Babylone, enabling Eclipse with framework assistance tooling

Mireille Blay-Fornarino and Estelle Ringenbach
University of Nice / I3S Laboratory
930, route des Colles
06903 Sophia-Antipolis, France
{blay|ringenba}@essi.fr

Abstract

In order to facilitate framework usage, and capture framework dependencies in Eclipse, we present the Active HotSpot Model and its implementation Babylone.

1 Introduction

Although Eclipse raised the standard of programmer assistance in providing concepts such as quick-fixes, refactorings or code templates, it falls short of help in the usage of framework. This article presents Babylone an eclipse implementation of the Active HotSpot Model(AHSM). After an introduction of the problematic, we describe the model and its integration in eclipse, before giving perspectives on this work.

2 Background, problematic

In a few years, frameworks became the cornerstones of application developments covering domains as various as enterprise management, software components or IDE. Fitted with well-known hot spots, framework ready-to-use structure enables the creation of bigger and bigger applications in less and less time, since the applications developers using frameworks only have to focus on their business code. Although hot spots clearly identify the feature they will enable, fulfilling one of these is not always easy, since it is often required to understand the inner behavior of the framework to fully take advantage of it. This would not happen if some documentation and information available at design-time were made available at reuse-time.

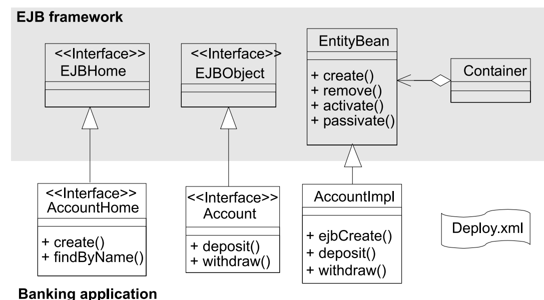

Figure 1: EJB framework and a customization

In order to facilitate framework usage and avoid the loss of design-time information, a twofold model named Active HotSpot Model has been developed [5, 6]. This language independent model focus on the expression of structural and behavioral dependencies from design- to reuse- time and so promote the documentation to be a major actor of the user assistance. In this article we only focus on the structural aspect and the underlying problems.

Structural dependencies capture framework usage constraints that must be respected when a customization is done by the framework user. For a given framework, those constraints are the expressions of design choices made by the framework developer. On a pragmatic point of view, structural dependencies are relations binding two or more heterogeneous elements of a framework (class, method, file, etc.) in the way that if an action is done on one element (creation/deletion/modification) an action should be done on the others.

In order to show an example of what structural dependencies are, we use the EJB framework (which allows to build distributed Java components, see figure 1) to create an EJB compo-

nent called `Account`. The resulting customization will be correct if the following constraints are respected:

- `Account`, the component interface, must implement the framework interface `EJBObject`. `AccountImpl`, the implementation of the account component, must extend `EntityBean`. `AccountHome`, the factory for the account component, must implement `EJBHome` – Here the constraints are on the inheritance relationship.
- `AccountImpl` must implement all methods from `Account` because `AccountImpl` does not implement `Account` – Here there is a constraint on the addition of methods.
- `AccountImpl` must contain one `ejbCreate` method per creation method on `AccountHome`, the factory. The framework will manage the correspondence between the specification in the interface and the implementation in the class following naming constraints.
- Last but not least, a file named `deploy.xml` must be filled with the name of all the classes created – This shows that dependencies can be among heterogeneous elements.

From this shorten example that could have been extracted from a cookbook, one can guess that dealing with those dependencies is burdensome and error prone if no help is provided. Those errors are even more tricky to find when they are not caught before runtime. To answer these problems, models like the Hooks [2] or the specialization patterns [3] have been proposed but none is language independent, or offers a clear way of expressing dependencies between elements that do not exist. Of course framework specific tools are available, but they are costly to develop and maintain, and can only be written once the framework is stable. Moreover those tools often propose a simple wizard type of assistance to the consistence which is not suitable, since nothing prevents the user to make change in the generated skeletons and jeopardize the consistency carefully generated. In fact a user may expect assistance and verification facilities based on a language independent model allowing reasoning on non-existing elements. It is to fulfill these goals that the AHSM model has been developed.

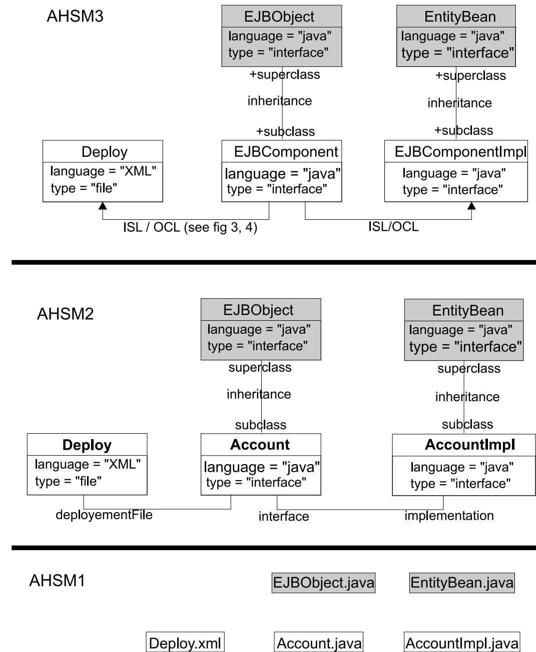

Figure 2: Modelling layers

3 The Active HotSpot Model

The overall goal of the Active HotSpot Model is to provide framework users with active and passive assistance while creating framework customization. To make this assistance possible, the framework developer has to provide the framework with its structural dependencies; this is also called consistency model.

3.1 Layers overview

AHSM is made of three layers represented figure 2.

- AHSM1: the physical layer. It represents the actual framework resources and the resources resulting from the customization. The elements found in this level can be of any kind: java files, XML files, database tables, etc. In our example, it consists of `EJBObject.java`, `EntityBean.java`, `Account.java`, `deploy.xml`,..
- AHSM2: the consistency description layer. It consists of a partial but enhanced representation of AHSM1. This AHSM suitable representation of the framework depicts elements from the framework and the customization.

In this layer, only information relevant for consistency management (the elements implicated in dependencies) are reified in matter of elements called `ElementDescriptors`. Relation between element descriptors are kept through `Link`. Those links are typically used to represent relations that will find a direct signification at the physical layer (for example the inheritance relation between `Account` and `EJBObject`). More importantly links also represent meta-information like the roles that two elements play relatively to each others (for example `AccountImpl` is the implementation of `Account`). This is typically the way some design-time information will be captured and made available at reuse-time. `ElementDescriptors` contain the following information: a name, a type and a language (to support heterogenous elements) and a reference to the physical element.

- AHSM3: the framework consistency model layer. It is the core level in which structural dependencies that should be enforced and verified on customization are expressed. It consists of a reified representation of framework elements into roles (called `RoleDescriptors`) connected by `ConsistencyRules`. A role descriptor captures the role played by elements appearing at some point in time in AHSM2 (they may already exist). It is made of a name, a language, a type, and a textual description of the role in the framework. The consistency rules, connecting the role descriptors, are carrying the precious information about the consistency: ISL [1] and OCL rules [4].

AHSM2 and AHSM3 entities are specified in a meta-model defining the semantics of the relations.

3.2 Relation between layers

We now gives an overview of the layers and their relations.

3.2.1 AHSM1, AHSM2

The consistency description layer and physical layer are related by an operation called mapping. This operation works both ways and provides a mapping from element descriptors and links (found in AHSM2) to physical elements (found in AHSM1) and vice-versa. This mapping, based on the type and language information available in the element descriptors, is key to the language independence of the model. To ensure this property, this mapping is implemented by a Factory which can be compared to the projection operation defined in the MDA.

```
interaction Component(RoleDescriptor EJBComponent,
                      RoleDescriptor EJBComponentImpl){
   EJBComponent.realize(String name) ->
   cptED = EJBComponent.realize(name);
   implED = EJBComponentImpl.realize(name + "Impl");
   newLink(cptED, "interface", implED, "implementation");
}
```

Figure 3: ISL rule

3.2.2 AHSM2, AHSM3

AHSM3 and AHSM2 are related by two operations, one for active assistance called *realization*, and the other for passive assistance called *verification*.

The realization is in charge of helping framework usage to ensure the correctness of customization when creation, modification or deletion is done. It generates an element descriptor which type and language are derived from the role descriptor. Then the ISL part of the consistency rule attached to the role descriptor being realized is executed. This ISL rule describes other realization that will be automatically executed to retain consistency of the customization. For example, the rule shown figure 3 indicates that the realization of `EJBComponent` will trigger the realization of `EJBComponentImpl` and the resulting element descriptors will be linked together. It can be noticed that role descriptors keep track of the generated elements descriptors.

The verification, contrary to the realization, must be explicitly asked by the user to check the consistency of a framework customization. This is expressed in the OCL part of the consistency rule and rely on the links connecting elements descriptors as well as the tracking of element descriptors done by role descriptors. For example, figure 4 shows an OCL rule verifying that for every component there exits a corresponding implementation class.

```
Context  EJBComponent  inv:
 EJBComponent.get("eltDescriptors")->forAll(s |
     EJBComponentImpl.get("eltDescriptors")->
     exists (se| s.get("name")+"Impl" == se.get("name")) )
```

Figure 4: OCL rule

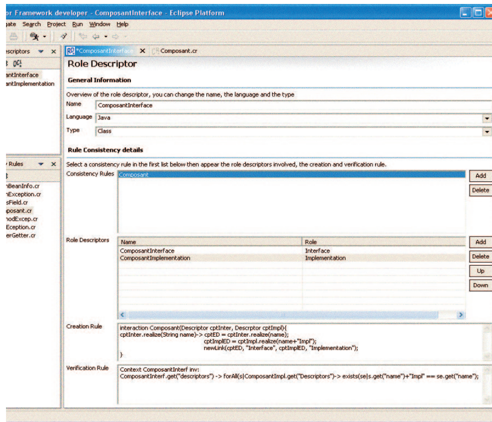

Figure 5: Framework developer perspective

4 Babylone: the integration of AHSM in Eclipse

As mentioned in the introduction, Babylone is the implementation of the AHSM in Eclipse. It features a full implementation of the model and a support for the java language.

4.1 Using Babylone

Babylone keeps the separation between the framework developer and the framework user in proposing two different user interfaces.

4.1.1 The Babylone perspective

The Babylone perspective shown figure 5 is mainly dedicated to the framework developer. It provides facilities to create the consistency model (AHSM3) of a framework as while this one being developed. It includes a view and an editor to manage the dependencies library as well as a view and an editor to associate consistency rules with role descriptors. The consistency model is saved in a separated file which will be delivered with the framework.

4.1.2 The framework user part

Because as developers we do not like changing our habits and being disturbed, we did not created a framework user perspective and the integration of Babylone has been made as discreet as possible while maximizing the assistance. To start receiving dynamic assistance, the user has to associate its project with a consistency model. Once this is done, it will not receive assistance before opening the Follow-up view. This view proposes role descriptors that matches the element the user is currently editing. For example, in the context of a project associated with the EJB consistency model, if the user creates an interface extending `EJBObject`, the Follow-up view will propose to associate this interface with the role descriptor `EJBComponent`. Of course, the user has nothing to do if he does not want to take advantage of the assistance. In contrary, if he decides to, he just has to select the element in the Follow-up view and says associate. From then on, the newly EJB component created (for example `Account`) will be controlled and an implementation class (ex. `AccountImpl`) and a `deploy.xml` file will be automatically created. Note that the automatic creation can be turned off and replaced by the corresponding tasks in the Task List. This will give users the opportunity to accept or deny the proposed operations.

If the user misses an association, it is still possible to use the browser associated with the targeted language and use contextual menus to associate an element with a role descriptor.

The verification is triggered explicitly by the user on a per project basis. The result indicates which constraints are violated if any.

4.2 Implementation in eclipse

Babylone is composed of 3 plugins. The babylone.core plugin implements the model, babylone.developerUI plugin for the framework developer interface and babylone.userUI for the framework user interface.

4.2.1 Babylone core

Since our model is language independent the core plugin exposes an extension point to allow extra language support to be plugged-in. Providing a new language consists in implementing IFactory and IReverseFactory to map ele-

ments descriptors to physical elements and vice-versa (see 3.2.1), and IListener to capture the creation/deletion/modification in the user code.

The implementation of a few wizards may be required when no context is available for the creation of a physical element (they are generally derived from the targeted language support).

Because Babylone is really language independent, no integration with language specific views is provided, and it is up to the person providing new languages to do the integration.

4.2.2 jBabylone

jBabylone is the name given to the plugin offering support for the java language. It extends the extension point previously described and rely on JDT. It uses a JavaCore listener to capture the java delta and the java code is manipulated thanks to the JavaCore API. However because the granularity of java delta was too big, we had to rely on some internal classes to edit elements like return type, parameters, etc.

jBabylone is seamlessly integrated with the JDT views by providing contextual menus to enable the association of elements, start the verification, etc. It also reuses JDT wizards.

5 Conclusion and further work

In this communication we presented Babylone and AHSM, an eclipse integrated tool and its model providing assistance when developing with frameworks. We also gave examples of how Babylone can help eclipse in dealing with framework, and more generally structural dependencies.

We are now envisioning several directions to this work. On a model aspect, we want to study the relation with the MDA. On a implementation aspect, we would like to add support for new languages, study the advantages we could take of EMF and GMT and try to change some complex Websphere Studio wizards into "always-running wizards" by using our model. On a usability aspect, we will carry tests on users and see if the tooling really helps developers productivity and framework documentation.

Acknowledgements

The authors would like to thank: the Eclipse Grant for funding part of this work, the UQÀM for hosting Estelle, and Pascal Rapicault for his continuous assistance.

About the authors

Mireille Blay-Fornarino is an assistant professor at the University of Nice and researcher at the I3S-CNRS. She has been involved with OO since late 80's. She is the co-founder of the Rainbow project and her main research interest includes meta-programming, meta-modelling, and component models.

Estelle Ringenbach is a software engineer. She earns a master degree from the ESSI. After two years of work in the industry, she joined the Rainbow project as the main contributor to Babylone.

References

[1] A.M. Dery, M. Fornarino, B. Arcier, L. Mule, and S. Moisan. Distributed access knowledge-based system: Reified interaction service for trace and control. In *3rd Int'al Symposium on Distributed Object Applications*, 2001.

[2] G. Froehlich, H. Hoover, L. Liu, and Paul Sorenson. Reusing hooks. In *Building Application Frameworks*. Wiley, 1999.

[3] M. Hakala, J. Hautamäki, K. Koskimies, J. Paakki, A. Viljamaa, and J. Viljamaa. Generating application development environments for java frameworks. In *Generative and CBSE*. Springer-Verlag, 2001.

[4] OMG. Object constraint language specification, 1997.

[5] P. Rapicault. *Models and techniques to specify, develop and use a framework: a meta-modelling approach*. PhD thesis, University of Nice, May 2002.

[6] P. Rapicault, J.P. Rigault, and L. Bourlier. Model, notation, and tools for verification of protocol-based components assembly. In *Component Deployment*, LNCS 2370. Springer-Verlag, 2002.