UDC 004.318

V. TSUKUR, O. DROBNYCH, Yu. PLAKOSH

IMPLEMENTATION OF INTERPRETER OBJECT ORIENTED DESIGN PATTERN ON BASIS OF XML LANGUAGE

Abstract: This article describes programmatic issues and solutions while implementing XML-based Interpreter Object Oriented Design Pattern. Aims and usage of such Interpreter are presented. Language elements and their implementation for Java 2 are described.
Key words: object Oriented Design Patterns, XML.

Анотація: У статті описані задачі та їх реалізація при програмуванні об'єктно-орієнтованого шаблона проектування «Інтерпретатор» на основі мови XML. Представлено мету та приклади застосування такого інтерпретатора. Описано елементи мови та їх реалізація на мові Java 2.
Ключові слова: шаблони об'єктно-орієнтовного проектування, XML.

Аннотация: В статье описаны задачи и их решения при программировании объектно-ориентированного шаблона проектирования «Интерпретатор» на основе языка XML. Представлены цели и примеры использования такого интерпретатора. Описаны элементы языка и их реализация на языке Java 2.
Ключевые слова: шаблоны объектно-ориентированного проектирования, XML.

1. Introduction

We are representing academic project named **mappinG_Ray** which is XML [1] based scripting language for broad usage developed on Java 2 [2] language.

First of all it is important to notice that **mappinG_Ray** is not an independent standalone script language. It is *heavy specification* and *implementation* of **Interpreter [3]** design pattern[4] for your **Java** code as a whole. It is a mean for controlling your classes in some way. The specific of this control type is that it is scripted. Your **mappinG_Ray** code is not an ordinary source code, because it is not compiled, but interpreted. You can change the behavior of your system without recompilation, just by changing script sources. **mappinG_Ray** doesn't pretend to replace **Java** – it is just another way to work with its objects. **mappinG_Ray** is based on **Java** *reflection [5]*.

The idea of creating **mappinG_Ray** was founded while developing one serious external protocols communication system. Let's imagine a big **Java**-based system (A), which needs to "communicate" with other systems using specific protocols (B). A wants to send request to B and get response on it (work sample of synchronous protocol). That request can be in the form of XML, specific text format recognized by external system or it can even use **Java** objects to hold data to be carried as request. How should we form a request? Well, we can hardcode request logic in **Java** sources and put an end to it. Our **Java** code will take A system objects and will form request in appropriate type. But if mapping will change we will need to change **Java** code heavily and make recompilation. It is rather slow and inconvenient, especially when system is rather big/complicated and developers are split. There is another, more flexible way of resolving this issue. By adding one more tier we can delimit request mapping from other request details. Let **Java** code know, how to send request and how to get answer but mapping itself is rather fluid and it is better to move this piece of work somewhere else. This "somewhere else" is **mappinG_Ray** code. While developing A system, mentioned before, we did use approach of scripted tier, but it was too dumb and inconvenient. The word *mapping* here shows the general task we had while developing A system: mapping data from native system to protocol-specific format or vice versa.

Originally, **mappinG_Ray** was developed as XML specification and sequence of operators corresponded to tags with attributes. Why not in ordinary text form? Using of XML give some advantages.

First of all, it is much easier to create code processing system when data (code) is held in XML, because there are great amount of XML parsers available, which are easy in use and understanding. So it is simple to create own **mappinG_Ray** processing engine for client programmer, if some implementation of parsing and evaluation system doesn't satisfy his needs. Besides, the idea was to make **mappinG_Ray** extensible. Extensibility is a basic feature of XML document. Known tag is processed. Unknown tag is ignored. The same rule is for operators. Known is recognized and executed and unknown stays untouched.

mappinG_Ray unit, called *stream*, consists of <imports> section, which holds definitions of external classes being used in the code, <operations> section, which holds additional processing functions, and <main> section, processing core. **mappinG_Ray** code itself is not object-oriented: it doesn't give an ability to define own classes, interfaces and so forth. Why? *There is no need in it*: its purpose is to control existing code in some structured manner, but not to give extended abilities in code organization. And at the same time it's wrong to think that **mappinG_Ray** code doesn't use objects. It does use them. Objects can be created and controlled. You can control objects with the help of primary and temporary references. Primary references point to the objects, which hold input and output script data. As far as **mappinG_Ray** unit can be called an algorithm, we can talk about its input and output. In the case of **mappinG_Ray** these input and output objects are referenced by so called primary references. By using these references in **mappinG_Ray** code, programmer can extract input data and update output data. The purpose of temporary references is the work with intermediate objects in the *stream*. You can create any object, which is being included in **mappinG_Ray** code caller project.

It is worth to admit that **mappinG_Ray** suggests a large variety of operations to use in the code. Actually, it gives you an ability to implement any algorithm you want by supporting linear, conditional and cyclic structures.

New temporary references can be created with the help of **set** operator. This operator is also used for existing references resetting or changing attributes of the objects they point to. **call** operator is used when we have to call some method from the object and when the result of this call is not significant for us, i.e. ignored. **delete** operator deletes temporary references. Programmer can organize conditional branches by using **if** and **switch** operators (**if** operator supports **else** and **elseif** branches, **switch** operator is even more flexible than the one in **C/C++/C#/Java**). Cycles are declared as **for**, **while**, **do-while**, **until**, **do-until**. You can call **break** and **continue** operators in cycles. Their meaning is the same as in **Java**. But they are a little more flexible: you don't use labels to switch from inner cycles to outer cycles, but specify number of levels to skip up! Also it is worth to admit **foreach** cycle. It works as **foreach** in **Java** and **C#** and can enumerate any array, java.util.Collection, java.util.Map, java.util.Iterator instances. **return** command is used when there is a need to break operation's (*stream* function) work. Also it is used to end the work of main processing core. **block** operator is used define inner code scopes. **mappinG_Ray** supports exception handing and defines truly functioning **try-catch-finally** sequence along with flexible **throw** operator. Logger info can be written with the help of specialized **logger** operator. A special extension is an ability to write **Jython** code in **jython** operator.

2. The concept of source and destination objects

Single **mappinG_Ray** unit can be treated as algorithm. Any algorithm is characterized by input data, processing routine (sometimes it is called a set of rules) and output data. Middle component is the main thing we'll look at through this document. Now let's concentrate on first and last concepts. So what is input and what is output in **mappinG_Ray**?

Input objects are defined in their own holder. Client programmer creates an object, which implements java.util.Map<String, Object> interface, and fills it with references to input objects. Than this map instance is used to initialize com.mappingray.external.SourceMap object, which is actually a *safe* wrapper for that map. SourceMap class is a native implementation of com.mappingray.external.NonChangeableMap interface. Why is it important? Because **mappinG_Ray** processor accepts source objects in the form of NonChangeableMap and SourceMap is valid because it implements this interface. So SourceMap creation process in **Java** code may look like this:

Map<String, Object> sourceObjectsMap = new HashMap<String, Object> ();

sourceObjectsMap.put (…);

…

SourceMap inputData = new SourceMap (sourceObjectsMap);

SourceMap has only one constructor availaility, which expects map to wrap as a first parameter. Pay attention that map must contain java.lang.String objects as keys. This convention is automatically controlled by SourceMap class – you will be unable to create SourceMap if map specified in constructor doesn't follow this rule.

It was the task of **mappinG_Ray** developers to have some centralized container for input data, rather than working with separate references. We considered java.util.Map to be the best choice for this purposes. It is easy and beautifully to extract objects (or references, to be more precise) from the repository by their names. But we don't use it directly! What the reason for that? What is the purpose of wrapper introduction?

mappinG_Ray is capable of creating any **Java** object, calling any public **Java** method or accessing/mutating any public field in available **Java** object. java.util.Map is not a frozen container. It can be changed at any time. And if we shall give the instance of java.util.Map to **mappinG_Ray** programmer, he will be able to change the contents of this map. But input data must not be changed! Here we see the safety and reliability problem. It was admitted that SourceMap is a *safe* wrapper for java.util.Map. What does the word safe means here? It contains methods, which cannot influence map's contents. These include extraction of value object by its key, checking whether map contains some key or value object, defining map size and emptiness state. No direct access to map instance is granted and no modification operations are available! This makes map stable.

But, at the same time extracted objects *can be changed*. That is because **Java** and **mappinG_Ray** source code operate with object references, not directly with objects! And it is impossible to guarantee objects' constant state. **mappinG_Ray** gives some sort of solution for this problem. There is special implementer of NonChangeableMap interface: com.mappingray.external.CloneSourceMap class. It copies all objects from specified map to internal one and tries to make cloning for them. Object will be cloned successfully if it implements java.lang.Cloneable interface, it's clone() method is redefined, has

public access modifier and doesn't throw execution exceptions. So if cloning is important **mappinG_Ray** processor user can work with CloneSourceMap rather than with ordinary SourceMap.

All these means of safety decreases human factor heavily.

It is time to show how we can extract a reference to NonChangeable instance in **mappinG_Ray** code. The access to input data holder is provided by using so called primary references. One of them has reserved name of 'sourceMap':

```
<call script=" #sourceMap.get ('key') " />
```

The sample above extracts reference to a value object with the 'key' key. # is a special symbol in **mappinG_Ray**. It defines that the name being written is the name of *primary reference*. Point symbol accesses the interface of sourceMap primary reference, i.e. the interface of NonChangeableMap entity. Than we write the name of the method being called: 'get'. This method expects one parameter, which is the key for value element. So that is the way we get access to source objects. The sample above doesn't do anything important: it just shows extraction process.

Output objects are stored in the instance of DestinationMap class (also uses the concept of Façade design pattern). The problem of safety we were speaking about in the preceding paragraph is not actual for output system as far as output data must be updatable. That's why DestinationMap class contains methods for registering objects in container (add method) and striking them off the container (remove and clear methods). Also class implements additional query logic: returning sets of keys or entries and collection of values. Actually DestinationMap implements com.mappingray.external.ChangeableMap interface. The process of creating DestinationMap instance in **Java** code is the same as for SourceMap. The sample of destination map usage in **mappinG_Ray** code is shown here:

```
<call script=" #destinationMap.put ('keyForOutputObject', Object @outputObject) "/>
```

destinationMap is a reserved primary reference name. Reference itself points to actual DestinationMap itself, which is defined for **mappinG_Ray** processor. In the sample above outputObject is registered in destination map container with the key keyForOutputObject. @ symbol defines temporary reference. We'll speak about temporary references later. All you have to know for now is that it is a custom reference, being created in the **mappinG_Ray** code.

Methods of SourceMap/CloneSourceMap and DestinationMap classes doesn't catch internal map exceptions: all of them will be thrown to outer levels. Both of them doesn't reference map specified in constructor – they copy its entries into internal map. If SourceMap/CloneSourceMap or DestinationMap constructors will get null reference to wrapping map, internal map instance will be empty. All these implementations support intelligent equals check, hash code and string representation generation, so they can be used for comparison, hash storages and in output.

mappinG_Ray code unit is considered to be an algorithm with its input and output. The work with input is organized with sourceMap primary reference, which represents non-changeable container (SourceMap or CloneSourceMap instance). Output is represented by changeable container in the face of destinationMap reference (DestinationMap instance) and single output object.

3. Let's interact with the help of data pool

Let's pay attention at interaction of **mappinG_Ray** code and caller's code. The question is "How shall we specify source objects for **mappinG_Ray** processor and how shall we get the result (destination objects) of the processing?". In such a way InteractionDataPool class appears on the horizon. What is it and how shall we use it? InteractionDataPool is a holder for both input and output data. It doesn't mean that it holds SourceMap and DestinationMap instances. Generally it holds any implementation of NonChangeableMap and ChangeableMap interfaces (SourceMap and DestinationMap are implementations of these). All client programmer has to do is to create instance of InteractionDataPool, set input data and specify InteractionDataPool to **mappinG_Ray** processor. After processing, client programmer can extract DestinationMap from InteractionDataPool. Method getReturnObject() returns the return object, that additional concept of passing output object to the **mappinG_Ray** code caller we were speaking about:

```
Map<String, Object> sourceObjectsMap = new HashMap<String, Object> ();
sourceObjectsMap.add (…);
…
// Setting instance of InteractionDataPool to processor and running the script.
comServiceProvider.setInteractionDataPool (new InteractionDataPool (new SourceMap (sourceObjectsMap), null));
comServiceProvider.buildCodeObjectModel ().execute ();
```

This sample shows that instance of InteractionDataPool can be created using overloaded constructor. And it is not necessary to specify DestinationMap. If it is null, it will be created automatically by the processor (will wrap empty java.util.HashMap container, obeyed by each **mappinG_Ray** processor).

General structure of **mappinG_Ray** unit.

Here we'll pay attention at XML implementation of **mappinG_Ray**. The high-level stream structure looks like this:

```
<stream>
    <imports>
        …
    </imports>
    <operations>
        …
    </operations>
    <main id='identifier'>
        …
    </main>
</stream>
```

<stream> is the highest level node. Its task is to be a holder for everything else. Parser will throw exception if <stream> tag will not be found on the highest level. <imports> tag contains the list of class declarations, which are used in short class names. <operations> tag contains stream operations (functions). These custom functions can be called from <main> section. Both <imports> and <operations> tags are optional and no reserved attributes are subscribed for them. If there are more than one <imports>

tag (or <operations> tag), all of them will be processed. <main> tag represents the core processing function, holding the sequence of operators to execute. Its presence and uniqueness is necessary. Id attribute is mandatory for <main> tag. It is for helper usage. Direct parsing is done in such a sequence:

1. <stream> tag determination.
2. <imports>, <operations> and <main> tags determination.
3. <imports> tag parsing and processing.
4. <operations> tag parsing.
5. <main> tag parsing.

4. The core

Now it is time to have a look at the core of **mappinG_Ray** stream – main processing sequence. This section is defined by <main> tag with id parameter. This tag is the parent node for all nodes, which represents actual processing. Everything the script has to do is enclosed inside <main>. The choice of tag's name is influenced by general convention of entry point function's name in **C/C++/C#/Java** (main or Main).

5. Creating references

It was told already that references do exist in **mappinG_Ray** and so called primary references were explained already. Understanding temporary references is extremely important. First why do we call these references temporary? Because they are used to hold intermediate processing results. Temporary references are not used by high-level systems. And you must not think about their physical removal from memory. All objects being created are controlled, i.e. removed automatically and safely by **Java** machine garbage collector. This feature is important because it makes the code more reliable and follows the concepts of present-day high-level languages. Actually **mappinG_Ray** processor implementers did nothing to support garbage collection. Nothing, except using **Java** as implementation language for **mappinG_Ray** processor.

Let's have a look at the example:

<set script="@name = new String ('user')" />

It is sample creation script, which can be divided into three parts: temporary reference name/field name (@name), assignment operator (=) and value expression (new String ('user')):

@temporary_reference_name = value_expression

Here '@' is a special symbol which shows that temporary reference name will be specified next. Reference name must start with UNICODE letter or underline symbol and can contain UNICODE letters, digits or underline symbols. The length of the name must be from 1 till 2147483647 and is case sensitive. Value expression is specified right to assignment. You'll discover all variety of value expressions later. Here you can see that right expression is much alike **Java** code. Reserved **mappinG_Ray new** keyword is used for objects' creation. The name of the class being instantiated follows new. Now about class names. What are they? Where do they come from? The name of the class you specify is actual **Java** class, being used by script caller program. So you always work with **Java** classes and **Java** objects. You'll not find something like **mappinG_Ray** type. They don't exist. **Java** is mastering here and it is important to remember this fact. Pay attention that the name of **Java** class in the sample is not full: String is specified,

but actual class is java.lang.String. There is no error here. **mappinG_Ray** automatically identifies classes in java.lang package. Just like **Java** do. For all other classes package name is mandatory. What is the result of **new** operation? The result is the reference to newly created object, which than set as value for temporary reference.

Temporary references aren't characterized by type or class. You simply write its name while creating. To be more precise, each reference points to the instance of java.lang.Object class, base class for all other classes in **Java**. So you can reference any object you wish. Such abstractness doesn't mean that you are able to call Object's methods only (it will occur in **Java** code if you have reference of type java.lang.Object). You are able to call any method and access any field of the object which is being referenced at particular time. Method/field and actual class compatibility is defined at run-time. To understand the things mentioned here, it is worth to compare **Java** code and **mappinG_Ray** code behavior. For example in **Java** you write:

Map map = new HashMap ();

map.put ("key", "value");

The same code in **mappinG_Ray** looks like this:

<set script=" @map = new java.util.HashMap () "/>

<call script=" @map.put ('key', Object 'value') "/>

Let's pay attention at **Java** sample. Programmer specifies new reference with name map of type HashMap and initializes it with new instance of HashMap class. Than put method is called in order to register reference in map. Put method can be called, because it is defined in java.util.Map interface. **mappinG_Ray** code also creates reference but doesn't need its type to be specified. It is just initialized with new HashMap instance. So how does call for put method works correctly? How does it know that map contains not an abstract Object but precise HashMap instance? The key is in **Java** reflection. **mappinG_Ray** reflection expression evaluation system (REE) identifies the class of the object being referenced, tries to find available method/field being called/accessed and if succeeds REE executes appropriate logic. Everything is done at run time. No need to specify reference type – everything can be extracted from object using REE system.

Actually, set operator is used for both creating and resetting references. E. g. you can write:

<set script="@name = new String ('user')"/>

…

<set script="@name = new String ('user 2')"/>

First call creates temporary reference, creates object and assigns reference for created object. Second call doesn't create temporary reference! It sets reference object to "user 2" String object.

6. Literals

mappinG_Ray allows client programmer to specify literals, which are automatically converted to appropriate **Java** objects. There are numeric (long integer, double), boolean, string and reference literals.

Each time a number occurs in the script as a standalone token, it is treated as long integer literal if it doesn't have fractional part specified. Negation symbol is allowed before the token (with no white spaces before constant itself). + symbol before numeric value can be specified but is ignored. Samples of long

integer constants are 1, 2, -10, 2187879. Long integer constants can be specified as decimal values only and their values range from -9223372036854775808 to +9223372036854775807. Something like 1.0, .5 or -99.9 is treated as double constant already. Integer and fractional parts are divided by point symbol. Values range from 4.94065645841246544e-324 to 1.79769313486231570e+308. Exponential part is not supported by **mappinG_Ray**. Long integer literal value is wrapped inside an instance of java.lang.Long class and floating numbers are wrapped inside an instance of java.lang.Double class.

Boolean literals are presented by true and false. These values are wrapped inside java.lang.Boolean.

String literals are enclosed with single quotation marks and can contain any sequence of UNICODE characters: "non-empty string", "another non-empty string". Escape-sequences are not supported yet. All string literals are actually instantiated as java.lang.String objects.

Reference literal is represented by one value: null. It is used to specify absence of object attached to reference.

As far as any literal is represented as an object, these literals can take part in all operations where objects are applicable. For example programmer can create a variety of temporary references in such a way:

```
<set script="@longNumber = 1 "/>
<set script="@doubleNumber = -1.4"/>
<set script="@boolean = true"/>
<set script="@string = 'string value'"/>
<set script="@nullReference = null"/>
```

7. Conclusion

The plans of **mappinG_Ray** development include development of non-XML form of the script, which will be easier to write and read. There are lots of ideas in language commands' improvement. Also the **mappinG_Ray** developers think about the introduction of *abstract object model*, which will be based on XML and which will be programming language independent (some **SOAP** alternative).

Summing up all that was said we come to a conclusion that **mappinG_Ray** can be used as data transformer and controller, programmable mapping assistant, scripted tier in multi-tiered system.

REFERENCES
1. Spencer P. XML design and implementation. – Wrox, 1999. – 426 p.
2. Eckel B. Thinking in Java. Third Edition. – Prentice Hall, 2003. – 820 p.
3. Eckel B. Thinking in Patterns. – Prentice Hall, 2002. – 520 p.
4. Гамма Э., Хелм Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – Питер, 2001. – 344 с.
5. Forman Ira R., Forman Nate Java Reflection in Action. – Manning, 2004. – 300 p.