A Meta-Model and Code Generator for Evolving Software Product Lines

Tijana Lalošević, Željko Vuković, Gordana Milosavljević
Faculty of Technical Sciences, University in Novi Sad, Serbia
tijana.vdn@gmail.com, zeljkov@uns.ac.rs, grist@uns.ac.rs

***Abstract*—Software Product Line (SPL) engineering is a paradigm which allows reducing development time, effort, and costs for development of products with the same core features and some variations needed for every client that purchases the product. Instead of writing the variations code from scratch, we can follow the Model-Driven development approach, which aims to generate software from design models automatically. Incorporating Model-Driven Software development in an existing large scale software product line (SPL) can be challenging and full of obstacles due to constant development and changes in the SPL core and product architecture. The introduction of the Model-Driven approach to such solutions often must be done in an iterative and incremental manner to embrace the changes. In order to achieve this goal, it is necessary to fulfill two preconditions: the existence of the domain-specific modeling language and transformation programs for automatic code generation from a model. This paper presents a meta-model and a code generator that enables rapid development and customization of the SPL applications. Our solution enables the core product line to be automatically expanded in any segment (e.g. method, data structure, etc.).**

Keywords: MDE, Code Generator, SPL, meta-model, Domain-specific modeling language

I. Introduction

The software industry has experienced expansion over the last twenty years. Dr William Raduchel, a professor at Harvard and later a chief executive at Sun Microsystems, Xerox and AOL Time Warner, describes the software as "the core of most modern organizations, products and services."[1] Most companies produce products for a specific market, therefore the products contain a common basis that often requires customization for a particular client. Development software market like any other market falls under the rules of supply and demand. This market is highly variable and customer appetites are increasing, so it is crucial for software companies to quickly develop a product in order to sell and retain customers and stay competitive. One solution to this problem is Model-Driven Software Development (MDSD).

For companies that are already building product lines, MDSD can further increase productivity because[1]:

- Variability can be described more concisely since, in addition to the traditional mechanisms, variability is also described on model level.
- The mapping from problem to solution domain can be formally described and automated using model-to-model transformations.
- Aspect-oriented techniques enable the explicit expression and modularization of crosscutting variability on the model, code, and generator level.
- Fine-grained traceability is supported since tracing is done on the model element level rather than on the level of code artifacts.

Model-Driven Software Development (MDSD) is a paradigm that focuses on models as the primary concept. From the standpoint of MDSD, modeling is successful if the model makes sense from the perspective of the user who is familiar with the domain and if they can serve as the basis for implementing the system.

Agile methodologies (**Figure 1**) are project management methodologies that uses short development cycles to prioritize continual improvement in the development of a product or service where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. Agile project management is an approach based on delivering requirements iteratively and incrementally throughout the project life cycle. At the core of agile methodologies is the requirement to exhibit central values and behaviors of trust, flexibility, empowerment, and collaboration. [3] These methodologies have proven to be the most effective today and becoming the industry standard, as it provides constant feedback from users (**Figure 2**). Creating models can be hard in situations where they require extensive communication between product managers, designers, developers, and users of application domains.

Software Product Line (SPL) engineering is concerned with systematically reusing development assets in an application domain. It is similar to mass customization in a traditional industry, aiming to develop and evolve software systems as quality products, with reduced development effort and time-to-market.[4] Systematic and planned reuse is facilitated within the system by integrating common and variable aspects into reusable artifacts. The integration of these artifacts is most often made possible by the fact that basic software components have an adaptive architecture. However, despite all the facilitation, just putting together the final product is usually a repetitive job. The most important factors in achieving profit in the field of software development are implementation time and the possibility of rapid adaptation to the ever-changing client requirements. Our solution can help reduce time and increase efficiency.

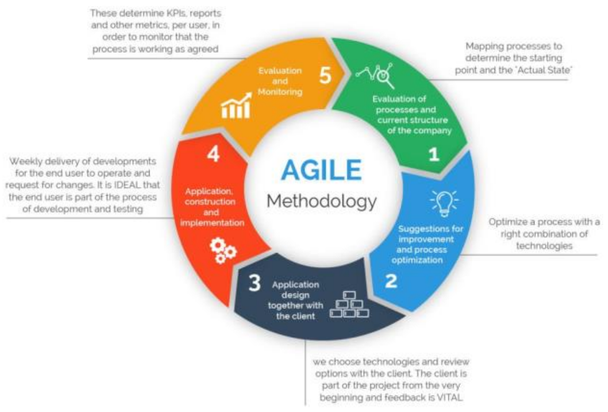

These determine KPIs, reports and other metrics, per user, in order to monitor that the process is working as agreed
Mapping processes to determine the starting point and the 'Actual State'
Evaluation and Monitoring
5
Evaluation of processes and current structure of the company
1
AGILE
Methodology
4
Application, construction and implementation
Weekly delivery of developments for the end user to operate and request for changes. It is IDEAL that the end user is part of the process of development and testing
Suggestions for improvement and process optimization
2
Optimize a process with a right combination of technologies
3
Application design together with the client
we choose technologies and review options with the client. The client is part of the project from the very beginning and feedback is VITAL

Figure 1 Iterative software development [4]

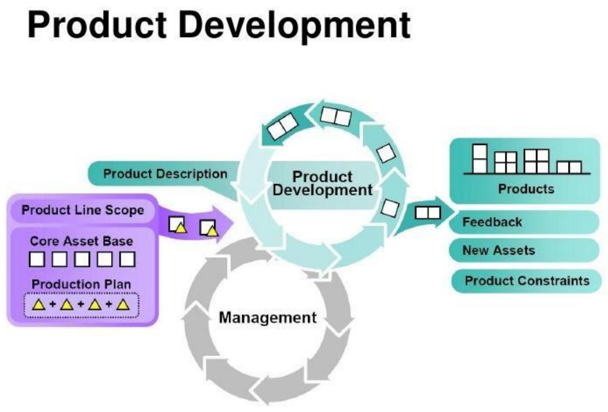


Figure 2 Product Development [5]

Domain-Specific Language (DSL) languages are languages that are designed for a specific, narrowly defined domain, context, or company to make it easier for people to describe the domain concepts. If the language is modeling oriented, it can also be called domain-specific modeling language (DSML) [5].

To facilitate the maintenance of existing SPLs and accelerate development of new large-scale SPLs using a Model-Driven Engineering (MDE) approach, we have developed a meta-model, a code generator, and accompanying tools. Since the introduction of MDE to the original SPL came in later phases of its development and deployment, it was not possible to alter the existing SPL architecture. This posed various challenges and constraints during this research and resulted in some specific solutions.

When a client purchases an existing SPL solution, its entities and processes usually have to be adapted to accommodate established business processes of the client's system. This customization of the software to individual client's needs is a good example of the use of the SPL workspace, in which SPL core is expanded and configured to get the product in accordance with the requirements.

Our goal is to build meta-model and code generator which provide the following activities for SPL developed using .NET technologies:

1. Extension of data models and configuration files;
2. Extension of the database schema used for data persistence;
3. Extending the user interface to allow the user to update and view documents;
4. Extending the validation of the documents entered and changing their life cycle.

II. Related Work

Model-driven software development (MDSE) within SPL is one of the current topics addressed by a number of scientific papers.

A. *Perspectives on combining model-driven engineering, software product line engineering, and version control*

This paper describes the SCT (specification-configuration-templates) specification of a source code generator that is independent of the target programming language and problem domain.[7] The code generator is defined as a multi-level structure, which allows the nesting of a generator, similar to nesting programming structures in structured and object-oriented programming.

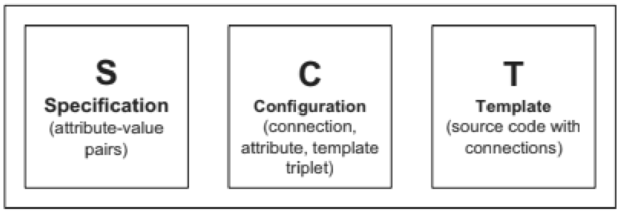

S
Specification
(attribute-value pairs)
C
Configuration
(connection, attribute, template triplet)
T
Template
(source code with connections)

Figure 3 SCT framework [7]

This solution is different from our CG (CodeGen) solution in that it contains inside its meta-model the information which template should be used to generate which element, as well as a list of all the templates. In CG, this information is stored as code generator configuration for every generated file type.

B. *Perspectives on Combining Model-Driven Engineering, Software Product Line Engineering, and Version Control*

Another challenge was file versioning, which is one of the biggest challenges with SPL where certain components are developed by different teams. In [8] the authors deal with the problem of mutual referencing of artifacts and their versioning. As stated in [8]: "The three most important aspects of modeling the evolution of architectures and product lines are versions, options, and variants. Versions record information about the evolution of architectures and elements like components, connectors, and interfaces. Options indicate points of variation in an architecture where the structure may vary by the inclusion or exclusion of an element or group of elements. Variants indicate points in an architecture where one of several alternatives may be substituted for an element or group of elements." However, this solution does not support the ability to insert a piece of code into an existing file depending on the context.

C. *ReingIS: A Toolset for Rapid Development and Reengeneering of Bussines Systems*

A particular challenge is reengineering, that is, transferring the solution to multiple versions. One of the interesting solutions is described in [8]. This solution involves the entire development framework, which in addition to generating code, also deals with system security. Its most complex element is the *analyzer*, a component that creates a specification of a business system user interface based on a database specification, with support for CRUD (create, update, delete) operations, lookup fields, form connections, etc. The latter implies that the analyzer can recognize the hierarchy of documents. This solution is one of the starting points for designing our generator.

III. CODEGEN REQUIREMENTS

The Code Gen requirements are quite complex and will be grouped into several sections for a better explanation. The first section will explain how the CG influences an existing solution, as well as the differences between conventional solutions and good practices. Next, the generation and integration requirements will be presented.

A. *Requirements related to the generation and integration of generated content*

The most important requirement is that CG should generate a new project, as well as to support the development of an existing project. Also, it should provide support to the various development teams that maintain certain technological segments (web, desktop, relational database, etc.). It is necessary to allow the user to extend a specific group of functionalities, that is, to make the graphical user interface intuitive and easy for the user with domain knowledge. For example, if a user wants to extend only the desktop application, the generator should provide him/her only with these extensions.

All generated files must be integrated with an existing project core. The desired result is an extended system that is successfully compiled and executed.

B. *Requirements that affect the structure of an existing solution*

MDSE's recommendation is to keep handwritten and generated code in separate files on disk and to use some of the known mechanisms for their integration: inheritance, extension (delegation), aspects, partial class mechanism etc. **Error! Reference source not found.**. Most of these mechanisms require careful design of the architecture of the system over which the code will be generated, before embarking on the development of the code generator.

However, when code generators are built for existing systems where the architecture must not be changed, different concepts are required, which will be outlined below.

The code generator directly changes handwritten code in situations where a strict separation of handwritten and generated code is not possible (various XML configuration files, XAML, HTML and javascript files). Both the developers and the CG are expected to work on these files. The C# classes are divided into two physically separated files (the concept of partial class from .NET), where one part of the class can be modified by the developer and the other part is under the responsibility of the CG. The developer should not modify the CG part.

If CG needs to change a part of the file that was modified by the developer and it finds out that there is a conflict, it must consult the user.

C. *An example of a concrete system*

The first challenge was to develop a meta-model that is not tied to any programming language so that any changes to the underlying architecture would result in minor changes to the code generators themselves.

The second challenge was to give context to the elements we want to generate that depends on domain knowledge. For example, the existence of different aggregations, where aggregation between type A and B and aggregation between type C and B do not behave in the same way.

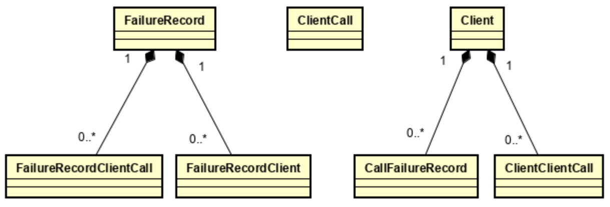
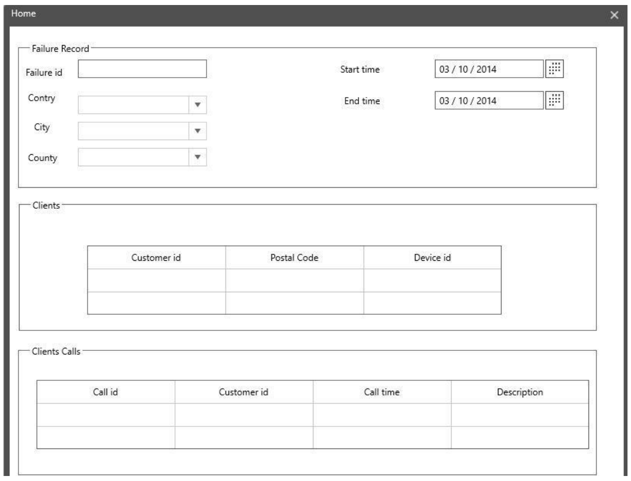


Figure 4 Example for aggregations with context

Thus, the documents themselves do not contain references to another type of document, but instead, contain auxiliary models that represent a "logical reference" to another type of document. In order to better understand the problem, a sketch of the screen format of the Failure Record document will be presented.

Using the sketch form of the Failure Record document screen (**Figure 5**), we can see that the fields of the other two documents (Client and ClientCall) are also contained within it. However, Client and ClientCall models were not used within the Failure Record, but auxiliary models were created represented by the diagram in the **Figure 4**.


Home

Failure Record

Failure id

Start time 03 / 10 / 2014

Contry

End time 03 / 10 / 2014

City

County

Clients

Customer id	Postal Code	Device id

Clients Calls

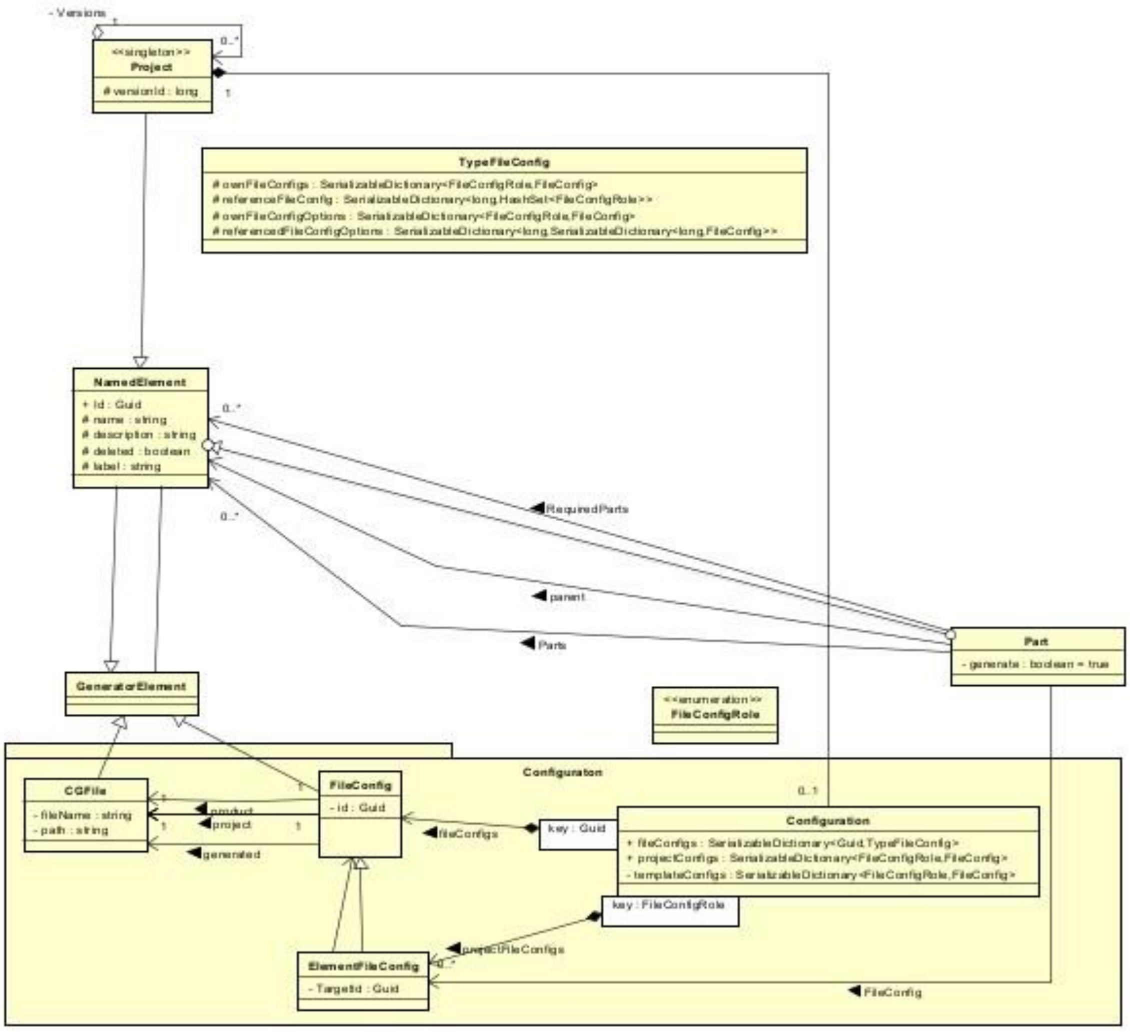
Call id	Customer id	Call time	Description

Figure 5 Screen format of the Failure Record document

IV. META – MODEL

A meta-model can be considered as a model of a modeling language. The term “meta” (“behind” or “above” something) is therefore relative – depending on the perspective, a model is either a model or a meta-model. It is important to note that a meta-model is a model at a different level of abstraction that makes statements about the structure of another model (or a whole set of other models), without making statements about their content.[11]

Figure 6 shows the core meta-classes that support the description of evolving SPL configuration.



- Versions
1
0..*
<<singleton>>
Project
# versionId : long
1
TypeFileConfig
# ownFileConfigs : SerializableDictionary<FileConfigRole,FileConfig>
# referenceFileConfig : SerializableDictionary<long,HashSet<FileConfigRole>>
# ownFileConfigOptions : SerializableDictionary<FileConfigRole,FileConfig>
# referencedFileConfigOptions : SerializableDictionary<long,SerializableDictionary<long,FileConfig>>
NamedElement
+ Id : Guid
# name : string
# description : string
# deleted : boolean
# label : string
0..*
RequiredParts
0..*
parent
Parts
Part
- generate : boolean = true
GeneratorElement
<<enumeration>>
FileConfigRole
Configuration
0..1
CGFile
- fileName : string
- path : string
FileConfig
- id : Guid
1
product
project
1
1
generated
fileConfigs
key : Guid
Configuration
+ fileConfigs : SerializableDictionary<Guid,TypeFileConfig>
+ projectConfigs : SerializableDictionary<FileConfigRole,FileConfig>
- templateConfigs : SerializableDictionary<FileConfigRole,FileConfig>
key : FileConfigRole
projectFileConfigs
0..*
ElementFileConfig
- TargetId : Guid
FileConfig

Figure 6 CG meta-model

The **NamedElement** meta-class is one of the basic concepts of most metamodels. It has a name and a unique identifier.

Project is a container of all specified elements. It contains a collection of earlier versions of the project, which enables comparison of the project versions and incremental code generation, so that code is generated only for new, edited or deleted elements, which minimizes the possibility of conflicts with manually edited code.

FileConfig is used to configure names and locations of project artifacts. It has three references to **CGFile**: **product** (artifact in SPL core), **project** (artifact in SPL customization that can be manually edited by the developer) and **generated** (artifact in SPL customization which is maintained exclusively by the code generator).

ElementFileConfig inherits FileConfig and has a reference to it in order to avoid redundant file descriptions if the situation arises that it should be expanded while expanding other documents.

Configuration consists of the following collections: **fileConfigs**, **projectFileConfigs**, and **templateFileConfigs**. **FileConfigs** is a collection of all configuration files. **ProjectFileConfigs** is a collection of configuration files that are independent of the specific project element (there is usually one such file in the entire project code base, e.g. configuration files for services or localization files). **TemplateFileConfigs** is a collection of the configuration files that are used when creating a new document.

Part is a generic element that is used to model any piece of either the resulting application or software development constructs used to implement it. Parts can be methods, method calls, different parameters depending on the referenced element, but also various dialogues, documents, document sections, etc. The Part meta-class contains a reference to the parent part as well as a list of required parts which generation is mandatory when generating the given part. Parts enable automation of development of mutually dependent documents and artifacts, when a change in one part of the document causes changes in other documents, across different artifact types and application layers (user interface, domain classes, database scripts, etc.).

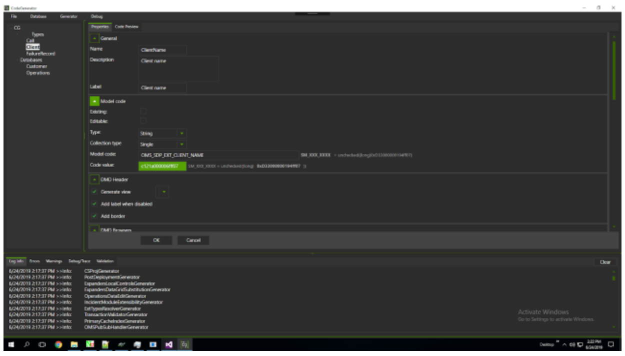
V. IMPLEMENTATION

As each client has specific features that SPL needs to address. The target software product is formed by expanding the SPL core using one of the seven mechanisms provided:

1. Inherit C # classes from core. The descendants are generated as partial classes [12], so CG and developers can have their dedicated parts to change;
2. Roslyn parser for C# can be used for direct manipulation of C# code in existing projects [13];
3. Extending the configuration of XML files – by using XML parser **Error! Reference source not found.**;
4. Extending XAML documents as XML-based syntax to describe the WPF (Windows Presentation Foundation) desktop application GUI – manipulate with elements by using parser [14] and also provide support for lay-outing;
5. HTML extension or web client GUI extension –by using XPath [15];
6. Javascript methods for web client – by using Jint parser [16];
7. SQL database extension – generating T-SQL code for new tables and changing the existing code using TSQL parser [17].

VI. CODEGEN USER INTERFACE

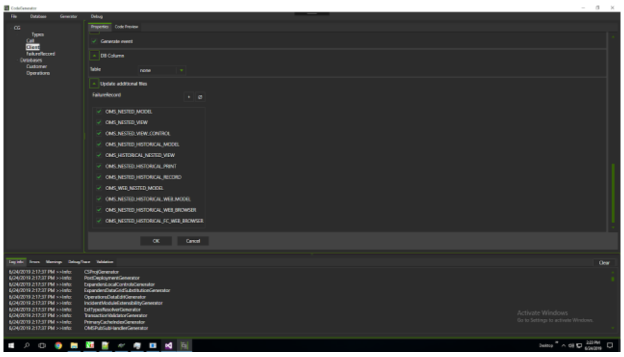
Figure 7 shows the graphical user interface (GUI) of CG when creating a blank project.



```
Name             ClientName
Description      Client name
Label            Client name
Existing:
Editable:
Type:            String
Collection type  Single
Model code:      OMS_SDP_EXT_CLIENT_NAME
Code value:      c121a000006ff07
✓ Generate view
✓ Add label when disabled
✓ Add border

6/24/2019 2:17:37 PM >>Info:   CSProjGenerator
6/24/2019 2:17:37 PM >>Info:   PostDeploymentGenerator
6/24/2019 2:17:37 PM >>Info:   ExpandersLocalControlsGenerator
6/24/2019 2:17:37 PM >>Info:   ExpandersDataGridSubstitutionGenerator
6/24/2019 2:17:37 PM >>Info:   OperationsDataEditGenerator
6/24/2019 2:17:37 PM >>Info:   IncidentModuleExtensibilityGenerator
6/24/2019 2:17:37 PM >>Info:   ExtTypesResolverGenerator
6/24/2019 2:17:37 PM >>Info:   TransactionValidatorGenerator
6/24/2019 2:17:37 PM >>Info:   PrimaryCacheIndexGenerator
6/24/2019 2:17:37 PM >>Info:   OMSPubSubHandlerGenerator
```

Figure 7 CG GUI

We can see on the left a tree view of the project structure, which shows which documents are being expanded in the project, as well as the core classes included in the project. On the right is a workspace. Activation of the Properties button, depending on the selected item in the tree, displays the attributes that the user can change to fit the requirements. At the top, we see that it is possible to select which section of the system we want to extend, whether it is a desktop application, web, database extension or configuration. This allows the user to iteratively extend the elements of the system according to the needs of the client and their development team.



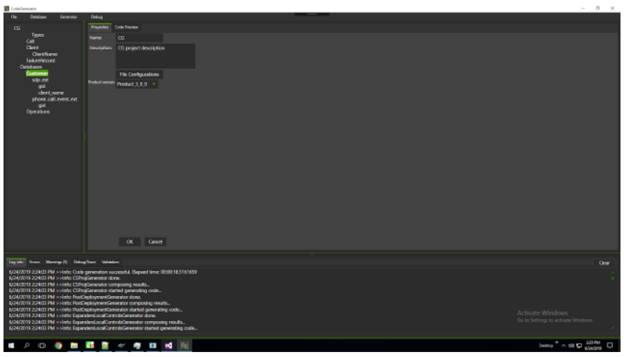
```
✓ Generate event
Table            none
FailureRecord
✓ OMS_NESTED_MODEL
✓ OMS_NESTED_VIEW
✓ OMS_NESTED_VIEW_CONTROL
✓ OMS_NESTED_HISTORICAL_MODEL
✓ OMS_HISTORICAL_NESTED_VIEW
✓ OMS_NESTED_HISTORICAL_PRINT
✓ OMS_NESTED_HISTORICAL_RECORD
✓ OMS_WEB_NESTED_MODEL
✓ OMS_NESTED_HISTORICAL_WEB_MODEL
✓ OMS_NESTED_HISTORICAL_WEB_BROWSER
✓ OMS_NESTED_HISTORICAL_FC_WEB_BROWSER

6/24/2019 2:17:37 PM >>Info:   CSProjGenerator
6/24/2019 2:17:37 PM >>Info:   PostDeploymentGenerator
6/24/2019 2:17:37 PM >>Info:   ExpandersLocalControlsGenerator
6/24/2019 2:17:37 PM >>Info:   ExpandersDataGridSubstitutionGenerator
6/24/2019 2:17:37 PM >>Info:   OperationsDataEditGenerator
6/24/2019 2:17:37 PM >>Info:   IncidentModuleExtensibilityGenerator
6/24/2019 2:17:37 PM >>Info:   ExtTypesResolverGenerator
6/24/2019 2:17:37 PM >>Info:   TransactionValidatorGenerator
6/24/2019 2:17:37 PM >>Info:   PrimaryCacheIndexGenerator
6/24/2019 2:17:37 PM >>Info:   OMSPubSubHandlerGenerator
```

Figure 8 Selection of parts of documents within a reference document

By selecting the Client document, we can add a new field to it. As it is shown in the previous picture, the user can specify the name, description and a label that will later be mapped to the GUI of the generated product. In addition, parts of the referenced document, are shown in the **Figure 8**.

By selecting a Part, the CG knows where to insert the code to make the extension visible within another document. The same goes for the web application.

After extending the model, it is necessary to perform object-relational mapping to make the changes visible in the database (**Figure 9**). This completes the project extension.



Name: CG
Description: CG project description
Product version: Product_3_8_0

```
6/24/2019 2:24:03 PM >>Info: Code generation successful. Elapsed time: 00:00:18.5161659
6/24/2019 2:24:03 PM >>Info: CSProjGenerator done.
6/24/2019 2:24:03 PM >>Info: CSProjGenerator composing results...
6/24/2019 2:24:03 PM >>Info: CSProjGenerator started generating code...
6/24/2019 2:24:03 PM >>Info: PostDeploymentGenerator done.
6/24/2019 2:24:03 PM >>Info: PostDeploymentGenerator composing results...
6/24/2019 2:24:03 PM >>Info: PostDeploymentGenerator started generating code...
6/24/2019 2:24:03 PM >>Info: ExpandersLocalControlsGenerator done.
6/24/2019 2:24:03 PM >>Info: ExpandersLocalControlsGenerator composing results...
6/24/2019 2:24:03 PM >>Info: ExpandersLocalControlsGenerator started generating code...
```

Figure 9 Object-relational mapping

VII. CONCLUSION

This paper presented a meta-model and associated code generator architecture designed to automate the development of mutually dependent documents within software product lines. One of the major challenges was the need to support development of existing projects, as well as newly created ones, which made necessary to use various types of extension mechanisms, usually not present in MDSD solutions:

- Direct manipulation of hand-written code using different kind of parsers.
- Collaborative work of CG and developers on the same software artifacts.

We plan future development in two directions:

1. To incorporate parsers for more languages to support development of SPL solutions based on different platforms.
2. To enable developers to create their own code generation templates, in order to enhance CG customizability.

REFERENCES

[1] Shapiro, R. J. (2014). The us software industry as an engine for economic growth and employment. *Georgetown McDonough School of Business Research Paper*, (2541673).

[2] Voelter, M., & Groher, I. (2007, September). Product line implementation using aspect-oriented and model-driven software development. In *11th International Software Product Line Conference (SPLC 2007)* (pp. 233-242). IEEE

[3] Agile project management manifesto https://medium.com/@sudarhtc/agile-project-management-methodology-manifesto-frameworks-and-process-f4c332ddb779 Web 5th April 2020.

[4] Urli, S., Blay-Fornarino, M., & Collet, P. (2014, September). Handling complex configurations in software product lines: a tooled approach. In *Proceedings of the 18th International Software Product Line Conference-Volume 1* (pp. 112-121).

[5] Software product lines presentation https://pt.slideshare.net/pagsousa/software-product-lines/14 Web 5th April 2020.

[6] Brambilla, M., Cabot, J., & Wimmer, M. (2017). Model-driven software engineering in practice. *Synthesis lectures on software engineering*, *3*(1), 1-207.

[7] Schwägerl, F., & Westfechtel, B. (2017, February). Perspectives on combining model-driven engineering, software product line engineering, and version control. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems* (pp. 76-83).

[8] Dashofy, E. M., Van der Hoek, A., & Taylor, R. N. (2002, May). An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 24th international conference on Software engineering* (pp. 266-276).

[9] Gordana Milosavljević, Željko Vuković. (2016). ReingIS: A Toolset for Rapid Develpoment and Reengeneering of Bussines Systems. In *ICIST 2016*.

[10] Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Nazari, P. M. S., Müller, K., ... & Rumpe, B. (2015, February). A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)* (pp. 74-85). IEEE.

[11] Models and meta-models, https://www.transentis.com/methods-techniques/models-and-metamodels/ Web 5th April 2020.

[12] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods Web 5th April 2020.

[13] Roslyn – C# parser, https://archive.codeplex.com/?p=roslyn Web 5th April 2020.

[14] Xml parser, https://github.com/KirillOsenkov/XmlParser Web 5th April 2020.

[15] XPath for html, https://docs.microsoft.com/en-us/dotnet/standard/data/xml/select-nodes-using-xpath-navigation Web 5th April 2020.

[16] Jint parser, https://archive.codeplex.com/?p=jint Web 5th April 2020.

[17] TSQL parser, https://docs.microsoft.com/en-us/dotnet/api/microsoft.sqlserver.transactsql.scriptdom.tsqlparser?view=sql-dacfx-140.3881.1 Web 5th April 2020.