HAL open science

Linking Data and Presentations: from Mapping to Active Transformations

Olivier Beaudoux, Arnaud Blouin

To cite this version:

Olivier Beaudoux, Arnaud Blouin. Linking Data and Presentations: from Mapping to Active Transformations. DocEng'10: Proceedings of the 2010 ACM symposium on Document engineering, 2010, Manchester, United Kingdom. pp.107-110. inria-00504671

HAL Id: inria-00504671

https://inria.hal.science/inria-00504671v1

Submitted on 21 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Linking Data and Presentations: from Mapping to Active Transformations

Olivier Beaudoux
ESEO-GRI
Angers, France
olivier.beaudoux@eseo.fr

Arnaud Blouin
INRIA
Rennes, France
arnaud.blouin@inria.fr

ABSTRACT

Modern GUI toolkits, and especially RIA ones, propose the concept of binding to dynamically link domain data and their presentations. Bindings are very simple to use for predefined graphical components. However, they remain dependent on the GUI platform, are not as expressive as transformation languages, and require specific coding when designing new graphical components. A solution to such issues is to use active transformations: an active transformation is a transformation that dynamically links source data to target data. Active transformations are however complex to write and/or to process. In this paper, we propose the use of the *AcT framework* that consists of: a platform-independent mapping language that masks the complexity of active transformations; a graphical mapping editor; and an implementation on the .NET platform.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Languages

Keywords

Mapping, Active transformation, Model Driven Engineering

1. INTRODUCTION

Visualizing and editing documents is performed by interactive systems that link documents to their graphical presentations. Such a separation of domain data from their presentation has been used for a while in computer science. The Model-View-Controller (MVC) design pattern synchronizes views with their model: as soon as a model changes, its views refresh their state [7]. Recent RIA toolkits, such as WPF [9], have introduced the concept of binding to simplify the specification of such model-to-view links. However, bindings offer limited features: they are not as expressive as transformation languages; they depend on the GUI platform; they require specific coding when designing new graphical components.

Transformations can be used to transform source document into target presentations. The best known document transformation language is XSLT. However, most XSLT processors are batch engine: they do not maintain the synchronization between the source and the target. *Incremental* (*i.e.* *active* or *live*) versions of transformation processes have been proposed to solve this issue [2, 1, 10]. However, developing an active transformation is not as easy as creating a batch transformation [1]. Model-Driven Engineering (MDE) has defined about thirty model transformation languages [4]. As for document transformation languages, most of them are not active.

Rather than hand-writing active transformations, recent works propose to specify the links between source models and target models using *declarative mapping* languages [3, 8]. In such a scheme, an active transformation is generated from a mapping, thus allowing the mapping to be applied by running its active transformation. To the best of our knowledge, mappings and their active transformations have not yet been applied in the context of interactive systems. The only exception is GMF [5], but it remains specialized for diagramming tools.

In this paper, we propose to use mappings and to generate their implementing active transformations in the context of GUIs. We explain, through the modeling of a "course schedule" document (section 2), how our *AcT framework* allows the specification of mappings between domain data and their presentations (section 3), and the execution of mappings through active transformations (section 4). The whole framework is freely available under the GPL license at *http:// gri.eseo.fr/software/act.*

2. MODELING DATA AND PRESENTATIONS

The first step of UI design consists of modeling the domain data and their presentations with class diagrams. Figure 1 gives the domain data model, edited with our AcT graphical editor *AcTeditor*, of a course schedule; it is used throughout the paper to illustrate how to use the AcT framework. An academic *week* is composed of five *day*s that contain *course*s. A course is related to a *topic*, is taught by a *teacher*, is located in a *room*, and *start*s and *end*s at *predef*ined *time*s.

Author version

If a teacher is not specified for a given course, it is considered to be the first *teacher* of the course *subject*.

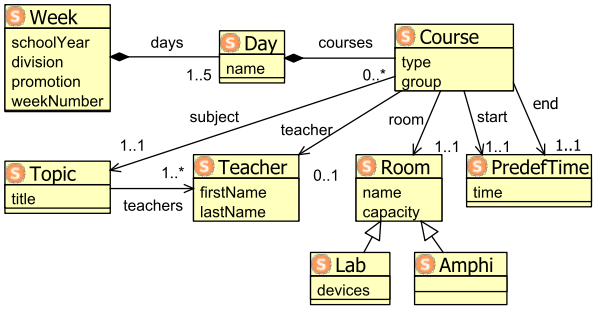

Figure 1: A model of a course schedule

Figure 2 gives the model of widget *ListBox* as defined by WPF (simplified view). A *ListBox* contains *ListBoxItem*s that may embed a *Control*, which can be a *TextBlock*.

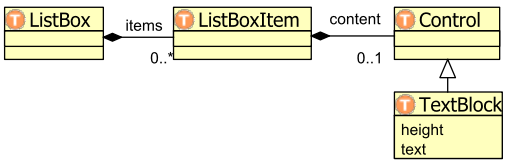

Figure 2: The "list-box" model

Figure 3 defines a graphical calendar with week view capabilities. A *CalendarCanvas* is graduated with *timeLine*s, and contains *eventStamps*. Each *EventStamp* has a user-defined *type*, defines its coordinates (*day*, *time* and *duration*), and contains textual *descriptions*.

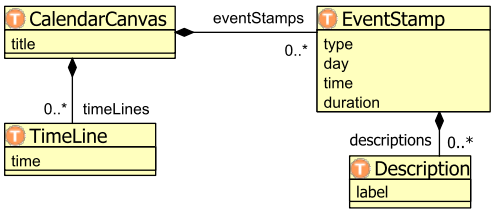

Figure 3: The "calendar canvas" user control model

3. MAPPING DATA TO PRESENTATIONS

3.1 Principle

The second step of the UI design consists of mapping the domain data with their presentations. A *mapping* establishes a persistent relation between source domain data and a target presentation. Figure 4 gives the model of AcT mappings.

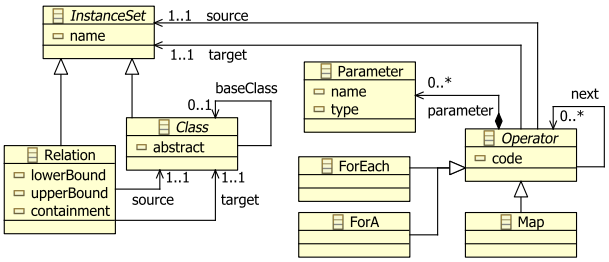

Figure 4: AcT mapping model

Both *Classes*, and *Relations* between *source* and *target* classes, are considered as sets of instances (class *InstanceSet*): a class C is the set of all the instances of C created while the application is running, and a relation $i.r$ of an instance i defines the set of instances related to i through r. Consequently, mappings between classes and relations are defined in a unified way: they are instance set mappings.

A mapping is a set of mapping operators. An *Operator* establishes a mapping between a *source* instance set S and a *target* instance set T. Operator *ForEach* defines, for each source $s \in S$, which target instance $t \in T$ is mapped to s. Operator *ForA* defines the same kind of mapping, but maps one selected source instance $s \in S$ with one target instance t. Once a *ForA* or a *ForEach* operator has mapped s with t, operator *Map* describes how s and t contents are mapped. This consists of transforming s properties and/or relations to t ones. An operator can define a specific *code* that currently consists of a small subset of the C# language. Operators are then chained through relation *next*. A mapping starts with at least one *ForA/ForEach* operator, each one followed by a *Map* operator. In turn, as soon as a *Map* operator establishes a mapping between source and target relations, the *Map* is chained to other *ForA/ForEach* operators that map these relations. Finally, an operator can define user *Parameter*s, which is useful whenever a user wants to modify a simple mapping parameter, such as a sort order.

As with any transformation language, a mapping uses navigation to specify which instances and properties of the source data are mapped to instances and properties of the target presentations. The AcT framework defines its own simple navigation language that uses the usual dotted notation. For example, the *week.days.courses* represents all *Course* instances for all *days* of the given *week*. A specific instance can be selected within an instance set by using the *[]* accessor. For example, the *week.days.courses[1]* is the first course of the given *week*, in chronological order. It is possible to navigate backwards a step with the *Parent()* function. For example, *week.days.courses[1].Parent()* gives the day of the first course.

3.2 Example 1

Figure 5 gives the AcT mapping "Teachers2ListBox" that maps all teachers of our "course schedule" example to a standard list-box widget.

The *ForEach* operator is graphically represented in the AcT mapping language by the double-arrow link «—». For

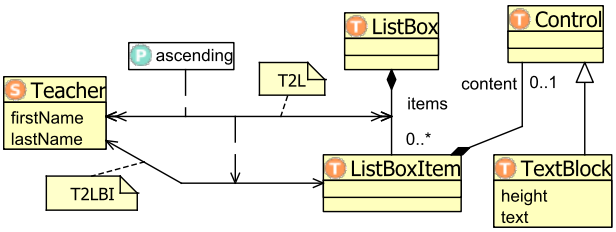

Figure 5: The "Teacher2ListBox" mapping

example, mapping operator *T2L* maps class *Teacher* to relation *ListBox.items*. The code of *T2L*, accessible through the *AcTeditor*'s property window, defines that the order of instances within relation *ListBox.items* is defined by a sort applied to the *Teachers*:

```
order = Teacher.SortedOrder(
    "lastName", ascending,
    "firstName", ascending);
```

The *order* keyword must receive an integer array, called an *order array*, that defines the order between the source and target instance sets. For a given order array Ω, each source instance $s = S[i]$ is mapped to the target instance $t = T[j]$ such as $j = \Omega[i]$. For example, $\Omega = \{1; 2; 3\}$ specifies the natural order of a set of three instances where each $S[i]$ is mapped to $T[i]$, while $\Omega = \{3; 2; 1\}$ specifies an inverse order where each $S[i]$ is mapped to $T[4 - i]$. Such an order capability is not present with bindings: sorting is rather defined by the graphical component itself, thus requiring its implementation for each new component.

Operator chaining is represented by a dashed arrow $\dashrightarrow$, and a map operator is represented by a single-arrow $\leftrightarrow$. The *ForEach* operator *T2L* is chained to map operator *T2LBI* that establishes the mapping between each *Teacher* instance and its corresponding *ListBoxItem* instance:

```
if (ListBoxItem.content == null)
    ListBoxItem.content = TextBlock.Create();
(ListBoxItem.content as TextBlock).text =
    Teacher.firstName + "␣" + Teacher.lastName;
```

3.3 Example 2

Figure 6 shows mapping "Week2Calendar" that maps a selected week of the schedule to the calendar canvas.

The mapping starts by a *ForA* operator, represented by arrow $\twoheadleftarrow\!\!\!\rightarrow$, that maps a user specified *Week* to an *CalendarCanvas*. The week is selected by the user through the *weekIndex* parameter within the following code:

```
select = Week[weekIndex];
```

The *select* keyword is used to indicate the user selection of the *ForA* operator. The operator is followed by a *Map* operator *W2CC* that maps *Week* properties to the *title* property of the *CalendarCanvas*:

```
CalendarCanvas.title =
    "Semaine" + Week.weekNumber + "␣–␣" +
    Week.schoolYear + Week.division +
    "␣(" + Week.promotion + ")";
```

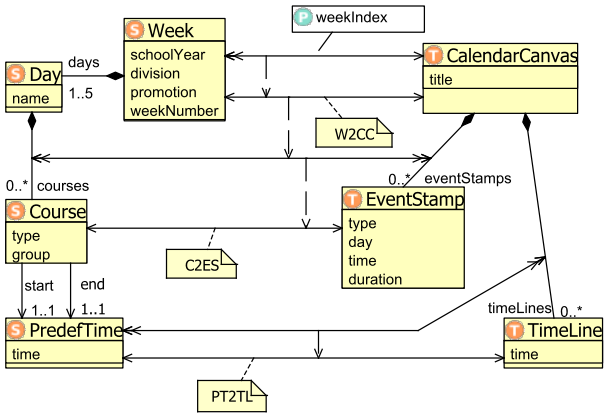

Figure 6: The "Week2Calendar" mapping

The mapping then continues with *a ForEach* operator that maps all week *courses* with the *eventStamps*. *Map* operator *C2ES* then maps each *Course* with an *EventStamp*; its code starts by defining *EventStamp* properties:

```
EventStamp.type = Course.type;
EventStamp.day = (Course.Parent() as Day).name;
EventStamp.time = Cource.start.time;
EventStamp.duration = Cource.end.time − Cource.start.time;
```

The second line of the above code (function *Parent*) cannot be defined with usual bindings without writing specific *ad hoc* code. The previous mapping then defines relation *EventStamp.descriptions*:

```
EventStamp.descriptions [1]. text = Course.topic. title ;
Teacher t = Course.teacher;
if (t == null)
    t = Course.subject. teachers [1];
EventStamp.descriptions [2]. text =
    t.firstName + "␣" + t.lastName;
EventStamp.descriptions [3]. text = Course.room.name;
if (Course.group != null)
    EventStamp.descriptions [4]. text = Course.group;
else
    EventStamp.descriptions [4] = null;
```

Once again, usual bindings cannot be used for linking relations with different cardinalities (*e.g. EventStamp.descriptions* with *Course.topic/room/group*).

Operator *PT2TL* maps the predefined times of the schedule to the time lines of the calendar canvas by following the same schema.

4. RUNNING THE MAPPING

The .NET implementation of AcT, called AcT.NET, is based on ADO.NET for the data platform [6], and on WPF for the GUI platform [9]. Figure 7 summarizes the AcT.NET development process applied to our course schedule example. The user first defines models of domain data and their presentations, and the mapping between them, by using the AcTeditor application ①. AcTeditor allows the generation of the final application code that consists of C# and XML Schema files ②. The source schema is specified in XML Schema document *CourseSchedule.xsd*, which is used

by ADO.NET to store and retrieve data in/from XML documents or DB tables. The C# file *CourseSchedule.cs* implements the source domain data model by allowing the loading and storage of source instances, and the navigation within the source instances. Similarly, the target presentation model is implemented in files *CalendarCanvas.cs* and *CalendarCanvas.xsd*.

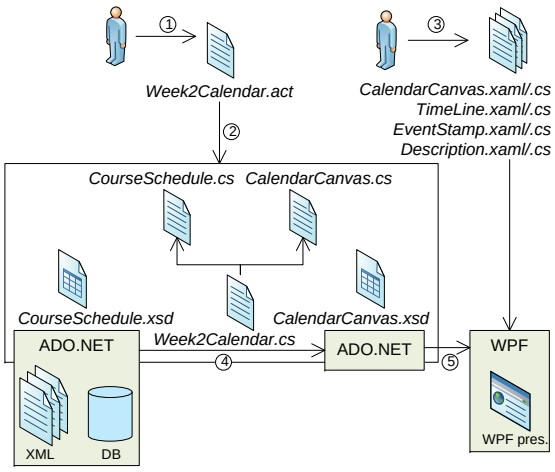

Figure 7: AcT.NET development process

The user needs to create WPF files if the target presentation requires the implementation of new specific controls, called "user controls" ③. In our example, user controls *CalendarCanvas*, *TimeLine*, *EventStamp* and *Description* must be defined using either VisualStudio .NET or Expression Blend. XAML files define the static structure of the user controls; for example, file *TimeLine.xaml* is defined as follows:

```
<UserControl>
    <Grid Name="lineUI" Canvas.Top="0" Height="800">
        <Line X1="0" Y1="0" X2="0" Y2="10000"/>
        <TextBlock Name="timeLabel" Text="" />
    </Grid>
</UserControl>
```

The C# code defines the behavior of the user control that consists of updating its static structure whenever a property of the user control changes. For example, file *TimeLine.xaml.cs* is defined as follows:

```
partial class TimeLine : UserControl {
    void PropertyChanged(
            string prop, object oldval, object newval) {
        int mins = 60 * (time.Hour-7) + time.Minute-45;
        lineUI.Canvas.SetLeft(mins * 2);
        timeLabel.Text = time.ToString();
    }
}
```

File *Week2Calendar.cs* contains the active transformation that implements mapping *Week2Calendar.act*. The transformation manages the link between the source data and the target presentation ④: the target presentation is initially built by the active transformation, *and* subsequently updated whenever the source data changes. The implementation is based on the observable/observer design pattern, which is fully implemented by ADO.NET: the transformation is the observer of observable ADO.NET source data. The graphical rendering of the ADO target presentation is finally performed by the WPF presentation ⑤. As for active transformations, a WPF presentation is an ADO observer: it observes the corresponding ADO target presentation in order to synchronize its rendering.

5. CONCLUSION AND PERSPECTIVES

In this paper, we propose to reap the benefits of defining the links between domain data and their presentations through *mappings*. This approach avoids the need of handwriting often complex active transformations: active transformations are generated from mappings, thus allowing the execution of the mappings on a GUI platform. We explain, through a "course schedule" example, how our AcT framework allows the specification of mappings between the course schedule document and its graphical presentation through a dedicated editor, and how they can be run through active transformations. The example illustrates the simplicity of the proposed framework.

However, the current version of the AcT language offer a limited expressiveness with respect to other mapping languages [3, 8]. For example, AcT mappings map only one source element to one target element. Moreover, it does not offer bidirectionality, which is an important feature for GUIs. The next step of our work is thus to extend the AcT.NET implementation so that it can generate active transformations for mappings expressed in the Malan language [3].

6. REFERENCES

[1] O. Beaudoux. XML active transformation (eXAcT): Transforming documents within interactive systems. In *Proc. of DocEng'05*, pages 146–148. ACM, 2005.

[2] O. Beaudoux, O. Barais, J.-M. Jézéquel, and A. Blouin. Active operations on collections. In *Proc. of MoDELS'10 (in press)*, 2010.

[3] A. Blouin, O. Beaudoux, and S. Loiseau. Malan: A mapping language for the data manipulation. In *Proc. of DocEng'08*. ACM, 2008.

[4] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[5] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language Toolkit*. Addison Wesley Professional, 2009 (to be published).

[6] B. Hamilton. *ADO.NET 3.5 Cookbook*. O'Reilly, 2008.

[7] G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of OOP*, pages 26–49, 1988.

[8] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In *Proc. of SIGMOD'07*, pages 461–472. ACM, 2007.

[9] C. Sells and I. Griffiths. *Programming Windows Presentation Foundation*. O'Reilly, 2005.

[10] L. Villard and N. Layaïda. An incremental XSLT transformation processor for XML document manipulation. In *Proc. of WWW'02*, pages 474–485. ACM, 2002.