Supporting Development of Enterprise JavaBeans Through Declarative Meta Programming

Johan Fabry*

Programming Technology Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium. Johan.Fabry@vub.ac.be

Abstract. Enterprise JavaBeans is a successful component model for the development of distributed business applications. Enterprise JavaBeans have to adhere to a set of rules and guidelines which, amongst others, require that a significant amount of glue code between the components is written. By using Declarative Meta Programming (DMP) we can codify these rules and guidelines as a logic program which operates on the Java code. Our DMP development tool can not only generate Java code, but can also verify compliance of developer code.

1 Introduction

Building the back-end servers of Internet information systems is a daunting task. The business application logic of the system is intertwined with services such as concurrency, transaction management, persistence and so on. To ease development of these servers, middleware technology such as Sun's Enterprise JavaBeans (EJB) component model[5, 6] has been developed.

In this model business logic is encapsulated into a number of Beans, while additional tasks, such as network communication, concurrency and transaction management, are performed by the EJB server. Beans cooperate to respond to network requests originating from the user interface, using data which is stored in the database.

The EJB specification defines two kinds of Beans: entity Beans and session Beans. Entity Beans are an objectified representation of data in a database and are therefore persistent. Session Beans encapsulate the business logic of the application and are not persistent.

To be considered a Bean, a business object must conform to a number of requirements, as given in the EJB specification. This ensures correct cooperation between the EJB server and the Bean. To satisfy these requirements a certain amount of repetitive tasks, which can be automated, have to be performed by the programmer.

So, although EJB is a significant step forward, it is still non-trivial for a developer to develop using this component model. Development support for building

* Author funded by a doctoral grant of the Flemish Institute for the advancement of scientific-technological research in the industry (IWT)

EJB's enhances the process by, for example, decreasing the amount of code that should be written by the developer. We propose the use of Declarative Meta Programming (DMP) to provide such development support. DMP has the added value of being able to reason about the code being written by the developer, which makes it possible to firstly extend the code generation capabilities beyond what is offered by current-day tools. Secondly, we can verify code compliance with regard to the EJB specifications, which allows us, for example, to detect possible runtime exceptions at coding time.

We will now give an introduction of DMP before showing how we use DMP to provide advanced development support.

2 Declarative Meta Programming

Checking whether a piece of source code matches some requirements, detecting violations of these requirements, and generating code according to the requirements are activities eminently suited to Declarative Meta Programming (DMP).

DMP[8, 4, 3] is a technique where a declarative meta language is merged with a standard OO base language. Base level programs are reified as logic facts at the meta level. This allows the logic interpreter to reason about the base program. Using DMP, we can specify requirements of the base program in logic code, query the interpreter for code that does not comply to these rules and we can generate base code.

For our experiments we extended the tool SOUL[9], to provide a version, named SOULJava, which reasons about base programs written in Java.

Working with SOULJava entails first parsing the Java code, which adds a parse tree representation of it to a Java code repository. The logic interpreter reasons about the code in this repository, and is able to add and modify it. To compile the code, the parse trees are exported to java source code format, and are compiled with the standard Java compiler.

Basic use of the EJB support only requires coding in Java, and providing a few logic facts to the interpreter. This does only require minimal knowledge of the logic language, as will be shown later.

3 Development Support

Using DMP we have built an EJB, named SJB (SoulJava ejBtool). In this section we first show how code generation similar to that done by existing tools[1, 7, 2] is done. Next we introduce the more advanced features of SJB in code generation and code verification.

3.1 Basic Code Generation

Basic usage of SJB consists of specifying properties of a Bean and letting SJB generate the required code. Indeed, we can generate fully functional entity Beans solely by specifying their properties. Consider the following specification for a Room EJB used in a computer room booking system:

```
ejbClass(Room, entity).
primaryKey(Room, number).
field(Room, number, String, <persistent>).
field(Room, numcomputers, int, <persistent,property>).
field(Room, computertype, String, <persistent,property>).
```

This specification consists of 6 logic facts written in SOULJava. We declare "Room" an entity bean, with "number" as primary key, followed by the instance variables of the bean. The arguments of the "field" declarations are, in order, the Bean to which it belongs, its name, type, and a list of attributes of the field. At this time, fields may have two attributes: whether it should be "persistent", and whether it is a "property", i.e. accessible to clients[1].

SJB contains a number of logic rules that transform this specification to a parse tree for the equivalent Java program. We illustrate this by examining the rules for fields. Consider first the rule for the instance variable declarations as given below:

```
fieldDeclaration(?class, ?field, ?tree) if
   field(?class, ?field, ?type, ?attrib),
   javaInstVarDecl(?class, public, ?type, ?field, ?tree).
```

This rule introduces new syntax: words prefixed with "?" are logic variables, "if" separates the signature from the body of a rule, and a comma in the body signifies a logical "AND". "field" is used to verify if the field exists for that class, and to obtain the type and attribute list. "javaInstVarDecl" is a logic rule which we will not discuss in detail here; it unifies its last argument with a parse tree according to the previous three arguments. In the rest of the text we adopt as convention that logic variables with the name "?tree" contain parse trees.

Note that due to the multi-way reasoning capability of SOULJava, the "field-Declaration" rule can be called with any variable unbound, and the results will contain all possible bindings for the unbound variables.

As we have seen above, if a field has the "property" attribute, clients must be able to access and modify it. The EJB specification states that this is done by using getter and setter methods which also have to be declared in a specific interface of the Bean. The names of the methods should be "getFieldName" and "setFieldName", where "FieldName" is the capitalized name of the field. We show the rule for a getter (the rules for setters and the interface declarations are similar, therefore we omit them):

```
ejbGetter(?class, ?field, ?tree) if
   field(?class, ?field, ?type, ?attrib),
   member(property, ?attrib),
   javaReturnStatementBody(?field, ?bdtree),
   capitalize(?field, ?capfd),
   javaMethodDecl(public, ?type, get+?capfd, <>, ?bdtree, ?tree).
```

[1] We use the term "clients" of a Bean to refer to all objects that use services offered by the Bean. Clients can be other Beans or user interface software.

Four new elements are introduced. Firstly, the "member" predicate verifies if "property" is contained in the attribute list. Secondly, a parse tree for a method body containing a "return" of the field is generated. Thirdly we capitalize the name of the field, and fourthly we create the getter method. In this last statement, "+" signifies concatenation and an empty list "<>" is given for the arguments.

Usage of this rule is analogous to "fieldDeclaration": if only "?class" is bound to a class name, it returns a list of all property fields of that class, and a list of the parse trees for the getters of that class.

When exporting code the above rules will be called at given times, and their results will be integrated into the Bean package.

3.2 Advanced Code Generation

Because DMP allows us to reason about Java code written by the developer, we can extract this information to generate additional code and configuration information.

Beans cooperate to implement the business logic by calling methods on other Beans. To be able to refer to another, external, Bean, some configuration issues need to be resolved. For example, at compile time and at runtime the interfaces of the external reference must be available. SJB also takes care of this by detecting all occurrences of external references, as shown in the following rules, and including the interface definitions where needed.

```
usesEJB(?used, ?methodtree) if
   ejbClass(?used, ?beantype),
   traverseMethodParseTree(?methodtree, <?msg, ?type>,
      messageSend, collectReturnType),
   or(equals(?type, ?used+Home), equals(?type, ?used+Remote)).

collectReturnType(messageSend(?msg), <?msg, ?type>) if
   messageReturnType(?msg, ?type).

usesEJB(?used, ?methodtree) if
   ejbClass(?used, ?beantype),
   traverseMethodParseTree(?methodtree, <?msg, ?type>,
      castExpression, collectCastType),
   or(equals(?type, ?used+Home), equals(?type, ?used+Remote)).

collectCastType(castExpression(?msg), <?msg, ?type>) if
   castType(?msg, ?type).
```

We use a generic parse tree traversal rule "traverseMethodParseTree" here, which will traverse the method parse tree, respectively find message sends and cast expressions, and collect the resulting type. ("messageSend" and "castExpression" are logic representations of parse tree elements.) "usesEJB" is called

on all methods at code generation time, and will return a list of used Beans. This list is then used to ensure that all needed interfaces are available.

3.3 Validating Code

When the developer starts adding non-generated code to the Bean, we can verify that this code complies to the specifications. It has been shown that DMP is eminently suited for verifying code, for example when regarding coding conventions and adherence to design patterns[4].

Detecting violations early in the development process shortens the development cycle. Using SJB we can perform code verification at the moment a method definition has been completed by the developer.

Consider cooperation between Beans: this is performed by method calls, however there are some restrictions on these method calls. For example: return and parameter types must be either primitives or Strings or serializable types or implement the Remote interface. The correct definition of serializable types can be quite complex, however in the most cases, Serializable types implement the Serializable interface.

Due to this complex definition, compliance cannot be fully checked at compile time, and the compiler can be quite forgiving. This allows that, at runtime, a type will not comply to this rule. In such cases a runtime exception is thrown and the operation is aborted. We want stronger compliance checking, and at least have a warning if there is ambiguity. This is achieved by calling the "beanMethodParametersCompliance" rule, shown below:

```
beanMethodParametersCompliance(?mtree) if
   findall(?typelist,
      traverseMethodParseTree(?mtree, <?msg, ?typelist>,
         messageSend, collectMessageTypes), ?typelistcoll).
   forall(member(?typeslist, ?typelistcoll), bmpComp(?typeslist)).

collectMessageTypes(messageSend(?msg), <?msg, ?typelist>) if
   messageArgumentTypes(?msg, ?typelist).

bmpComp(?typeslist) if
   forall(member(?type, ?typeslist), bmpTypeComp(?type)).

bmpTypeComp(?type) if primitive(?type).
bmpTypeComp(String).
bmpTypeComp(?type) if classOrSupersImplement(?type, Serializable).
bmpTypeComp(?type) if classOrSupersImplement(?type, Remote).
```

We first find the lists of argument types for all method calls on other Beans and collect them in "?typelistcoll". Then, for each member of the collection, we

verify if all the elements of that type list conform to the specification as given above.

This section detailed how we can use DMP to provide development support for building Beans. We have shown how our tool, SJB, can generate more code than traditional tools due to the ability to reason about the Java code. Furthermore we illustrated how this ability allows us to verify developer code compliance early in the development cycle.

4 Conclusions

We have shown here that DMP's ability to reason about code allows it to extract information from the developers' Java code, and use this to aid in development. This is done first by generating a larger amount of "support" code, required for EJB compliance, than other tools. Repetitive and labor-intensive tasks are significantly automated. Second, developer code is verified for compliance, as it is being written, so to speak. This allows us, for example, to detect possible runtime exceptions at the start of the implementation phase.

It is clear that making the programmer write less code, and verifying the code for compliance, lowers the chance for errors in the code, and therefore significantly supports the development cycle.

5 Acknowledgments

Thanks to Tom Mens, Werner Van Belle and Dirk van Deun for proofreading this paper.

References

[1] Cedric Beust. EJBGen. http://www.beust.com/cedric/ejbgen.

[2] Borland. JBuilder. http://www.borland.com/jbuilder/.

[3] Programming Technology Lab. Declarative meta programming pages. http://prog.vub.ac.be/research/DMP/.

[4] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert Systems with Applications*, 2002.

[5] Sun Microsystems. Enterprise JavaBeans specification. http://java.sun.com/products/ejb/docs.html.

[6] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2001.

[7] Pramati Technologies. Pramati studio 3.0. http://www.pramati.com/product/studio30.

[8] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proc. Int'l Conf. TOOLS USA'98*, pages 112–124. IEEE Computer Society Press, 1998.

[9] R. Wuyts and S. Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. *Multiparadigm Programming with Object-Oriented languages*, 7, 2001.