;
    private transient boolean check = false;
    @OneToMany (mappedBy="customer_Id")
    public Long getId() { return id; }
    public void setId(Long Id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) {
        this.name = name;
    }
    public transient boolean isCheck() { return check; }
}
```

It is a part of a JPA (Java Persistence API) object, which holds data from a database. It is a typical example, the details are not important. The name of the class corresponds to the name of the table from which the data are extracted. The class defines a single field, *name*, of type string, as well as getters/setters to access the data. It also specifies a relationship from another object via a reverse link, named *customer_Id*. Obviously, the whole source can be reconstructed from the three symbols *customer*, *customer_Id* and *name*, knowing that the first one is the name of the table, the second one, the name of the reverse link and the third one the name of the column. One could even, without knowing much of the language, create a second attribute of type string, or another data object with a different table name and different column names. The same set of symbols can then be used to generate the fields that must appear on the page to present a set of data or to fill the JPA. One just need a template for each component and the list of symbols with their purposes.

Template generators are used in the RAD tools and the wizards and many template libraries (Section 8) are available independently from these tools, but few offer an easy access to the specification structure (the list of symbols) and most require their users to develop their own data structures to store the specification data.

Moreover, the successive steps performed during the trials mentioned above are usually not logged and the developer thus quickly looses track of the successive steps she/he did to generate the application, which consequently becomes increasingly heterogeneous as the development goes along. A description of the architecture that is automatically kept up-to-date is indispensable.

The templates, the specification structure and the description file handled by our plugin are directly managed by the user. They are not hidden within the libraries. This is important, as these files provide a high-level model of the application, from which the aforementioned architecture can be derived. Moreover, as they are used to regenerate the whole application, the model they represent is automatically always up-to-date.

The commands we use in the templates are close to the ones defined in the JSON project. They contain no code or expressions, all the computations are made within the specification object.

Refactoring can be performed within the templates, which makes it extremely efficient and easy. Templates are highly reusable in other projects.

One consequence of the use of templates is that an error introduced in a template appears in all the components that use that template. However, this is not a problem, as this multiplicity makes it easier to find it and it can be fixed by a single modification.

3. THE WORKBENCH

We mostly use the JSON standard, in other words Javascript objects, but we had to slightly adapt its syntax to cover some cases that would have been difficult to handle without these modifications. One particular aspect had to be taken care of: the attributes of a JSON object have no ordering. However, the solution described in the next paragraph respects the JSON fundamentals.

3.1 Definition of the Templates and the Description Object

An example of the basic template commands available in the JSON template environment is shown in the left column below. The components of the object displayed on the second column are referred to by the commands of the template. The combination of the templates with the specification object produces the result shown on the third column.

Template	Object	Result
class	{	class
{aaa}	aaa : “Employee”,	Employee
{.section bbb}	bbb : {	
extends {xxx}	xxx : “Person”	extends Person
{.end}	},	
{		{
{.repeated section ccc}	ccc : [”name”,	String name;
String {@};	”initial”,	String initial;
{.end}	”surname”	String surname;
	]	
} // end {aaa}	}	} // end Employee

The symbol *class* has no particular marks. It is thus copied without any modification. *aaa* is placed between braces, it forms an attribute. Its value is replaced by the value given to the attribute with the same name in the object. The content of section *bbb* is expanded, because its name is available in the object. The value of its nested attribute, *xxx*, is given the value defined in the attribute nested in *bbb*. The content of the repeated section *ccc* is generated a number of times corresponding to the number of elements contained in the *ccc* array. The strings in the array elements have no attribute name, they are accessed with the @ sign. } *// end* is just copied to the resulting text and finally, {*aaa*} is expanded into *Employee* again.

The JSON objects and sub-objects contain attributes with strings, integers, sub-objects or sub-arrays. The arrays or sub-arrays contain strings, integers or sub-objects, but no attributes. The sections and the objects can be nested in any order and to any depth.

One solution to specify an ordered list of different elements, is to put them in the elements of an array, with a single attribute in each element, and use them in the template as indicated below:

	{	
{.repeated section rsec}	rsec : [{	y
{.section a} x	b: { ... }	x
{.end}	} , {	
{.section b} y	a: { ... }	
{.end}	}]	
{.end}	}	

During the first pass in the repeated section, the *y* is generated and during the second pass, the *x* is, which corresponds to the order defined in the array, not in the repeated section.

3.2 Enrichment of the specification object

A given part of the description object can be used in different templates, for example to create the columns of the data tables of the application, the lists of fields in which the end-user enters the data, and so on. The few commands available to specify a template cannot cope with all these situations, but rather than extending the set of commands, or introducing computation in the templates we have introduced some new possibilities of specification of the description object.

By default, our tool provides the following capabilities :

- Automatic insertion of attributes ”_index: n” in the array elements where ”n” is the rank of the element. They can be used to number elements.
- Automatic insertion of a capitalized version of attributes defined in a list (useful to create Java’s getters / setters or class and variable names).
- Copy of a subtree of the specification object into another attribute (useful to create relationships)
- Execution of predefined or user defined methods to complete the object

The copy of a subtree into the attribute of another subtree makes it possible to use the same elements at several levels of the repeated sections. This sometimes revealed to be needed, as the alternative would again have required the introduction of new types of sections.

The predefined or user defined methods are written in Java and called by reflexion. They are called from attributes that start with the sign #. They take attributes, arrays or objects as input and return objects or arrays that are inserted into an attribute, the name of which simply drops the #. An example follows:

#xxx : ”merge(aaa, ***options***)”,
options : [
  {id: ”x”, ***readwrite***},
  {id:”y”, ***readonly***}
]

4. THE EDITOR

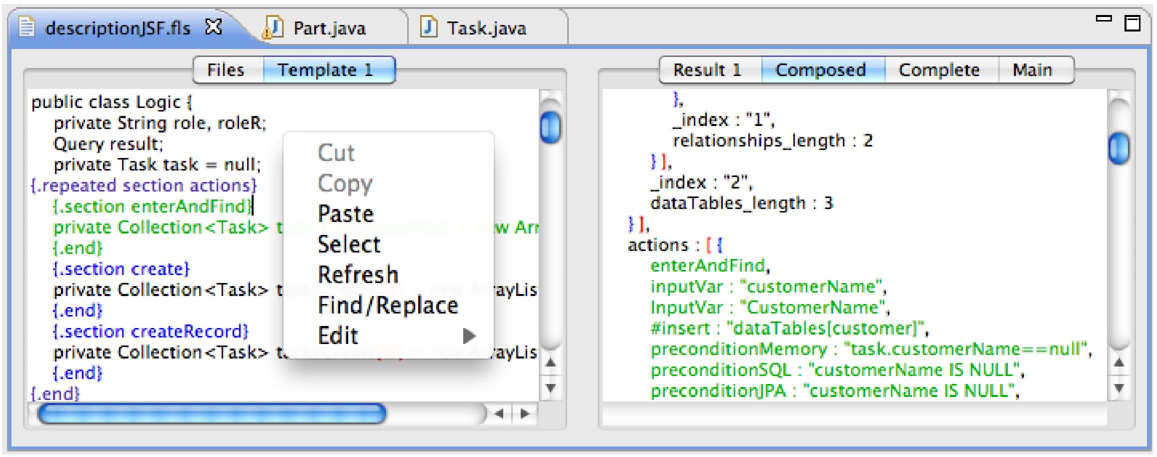
Our experience has shown us that it is very difficult to recognize what pieces in a template and in the corresponding object create a given element in the resulting text. We have thus developed an editor that offers several welcomed features to support the developer in this task.

Our editor (Fig. 1) is available as a plugin under Eclipse [1]. The editor highlights the syntax of the three kinds of files: the templates, the specification object and the description file (see below). It opens several windows that display the different files. A click on an element in any file highlights this element in the clicked window and the elements that have contributed to the creation of this element or that use this element in the other windows.

Moreover, a number of helpers are available to insert the commands both in a template and at the corresponding location in the description object.

5. CREATION OF THE FILES AND FOLDERS OF AN APPLICATION

The basic JSON template library has no provision to create the many files of an application from the same object. Thus, we have completed our tool to do just that.



```
public class Logic {
    private String role, roleR;
    Query result;
    private Task task = null;
{.repeated section actions}
    {.section enterAndFind}
    private Collection<Task> t                      w Arr
    {.end}
    {.section create}
    private Collection<Task> t                      ayLis
    {.end}
    {.section createRecord}
    private Collection<Task> t                      ayLis
    {.end}
{.end}
```

```
            },
            _index : "1",
            relationships_length : 2
        } ],
        _index : "2",
        dataTables_length : 3
    } ],
    actions : [ {
        enterAndFind,
        inputVar : "customerName",
        InputVar : "CustomerName",
        #insert : "dataTables[customer]",
        preconditionMemory : "task.customerName==null",
        preconditionSQL : "customerName IS NULL",
        preconditionJPA : "customerName IS NULL",
```

Figure 1: Snapshot of the editor's windows

5.1 Specification file

The descriptor file that describes which file is generated from which template looks like the following text, borrowed from one of our prototypes.

```
output directory ".." // relative to this file's folder

foreach dataTable
    expand "data/Customer.java"
    to "src/data/"+Id+".java"

for whole
    expand "data/Task.java"
    to "src/data/Task.java"
```

All the templates, the object and the descriptor file must be located in the same folder. If we store this folder in an Eclipse project, the ***output directory*** with the relative address that appears on the first line above is simply the surrounding project.

The ***foreach*** command must point to an array. Each array element is then used to create a separate file, the name of which may include some field borrowed from the array element. The template that must be used in combination with this object is specified by the name placed beside the ***expand*** keyword.

The ***for*** command is used to create a single file from the whole object (in this case). In this file, the *dataTable* referenced in the previous command can be used in its entirety in a repeated section to create a list of references to the other components (for example to create the database).

6. CREATING A DSL

The compiler generated by our tool only accepts sources that correspond to valid description objects. The compiler recreates the corresponding object and the workbench expands it to generate the application as usual. The corresponding language can be used to describe any application built with the same components as the ones found in the application under development, it is thus a DSL. In order to provide the developer with some form of documentation for the generated DSL (nobody has ever seen the generated DSL beforehand!) the tool also creates the source that should be compiled to recreate the current application. This source is close enough to the object for the developer to understand it.

Table 1 shows an example of a description object with the corresponding generated and optimized sources. This example has been used to implement an application for different platforms. It describes the data table shown in Paragraph 2. To save space, only one table is represented here, but the real application has several tables and a list of actions that can be performed by the end-user.

The first column shows the description object used by the templates. This object is assumed to be an object written by the developer during the first phases of the development. The table descriptions must be stored in an array (*dataTable*), because they must correspond to a *repeated section*. The *#insert* command passes the general application parameters into the right level of the template. The attributes are stored in an array specifying the column names and types in the format presented at Paragraph 3.1.

The source in the center column, automatically generated, uses the syntax *dataTable name* {. . .}, which remembers the syntax of a Java class. The table identifier will be used for the filename in which the table code will be stored. The developer is not constrained by the array and she/he can interleave the blocks with other blocks, group elements as she/he please to clarify the source code.

The third column shows the kinds of optimizations that can be made (manually) to make the language more friendly. Such optimizations are very easy to implement in the JavaCC file. We only had to modify the syntax part, not the Java statements, in such a way that after the modifications, the compiler continues to generate the same object as in the first version. Of course, this adaptation requires some knowledge of JavaCC, but it is performed only when the development team grows, other similar applications must be developed or the project becomes an asset of the company.

Original Object	Generated Source	Optimized Source
{	{	
dataTable : [{	dataTable customer {	program {
id : ”customer”,		dataTable customer {
#insert : ”application”,	#insert ”application”	finder string ”name”;
attributes : [{	attributes {	int ”number”;
stringAttribute : {	stringAttribute {	OneToMany to ”confirmedOrder”
varName : ”name”,	varName ”name”	}
finder	finder	}
}	}	
}, {		*DSL*
intAttribute: {	intAttribute {	
varName : ”number”	varName ”number”	*The table above defines a* ***customer***
}	}	*with two fields, a string and an integer*
}],	}	
relationships : [{	relationships {	*A finder will be created for the* ***name***
OneToMany : {	OneToMany {	
to : ”confirmedOrder”	to ”confirmedOrder”	*A relationship with another record,*
}	}	***confirmed order**, will be built*
}]	}	
}	}	
	}	

Table 1: Transformation of an object to two forms of sources

6.1 Generation of the Compiler

The compiler is obtained according to the following considerations. The object that describes an application can actually be understood as an abstract syntax tree generated by compiling a hypothetical source that would represent the application. Our tool generates thus both a source that is close to this object and the corresponding compiler. By compiling this source, the compiler recreates the original description object, which can be used to generate the application. This closed loop does not seem to be productive, but, unlike the general object editor, the compiler only accepts sources that contain information pertinent to the application. It is thus more comfortable to continue the development with the DSL than with the general editor, and the optimized DSL shown in Figure 1 can be used by the domain experts.

The compiler is created with (what else?) templates. A first operation traverses the tree of the object and creates a flattened representation of its components within an array of objects. This object is then used in conjunction with a template to generate the production rules of a JavaCC source. These rules are naturally recursive, which reinstalls the original recursion. The same template is used for all generations, which by the way shows the power of templates.

7. WORKING WITH THE APPROACH

As a test, we have developed four sets of templates corresponding to four environments to create a simple Web application introducing customers and parts in a database and creating the orders of parts made by the customers. The architecture has been devised to integrate these operations within a generic workflow. The platforms were :

- Elementary JSP with servlets and SQL
- JSF with JPA
- PHP with SQL
- Ruby on Rail

These four applications can be expanded with the same specification object. Considering the fact that by changing the object, other workflow applications may be generated and that by changing the templates the same application may be generated for different platforms, we can say that the templates correspond to the platform specific model and the object, to the platform independent model, to refer to MDA's terminology.

8. RELATED WORKS

Our industrial experience with WebLang [6] has shown us that the description of an application by a source code that allows a whole regeneration at any time, is very efficient.

As already mentioned, our approach is derived from the JSON templates [2]. Libraries to handle the objects and to expand the templates are freely available.

The use of templates to build applications has been relied on for a long time with approaches such as JSP (Java Server Pages [10]), PHP (the well-known language used to create Web pages) or ASP (Active Server Pages [11]). These tools allow developers to introduce programming statements (Java, C#, PHP code) into HTML pages to produce the variable parts of these pages. The information retrieved by these template-based generators have no standard data representation and the interleaving of text and code makes their templates difficult to read.

XSLT (EXtensible Stylesheet Language Transformations [9, 8]) is particularly well-suited to transform sources written in XML into different kinds of documents. A control file, called style sheet and written in XML too, defines what parts of the source must be extracted and translated to what format.

Note that a JSON object can easily be converted to an XML source and vice-versa, although some details must be taken care of: on the contrary of JSON, the XML actions have internal attributes, but no arrays. A partial treatment of this aspect has been proposed in Paragraph 3.1.

The Eclipse Modeling Project [3] contains a set of powerful tools for the development of DSLs. It contains, in particular, two tools that generate source code from templates: JET (Java Emitter Templates) and JMerge (Java Merge). JET is a generic template engine derived from JSP. JMerge can be used to regenerate sources in which a developer has already introduced her/his own code. The project also supports the creation of meta-models via different means, such as XMI, Java annotations, UML or an XML Schema. These tools require a huge initial investment and don't seem to have had much success yet.

Environments such as Ruby on Rails [12] allow developers to generate the sets of files and skeletons required to build an application by executing commands within a command line interpreter. With such systems, a fast prototype can quickly be created and shown to a customer. However, as the generated elements must often be modified they cannot be easily regenerated. There are few traces of the executed commands, so that the vision of the architecture gets gradually lost while the development proceeds. The workbench has not been thought to address other platforms and can hardly be reused for other purposes.

On Eclipse [13], the generation commands are called from menus to create the projects with a dedicated hierarchy of folders. A reduced access to the templates is available, but there is no edition facilities and new templates cannot be integrated without modifying the IDE plugins

All the tools available to create Web applications either don't give access to the underlying templates, at least not with a user-friendly environment, or require their user to cope with very complex data structures (JSP[10], ASP[11], JET[3]). Our approach is not linked to any particular platform (only to the IDE), it allows the user to display and handle the templates, the specification data and the file descriptors and offers tools to manage them comfortably. The creation of a DSL is particularly useful for companies specialized in some domain who want to structure their team with software and domain experts.

9. CONCLUSION

We have presented an approach supported by a workbench that allows developers to create, manage and expand their own templates. This workbench is available as an Eclipse plugin [1]. The plugin supports the stepwise development of templates and a specification object that describes which template is used to create which file.

This approach draws from a previous project: WebLang [6], which has been used to develop systems for several companies and for a pedagogical game used by more than one hundred students. WebLang can specify JEE components and create a complete application, but the definition of the templates and a DSL with it is less flexible than with the new approach.

We have successively used the new method to create several prototypes for different environments: a GUI started with Eclipse's Visual Editor, an application in plain JEE plus SQL, one with JSF plus JPA, another one with PHP plus SQL, one with Ruby on Rails, and finally the compiler of the DSL itself.

Our project follows the de facto industrial practices, and considers the benefits of the agile approach that recommends to work on intermediate steps. The availability of the templates and the objects makes the refactoring (an important aspect of agile methods) very efficient. The specification object provides a model of the application, which could be argued upon, but this assertion is supported by the capacity of our tool to create a DSL from this object.

A characteristic of our new approach is that it has a very smooth learning curve and requires only a few new natural notations. The effort required to use it is progressive and as the development enlarges its domain span and gets more complex, a greater involvement of the developer provides greater returns.

Acknowledgements

The workbench WebLang has been developed by Olivier Buchwalder and transferred to industry by Jean Bourgeois. The tools have been integrated into an Eclipse plugin by Paul-Louis Meylan. The scenarios of the workflow prototypes were proposed by Alain Wegmann and Biljana Bajic. I got explanations for XSLT from Christine Vanoirbeek and the Ruby on Rails version was developed by Nikhil Gupta.

10. REFERENCES

[1] Bottom up to a DSL : http://ltiwww.epfl.ch/EBUD
[2] JSON templates : http://org.json and http://code.google.com/p/json-template/
[3] R. C. Gronback, "Eclipse modeling project : a domain-specific language toolkit", Addison-Wesley, 2009.
[4] S. Cohen, A. Soffer, "Scrutinizing UML and OPM Modeling Capabilities with Respect to Systems Engineering", Proceedings of the 2007 International Conference on Systems Engineering and Modeling.
[5] S. Ambler, "Agile Modeling", JohnWiley& Sons, 2002.
[6] O. Buchwalder, C. Petitpierre, "WebLang: A Language for Modeling and Implementing Web Applications", SEKE/06. San Francisco, USA.
[7] JavaCC compiler: http://javacc.java.net
[8] XSLT: http://www.w3.org/TR/xslt
[9] XSLT example: http://zvon.org/xxl/XSLTutorial/Books/Output/example72_ch2.html
[10] Java server pages: http://www.oracle.com/technetwork/java/javaee/jsp/index.html
[11] M. MacDonald, A Freeman, "Pro ASP.NET 4 in C# 2010", Apress, 2010.
[12] S. Ruby, D. Thomas, D.H. Hansson, "Agile Web Development with Rails", The Pragmatic Bookshelf, 2011.
[13] N. Dai, L. Mandel, A. Ryman, "Eclipse Web Tools Platform", Addison-Wesley.