Upgrading Generator Scripting Model by Object Model Properties

Radošević, Danijel; Kliček, Božidar; Kozina, Melita

Abstract— *Generative Programming (GP) is a relatively new discipline of automatic programming, with the accent on program code optimization and flexibility in parallel development of generators and final applications. The Generator Scripting Model (GSM) is an aspect-based, graphic model of application generator. It is a static model based on higher level of scripts, called metascripts, unlike UML, which is based on classes and their processes. However, the basic concepts of object oriented programming, like encapsulation, inheritance, and polymorphism could be achieved in GSM. Benefits of introducing such concepts into GSM are: more precise application specification, more reusability of generator, simpler generator model and its easier implementation. The upgraded GSM is actually implemented in a form of a C++ library.*

Index Terms— *Generative Programming (GP), Generator Scripting Model (GSM), Object Oriented Programming (OOP), polymorphism, UML*

1. INTRODUCTION

GENERATIVE Programming (GP)[1] is a discipline of automatic programming which assumed that name in the late 1990's. According to the definition, GP represents "...designing and implementing software modules which can be combined to generate specialized and highly optimized systems fulfilling specific requirements" [2]. The main aspirations of GP are, in relation to other techniques of automatic programming, programming code optimization as well as making the process of building generators and final applications more flexible.

The Generator Scripting Model (GSM) was introduced by Radošević[8] and enables graphic way of modeling generators. GSM is oriented on modeling aspects i.e. features not strictly connected to individual program organizational units like functions or classes. Therefore, the model can appear within different application parts [6].

Manuscript received Juni 29, 2008.

D. Radosevic is with the Faculty of Organization and Informatics, University of Zagreb, Croatia

B. Klicek is with the Faculty of Organization and Informatics, University of Zagreb, Croatia

M. Kozina is with the Faculty of Organization and Informatics, University of Zagreb, Croatia

Concepts inherited from Object Oriented Programming (OOP) like encapsulation and inheritance are included in the basic GSM, which is now upgraded by concept of polymorphism. Inside GSM, polymorphism concerns late binding of metaprograms during the process of generation, depending on program specifications. Benefits of that concept should be: easier generator (because of lower number of generation levels), more precise application specification and more flexibility in generator development.

GSM (now including polymorphic features) is actually implemented in a form of C++ library [11]. Some older generators are written in Perl.

2. GP AS A POSSIBLE SUCCESSOR OF OBJECT ORIENTED PROGRAMMING

The idea of this paper is to offer GP as a possible successor of today's dominant programming paradigm - OOP. Some notion for it was given by Guerray, who marked Aspect Oriented Programming (AOP) as possible successor of object paradigm [3]. AOP is a discipline in base of GP, and there some aspect oriented varieties of standard programming languages, like AspectJ, AspectC++ etc.

To be a successor of OOP, GP should inherit the main concepts of OOP (encapsulation, inheritance and polymorphism) in a way the OOP inherits main concepts of structured programming (Figure 1).

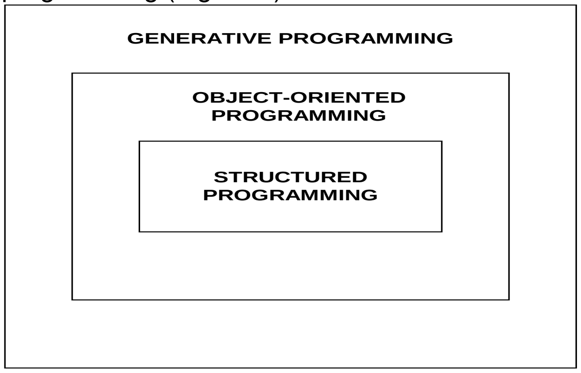


Figure 1: Relationship between structural programming, OOP and GP

GSM has been originally developed for the needs of GP based on scripting languages [10]. It's a connection model which defines the involvement of specific features (i.e. aspects) given inside program specification into programming code which should be generated. In terms of AOP, it is a type of a Join Point Model [4]. The specific difference to other Join Point Models is that the scripting model (similarly to scripting programming languages) is not based on types, i.e. it represents a type-free system. Thus, connecting points in the scripting model do not represent classes and their objects, only type-free connections between metaprograms and properties defined in program specification [10].

3. BASICS OF GSM

GSM is a graphic diagram which starts from the general (static) model of a generator (Figure 2), where that generator binds the program/component specification to metaprograms, producing final programming code, as described in [1].

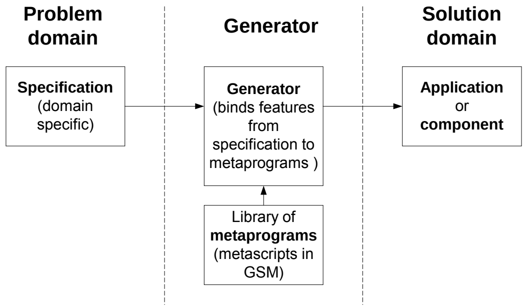

Problem domain
Generator
Solution domain
Specification (domain specific)
Generator (binds features from specification to metaprograms)
Application or component
Library of metaprograms (metascripts in GSM)

Figure 2: The general model of generator

GSM consists of two graphic diagrams:

- **The specification diagram** - defines the structure of application specification. Specification defines proposed properties (aspects) of the future application, which will be used in application generation.
- **The metascripts diagram** - defines distribution of specific properties, which are proposed by specification diagram, within application, and also their connecting to program code templates (called *metascripts inside GSM*).

The specification diagram is a hierarchic diagram defining proposed application features in a form of tree-like feature model, similar to the model given by Limbourg and Kochs [7]. Features are defined by attributes which are used for specifying application from the generator's domain. Attributes of specification have hierarchic structure, as shown on Figure 3.

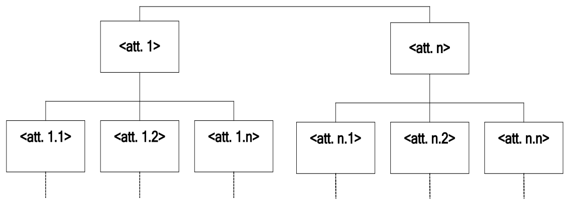


Figure 3: The specification diagram

The metascripts diagram defines distribution of specific properties proposed by the specification diagram within the application, and also their connection to program code templates (metascripts). It consists of three basic elements (Figure 4):

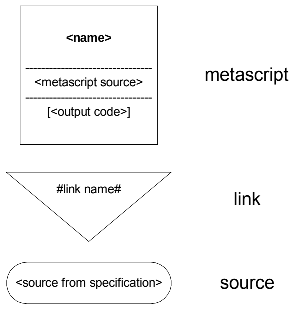


Figure 4: Elements of the metascripts diagram

Metascript is a template used in generation of its implementation - program code in target programming language, which is called *script*.

Link connects metascripts to property sources from application specification, and (optionally) to lower level metascripts.

Source represents a feature defined by appropriate attribute in the specification diagram. Sources can be defined as containers - in that case their further use must be defined on lower levels of the metascripts diagram.

Elements of the metascripts diagram are located in vertical columns. Each of the columns represents an appropriate level of metascripts, as shown on Figure 5:

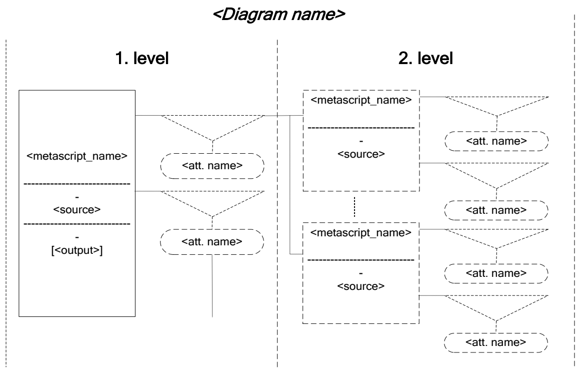


Figure 5: Arrangement of elements on the metascripts diagram

Number of levels in a metascripts diagram determines the level of the generator; therefore, the levels of generators in scripting model are:

- ***Single-level generator*** - simple generator consisting of only one metascript (its metascripts diagram has only one level; Figure 6) and

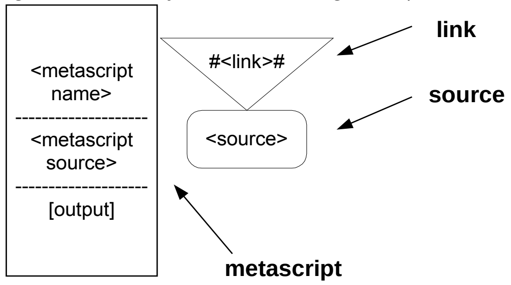


Figure 6: Single-level generator

- ***Multi-level generator*** - generator that includes more metascripts arranged on different levels (its metascripts diagram has more than one level). Each branch in the metascripts diagram defines the appropriate lower level generator. The lowermost level of each branch in the metascripts diagram defines one simple single-level generator. Multi-level generator is given by superposition of multiple one-level generators (Figure 7).

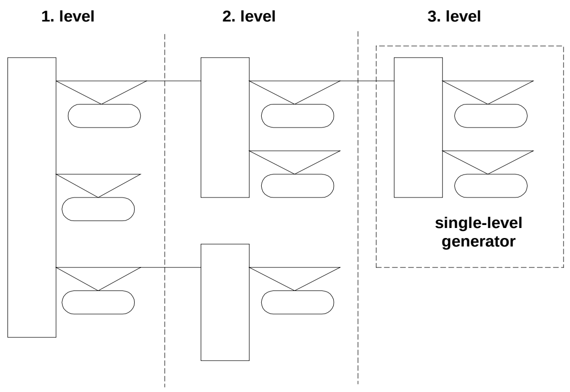


Figure 7. Multi-level generator is given by superposition of multiple one-level generators

4. MODELING LEVELS OF GSM IN RELATION TO UML

Architecture of software system development supported by UML diagram techniques is shown on Figure 8.

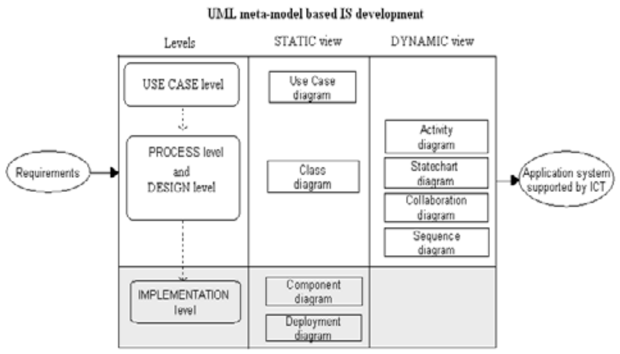


Figure 8: Architecture of software system development supported by UML diagram techniques [4]

According to this architecture, we recognize the following levels: the use case level; the processing level, the design level and the implementation level [4].

GSM defines only the static view, and includes two diagrams on two levels: the specification level and the processing level. The dynamic view is a part of programming code templates – (metascripts; Figure 9).

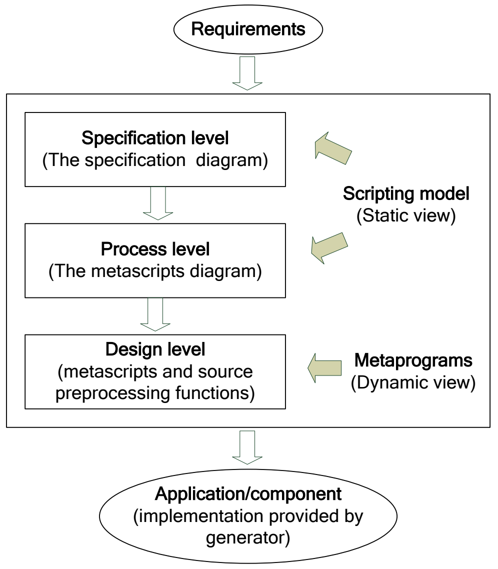

Requirements
Specification level
(The specification diagram)
Process level
(The metascripts diagram)
Design level
(metascripts and source preprocessing functions)
Scripting model
(Static view)
Metaprograms
(Dynamic view)
Application/component
(implementation provided by generator)

Figure 9: Levels of GSM architecture [9]

The paper especially emphasizes the defining of relationships between the generator scripting model and base levels of the UML model (Use Case and Process).

1. OBJECT MODEL PROPERTIES IN GSM

1.1. Encapsulation

Inside GSM, the basic encapsulated unit is the metascript. The metascript is, despite class, just a basic structure, which means that its instances (called *scripts*) share only the basic (higher level) structure, and could be of different sizes (Figure 10):

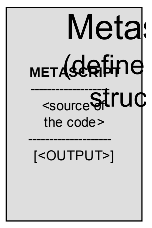


Figure 10: Metascript and instantiated scripts

1.2. Inheritance

Inheritance in the generator scripting model works in a way that base metascript inherits lower-level metascripts. Inheritance is selective and inclusion of a particular metascript depends on the existence of the appropriate program specification source (Figure 11):

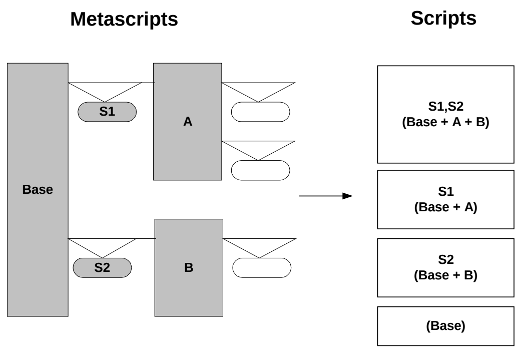


Figure 11: Inheritance in GSM

1.3. Polymorphism

Polymorphism inside generator scripting model is based on usage of *virtual metascripts*. Virtual metascripts are invoked by a mechanism of late binding during the process of program generation (according to program specification, as further described; Figure 12).

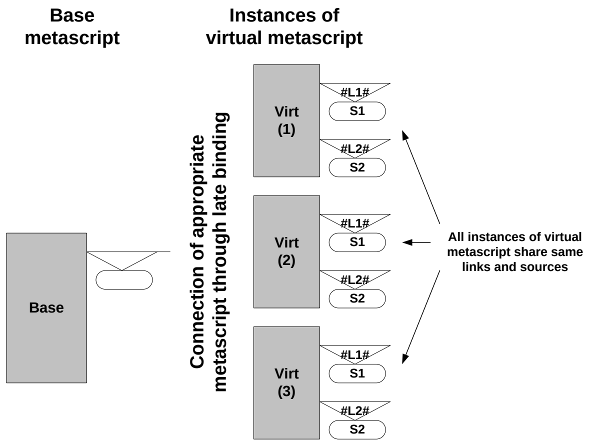
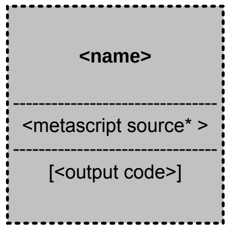

Base metascript
Instances of virtual metascript
Connection of appropriate metascript through late binding
Base
Virt (1)
Virt (2)
Virt (3)
#L1#
S1
#L2#
S2
All instances of virtual metascript share same links and sources

Figure 12. Instances of virtual metascript

As shown on Figure 12, all instances of virtual metascript share same links and connected sources, which is analogue to the object model, where subordinated classes share same virtual methods. Virtual metascript is represented by the dot line rectangle (Figure 13):



***** - contains variable part

Figure 13. Virtual metascript

Metascript source (usually a text file) contains a variable part marked by brackets (square or curly; depending on the type of invocation). This variable part is defined by appropriate specification source, as described below.

Invocation of virtual metascripts
Program specification consists of attribute-value pairs, meaning there are two invocation methods of virtual metascripts:
- By attribute and
- By value.

Invocation *'by attribute'* is marked by square brackets:

<prefix>[<attribute group>]<suffix>

For example (Figure 14), if metascript instance should be invoked according the attribute from the group *field_*, it should be specified as follows:

create_[field_].metascript

For the specification line

field_float:<some value>

which will produce the invocation of the metascript *create_field_float.metascript* .

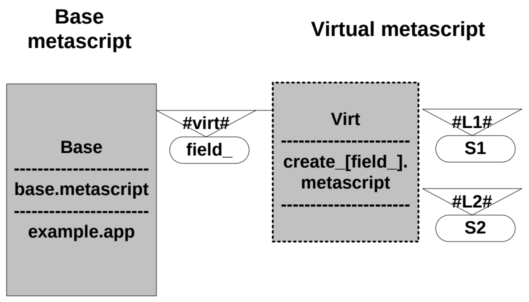


Figure 14. Example of invocation *by attribute*

The source *field_* is (on link *#virt#*) is a container source. The value inside the brackets has to be inside its source container.

Invocation *'by attribute*' is marked by curly brackets:

<prefix>{<attribute>}<suffix>

For example (Figure 15), if metascript instance should be invoked according the value of attribute *review*, it should be specified as follows:

show_{review}.metascript

For the specification line (specifies type of review)

review:combo

which produces the invocation of the metascript *show_combo.metascript* .

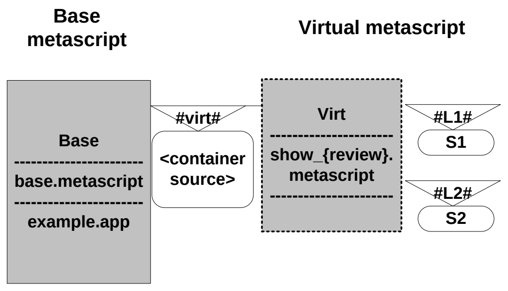


Figure 16. Example of invocation by value

Similar to previous type, the value inside brackets has to be inside its source container.

2. Illustrative Example: The Scripting Model of PHP Web Application Generator

Formation of a generator starts from the *application prototype*, i.e. from a possible application that could be made by the generator. Furthermore, the application specification should define all features differing the particular (generated) application from its prototype. This illustrative example contains a database table *students*, which should be managed by php scripts and html forms (Table 1).

Attributes	Data types
student_id (primary key)	integer
surname_name	varchar
year_of_study	integer
year_of_enrolment	integer

Table 1: Structure of prototype database table

According to Table 1, the specification of a prototype could look like this:

title:students
table:students
primary_key:student_id
field_char:surname_name
field_int:year_of_study
field_int:year_of_enrolment

Appropriate specification diagram is given in Figure 17:

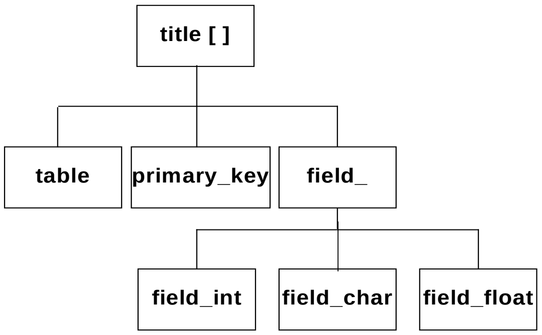
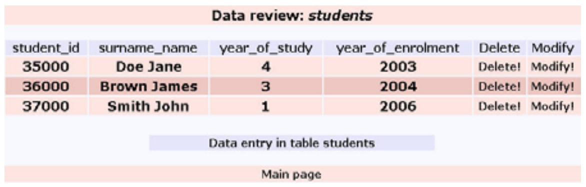


Figure 17: The specification diagram of the example generator

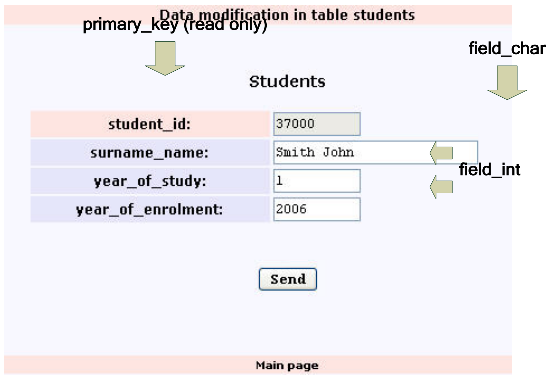
Features from the specification are dispersed through different parts of program code, enabling basic operations like data review (Figure 18) and existing record modification (Figure 19):



Data review: *students*

student_id	surname_name	year_of_study	year_of_enrolment	Delete	Modify
35000	Doe Jane	4	2003	Delete!	Modify!
36000	Brown James	3	2004	Delete!	Modify!
37000	Smith John	1	2006	Delete!	Modify!

Figure 18: Data modification



Data modification in table students

Students

student_id:	37000
surname_name:	Smith John
year_of_study:	1
year_of_enrolment:	2006

Figure 19: Existing record modification

As can be seen on Figure 18 and Figure 19, features from the specification are dispersed through different parts of the application, representing aspects. Those features are handled differently depending on metaprograms used, as defined in the metascripts diagram (Figure 20.).

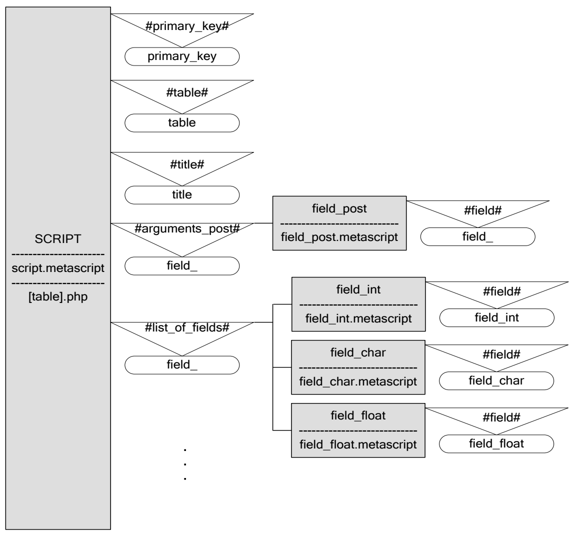


Figure 20: The metascript diagram of the example generator

3. Conclusion

GP, as a relatively new discipline of automatic programming, strives to make the application development process more flexible through the usage of generators as a knowledge base for development of new applications. GSM is a generative model that enables a graphic means of modeling generators, not particular applications, like UML. However, to be a successor of the currently dominant object paradigm, GP should involve all basic concepts of OOP. Therefore, the paper examines the usage of basic object oriented concepts, encapsulation, inheritance and polymorphism on a level of modeling generators using GSM. Encapsulation and inheritance were present in the original GSM, while polymorphism was introduced recently. It has shown that benefits of introducing polymorphism into GSM are recognized in more precise application specification, more reusability of generator, simpler generator model (lover number of levels) and its easier implementation.

For now, the improved GSM is implemented in form of a C++ library and used mainly in development of web application generators. In our future projects, we plan to work on the development of new programming platforms and new tools for making generators, as well as on the improvements to the prototype reengineering process.

REFERENCES

[1] Czarnecki, K., Eisenecker, U.W.: "Generative, Programming: Methods, Tools, and Applications", Addison Wesley, 2000.

[2] Eisenecker, U. : "Generative Programming: Beyond Generic Programming", Proc. Dagstuhl Seminar on Generic Programming, April 27--May 1, 1998, Schloß Dagstuhl, Wadern, Germany, 1998.

[3] Guerraoui R. : "Strategic directions in object-oriented programming", ACM Computing Surveys, Baltimore, december 1996.

[4] Jacobson, I., Booch, G., Rumbaugh, J.: "The Unified Software Development Process", Addison-Wessley, 1999.

[5] Kandé, M.M., Kienzle,J., Strohmeier, A.: "From AOP to UML - A Bottom-Up Approach", 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, 2002.

[6] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.:“Aspect-Oriented Programming”. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[7] Limbourg, P., Kochs, H.D.: "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, Pages 815-828, 2008.

[8] Radošević, D.: "Integration of Generative Programming and Scripting Languages", Doctoral thesis, Faculty of Organization and Informatics, Varaždin, 2005.

[9] Radošević D., Kliček, B., Kozina, M.:"Conceptual Similarities and Differences Between Object Model and Generator Application Scripting Model", DAAAM International Scientific Book 2006, DAAAM International, Vienna, Austria 2006.

[10] Radošević, D., Kliček, B., Dobša, J. "Generative Development Using Scripting Model of Application Generator", DAAAM International Scientific Book 2006, DAAAM International, Vienna, Austria, 2006.

[11] Radošević, D., Orehovački, T., Konecki, M.:"PHP Scripts Generator for Remote Database Administration based on C++ Generative Objects", Proceedings of "Mipro 2007" conference, Opatija, 21.05.-25.05.2007.