sciendo

BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **20**, No 3
Sofia • 2020 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2020-0024

CRUDyLeaf: A DSL for Generating Spring Boot REST APIs from Entity CRUD Operations

Omar S. Gómez[1], *Raúl H. Rosero*[1], *Karen Cortés-Verdín*[2]

[1]*GrIISoft Research Group – Escuela Superior Politecnica de Chimborazo, Riobamba 60155, Ecuador*
[2]*Universidad Veracruzana, Xalapa 91090, Mexico*
E-mails: ogomez@espoch.edu.ec rrosero@espoch.edu.ec kcortes@uv.mx

***Abstract*:** *Domain-Specific Languages* (*DSLs*) *are programming languages designed specifically to express solutions to problems in a particular domain. It is said they foster productivity and quality. In this work we describe CRUDyLeaf, a DSL focused on the generation of Spring Boot REST APIs from entity CRUD operations. Spring Boot is an open source Java-based framework used to implement the REST architecture style. It has gained popularity among developers mainly because it allows to build stand-alone and production ready software applications* (*avoiding the use of an application server*). *Through seven proposed stages* (*domain immersion, golden application implementation, syntax definition, meta model generation, code generator implementation, deployment, and refinement*) *we describe the development of this DSL. We also exemplify and evaluate the proposed DSL. Our findings suggest a yield automation rate of 32.1 LOC* (*Lines Of Code*) *for each LOC written in this DSL, among other observed benefits.*

***Keywords*:** *CRUDyLeaf, DSL, Domain-Specific Language, Software Engineering, Spring Boot.*

1. Introduction

Nowadays Domain-Specific Languages (DSLs) are so popular that they have begun to be used in different domains such as: law [1], blockchain smart contracts [2], genomics [3], automotive [4], seismology [5], cybersecurity [6], electrical engineering [7], mathematics [8], among other disciplines. DSLs are specific languages designed and implemented to address problems from particular application domains, as the ones previously mentioned.

At a glance, a DSL is composed of a concrete syntax, which defines the notation for implementing a program. It may be textual, graphical or tabular [9]. It also consists of an in-memory representation of the concrete syntax known as the meta-model that contains semantical information about the language notation, i.e., the abstract representation of the language. Optionally, code generation can be part of a DSL. Once having the meta-model, it is possible to generate code from the given DSL.

According to [10], DSLs can be developed under a model-based approach or a text-based approach. Model-based implementation represents the language as a meta model, so the coded program is an instance of the given meta model. On the other hand, text-based approaches represent the language as a set of grammar rules and the program as a text that conforms to the given rules.

DSLs can also be classified as internal or external ones. An internal DSL is either built on top of an existing language (general programming language) or it extends another language which is commonly packaged as a language library. Regarding an external DSL, it is implemented through an independent interpreter or compiler, i.e., the language itself is separate from the language used for developing it [11].

In this work we present CRUDyLeaf, an external text-based DSL used for building Spring Boot REST endpoints (APIs) by defining entities and CRUD (Create, Read, Update, Delete) persistent storage operations. Spring Boot is a widely adopted open source Java-based framework mainly used to implement and expose REST resources (also known as RESTful APIs, endpoints, services or microservices). The DSL here presented has been developed by the following seven proposed stages.

The rest of the document is organized as follows: Section 2 describes the related work. Section 3 describes the architecture of a typical Spring Boot application that implements the REST architectural style. Section 4 describes the structure of the proposed DSL. Section 5 exemplifies and evaluates the use of the proposed DSL. Finally, Section 6 presents the discussion and conclusions.

2. Related work

In the context of the present work, we found a couple of works that address the use of domain-specific languages in the domain of the REST architecture style. In the first work found [16], authors propose a DSL for specifying REST based contracts that can be translated into source code. Authors implement a DSL and a code generator written in the Haskell programming language. Authors implement a pluggable architecture for the code generator, which generates Swagger, Python, and Java source code. The aim of this DSL is to support the definition and generation of REST services for a proprietary platform designed and used by the Brazilian Army.

Concerning the second work found [17], authors develop a model-driven software tool that supports specification and code generation related to the REST software architecture style under a distributed system. Authors used the Xtext and Sirius frameworks for developing the textual and graphical concrete syntax, respectively. Authors implement a series of code generators with the Xtend programming language. The generated code is able to run under a distributed or cloud software architecture, so it uses libraries such as Zuul, Eureka, Turbine, Hystrics and the Spring Cloud framework. The defined REST APIs comprise basic CRUD operations over the defined entities.

Finally, another related work is reported in [18], although authors do not specify a DSL, they use a domain model (Ecore model) as starting point for generating REST APIs in the Java programming language. Authors also map CRUD operations from

the specified entities to REST APIs. The generated code is aimed to be deployed in a Java EE application server, so it uses technologies such as JAXB, JPA, EJB, CDI and JAX-RS.

3. Spring Boot REST application architecture

Spring Boot has gained popularity among Java developers. It is used as an alternative to deploy software products in application servers. It is considered the de facto standard for microservice development [12]. Spring Boot has a built-in server by which the process of implementing a REST application is significantly simplified [22]. A Spring Boot application that exposes REST resources from CRUD operations can be modelled as the one shown in Fig. 1.

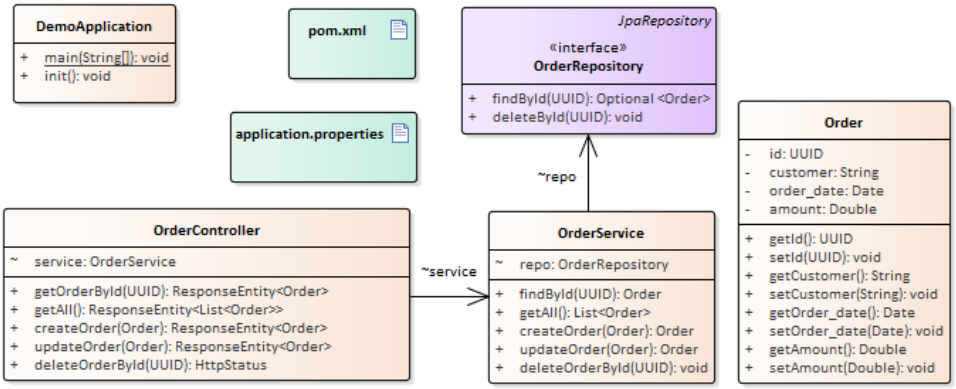

Fig. 1. Example of a Spring Boot application that exposes REST resources from CRUD operations

A Spring Boot application usually is configured as a Maven project, the pom.xml file contains the dependencies needed for building and running this kind of applications. The application properties file contains configuration properties such as the persistence storage used (database), web server port, among other settings. A main class is used as the starting point for running the application (in this example, DemoApplication).

As shown in Fig. 1, an entity class is modelled to be an object wrapper for a database table (Order class), so its attributes are mapped to columns on the database table. It is done by using an object-relational mapping framework like Hibernate. A repository interface is used to access CRUD operations. In this case, OrderRepository extends from the JpaRepository interface, which, among other functionalities, provides CRUD operations. A service class is used to wrap the repository interface; as shown Fig. 1, the class OrderService wraps the CRUD operations exposed by the repository. Finally, a controller class contains the code used for exposing CRUD operations as resources (endpoints); in this case five resources are exposed, all related to the methods: getOrderById, getAll, createOrder, updateOrder, and deleteOrderById. The Tomcat HTTP web server is used for exposing the implemented resources.

4. Proposed DSL

CRUDyLeaf was developed through the following proposed stages: domain immersion, golden application implementation, syntax definition, meta model generation, code generator implementation, deployment, and refinement. These stages are explained below.

4.1. Domain immersion

This stage consists in learning about the domain in which the DSL will be built. In our case, we examined the Spring Boot framework, particularly how typical REST applications with CRUD operations are implemented. A series of exercises were coded in order to gain more understanding of this domain.

4.2. Golden application implementation

Once immersed in the domain of interest, we implemented a Spring Boot application that exposes CRUD operations as REST resources. This application was used as a golden template in order to derive the syntax for the DSL, this golden application is also used as reference for implementing the code generator.

4.3. Syntax definition

```
5  Domainmodel:
6      element=Body;
7
8  Body:
9      {ConfigDeclaration}
10      group=Group
11      artifact=Artifact
12      api=API
13      timezone=Timezone
14      entity=EntityDeclaration;
15
16 Group:
17     'group' ':' name=QualifiedName;
18
19 Artifact:
20     'artifact' ':' name=ID;
21
22 API:
23     'api_prefix' ':' name=QualifiedNameAPI;
```

Fig. 2. Concrete syntax excerpt of the CRUDyLeaf DSL

With a primitive version of the syntax, in this stage we refined and finished the concrete syntax and built the DSL. We implemented it using Xtext [13]. Xtext is a powerful framework for building language workbenches for textual DSLs. Xtext only requires specifying a grammar file. With this syntax file, Xtext defines the language and creates the required infrastructure such as the parser, linker, type checker as well includes an editor for the Eclipse IDE. Fig. 2 shows an excerpt of the defined concrete syntax.

4.4. Meta model generation

The concrete syntax serves as an input for generating the semantic model (meta model). Among the infrastructure generated by Xtext, an Ecore model is generated.

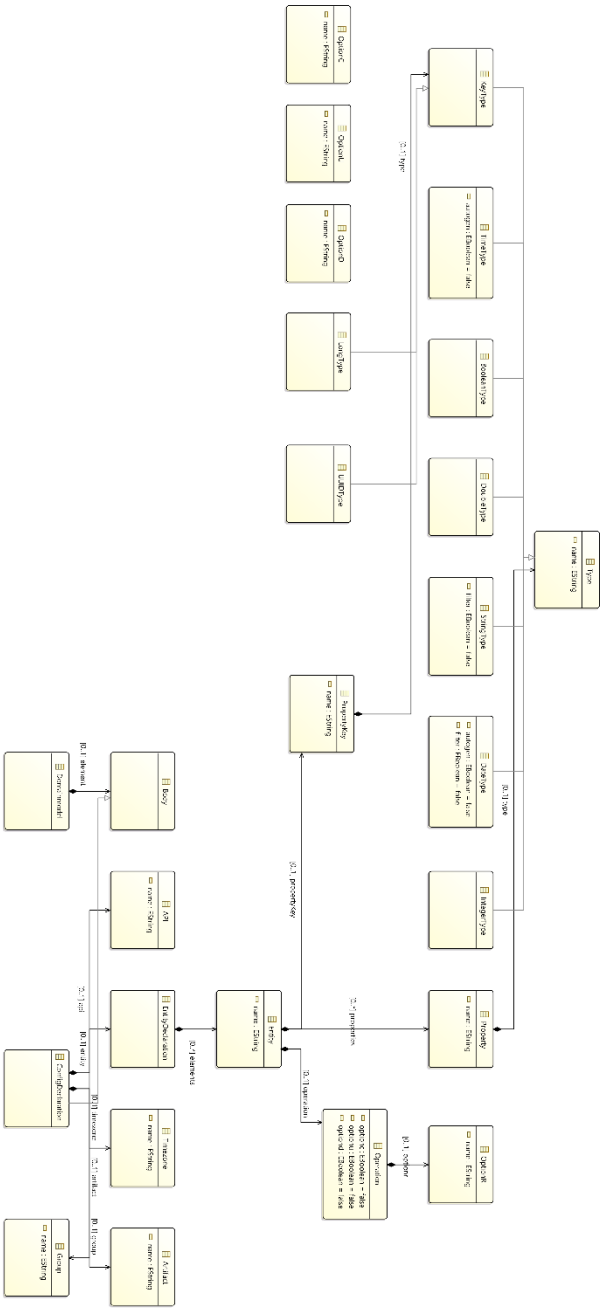

Fig. 3. Ecore meta model excerpt of the CRUDyLeaf DSL

It represents the Abstract Syntax Tree (AST). This Ecore model can be used to perform Model to Text (M2T) transformations, i.e., generate source code from the model. This stage is automatically conducted in the Eclipse IDE by running the Xtext artifacts generation function. Fig. 3 shows the Ecore model derived from the concrete syntax (also available at **http://osgg.net/crudyleaf/crudyleaf_ecore.png**).

4.5. Code generator implementation

We used the Xtend language [14] in order to write the code generator from the proposed DSL. Xtend is a statically-typed programming language that was initially released with Xtext. We wrote different code templates in order to automatically generate all the Spring Boot Java files that were necessary. Fig. 4 shows an excerpt of the Xtend language used for implementing the code generator. This excerpt is related to the service classes generation.

```
513     //SERVICE
514     def writeServiceClasses(Entity e)'''
515     import org.springframework.stereotype.Service;
516     import org.springframework.beans.factory.annotation.Autowired;
517     import java.util.ArrayList;
518     import java.util.List;
519     import java.util.Optional;
520     «var String aux = null»
521     «FOR p : e.properties»
522         «IF p.type.name == "Date" && (p.type as DateType).filter»
523             «{aux = "filter"; ""}»
524         «ENDIF»
525     «ENDFOR»
526     «IF aux !== null»
527     import java.util.Date;
528     «ENDIF»
529     «IF e.propertyKey.type.name == "UUID"»
530     import java.util.UUID;
531     «ENDIF»
532
533     @Service
534     public class «e.name»Service {
535
536         @Autowired
537         «e.name»Repository repo;
```

Fig. 4. Xtend code excerpt of the implemented code generator

4.6. Deployment

The aim of this stage is to make available the developed DSL in order to be used by other people. In our case, we generated an eclipse plugin and published it on the Eclipse Marketplace. This allows anyone interested in this DSL to install it in his/her Eclipse IDE. Fig. 5 shows a screenshot of the published DSL in the Eclipse Marketplace.

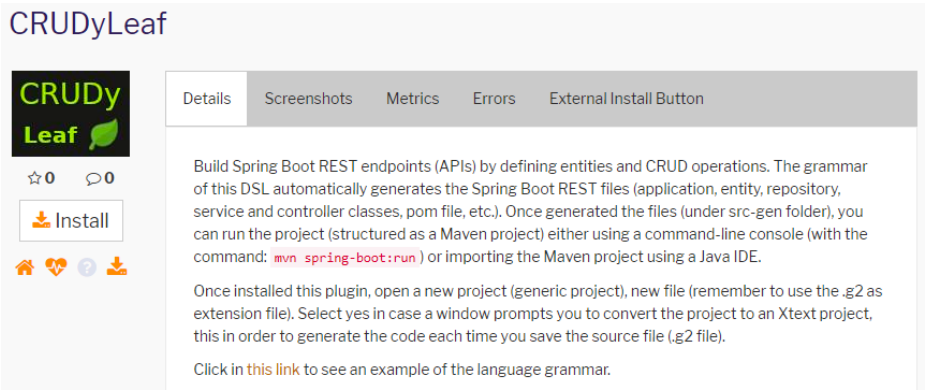

Fig. 5. Published DSL in the Eclipse Marketplace (**https://marketplace.eclipse.org/content/crudyleaf**)

4.7. Refinement

Building a DSL is a cyclical process, although the previous stages seem to be sequential, stages can be overlapped or jointly addressed. For example, when we worked on the golden application we also started to identify the grammar of the language. We also worked interchangeably between the code generator and the syntax definition. Finally, once the DSL was deployed onto the Eclipse Marketplace, some issues were addressed mainly in the code generator.

5. Exemplification and evaluation of the proposed DSL

In this section we exemplify the use of the proposed DSL through a scenario. Let assume we want to expose some REST endpoints from CRUD operations belonging to two entities: Car and Order. This scenario can be specified with the proposed DSL as shown in Fig. 6.

example.g2

```
group:net.osgg
artifact:Demo
api_prefix:/api/v1
timezone:America/Guayaquil
entities{
    entity Car{
        operations: r c u d
        key: Long pk
        name: String filter
    }
    entity Order{
        operations: r c u d
        id: UUID pk
        customer: String
        orderDate: Date auto filter
        amount: Double
    }
}
```

Fig. 6. Grammar example of CRUDyLeaf

This DSL is composed of 22 reserved words which are: *group*, *artifact*, *api_prefix*, *timezone*, *entities*, *entity*, *operations*, *r*, *c*, *u*, *d*, *pk*, *filter*, *auto*, *UUID*, *Long*, *Integer*, *Double*, *Boolean*, *String*, *Date*, and *Time*.

As shown in Fig. 6, the DSL will produce the corresponding Spring Boot Java files, such as: Car.java (entity), CarRepository.java, CarService.java, CarController.java, Order.java (entity), OrderRepository.java, OrderService.java, OrderController.java, also the main file DemoAppApplication.java and the Maven pom file pom.xml. All the source code from this DSL is generated under the src-gen folder. The *group* and *artifact* reserved words are used to specify the package and application name, respectively. The *api_prefix* reserved word is used to specify the location of the exposed REST resources (APIs). The CRUD operations specified in the entities are exposed as REST services in the CarController.java and

OrderController.java classes. In this case the four operations were specified (at least a "Read" operation should be defined).

By default, the H2 database is used to persist the entities, but it can be changed to other databases by editing the application.properties file. The source code generated is structured as a Maven project, which can be run in an IDE or through a console using a command prompt. Fig. 7 shows an excerpt of the code generated by the DSL, in this case the code related to the OrderService.java file.

The data types supported for this DSL are: *Long*, *UUID*, *String*, *Integer*, *Double*, *Boolean*, *Date* and *Time*. The first property of an entity is specified with the "*pk*" reserved word after defining the datatype. This keyword is used to specify the primary key of its corresponding database table column. This first property should be defined as either a *Long* or *UUID* data type.

```
23 @Service
24 public class OrderService {
25
26     @Autowired
27     OrderRepository repo;
28
29     public List<Order> getAll(){
30         List<Order> orderList = repo.findAll();
31         if(orderList.size() > 0) {
32             return orderList;
33         } else {
34             return new ArrayList<Order>();
35         }
36     }
37
38     public Order findById(UUID id) throws RecordNotFoundException{
39         Optional<Order> order = repo.findById(id);
40         if(order.isPresent()) {
41             return order.get();
42         } else {
43             throw new RecordNotFoundException("Record does not exist for the given Id");
44         }
45     }
46
47     public Order createOrder(Order order){
48         return repo.save(order);
49     }
```

Fig. 7. Excerpt of the code generated by CRUDyLeaf

As shown in Fig. 7, searches in the database can be enabled by using the "*filter*" keyword. In this sense, an entity property can be filtered as long as *String* or *Date* datatypes are defined in a given entity property. When this keyword is specified, the DSL generates the corresponding endpoints (controllers) for searching in the corresponding database table columns. In this case, the DSL generates the necessary code for searching in the "name" column of the Car database table (which is mapped to the "name" property of the Car entity), and also generates the code for searching in the "orderDate" column of the Order database table.

Date and *Time* data types can automatically generate values each time a successful post is requested, just specifying the reserved word "*auto*" after the *Date* or *Time* datatypes, as shown in Fig. 7. In this example, each time a successful post is requested from the corresponding API REST, the current date from the system is retrieved and stored in the database.

Concerning the time zone, it is possible to specify any available time zones from the TimeZone.getAvailableIDs() method, so the web server will use the specified time zone.

The endpoints generated are also documented using OpenAPI and Swagger [15]. This documentation is available once the generated Spring Boot application is run. Fig. 8 shows a screenshot with the information of the exposed endpoints. The documented endpoints can also be tested by the user.

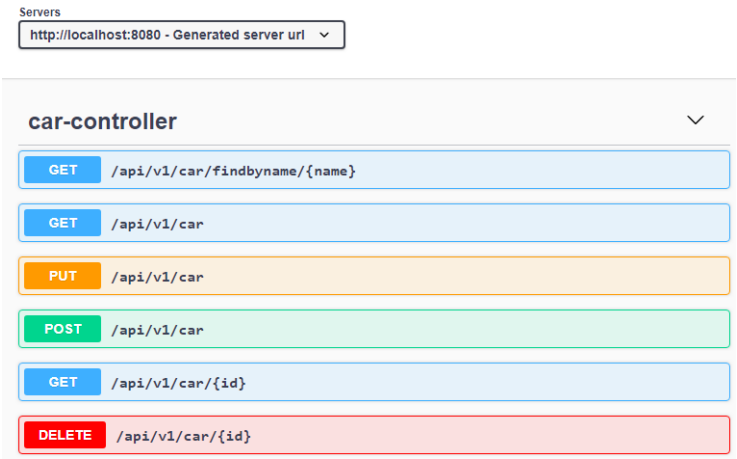

Fig. 8. Example of Swagger documented endpoints generated by the proposed DSL

5.1. Evaluation

Once exemplified the use of this DSL, following we describe the conducted evaluation. With the grammar example previously discussed (see Fig. 6), 578 Lines Of Code (LOC) were automatically generated from the 18 LOC written in this DSL. Table 1 shows the accounting of LOC of every file automatically generated by this DSL. For this example, the yielded automation rate was 32.1 LOC for each LOC written in this DSL (32:1).

Table 1. Accounting of the source code generated from 18 LOC written in the CRUDyLeaf DSL

Generated file	LOC
pom.xml	58
application.properties	10
DemoApplication.java	47
DemoApplicationTests.java	19
Car.java	36
CarController.java	63
CarRepository.java	22
CarService.java	68
Order.java	66
OrderController.java	66
OrderRepository.java	30
OrderService.java	70
RecordNotFoundException.java	23
Total	*578*

With regard to productivity measured as the LOC per 1 h, we have found individual productivity rates of 20 LOC per 1 h [19], 13.3 LOC per 1 h [20], and 10 LOC per 1 h [21] reported in the literature. Averaging these three measurements, we assume an average productivity of 14.4 LOC per 1 h for our DSL assessment. The 578 LOC generated by the DSL will require 40.13 hours, approximately five full workdays. On average, the proposed DSL spent two seconds generating the source code files shown in Table 1, for this example, a yield of 963,333 LOC per 1 h was estimated.

6. Discussion and conclusions

Concerning related works, none of the previously mentioned proposals are currently available, so it was not possible to use or test them. The work reported in [17] is the one more closely related with our proposed DSL, however since our DSL is related with the Spring Boot framework, the concrete syntax is simpler than the one reported in [17]. Having a simple syntax helps to improve the understanding of this kind of Spring Boot applications to persons that start to study this technology. On the other hand, for all those with more knowledge of the Spring Boot framework, our proposed DSL speeds up the development of this kind of applications.

The proposed DSL fits well in current software industry approaches such as DevOps in which software products are delivered quickly and with high reliability [23]. The REST resources generated by CRUDyLeaf are automatically coded as a Spring Boot application (structured as a Maven project), so the resulting application can be automatically built and deployed in a fraction of time, in line with the DevOps approach [24].

In this work we have presented CRUDyLeaf, an external text-based DSL that can be used for automating the building of Spring Boot REST resources by defining entities and CRUD persistent storage operations. The example described in the present work yielded an automation rate of 32.1 LOC for each LOC written in this DSL (32:1), i.e., 578 LOC were automatically generated by coding 18 LOC through this DSL. The main contributions of the presented work are twofold:

- The development and deployment (available in the Eclipse Marketplace) of a DSL that is speeds up the development of Spring Boot REST resources from entity CRUD operations.
- An exemplified and proposed DSL development process conformed of seven stages (domain immersion, golden application implementation, syntax definition, meta model generation, code generator implementation, deployment, and refinement) where this process can be used as reference for building a DSL.

As future work, we plan to implement a validator in order to address better the error markers shown in the IDE. This component will be active in the background while the user of the proposed DSL is typing in the IDE editor, so the user will get an immediate feedback of the typed syntax.

References

1. Alves, A., P. Ventura, A. Rodrigues. LegalLanguage: A Domain-Specific Language for Legal Contexts. – In: D. Aveiro, G. Guizzardi, J. Borbinha, Eds. Advances in Enterprise Engineering XIII. EEWC 2019, Lecture Notes in Business Information Processing, Vol. **374**, 2020, pp. 33-51.
2. Skotnica, M., R. Pergl. Das Contract – A Visual Domain Specific Language for Modeling Blockchain Smart Contracts. – In: D. Aveiro, G. Guizzardi, J. Borbinha, Eds. Advances in Enterprise Engineering XIII. EEWC 2019. Lecture Notes in Business Information Processing, Vol. **374**, 2020, pp. 149-166.
3. Coelho, L. P., R. Alves, P. Monteiro, J. Huerta-Cepas, A. T. Freitas, P. Bork. NG-Meta-Profiler: Fast Processing of Metagenomes Using NGLess, a Domain-Specific Language. – Microbiome, Vol. **7**, 2019, No 84, pp. 1-10.
4. Maschotta, R., A. Wichmann, A. Zimmermann, K. Gruber. Integrated Automotive Requirements Engineering with a SysML-Based Domain-Specific Language. – In: Proc. of IEEE International Conference on Mechatronics (ICM), IEEE, 2019.
5. Louboutin, M., M. Lange, F. Luporini, N. Kukreja, P. Witte, F. Herrmann, P. Velesko, G. Gorman. Devito (V3.1.0): An Embedded Domain-Specific Language for Finite Differences and Geophysical Exploration. – Geoscientific Model Development, Vol. **12**, 2019, No 3, pp. 1165-1187.
6. Caramujo, J., A. Rodrigues, S. Monfared, A. Ribeiro, P. Calado, T. Breaux. RSL-IL4Privacy: A Domain-Specific Language for the Rigorous Specification of Privacy Policies. – Requirements Engineering, Vol. **24**, 2019, No 1, pp. 1-26.
7. Monjardim, G. E., A. Rodrigues, F. M. Varejão, V. E. Silva, M. P. Ribeiro. A Domain-Specific Language for Fault Diagnosis in Electrical Submersible Pumps. – In: Proc. of 16th International Conference on Industrial Informatics (INDIN), IEEE, 2018.
8. Earl, C., M. Might, A. Bagusetty, J. C. Sutherland. Nebo: An Efficient, Parallel, and Portable Domain-Specific Language for Numerically Solving Partial Differential Equations. – Journal of Systems and Software, Vol. **125**, 2017, pp. 389-400.
9. Voelter, M. DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. Create Space Independent Publishing Platform, 2013.
10. Negm, E., S. Makady, A. Salah. Survey on Domain Specific Languages Implementation Aspects. – International Journal of Advanced Computer Science and Applications, Vol. **10**, 2019, No 11, pp. 624-633.
11. Fowler, M. Domain-Specific Languages. Addison-Wesley Professional, 2010.
12. JetBrains (last accessed 13.02.2020).
    https://www.jetbrains.com/lp/devecosystem-2019/java/
13. Xtext (last accessed 02.02.2020).
    https://www.eclipse.org/Xtext/
14. Xtend (last accessed 09.02.2020).
    https://www.eclipse.org/xtend/
15. Swagger (last accessed 22.02.2020).
    https://swagger.io/docs/specification/about/
16. Lima, L., R. Bonifácio, E. Canedo. NeoIDL: A Domain Specific Language for Specifying REST Contracts Detailed Design and Extended Evaluation. – International Journal of Software Engineering and Knowledge Engineering, Vol. **25**, 2015, No 9-10, pp. 1653-1675.
17. Terzić, B., V. Dimitrieski, S. Kordić, G. Milosavljević, I. Luković. Development and Evaluation of MicroBuilder: A Model-Driven Tool for the Specification of REST Microservice Software Architectures. – Enterprise Information Systems, Vol. **12**, 2018, No 8-9, pp. 1034-1057.
18. El-Douibi, H., J. L. Izquierdo, A. Gómez, M. Tisi, J. Cabot. EMF-REST: Generation of RESTful APIs from Models. – In: Proc. of Symposium on Applied Computing (SAC'16), 2016.
19. Roy, D. M. The Personal Software Process: Downscaling the Factory. – In: Proc. of 19th Annual Software Engineering Workshop, 1994.

20. G w a k, T., Y. J a n g. An Empirical Study on SW Metrics for Embedded System. – In: Q. Wang, D. Pfahl, D. M. Raffo, P. Wernick, Eds. Software Process Change. SPW 2006. Lecture Notes in Computer Science, Vol. **3966**, 2006, pp. 302-313.
21. B a h e t i, P., L. W i l l i a m s, E. G e h r i n g e r. Distributed Pair Programming: Empirical Studies and Supporting Environments. – Technical Report, TR02-010, Univ. of North Carolina at Chapel Hill, 2002, pp. 1-11.
22. G a j e w s k i, M., W. Z a b I e r o w s k i. Analysis and Comparison of the Spring Framework and Play Framework Performance, Used to Create Web Applications in Java. – In: Proc. of XV International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH'19), IEEE, 2019.
23. A n g a r a, J., S. P r a s a d, G. S r i d e v i. DevOps Project Management Tools for Sprint Planning, Estimation and Execution Maturity. – Cybernetics and Information Technologies, Vol. **20**, No 2, 2020, pp. 79-92.
24. E b e r t, C., G. G a l l a r d o, J. H e r n a n t e s, N. S e r r a n o. DevOps. – IEEE Software, Vol. **33**, 2016, No 3, pp. 94-100.

Received: 27.03.2020; Second Version: 07.07.2020; Accepted: 22.07.2020