

# Introducing Polymorphic Features into a Scripting Model of Generator

---

Danijel Radošević and Ivan Magdalenić

Faculty of Organization and Informatics, University of Zagreb, Varaždin, Croatia

Generative programming is a discipline of Automatic programming which strives to make application and the generator development process flexible and generated program code optimized. Because of the lack of appropriate graphic and aspect based generator models, we developed the Scripting model of generator, as a static generator model based on higher level scripts. This paper gives a formal definition of the Scripting model and describes how basic object model properties, like encapsulation, inheritance, and now, polymorphism are achieved. This offers some advantages in generative application development, such as more precise application specification, better generator reusability, and simpler generator model and its easier implementation. The introduced polymorphic features are presented in an illustrative example of a Java application generator.

*Keywords:* generative programming, scripting model, polymorphism

## 1. Introduction

Generative programming [2] is a discipline of Automatic programming introduced, under this name, in the late 1990's. According to a definition, Generative programming represents designing and implementing software modules which can be combined to generate specialized and highly optimized systems fulfilling specific requirements [5]. The main specific difference from other techniques of automatic programming is aspect orientation of the model and independency from the targeted programming language. The programming code contains only instructions necessary for execution of desired task, which makes programming code more optimized.

The Scripting model of generator is a graphic model, originally developed for the needs of

Generative programming based on the scripting languages by Radošević [17]. Different to the object model given by the UML diagrams, it is aspect-oriented, i.e. oriented to define the specific aspects of a future application within its problem domain. Aspects represent the features that are not closely connected to the individual program organizational units, like functions or classes. Consequently, they can appear within different application parts which require a connection model, according to Kiczales [10]. The Scripting model is a kind of a Join point model [9] known from the Aspect oriented programming. A specific difference is in that Scripting model is not type-based i.e. it represents a type-free system because its connecting points are only the connections between the metaprograms and the properties defined in the application specification rather than classes and their objects [15].

The Scripting model, introduced in [17], includes properties which can be compared with object-oriented concepts: encapsulation and inheritance. Introduction of polymorphic features into Scripting model offers all main object-oriented concepts to Generative programming.

The expected scientific contribution of this paper is in:

- Formal definition of Scripting model. The formal definition of Scripting model is described in Section 3. Previous version of Scripting model presented in [17] was defined as a graphic model.
- The addition of the polymorphic features into Scripting model, now covering all basic concepts from the Object oriented programming, which is an important prerequisite for

becoming alternative regarding Object oriented paradigm.

The paper is organized as follows: The related work is presented in Section 2. The formal definition of the Scripting model is presented in Section 3. Section 4 describes implementation of the object model properties in the Scripting model and Section 5 presents polymorphism in the Scripting model. Section 6 deals with open implementation issues. Section 7 gives illustrative example of usage of polymorphism in the Scripting model. The conclusion is given in Section 8.

## 2. Related Work

Generative programming is mostly observed as a discipline based on the Object-oriented programming (OOP). Advanced OOP concepts, like generic classes, along with techniques of object modelling, such as UML, are used in building generators and automatization of programming [2]. Except the OOP, there some other disciplines in the base of generative programming. Aspect oriented programming (AOP) was marked by Guerray [8] as one of the possible successors of OOP. AOP deals with so called crosscutting concerns i.e. features that are shared among more program organizational units like functions or classes [9]. Crosscutting concerns can hardly be handled by parameters, so appropriate code fragments, called aspects, have to appear within different application parts which require a connection model, according to Kiczales [10]. The Scripting model is a kind of Join point model [6][9] known from AOP. A specific difference from original AOP approach is that the Scripting model is not type-based i.e. connecting points have only their names, without any property.

Generators use so called Domain-specific languages (DSL) to specify applications to be generated i.e. DSL can be used to generate members of a family of systems in an application domain [4]. A Domain engineering as a discipline which deals with DSL-s is in the base of Generative programming [2]. The Scripting model uses hierarchical tree-like specification form to propose elements of DSL to be used in generation of applications, which is described in chapter 3.1.

Metaprogramming is, according to Cordy and Shukla [1], the process of specifying generic software source templates from which classes of software components, or parts thereof, can be automatically instantiated to produce new software components. The source templates used in the Scripting model are called metascripts. Metascripts are code fragments (or scripts) which have to be completed by elements of application specification and other metascripts by an automatized process of generation.

After OOP languages, the scripting languages are also introduced into Generative programming, as announced by Sells [18]. The advantages of the scripting languages should be reached by avoidance of certain weaknesses of the Object-oriented programming, which are, according to Ousterhout [16], rigidity of the object model, high level of typing and the need of the translation/compilation phase during the program development. There are some scripting language characteristics that could be useful for Generative programming: a scripting language abilities in the character strings processing [13][19], the possibilities of connecting completed components written in the target programming languages [19] and flexibility of the scripting languages syntax which arise from a low typing level [16][19]. Except of using existing scripting languages, some projects on making specialized scripting languages aimed at Generative programming were conceived, such as Open Promol [19] and CodeWorker [13].

The idea of the Scripting model is to offer model of generator with good properties of scripting languages, primarily to avoid the rigidity of object model. Complex classes, which are used as connection points in Join point model [9], are replaced with simple links which are type-less. A program code is assembled from fragments of source code called metascripts. On the other hand, introducing properties which are characteristic to object model, like encapsulation, inheritance, and polymorphism, could be interesting to achieve higher level of model elements reusability (so less complex model) and easier upgrading of generator.

The Scripting model is a static model of a generator which does not suggest any particular technology (component or other) used in its implementation, unlike some other Generative programming based projects, like Uniframe [21].

Consequently, there is a difference in relation to metaclass-based approaches, as described by Grigorenko et al. [7] Tolvanen and Rossi [20] and De Lara and Vangheluwe [3]. Metascripts are not strictly tied to particular classes and other program units to be generated. Their purpose is to implement some crosscutting concern in all needed parts of the program. Another important difference between the object model and the Scripting model is that the latter is based on the dealing with crosscutting concerns, which is recognized as a problem within the object model by Kühl [11] and Lee [12]. Furthermore, an object model is aimed to define the individual application, while a Scripting model defines the application generator for a proposed problem domain.

### 3. Scripting Model Definition

Scripting model is oriented to produce software variants i.e. building Software Product Lines (SPL). According to the Scripting model, the generator is software that produces a source code of a one particular application from a family of possible applications. Scripting model defines generator as a multi-level structure (Figure 1) consisting of three kinds of elements: application Specification (**S**), Metascripts (**M**; code templates in Scripting model) and Links (**L**), where links are base elements containing all lower-level structures. Each level starts with its base Metascript and defines one particular generator. For each particular application the Scripting model is defined by this 6-tuple:

$$\mathbf{SM} = \{\mathbf{S}, \mathbf{M}, \mathbf{L}, \mathbf{P}, \mathbf{m}_0, \mathbf{g}\}.$$

where

**S** is a whole specification (of particular application to be generated),

**M** is a set of metascripts (program code templates in Scripting model),

$$\mathbf{M} = \{\mathbf{m}_0, \mathbf{m}_1, \dots, \mathbf{m}_m\}$$

**L** is a set of links (each link occurs  $0..N$  times in  $0..M$  metascripts). The link is a user defined mark, which is replaced with the source code in the process of generation.

$$\mathbf{L} = \{\mathbf{l}_1, \mathbf{l}_2, \dots, \mathbf{l}_n\}$$

**P** is a generated program (source code)

$\mathbf{m}_0$  is a highest-level metascript (base of the appropriate generator)

**g** is a function that produces source code (generator)

Generated program is a function of Specification (**S**), Links (**L**) and highest level metascript  $\mathbf{m}_0$ :

$$\mathbf{P} = \mathbf{g}(\mathbf{S}, \mathbf{L}, \mathbf{m}_0)$$

Generator (**g**) is a recursive function. Each particular metascript ( $\mathbf{m}_i$ ) can contain fragments of a program code (**p**) and subset of links ( $L_{mi}$ ), from all possible links in **L**, toward lower-level structures:

$$\mathbf{m}_i = \{\mathbf{p}, L_{mi}\}$$

**p** are fragments of the program code (to be included in generated source code)

$L_{mi}$  is a set of links (connections toward lower-level structures) in this particular metascript,

$$L_{mi} = \{\mathbf{l}_0, \mathbf{l}_1, \dots, \mathbf{l}_k\}.$$

Each link ( $l_i \in L_{mi}$ ) in metascripts  $\mathbf{m}_i$  is replaced in the process of generation with one of the following:

- appropriate three elements groups  $\{\mathbf{S}, L_{mj}, \mathbf{m}_j\}$ , which can be observed as a new generator or
- a fragment of program code (**p**) from metascript  $\mathbf{m}_j$ .

Higher-level generator is given as a superposition of lower-level generators (Figure 1). Particular links can be used more times, in different templates, enabling reuse of whole connected sub-levels.

#### 3.1. Graphic representation of Scripting model

The graphic representation of Scripting model shows the structure of application specification, by the Specification diagram, and configuration of generator, by the Metascripts diagram.

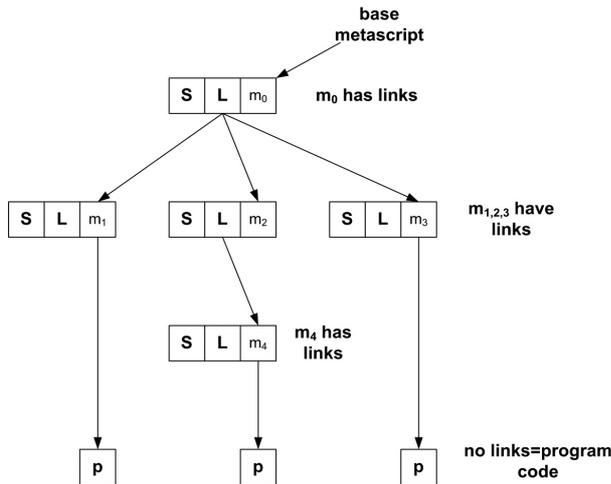


Figure 1. Generator as a multi-level structure.

### 3.1.1. Specification diagram

The Specification diagram gives the structure of application specification in the form of a tree-like feature model (similar approach: Limbourg and Kochs [14]). The application specification consists of attributes, defining particular features of application to be generated within its problem domain and their values, as shown in Figure 2. All attributes are optional, so they can be omitted or replicated as many times as required.

```

<att 1>      : <value 1>
<att 1.1>   : <value 1.1>
<att 1.2>   : <value 1.2>
<att 1.n>   : <value 1.n>
<att n>     : <value n>
...
...
<att n.1>   : <value n.1>
<att n.2>   : <value n.2>
<att n.n>   : <value n.n>
    
```

Figure 2. Application specification.

An example of the concrete application specification is shown later in Figure 12.

The hierarchy of the attributes is given by the Specification diagram (Figure 3). Higher-level attributes are containers for the lower-level attributes.

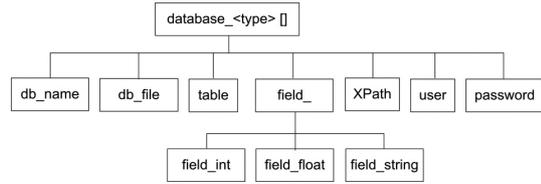


Figure 3. Example of the Specification diagram.

The attribute *database\_<type>* is a container for all other specification attributes. It can be repeated an arbitrary number of times. The attributes *db\_name*, *db\_file*, *table*, *XPath*, *user* and *password* are optional and can appear only once per *database\_<type>*, while the attributes from the group *field\_* (*field\_int*, *field\_float* and *field\_string*) can appear an arbitrary number of times. As shown in Figure 2, containers are marked by square brackets, or end by ‘\_’ sign, which is used for groups e.g. *field\_int*, *field\_float* and *field\_string* could be treated in both ways: separately and as members of group *field\_*. This diagram is used later to describe example application in subsection 6.1.

### 3.1.2. Metascripts diagram

The Metascripts diagram defines generator configuration, i.e. connections between application specification and metascripts (code templates in Scripting model). Specification attributes, given in the Specification diagram, represent *sources* in the Metascripts diagram, while *links* represent replacing tags in metascripts.

There are three basic elements used in the Metascripts diagram (Figure 4):

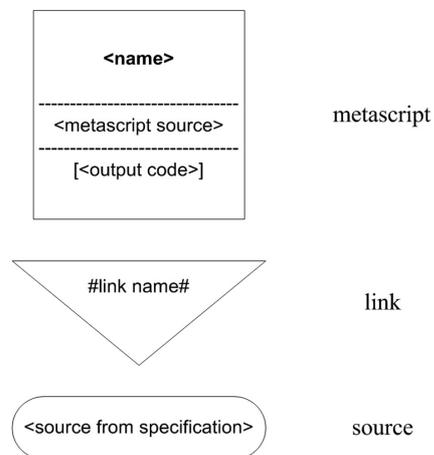


Figure 4. Elements of the Metascripts diagram.

A *metascript* is a template used in generation of its implementation – a program code in the target programming language. It is represented by a rectangle. The element contains data such as: *metascript* name, *metascript* source (filename or the name of another used source), and output code (filename or name of another output which contains the generated program code). *Metascript* contain replacing tags, called *links*, in form of ‘#’ marks. *Links* physically belong to their *metascripts*, but it is shown outside. A *link* connects the metascript(s) with the feature source from the application specification and (optionally) to the lower level metascript(s). It is represented by a triangle with one corner oriented downwards. A link is typeless and has only its name. During generation process, the *links* inside *metascripts* are replaced with the data or a program code in either of the two possible ways. The first way is the direct replacement of a *link* with the *source* data (if there are no connected lower-level *metascripts*). The second way is the replacement of a *link* with a lower-level *metascript*. It is important that the *links* inside a particular metascript could appear more than once. All appearances are shown by a single triangle in the Metascripts diagram. The *sources* can be defined as containers. In that case, their further use must be defined on the lower levels of the Metascripts diagram.

The example of the Metascripts diagram is shown in Figure 5. Metascript *Main* has link to metascript *Database*. Metascript *Database* has links to metascripts *Fields*, *Table\_fields*, *Oracle\_fields*, and *XML\_fields*.

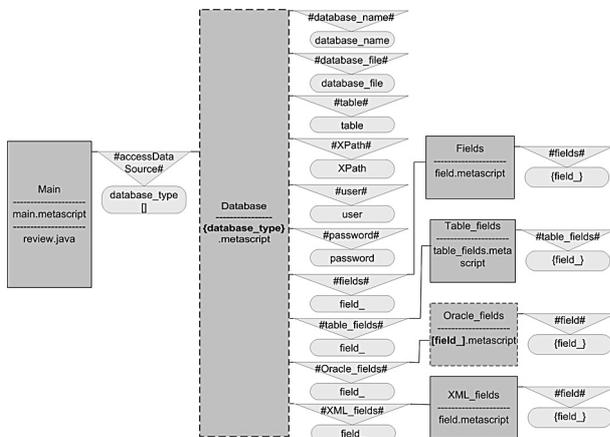


Figure 5. Example of the Metascripts diagram.

Dashed line represents *virtual metascript* (metascript subjected to late binding process), as described in Section 5. This diagram is used later to describe example application in subsection 6.2.

#### 4. Implementation of the Object Model Properties in the Scripting Model

Object-oriented programming uses encapsulation, inheritance and polymorphism as its basic concepts. Similar concepts are used in Scripting model at generation level. These object-oriented features are applied in the context of generation of the source code regardless of the target programming language.

Implementation of some features that are characteristic for object-oriented programming could be interesting in achieving better features of the generator system, like higher reusability of model elements, more precise application specification and simpler model of particular generator.

The Scripting model, as introduced in [17], includes basic concepts of encapsulation and inheritance. These two features of the Scripting model are shortly described in next two subsections.

##### 4.1. Encapsulation

The basic unit of encapsulation in the Scripting model of generator is a *metascript*. The metascript encapsulates a program code together with links (replacing marks). A program code of the metascript is static and serves as interface for combining with other metascripts. Links in metascripts can be observed as full implementation, which depends on application specification. Unlike a class, the *metascript* is a dynamic structure, which means that its instances (*scripts*) share only a basic structure and can be sized differently (Figure 6).

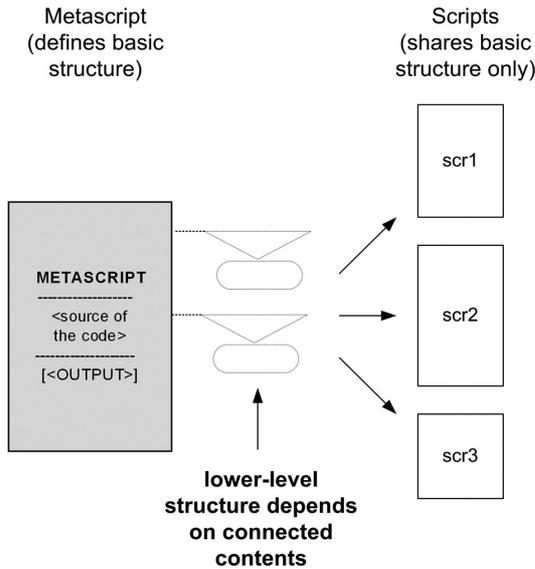


Figure 6. Metascript and its instances.

The structure of *scripts* depends on its *metascripts* and their connected contents (the *sources* from a specification and, possibly, the inherited *metascripts*).

### 4.2. Inheritance

Inheritance in the Scripting model occurs when a base *metascript* inherits the lower-level *metascripts*. The lower-level *metascripts* in context of Scripting model are parents regarding the base *metascript*. In Figure 7 *metascripts* A and B are parents of *metascript* Base. The inheritance is selective and inclusion of every particular parent *metascript* depends on the existence of the appropriate *source* from the program specification (Sources S1 and S2 in the example in Figure 7).

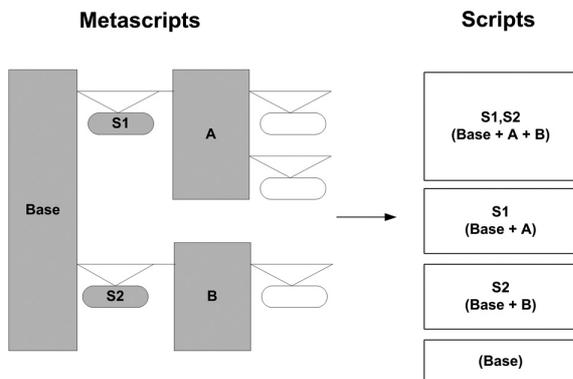


Figure 7. Inheritance in a Scripting model.

The Scripting model inheritance mechanism differs from standard function call mechanism (used in structural programming and OOP) in several ways: in the level of inheritance, in selective inheritance, and using of application specification values instead of function parameters. Function calls exist at application runtime level and are used for process sharing. Inheritance in the Scripting model exists at the generation level and is oriented on static structure of generated application. The selective inheritance is a base for achieving optimization (in relation to generic approaches): final application could use only a subset of available *metascripts*, depending on its specification.

### 4.3. Polymorphism in the Scripting model

Polymorphism in the Scripting model is based on the use of the *virtual metascripts*. These *metascripts* are invoked by the mechanism of late binding during the program generation (according to the program specification, as described further; Figure 8).

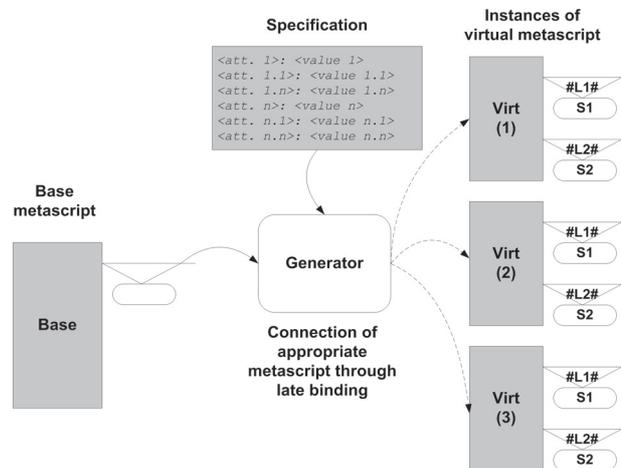
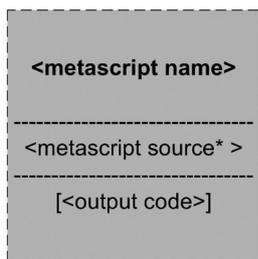


Figure 8. The principle of using virtual metascripts.

Polymorphism extends the Scripting model by providing possibility that all *metascripts* sources do not have to be known before application generation. A name of the *metascript* is determined by the values from application specification during the process of generation. This allows later addition of new functionalities by introduction of new *metascripts*.

The advantage of introducing polymorphism is expandability in a way that does not touch the model of generator and its implementation. Addition of new functionality by using polymorphism requires specifying new values in the application specification and adding the appropriate *metascripts*.

The *virtual metascript* is represented by a dashed line rectangle (Figure 9).



\* - contains a variable part

Figure 9. Virtual metascript.

A *metascript* source contains a variable part marked by the square brackets. The following example shows the usage of attribute name for invocation of appropriate *virtual metascript*:

*create\_[field\_].metascript*

Figure 10 shows such definition of *virtual metascript*.

Figure 11 shows how invocation of *virtual metascripts* works. Specification defines the usage of attribute *field\_char*. During the process of generation, generator uses this value to create name of metascript *create\_char.metascript* and then invokes it.

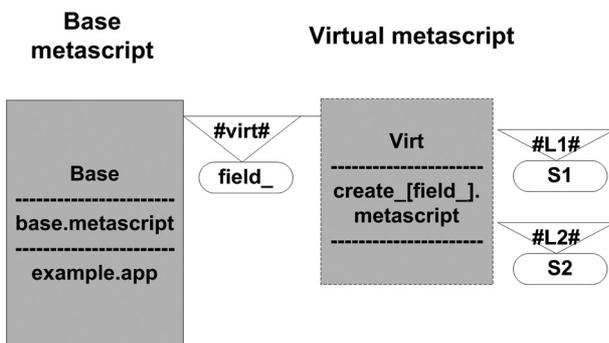


Figure 10. An example of virtual metascript definition.

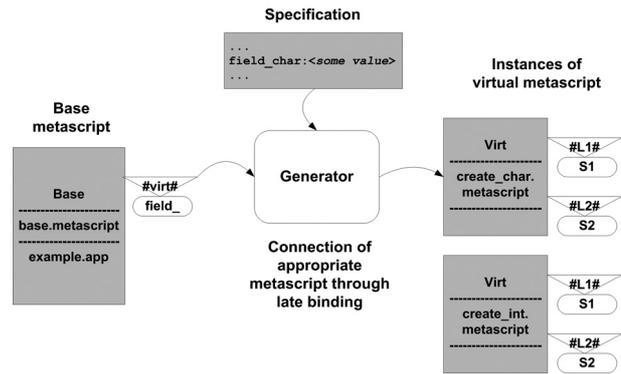


Figure 11. An example of virtual metascript invocation.

In some other cases the real *metascript* source could be e.g. *create\_field\_int.metascript*, depending on the specification attributes used.

### 5. Open Implementation Issues

The upgraded Scripting model of generator defines the way of building and documenting of generator by its specification (represented by the Specification diagram), configuration (represented by the Metascripts diagram) and set of *metascripts* (code templates within the Scripting model). Some issues regarding checking consistency that concern implementations of the model still remain for future work. There are three kinds of inconsistencies that could happen in implementation of the model: the inconsistent syntax of the model, syntactic incorrectness of the generated code and logical incorrectness of the generated code.

**Inconsistent syntax of the model** could be, in the simplest form, a result of using unsupported specification attributes or disregarding of the specification hierarchy. Referencing of non-existing *metascripts* could be a circumstance of polymorphism if some of the *virtual metascripts* have not their implementation. It's possible to use replacing *metascripts*, or to stop the generation process. Dealing with links in generator configuration could cause two kinds of errors:

- usage of connections that don't appear in used metascripts. Could be wrong connection or redundant generator configuration,
- usage of connections in metascript which are not supported in generator configuration. This could result in links (usually in '#')

signs) remaining in generated code resulting in syntax and/or logical errors.

**Syntactic incorrectness of the generated code** often comes from insufficient specification, where some necessary attribute values are not specified. This results in links remaining in generated code. Usage of unsafe names in metascripts (variables, functions, classes etc.) could cause the collision with attribute values in specification. Some programming languages require that e.g. functions have to be defined prior to their calls. Order of specification attributes could cause breaking of that rule.

**Logical incorrectness of the generated code** could be violated by usage of unsafe names or by links remaining in generated code (if these do not cause the syntactic incorrectness) or by breaking program restrictions. For example, exceeding of size limits for attribute values could cause improper work or instability of generated applications.

## 6. Illustrative Example

The problem domain of an illustrative example is a small Java application for reviewing data from different data storages (Oracle database and XML). The generator uses attribute names for handling different data types by invocation of *virtual metascripts*.

### 6.1. Specification of the example application

The Specification diagram of the example application is presented in Figure 3 and described in subsection 3.1.1. It defines main parameters of the used data storage and the used fields together with their types. The appropriate specification is shown in Figure 12.

```
package hr.foi.gp;
// import packages
public class DataAccessObject{
    public static void showStudentDetails() throws Exception{
        #accessDataSource#
    }
}
```

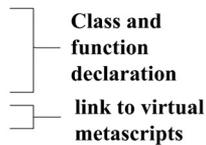


Figure 13. The highest level metascript.

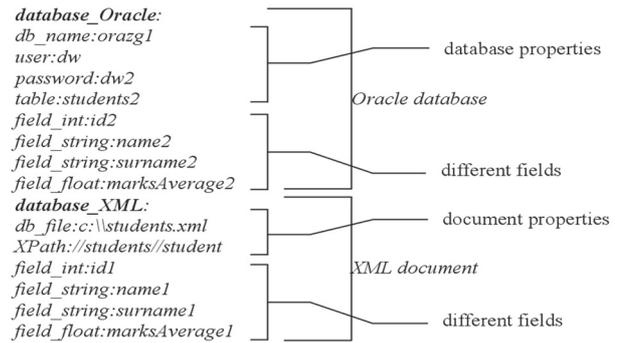


Figure 12. The specification of the example application.

The specification consists of two parts: the first part defines the structure of *Oracle* database, and the second part defines XML data structure. Attribute group 'database\_' is connected to virtual metascript marked as 'database\_[database\_type].metascript'. The real metascript (Oracle or XML) is attached during the process of generation.

### 6.2. Working with the metascripts

The example generator generates only one program file, which has the part for handling appropriate data storage. The Metascript diagram of the example application is presented in Figure 5 and described in subsection 3.1.2.

Hence the main *metascript* contains the parts common to all generated applications, including its general structure:

The link `#accessDataSource#`, from Figure 13, is replaced by a program code in the process of generation. The virtual metascript *Database* is used for generating a code and it has two instances (for Oracle database and for XML document). The metascript *Oracle\_fields* for generating the *Oracle* code that handles different data types is also virtual because of the three field

types used (*field\_int*, *field\_float* and *field\_string*; Figure 5).

Some of the links on *Database* virtual metascript are not common to all instances of the *virtual metascript*. Therefore, they are ignored in these instances. Finally, the generated code for the example specification is shown in Figure 14.

Figure 14 shows the source code which is used for access data about students from two different data sources. The first data source is table in *Oracle* database and the second data source is *XML* file in file system. The source code used for accessing these data sources differs significantly because data sources are of different kinds.

The source code for each data source comes from separate *metascripts*. The element *database\_<type>* of application specification, as shown in Figure 12, specifies which metascript

file is used. The bolded words in source code come from application specification presented in Figure 12.

Because some fields in Oracle table are of different types, different source code should be generated. That is the reason why *Oracle.fields.metascript* is *virtual metascript* which can generate source code for different data types depending on application specification, e.g. type *int* and type *String* in Figure 14.

The main benefits of using polymorphism in this example are simpler application specification and simpler extension of the generator.

The application specification (shown in Figure 12) is simplified by usage of virtual metascript (*[database].metascript*). This enables usage of same specification attributes (e.g. *field\_int*, *field\_string* etc.) after the type of database is

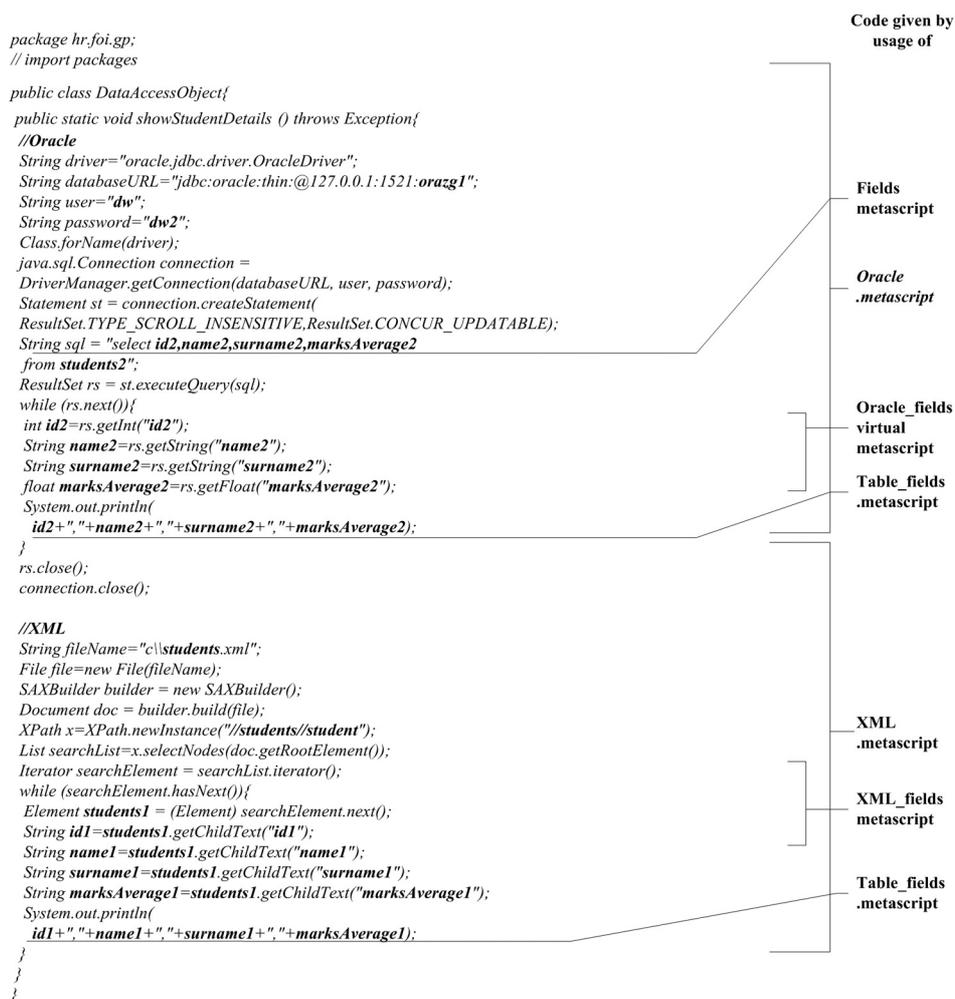


Figure 14. Generated source code.

specified (*database\_Oracle* or *database\_XML*).

Addition of new type of data source requires building a new metascript for this type and addition of appropriate elements into application specification.

This and some other examples are available<sup>1</sup>.

## 7. Conclusion

This paper gives formal definition of the Scripting model and introduces polymorphic features into the Scripting model. Scripting model has preserved all of previously obtained features, such as aspect orientation and type-less join points. The paper shows that the basic concepts of the Object-oriented programming, e.g. encapsulation, inheritance and, finally, polymorphism can be implemented within Scripting model of generator.

The main goals of introducing polymorphic features in the Scripting model are: a more precise application specification, a better reusability of generator elements, and a simpler generator model and its implementation. A more precise application specification is enabled by variety of metascripts under the same name that are subjected to late binding. The problem domain covered by a generator can be extended by usage of virtual metascripts and change of a generator aimed at introducing new program templates is not always necessary. The virtual metascript covers more real metascripts, which reduces the number of metascripts in the Metascripts diagram.

The advantage of introducing polymorphism is expandability in a way that does not touch the model of generator and its implementation. Addition of new functionality by using polymorphism requires specifying new values in the application specification and adding the appropriate *metascripts*.

In our future work we plan to focus on the problems of checking consistency of the model implementation.

## References

- [1] J. R. CORDY, M. SHUKLA, Practical Metaprogramming. *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research*, (November 09–12, 1992), Toronto, Ontario, Canada.
- [2] K. CZARNECKI, U. W. EISENECKER, *Generative Programming: Methods, Tools, and Applications*, Addison Wesley, 2000.
- [3] J. DE LARA, H. VANGHELuwe, Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages and Computing*, Vol. 15 (2004), pp. 309–330.
- [4] A. V. DEURSEN, P. KLINT, Domain-specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology (CIT)*, Vol. 10, No. 1 (2002).
- [5] U. EISENECKER, *Generative Programming: Beyond Generic Programming. Proc. Dagstuhl Seminar on Generic Programming*, (April 27–May 1, 1998), Schloß Dagstuhl, Wadern, Germany.
- [6] J. GRAY, Y. LIN, J. ZHANG, Levels of Independence in Aspect-oriented Modelling. Available on: <http://www.gray-area.org/Pubs/middle-ware-2003.pdf>, 2003.
- [7] P. GRIGORENKO, A. SAABAS, E. TYUGU, Visual Tool for Generative Programming. *Proc. of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13)*, ACM Publ., (2005), pp. 249–252.
- [8] R. GUERRAOUI, *Strategic directions in object-oriented programming*. ACM Computing Surveys, Baltimore, December 1996.
- [9] M. M. KANDÉ, J. KIENZLE, A. STROHMEIER, From AOP to UML – A Bottom-Up Approach. *1st International Conference on Aspect-oriented Software Development*, (2002), Enschede, The Netherlands.
- [10] G. KICZALES, J. LAMPING, A. MENDHEKAR, C. MAEDA, C. VIDEIRA LOPES, J.-M. LOINGTIER, J. IRWIN, Aspect-oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, (June 1997), Springer-Verlag LNCS 1241, Finland.
- [11] D. KÜHL, STL and OO Don't Easily Mix. *Proceedings of the GSCE, Workshop on C++ Template Programming*, Erfurt 2000, Available on: <http://www.oonumerics.org/tmpw00/kuehl.html>, 2000.
- [12] K. W. K. LEE, An Introduction to Aspect-oriented Programming. *COMP610E: Course of Software Development of E-Business Applications*, (Spring 2002), Hong Kong University of Science and Technology.

<sup>1</sup> <http://barok.foi.hr/~darados/smg/>

- [13] C. LEMAIRE, *CODEWORKER Parsing tool and Code generator – User’s guide & Reference manual*, (Release 4.4.), <http://codeworker.free.fr/CodeWorker.pdf>, 2007.
- [14] P. LIMBOURG, H. D. KOCHS, Multi-objective optimization of generalized reliability design problems using feature models – A concept for early design stages. *Reliability Engineering & System Safety*, Volume 93, Issue 6 (2008), pp. 815–828.
- [15] I. MAGDALENIĆ, D. RADOŠEVIĆ, Z. SKOČIR, Dynamic Generation of Web Services for Data Retrieval Using Ontology, *INFORMATICA*, Vol. 20, No. 3 (2009), pp. 397–416, ISSN 0868-4952, Institute of Mathematics and Informatics, Vilnius, Lithuania.
- [16] J. K. OUSTERHOUT, Scripting: Higher Level programming for the 21st Century, *IEEE computer magazine*, (March 1998).
- [17] D. RADOŠEVIĆ, B. KLIČEK, J. DOBŠA, *Generative Development Using Scripting Model of Application Generator*. DAAAM International Scientific Book 2006, DAAAM International, Vienna, Austria 2006.
- [18] C. SELLS, Generative programming: Modern Techniques to Automate Repetitive Programming Tasks. *MSDN Magazine*, (December 2001). Available on: <http://msdn.microsoft.com/msdnmag/issues/01/12/GenProg/GenProg.asp>, 2001.
- [19] V. ŠTUIKYS, R. DAMAŠEVIČIUS, G. ZIBERKAS, *Open PROMOL: An Experimental Language for Target Program Modification*, Software Engineering Department, Kaunas University of Technology, Kaunas, Lithuania, 2001.
- [20] J. P. TOLVANEN, M. ROSSI, *Metaedit+: Defining and using domain-specific modeling languages and code generators*. In OOPSLA 2003 demonstration, 2003.
- [21] UNIFRAME WEB SITE. Available on: <http://www.cs.iupui.edu/uniFrame/>, last accessed 05-13-2008.

Received: October, 2009  
Revised: December, 2010  
Accepted: February, 2011

Contact addresses:

Danijel Radošević  
Faculty of Organization and Informatics  
University of Zagreb  
Pavlinska 2  
42000 Varaždin  
Croatia  
e-mail: danijel.radosevic@foi.hr

Ivan Magdalenic  
Faculty of Organization and Informatics  
University of Zagreb  
Pavlinska 2  
42000 Varaždin  
Croatia  
e-mail: ivan.magdalenic@foi.hr

---

DANIJEL RADOŠEVIĆ, PhD, is an associate professor at the University of Zagreb, Faculty of Organization and Informatics. He teaches at different programming courses. His research focuses on programming languages, generative programming and educational software.

---



---

IVAN MAGDALENIĆ, PhD, is an assistant at the University of Zagreb, Faculty of Organization and Informatics in Varaždin. His research interests are in e-business, web technology, semantic web technology and generative programming. He has been involved in several projects of e-business adoption in Croatia. He is a member of National Council for e-business.

---