Development and Maintenance of Model-Oriented Software with Visualization – Exploratory and Experimental Study

Thiago Gottardi[1], **Rosana T. Vaccare Braga**[1]

[1]Institute of Mathematics and Computer Sciences – University of São Paulo
P.O. Box 668 – São Carlos – São Paulo – Brazil

{gottardi,rtvb}@icmc.usp.br

Abstract. *Several development methods employ models throughout the software life-cycle, such as: Unified Process, Model-Driven Engineering, Model-Oriented Programming and Models at Run-time. In previous works, we have studied how to generalize concepts from these methods into a generic Model-Oriented (MO) Software paradigm. In this paper, we present new visualization tools and an exploratory study to discuss new opportunities that arise from the MO software concepts, e.g., representing code and run-time data as diagrams. We also include a preliminary experiment where developers employ diagrams created by these tools to perform maintenance tasks. As results, the participants managed to perform the tasks correctly. We conclude that diagrams can be used to inspect the Model-Oriented Software and show how object diagrams can be made useful according to agile modeling principles. The experiment also indicates that visualization tools could be used for MO software maintenance.*

1. Introduction

Modeling languages and tools allow to represent software artifacts and to improve their comprehension by humans [Ambler, 2002; Brambilla et al., 2017]. The availability of Unified Modeling Language (UML) modeling tools has fostered the software development according to the Unified Processes (UP) [Larman, 2005]. Indeed, these processes include instructions to developers on how different kinds of UML diagrams could be employed. In the UP life-cycle, object diagrams can be used by the development team to express the software dynamic behavior. However, agile practitioners use models on the design phase, while object diagrams are seen as unnecessarily complex [Ambler, 2002].

Model-Driven Engineering (MDE) allows to create software using models as the main development artifact [Brambilla et al., 2017]. Model-Oriented Programming (MOP) is a software programming paradigm that was created by employing MDE concepts into a new programming language. It allows to further tighten the gap between code and modeling, by using a model-oriented programming language e.g., Umple. It allows programmers to write code without losing semantics from the design models, as well as rapid prototyping thanks to round-trip generation tools Badreddin et al. [2012]. Models at Run-time systems are systems that employ software models during run-time. These systems are being tested as a solution for self-adaptive and self-aware systems that are capable of self-healing and automatic integration [Aßmann et al., 2014].

After tools for Model-Driven Engineering (MDE) were developed, e.g., Eclipse Modeling Framework[1] (EMF), opportunities appeared for studying how modeling tools

[1]https://eclipse.org/modeling/emf/

could be employed for software development besides their original intent. After conducting a secondary study on this context [Gottardi and Braga, 2018], we have identified the opportunity of further extending MO Programming with Models at Run-Time concepts into a broader software development paradigm that we call Model-Oriented (MO) Software. In this paradigm, abstractions are generalized, virtually eliminating the gap between: **(1)** code and models; **(2)** run-time objects and models.

In this paper, we explore how this gap reduction allows developers to visualize and manipulate code and data by using modeling tools interactively during development and execution. We include an exploratory study along with a preliminary experimental study on MO software maintenance by using diagrams generated by these tools. The contributions of these studies show how MO software development could foster constant visualization for classes and objects. In this manner, UML models can be used to inspect the software and finally make UML object diagrams more useful for development.

This paper is structured as follows: In Section 2, related works are cited and compared to ours. Section 3 contains MO concepts and discusses how they were employed to support visualization as an exploratory study. In Section 4, we present an experimental study to verify if the diagrams generated by one of the visualization tools is useful during MO software maintenance tasks. Finally, Section 5 contains the conclusions and prospections for future works.

2. Related Works

Modeling tools have been developed for supporting Model-Based and Model-Driven software development practices. Instances of tools for model comparison and modeling tool construction frameworks have been reported in the literature [Brambilla et al., 2017]. We have also conducted a secondary study with the intent of listing related tools and approaches [Gottardi and Braga, 2018].

Since this work is related to Model-Oriented Programming, it is worth mentioning the set of tools created for Umple coding, which allows programmers to transform code into UML models and back at any time throughout development cycle without semantic loss. Following the creation of Umple, Badreddin et al. [2012] have also conducted an empirical study on the comprehension of UML, Umple and Java. Afterwards, they discussed how Java had the worst comprehension ratios among these languages. Their study is the most related work presented in this section. In comparison, in our work, we propose to create a broader view of model-orientation that is not tied to programming languages, making it platform independent while complying to existing modeling tools. Instead of evaluating comprehensibility of artifacts, we conducted a study that involved coding activities with the intent of verifying if there are benefits of employing the visualization tools in practice. In our work, we are also interested on visualizing dynamic models, showing objects from the run-time application, which is similar to Models@Run.time approaches [Aßmann et al., 2014], e.g., Kevoree[2].

The difference between our study and the related studies is that we are focusing on studying how MO software could be visualized dynamically and statically. The MO principles allow us to generate model representations of the software, which can be visualized

[2]http://kevoree.org/

as diagrams, texts or other artifacts for improving human comprehension. Therefore, in this paper we discuss a exploratory study on how MO principles could be supported by visualization tools. We also present an experimental study assessing how this visualization could help programmers to write code for MO software.

3. Model-Oriented Software Development and Visualization

Model-Oriented Software is a hypothetical software category that blends principles from both Model-Oriented Programming and Models at Run-time, along with an MDE method for building software. From the perspective of developers, a MO system can be developed by using modeling tools. It includes optional support for round-trip software engineering, while avoiding semantic loss, thanks to its MOP properties. From the perspective of users, a MO system is configured by using models and runs by using models thanks to its Models at Run-time properties [Gottardi, 2018].

MO software was planned as a platform-independent definition for software. This was considered to avoid forcing developers to use a new language or platform. Therefore, it was intended to provide a set of requirements that would fit existing languages, libraries, frameworks and tools. This condition makes MO software more generic than MO programming, which depends on programming language constructions.

MO software handles models at run-time and includes reflection support. This allows the system to be configured by these models and to store data to serve as parametric configuration of the system. The state of the systems can also be stored as a model. Furthermore, this generalization empowers the developers to utilize modeling tools to edit and visualize the configuration and the execution. It is arguable that this support for visualization could also be beneficial to end-users who wish to understand the data of the system, i.e., not only developers.

As the data is stored as models, the transmission of data among modules of the system could also be carried by models. This suggests that the system could be built by using a Domain Specific Language (DSL) [Brambilla et al., 2017] that has been optimized to properly represent the domain concepts of the MO software application. Models can be alternatively represented in a different concrete syntax, bridging the gap between the data that is handled by the machines and the humans who wish to read this as graphical or textual information. When comparing MO software to MDE, the main difference to the end-users is that the final software developed with MDE methods may be completely unrelated to modeling techniques, while the resulting MO software is always based on models because it employs models at run-time.

Table 1. Basic Comparison Between MDE and MO Software

Level	Name	MDE	MO Software
M3	Meta-metamodel	Language for Metamodeling	Language for Metamodeling
M2	Metamodel	Language Specification	System Design
M1	Model	Development Artifact	Data File
M0	Code	Real World Objects	Program or Model Interpreter
M-1	Data Instance	–	Models in Memory

Table 1 compares MDE versus MO basic concepts. While M3 level is common, other levels are shifted. For example, while in MDE level M2 refers to metamodels, in

MO it is used as the system design. The design model of a MO software is mapped to a metamodel, which can be visualized graphically as a diagram at any moment throughout the life-cycle. This differs from how meta-levels are described within the MDE literature [Brambilla et al., 2017]. This allows MO to employ models (M1 and M-1) as data. Therefore, data can be visualized as diagrams or exported to model files to be manipulated by modeling tools. MO software also employs model interpretation besides code generation (M0) to increase flexibility, as in Models at Run-time applications.

3.1. MO Software Visualization: A Retail System Example

In this section, an example of an online retail system is described to illustrate MO systems. This system was developed as a web services system composed by a server and a client. The client manages the cart of the customer, while the server stores the product information, collects the final cart managed by the client and executes the final checkout.

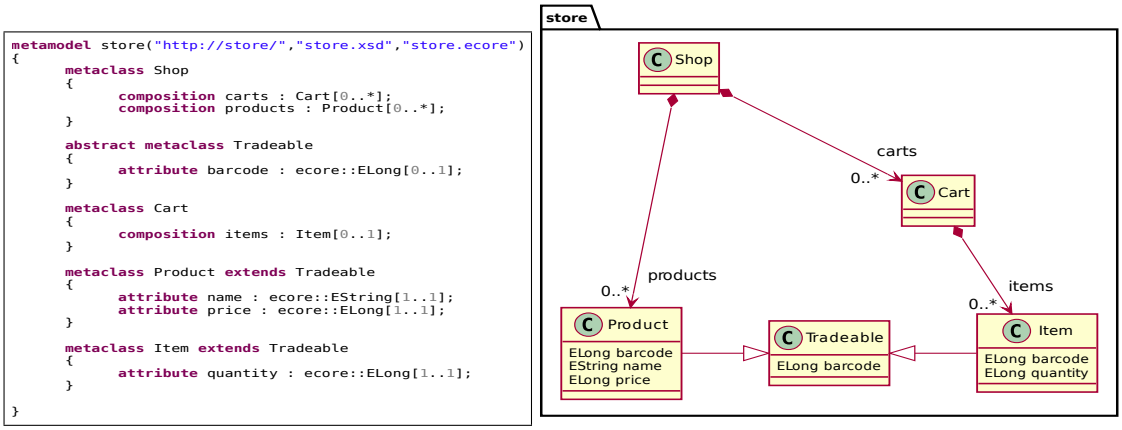

Figure 1. Cashier Application Static Development Models

As presented in Figure 1, the design of the data types is represented as a metamodel, i.e., the static portion of the software that should not change at run-time. The figure (left side) contains a textual representation using a definition language created within this project. Since this language is similar to KM3 (part of EMF), describing its constructs is not the focus of this paper. The figure also includes a class diagram representation (right side), which is generated by the tools created within our project.

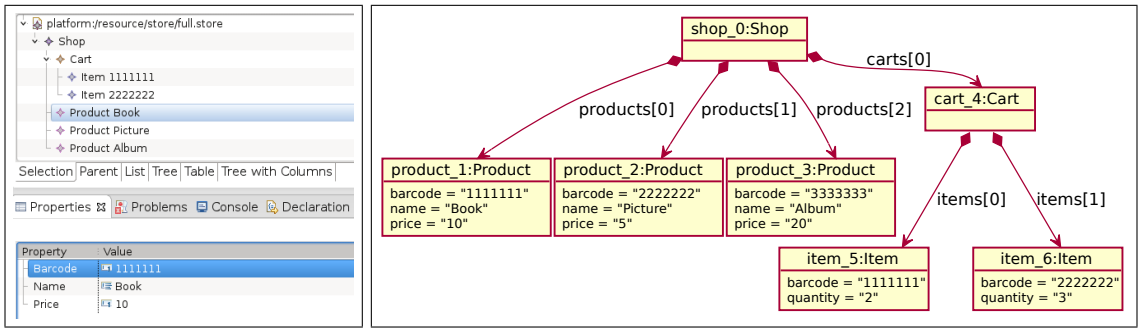

Figure 2. Cashier Application Dynamic Execution Models

The classes are divided into Shop (which stores the cart data and product data); Carts (which stores the items selected by customers); Products (which are the registry

of products sold by the shop); and Items (which are the quantifier objects of products); *Tradeable* is the generalization for the "barcode" attribute. Figure 2 contains the dynamic portion of the sample MO software. The run-time data is represented graphically using model editor tools (left) and diagrams (right).

3.2. Tool Set for Model-Oriented Development and Visualization

A set of cooperating tools has been developed for assisting developers who wish to build MO software systems. All referenced tools are functional[3]. We have also developed diagram generators for class and object models. As MO software has both its static and dynamic representation mapped to models, these tools allow developers to inspect the software at any time during development or run-time. The class diagram generator can also be used in conjunction with the round-trip strategy of metamodel creation using both ECore and the proposed definition language, allowing quick visualization. When using different versions of the design model, it is possible to generate a diagram that represents the changes between them. For example, according to the design represented in in Figure 3, changes were made to add a "Customer" class (highlighted). These diagrams have been employed in the experimental study described in Section 4.

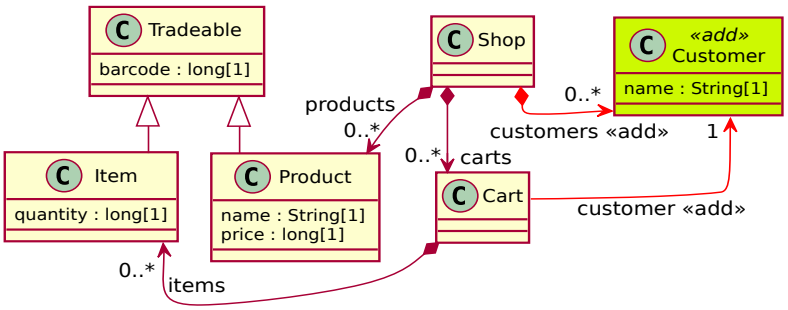

Figure 3. Cashier Application Static Maintenance Diagram

The object diagram generator is an advanced tool capable of dynamically decoding the metamodel. It interprets the also dynamic model instance from the MO software run-time data and then generates an object diagram for any MO software. This tool reads the metamodels and parses their definitions to seek the classes and their relationships that are referenced by the model instance dynamically. This was necessary to provide a tool that is completely metamodel independent, i.e., it can generate object diagrams from any MO run-time data. Our tools are compatible to EMF and also make use of PlantUML[4] to output the object diagram into image files (e.g., Figure 2).

It is also worth mentioning other development tools that generate MO software code. Since ECore is a metamodel language devised as part of EMF, we have developed tools to transform our metamodel definition language into ECore metamodels and back (round-trip method). The resulting ECore artifact is created to represent the metamodel specified by the definition language code. This metamodel is intended to be used as the data types of the MO system. It is eventually converted to data structures in programming language and XSD to represent the XML structure. This XSD is used to represent the

[3] http://tiny.cc/mo-tools
[4] http://plantuml.com/

model instance handling when using the XMI (XML format) standard[5]. The XSD generation tool creates an XSD instance without using ECore as intermediate language. Code generation tools for Java and C++ were also developed. They generate model parsing and handling modules to assist the development of MO software that is fully compatible to standard XMI model editors.

4. Experimental Study on Model-Oriented Software Maintenance

This section presents an experimental study conducted to evaluate the usage of a design diagram as instruction for the maintenance task of MO software systems. This study also allows to compare the efforts of using Java versus the proposed textual language for these maintenance tasks. It is expected that the outcome of this study would elucidate if the diagram generated by our tool is sufficient for instructing the developer. The comparison allows to identify which method takes less effort. Experiment packing is available[6].

4.1. Method

In this experiment, the maintenance of portions of a MO software is considered using different development methods. The experiment was applied in the context of graduate students properly trained in software development activities. The study was carried out from the perspective of the researchers, however, it was also intended to cover the expectations from software project managers.

The object of this study is the effort to perform a maintenance task on MO software while using the provided diagrams for instructions. The object was treated with two different development methods. While both treatments employed the same diagrams, they varied in the text language used for coding. The first method is based on writing Java code, while the second method involves our custom metamodel definition language (discussed in Section 3). The participants received the instructions in print, therefore, they could not use the computer to search or copy the content of the diagrams in the instructions. The diagrams within the maintenance task instructions were generated by our MO static visualization tool (example in Figure 3) for four different applications (different names for classes and relationships but in equal quantities).

The first dependent variable captures the time for the Java treatment, while the second variable captures the time for our custom metamodel definition language. Both variables indicate the time taken to complete the given task correctly. Participants had a maximum of 30 minutes for each task and were reserved the right to quit if they believe that they would not be able to finish.

4.2. Results

Next, we present the raw data collected from the study executions. The elapsed time data for completing each implementation task is presented on Table 2. The raw timings data was collected automatically in seconds (with millisecond precision), however, the data is presented as minutes and seconds on this table for readability.

Each row of the table represents a participant. The table allows to compare the data of Java and modeling tasks in pairs: data from the first task using Java coding (Java 1)

[5] https://www.omg.org/spec/XMI/About-XMI/

[6] http://tiny.cc/mo-maintenance-pack

/ data from the same task using the proposed modeling tool (Model 1); and data for the second task using Java and the proposed modeling tool (Java 2 & Model 2).

It is important to remind that all tasks were designed to have the same complexity, however, the experience gain from participants could help them to complete the second task faster. Finally, the last column indicates the group of the participant (Group): Group 1 participants performed the task in order Java 1, Model 1, Java 2, Model 2, while Group 2 participants performed the task in order Model 1, Java 1, Model 2, Java 2.

Table 2. Maintenance Task Timings

Participant	Java 1	Model 1	Java 2	Model 2	Group
1	04m59s	03m39s	01m08s	00m33s	1
2	28m18s	02m00s	27m57s	02m48s	1
3	08m51s	15m19s	28m12s	02m25s	1
4	05m02s	01m26s	02m14s	01m33s	2
5	09m41s	04m13s	12m09s	03m16s	2

Statistical T-Test was calculated for treatment comparison. The computed t-value is -2.341229, while the p-value is 0.04393179. Therefore, the probability of the Null hypothesis being true is below 5%, thus favoring the alternate hypothesis. As the estimated mean is negative, it is suggested that the modeling tool usage was beneficial for the tasks. Besides this comparison, all participants managed to finish the tasks during the allotted time, suggesting that the provided diagrams (e.g., Figure 3) were sufficient for their tasks.

4.3. Threats to Validity

Internal validity: The different levels of knowledge of the participants could have compromised the data. To mitigate this threat, the participants were thoroughly trained before the study tasks. Different computers and configurations could have affected the recorded logs. However, participants worked by using the computers with the same make and model in the same room and at the same time.

Validity by construction: The participants' expectations could have affected the results. To mitigate this threat, we have collected as much data as possible and asked the participants to perform as natural as possible. Also, we have concealed the objective of the experiment to avoid them from actively affecting their data towards a specific result.

External validity: It is possible that the exercises were not accurate for the real world. The experimental systems had simple requirements and their scalability was not evaluated. To mitigate this threat, we designed the exercises based on functional software inspired by real world applications.

Conclusion validity: The data collection and measurements precision could have affected the study. To mitigate this threat, all data was captured automatically as soon as the participants concluded each activity in order to allow better precision; Since we have a small population, we applied T-Tests to analyze the experiment data statistically to avoid the issues with low statistic power. Besides that, we are working on larger scale experiments and applications for the proposed methods.

5. Conclusions and Future Works

In this paper, we presented how Model-Oriented software could be visualized thanks to its concepts that map both the code and data into models. The employed tools allow to visualize the models as text and diagrams. These tools have been explained as part of an exploratory study. This exploratory study also included an example of MO software, which was used as a basis for a experimental study. In this experiment, the participants were asked to perform a maintenance task on MO software. The results indicate that every participant could finish within the allotted time, i.e., all 20 maintenance task attempts were successful. According to statistical testing, the proposed textual language also increased their productivity. Despite its preliminary status, results suggest that the diagrams created by our tools can instruct the developers to carry their task. Therefore, we conclude that the presented studies are references for further studies and tool developments.

We are working on new tools for MO development and visualization. This includes new compilers, code generators and diagram editors. Among tools for class declarations, new tools on verification and testing are under development. It is also worth mentioning a development tool for MO software that employs the object models for code generation. In conjunction to the existing class and object model visualization, this tool allows to automatize data handling and instantiation, which is being considered as part of a new experimental study on MO software testing and debugging. We also intend to verify the impact of version control on the artifacts of MO software, analyze their impact on large scale projects as well as the efforts of teaching this paradigm to students.

Acknowledgements

This work is derivative from projects funded by CAPES (DS-8428398/D, BEX 3482-15-4) and FAPESP (2016/05129-0). Special thanks to Lilian Passos Scatalon, whose suggestions have been incorporated to the presented studies. Many thanks to study participants.

References

Ambler, S. (2002). *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. Wiley.

Aßmann, U., Götz, S., Jézéquel, J.-M., Morin, B., and Trapp, M. (2014). *A Reference Architecture and Roadmap for Models@run.time Systems*, pages 1–18. Springer International Publishing, Cham, Switzerland.

Badreddin, O., Forward, A., and Lethbridge, T. C. (2012). Model oriented programming: An empirical study of comprehension. In *Proceedings*, CASCON '12, pages 73–86, Riverton, NJ, USA. IBM Corp.

Brambilla, M., Cabot, J., Wimmer, M., and Baresi, L. (2017). *Model-Driven Software Engineering in Practice: Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers.

Gottardi, T. (2018). *A proposal for the evolution of model-driven software engineering*. PhD thesis, ICMC-USP. Advisor: Prof. Dr. Rosana T. V. Braga, 297 pages.

Gottardi, T. and Braga, R. T. V. (2018). Understanding the successes and challenges of model-driven software engineering - a comprehensive systematic mapping. In *Proceedings of CLEI 2018*. IEEE.

Larman, C. (2005). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Object-Oriented Design (Computer Science). Prentice Hall, 3rd edition.