2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)

Evaluating the Ability of Developers to Use Metamodels in Model-Oriented Development

Thiago Gottardi
Institute of Mathematics and Computer Sciences
University of São Paulo
São Carlos, Brazil
gottardi@icmc.usp.br

Rosana Teresinha Vaccare Braga
Institute of Mathematics and Computer Sciences
University of São Paulo
São Carlos, Brazil
rtvb@icmc.usp.br

***Abstract*—The applicability of models has evolved throughout the history of software engineering, from documentation, development and beyond. In this context, we study how to employ models for a common language shared by humans and computers. After studying a model-oriented development method for models at run-time systems, we have identified that this method would heavily rely on metamodels. Therefore, it is important to evaluate if developers are able to use metamodels in software development. In this paper we present a controlled experiment to evaluate the ability and efforts of professional and novice developers to effectively use metamodels. Participants of the experiment had access to newly created metamodeling definition tools, as well as standard Java code and UML diagrams in order to complete their tasks. Results indicate that the definition language was easy to be learned by experienced Java developers, who were able to comprehend metamodeling development artifacts without struggling with modeling concepts. We conclude developers would be able to adapt to new modeling concepts and tools as required by different systems that handle models at run-time.**

***Index Terms*—metamodeling, model-oriented software, experimental study, development tools, model comprehension**

I. Introduction

Models and modeling tools are used by software engineers to express how they think during software development. These models have led to the definition of model-based development processes. For example, the Rational Unified Process is a software development process model that employs the Unified Modeling Language (UML) to document concepts, fostering communication among the development team members [1].

Throughout the history of software engineering, new uses for models were added. In MDE (Model-Driven Engineering), models are used as the main development artifact to create software [2]. Besides employing models during development, Models at Run-time (MRT) are systems that employ software models during run-time. [3]. In another research effort, Model-Oriented Programming (MOP) was defined as a software programming paradigm based on MDE concepts. MOP further tightens the gap between code and modeling [4].

On top of employing models for software development and execution, it is possible to further extend the usage of models. By generalizing principles from MDE, MOP and MRT, there is the generic concept of Model-Oriented (MO) Software. A MO software can be developed by using modeling tools, includes optional support for round-trip software engineering, is configured by using models and runs by using models. For example, when applied to the Web Services (WS) domain, they give rise to Model-Oriented Web Services (MO WS) [5], a type of models at run-time system which allow machines to communicate by using the same models used by humans, thus creating opportunities for a common language for knowledge sharing, cognitive computing and natural language processing.

However, we have identified that the successes of developing MRT systems depend on the ability of developers to define and use metamodels. In this manner, new questions were raised regarding MRT system development effort and feasibility, since developers are not often familiar with models and metamodeling.

In this paper, we evaluate the ability of developers to comprehend metamodels compared to the usual artifacts defined throughout development, i.e. code (programming) and class design. This evaluation was conducted via an experimental study where we asked experienced developers to extract information from both a textual metamodel and a Java code.

This paper is structured as follows: in Section II, a background is presented on the concepts employed for the present study. In Section III, works related to this one are cited and discussed. In Section IV, the experimental assessment of textual metamodeling efforts is presented. In Section V there are the conclusions and prospection for future works.

II. Background

Models are actively used during development in MDE (Model-Driven Engineering). In this development method, models drive the software development and are the main artifact throughout the life-cycle. A Domain-Specific Language (DSL) is a language defined to closely represent concepts of a specific domain, which could be a software domain or some group of concepts that exist in the real world. In the context of MDE, using a DSL as a model allows developers to describe real world problems related to the established domain for generating a software solution, instead of coding in low level. Therefore, the gap between problem and solution is reduced [6]. Metamodels are often used to define a DSL [2].

Work derived from projects funded by CAPES and FAPESP.

2575-4475/19/$31.00 ©2019 IEEE
DOI 10.1109/MiSE.2019.00012

Model-Oriented Programming (MOP) is a software programming paradigm created by employing MDE concepts. It allows to further tighten the gap between code and modeling, which is attained by using a high level model-oriented programming language, as is the case of Umple, which allows programmers to write code without losing semantics from the design models, as well as rapid prototyping provided by round-trip model transformation and code generation tools [7].

Models at Run-time (MRT) systems are systems that employ software models during run-time. These systems are being tested as a solution for self-adaptive, self-aware and self-healing systems supporting automatic integration [3].

Model-Oriented Software blends principles from both Model-Oriented Programming and MRT 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, avoiding semantic loss from transformations, due to its MOP properties. From the perspective of users, a MO system is configured by using models and runs by using models facilitated by its Models at Run-time properties [5]. These systems can be built by creating meta-modeling specification. As these systems handle data by using models, they allow machine and humans to communicate by using domain-specific languages, which are built to improve the representation of the concepts from a specific domain [8].

III. Related Works

Modeling tools have been developed for supporting software development practices. Instances of tools for model comparison and modeling tool construction frameworks have been reported in the literature [2]. Further listing of similar frameworks and approaches were listed on a secondary study we have conducted as a previous work [9].

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 [7]. These authors have also conducted an empirical study on the comprehension of UML, Umple and Java [4]. 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 this 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. In our work, we are also interested on visualizing both metamodels and object models. This allows to show objects from the run-time application, which is similar to Models@Run.time approaches [3], e.g., Kevoree[1].

As related examples of comprehension study of UML diagrams: Scanniello *et al.* have aggregated experimental studies on how comprehension of UML diagrams differ from code [10]. Störrle conducted a study on finding the optimal layout and size of diagrams for comprehension [11]. In comparison to our study, we focus on metamodel comprehension. As for service oriented system development, we can cite a survey conducted to validate a technique for service oriented systems [12], although our work is focused on building MO systems.

As an example of related metamodeling language readability study, we can cite the work by Hinkel and Strittmatter, who created metrics for metamodel modularity perception. Their work includes an experimental study considering the modularity of 32 metamodels, which were rated by both using these metrics and by architects. Then, the authors compared the answers of the architects and the metrics results for validation [13]. In comparison to our work, they focus on existing metamodels for MDE, while we focus on metamodels for MO system development. However, it is worth mentioning that their metrics may be applied in further studies.

IV. Experimental Assessment

A. Objectives

An experiment was conducted with the objective of verifying whether developers are able to comprehend metamodels as design. The experiment compares the use of a textual metamodeling language based on KM3[2] in contrast to conventional Java code. The usage of a new language was planned to assess how participants would learn to use it (further details provided as study packing[3]). The participants are supposed to perform activities to read models and code in order to identify its semantics.

Our premise is that by evaluating the understanding of metamodels, we are indirectly evaluating the ability of developing metamodels for MO development. Therefore, the rationale behind comparing Java and textual metamodeling for data schema definition was beyond simply identifying whenever a design language or a programming language was preferable for activities, since this is required for metamodel definition.

B. Operation

1) Context Selection: The invitations to participate in the experiment were sent via different mail-lists, including professionals, undergraduate students and graduate students, effectively reaching 118 participants. The participants were openly invited, however, their profile was employed for selection. The basic profile requirement was to select participants with experience in programming. Still, the participants also had the right to quit at any time, which reduced the number of participants who completed the study.

The operation was carried out by employing online forms that did not reject participants without the basic selection requirements. This allowed 42 participants to complete the form, however, 34 were selected after discarding the participants who lacked any programming skill. Among these participants, there are: 15 professional software developers, 9 graduate students (computing) and 10 professors (computing).

[1] http://kevoree.org/

[2] http://wiki.eclipse.org/KM3

[3] http://tiny.cc/metamodel-study-pack

The list of selected participants, including occupation and experience, are shown on Table I. The participants were treated as anonymous, the provided participant numbers are just for identifying them inside this section.

TABLE I
PARTICIPANT LIST AND PROFILE

P	Occupation	Exp.(years)
1	Graduate Student	3
2	Graduate Student	6
3	Graduate Student	4
4	Graduate Student	5
5	Graduate Student	5
6	Graduate Student	2
7	Graduate Student	3
8	Graduate Student	10
9	Professional Developer & Systems Analyst	17
10	Undergraduate Student & Professional Developer & Systems Analyst	4
11	Professional Developer & Systems Analyst	8
12	Professional Developer & Systems Analyst	8
13	Professional Tester	4
14	Professor & Professional Developer & Systems Analyst	10
15	Professor	10
16	Software Support	6
17	Professor & Professional Developer	22
18	Software Support & Researcher	2
19	Professor	12
20	Professional Developer & Systems Analyst & Tester	20
21	Professional Developer & Systems Analyst & Software Support & Researcher	5
22	Software Support & Researcher	4
23	Professor	20
24	Professor	10
25	Professor	10
26	Professor	30
27	Professor & Researcher	26
28	Professor & Researcher	10
29	Systems Analyst	8
30	Teacher & Systems Analyst	7
31	Graduate Student	13
32	Professor & Researcher	12
33	Professor	20
34	Professional Developer & Teacher	6

2) Preparation: The participants received training material as part of the preparation.

3) Instrumentation: The participants were required to read the training material, which included all tasks and examples that illustrated how to identify classes, lists, relationships and match diagrams for both languages.

The instructions included only a vague description about the study objectives, thus avoiding the participants expectations from affecting the actual objective of the study. The participants were told that the study objective was simply to compare both languages, so they were not aware of our interest on verifying MO system development abilities. The training included examples of the artifacts to be used within the tasks. These examples included the diagram in Figure 1, the Java code in Figure 2, and the text metamodel code in Figure 3.

The instrumentation text included instructions on how to identify the classes, properties and relationships. For relationships and properties, there were proper instructions on how to identify each kind of multiplicity, i.e. [0..1] indicates 0 or 1; [1] indicates always 1; [1..*] indicates at least one (or more); [0..*] indicates any natural number.

Following the diagram, a design report presented the names

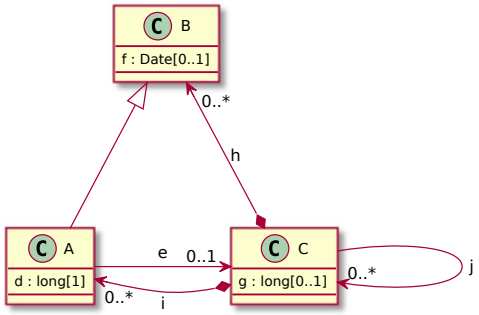

Fig. 1. UML Example used in training introduction.

A.java
```
package model;
class A extends B {
B d;
}
```
B.java
```
package model;
class B {
Date e;
List<C> f;
}
```
C.java
```
package model;
class C {
String g;
char h;
List<B> i;
List<B> j;
}
```

Fig. 2. Java Example used in training introduction.

model.mowsdl
```
metamodel model( "http://example.com/model" ,"model.xsd","model.ecore" ) {
 class A extends B {
  attribute d: long[ 1];
  reference e: C[ 0..1];
 }
 class B {
  attribute f: Date[ 0..1];
 }
 class C {
  attribute g: long[ 0..1];
  composition h: B[ 0..*];
  composition i: A[ 0..*];
  reference j: C[ 0..*];
 }
}
```

Fig. 3. Text Metamodel Example used for training introduction.

and count of Classes, Attributes (Properties), Compositions, References (Association links) and Generalizations (Inheritance), as shown on Table II.

TABLE II
DESIGN REPORT FOR CLASS DIAGRAM USED IN TRAINING

Classes	Attributes	Compositions	References	Generalizations
3 { A, B, C }	3 { d, f, g }	2 { i, h }	2 { e, j }	1 { A : B }

Semantics are the most important knowledge required to answer the questions of the tasks, since they are focused on identifying the design semantics of the languages. It is important to remind that specific language details were stripped from the study, which only focuses on visible and common semantics.

All classes, properties and relationships were randomly generated. These artifacts were generated in triples (UML class diagram, Java code and text metamodel) to represent the same intended semantics to be identified by participants, however, their declaration orders (i.e. the order in which classes appear on page) were changed, which does not impact the semantics.

Despite the randomness, the task order was fixed to avoid inserting new variables. The randomness was planned to make it harder to the participants to remember the previously used artifacts. Indeed, all participants to whom the researchers could talk to after the study did not even identify that the tasks had paired artifacts.

The first three tasks were focused on identifying elements. These tasks were focused on counting elements and were composed by eight sub-tasks: four Java artifacts paired with four text metamodel artifacts. Sub-tasks were shown in random order to avoid participants from finding out their similarities.

The first task involved identifying and counting classes. The second task was carried out to measure how the participants could identify lists. In the case of Java code, the participants had to identify the "List" (a Java Interface) usage on code. In the case of text metamodel, the participants had to search for [0..*] and [1..*] multiplicities.

In the third task, the participants had to identify the relationships that referred to a class type. In the case of Java, they had to identify the type name and find it among the classes. In the case of text metamodel, the participants could verify the type name or search for the usage of the reserved words "reference" and "composition".

The last task involved comparing the provided artifact against a UML class diagram. The participants had to read a single code written either in Java or text metamodel to identify the corresponding UML class diagrams. There were three UML diagrams for each code example, where exactly one of them was the correct answer.

At the end of each task, the participants were asked their opinion by using five ordinals: "Java was Much Better", "Java was a Bit Better", "Seem equivalent", "text metamodel was a Bit Better", "text metamodel was Much Better". The participants were also invited to provide qualitative feedback on the survey, as discussed in Subsection IV-E.

4) Execution: Initially, the participants had to accept a consent form and then answered a profile characterization form. The latter had questions regarding knowledge on programming, UML diagrams and database development. Afterwards, the participants were led to a general instruction for the training. Each task had a specific training session. Tasks were carried out in fixed sequence, however, the order of the sub-tasks were random.

5) Data Validation: The forms filled by the participants have been programmed to restrict the input data to valid values. Characterization and feedback forms also allowed the researchers to cross-examine the answers in order to find contradictions.

6) Data Collection: Since the study was performed via online forms, the data was captured on the submission of each page. This included their answers for characterization and tasks, as well as the timings required for each task. The study also included a qualitative feedback form, which is not used for the quantitative analyses.

C. Data

Table III contains the aggregate data collected from the study tasks. As the tasks are divided into four sub-tasks, J.A lists the correctness count result for the first task using Java, while M.A has the result for text metamodel. Each task contains two rows presenting aggregates for the raw data. Table IV contains the aggregate data for the timings required to complete each task (in seconds).

TABLE III
AGGREGATED ANSWERS DATA

Task	Task correctness (ratios)							
	J.A	M.A	J.B	M.B	J.C	M.C	J.D	M.D
Class	34/34	33/34	34/34	32/34	32/34	32/34	33/34	34/34
	100.00%	97.06%	100%	94.12%	94.12%	94.12%	97.06%	100%
List	33/34	16/34	32/34	14/34	33/34	18/34	33/34	13/34
	97.06%	47.06%	94.12%	41.18%	97.06%	52.94%	97.06%	38.24%
Relation	19/34	24/34	16/34	25/34	22/34	25/34	24/34	27/34
	55.88%	70.59%	47.06%	73.53%	64.71%	73.53%	70.59%	79.41%
Diagram	25/34	24/34	28/34	28/34	27/34	27/34	30/34	27/34
	73.53%	70.59%	82.35%	82.35%	79.41%	79.41%	88.24%	79.41%

TABLE IV
AGGREGATED TASK TIME

Task	Agg.	Task Elapsed Time (seconds)							
		J.A	M.A	J.B	M.B	J.C	M.C	J.D	M.D
Class	Avg.	16.594	17.032	16.794	16.333	26.304	25.713	21.119	17.419
	Med.	14.580	13.185	15.165	14.405	18.185	18.000	16.950	15.365
List	Avg.	12.063	16.916	12.465	17.766	13.668	18.169	12.425	13.985
	Med.	10.620	12.630	9.620	11.280	10.705	12.880	9.585	10.370
Relation	Avg.	21.061	15.262	17.589	17.385	15.153	14.823	14.919	14.731
	Med.	15.140	13.075	12.765	13.210	13.930	11.930	12.120	10.675
Diagram	Avg.	59.232	65.321	55.151	46.252	42.478	59.259	56.835	48.171
	Med.	51.135	44.415	35.920	42.980	31.500	51.100	38.530	39.935

The timings data for the class identification is also provided as a plot in Fig. 4. The bar sizes represent the allotted time in seconds. Each treatment task is divided into four sub-tasks (A, B, C, D), which are stacked to represent the total time for each participant.

In the same sense, the time data for the list identification task is also provided as a plot in Fig. 5. The bar sizes represent the allotted time in seconds and are stacked to represent the

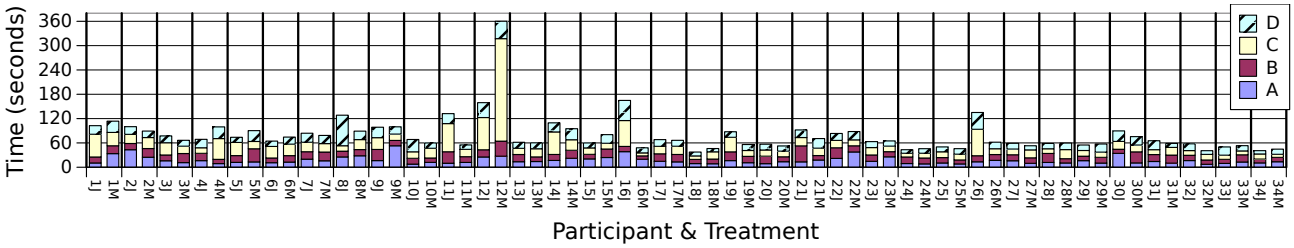

Fig. 4. Plot for Class Counting Task Time

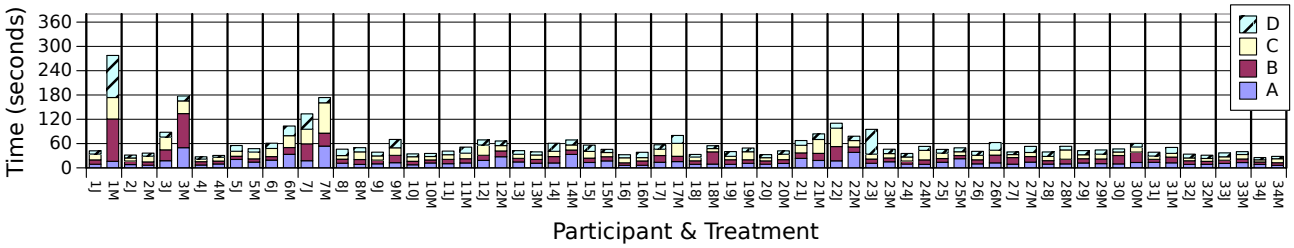

Fig. 5. Plot for List Counting Task Time

total time each participant took to complete the task including the four sub-tasks.

Similarly, the plot for the timings data for the relationship identification is provided in Fig. 6 and the plot for the diagram matching timings data is provided in Fig. 7.

D. Results

1) Class Identification: The input data for this test compares correct against incorrect answers. The hypotheses were statistically tested using paired tests, as shown on Table V. The considered null hypothesis is the "Equivalent", while the considered alternate Hypothesis is the "Java is more Accurate", since the sum of correct results for Java is greater. These results indicate that the significance of the difference between both treatments cannot be used to reject the null hypothesis. Therefore, it is likely that the languages are equivalent for this task.

TABLE V
HYPOTHESES TESTING FOR CLASS TASK CORRECT ANSWERS

Property	Value
Null Hypothesis	Difference is Zero;
Alternate Hypothesis	Difference is Negative;
Test type	Exact Wilcoxon Ranked Sign Test;
Test Statistic	13.5;
p-value	0.7265625;

Table VI contains the hypothesis testing results for the class identification timings. Since the estimated mean is negative, the alternate hypothesis is "text metamodel is Faster", which would imply that Java takes longer for this task. As usual, the null hypothesis is the zero difference, i.e. "Equivalent". The calculations for this test applied the paired t-test. The test results suggest that the probability for the null hypothesis is also high, as the previous comparison for class identification. Moreover, the participants answered an opinion form after each task. As shown on Table XI, over 80% believe that the languages were equivalent for the class task.

TABLE VI
HYPOTHESES TESTING FOR CLASS TASK TIMING

Property	Value
Null Hypothesis	Difference is Zero;
Alternate Hypothesis	Difference is Negative;
Test type	Paired T-Test;
Estimated Mean	-1.139545;
Degrees of Freedom	131;
t-value	-0.6329999;
p-value	0.5278371;

2) List Identification: Table VII contains the statistical testing results for the list identification task. It compares the correct and incorrect answers. The alternate hypothesis is "Java is more Accurate", since the correct answers for the Java artifacts were higher. According to statistical testing, the probability of the null hypothesis being valid is very low, favoring Java in this comparison.

TABLE VII
HYPOTHESES TESTING FOR LIST TASK CORRECT ANSWERS

Property	Value
Null Hypothesis	Difference is Zero;
Alternate Hypothesis	Difference is Negative;
Test type	Wilcoxon Ranked Sign Test;
Test Statistic	36.5;
p-value	$1.633055 \cdot 10^{-16}$;

Table VIII contains the statistical testing results for the timings to complete the list identification task. The alternate hypothesis is the Java is Faster, since the time required for

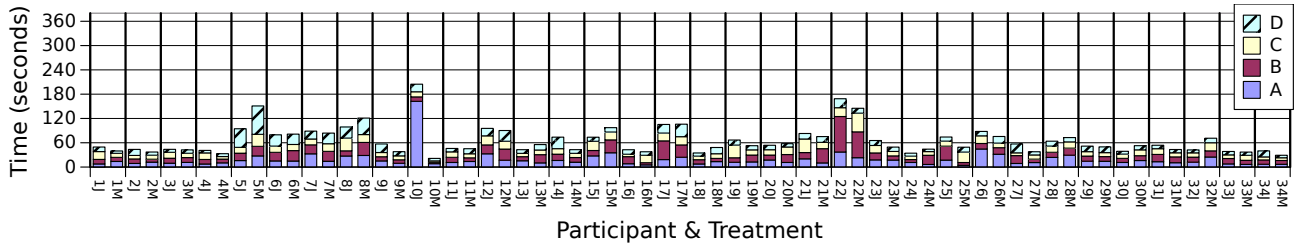

Fig. 6. Plot for Relationship Counting Task Time

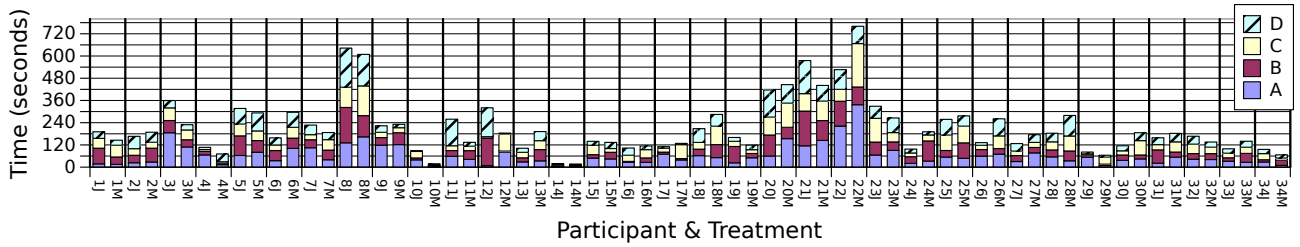

Fig. 7. Plot for Diagram Matching Task Time

TABLE VIII
HYPOTHESES TESTING FOR LIST TASK TIMING

Property	Value
Null Hypothesis	Difference is Zero;
Alternate Hypothesis	Difference is Positive;
Test type	Paired T-Test;
Estimated Mean	4.155303;
Degrees of Freedom	131;
t-value	2.973047;
p-value	0.003510379;

the text metamodel version was higher. The result for the paired t-test suggest a low probability for the null hypothesis, favoring Java in this study. The opinion of the participants after completing this task is also presented on Table XI, where 20 out of 34 believe that Java was better for list identification.

3) Relationship Identification: Table IX lists the results for the statistical testing used for the hypotheses of the relationship identification task. The "text metamodel is more accurate" is the alternate hypothesis, since the correct rate for the text metamodel was higher. This test returned a very low probability for the null hypothesis, suggesting that text metamodel has outperformed Java in this comparison.

TABLE IX
HYPOTHESES TESTING FOR RELATIONSHIP TASK CORRECT ANSWERS

Property	Value
Null Hypothesis	Difference is Zero;
Alternate Hypothesis	Difference is Positive;
Test type	Exact Wilcoxon Ranked Sign Test;
Test Statistic	387.5;
p-value	0.0003249142;

Table X contains the paired t-test results for the timings required to complete the relationship identification task. The alternate hypothesis is "Text metamodel is Faster", since the Java version took longer to complete in average. Despite the results favoring the text metamodel, the probability for the null hypothesis is too high to reject the null hypotheses.

TABLE X
HYPOTHESES TESTING FOR RELATIONSHIP TASK TIMING

Property	Value
Null Hypothesis	Difference is Zero;
Alternate Hypothesis	Difference is Negative;
Test type	Paired T-Test;
Estimated Mean	-1.598258;
Degrees of Freedom	131;
t-value	-1.143369;
p-value	0.2549705;

The results for the opinions answered by the participants for the relationship identification task are also on Table XI. This suggests a scattered opinion frequency unlike previous answers. 15 out of 34 of the participants indicated that the languages are equivalent. 12 out of 34 of the participants favored text metamodel while 7 out of 34 of the participants prefer Java for this task.

4) Diagram Matching: Table XII contains the hypothesis testing for the correct answers of the diagram matching task. The alternate hypothesis is "Java is more accurate" since the Java version had higher correct count. The probability for the null hypothesis is high for this study, which could indicate that the languages are equivalent.

The timings for identifying the diagram for the text metamodel version took slightly longer than Java. Therefore, the alternate hypothesis considered for the statistical testing shown on Table XIII considers the "Java is Faster". Following the same conclusion as the previous test, it is not possible to reject the null hypothesis.

TABLE XI
OPINION AFTER TASKS: FREQUENCY DISTRIBUTION

Answer Text Choice	Class		List		Relationship		Diagram Matching	
	Frequency	Percent	Frequency	Percent	Frequency	Percent	Frequency	Percent
Java is Much Better	2/34	05.88%	6/34	17.64%	1/34	02.94%	1/34	02.94%
Java is a Bit Better	1/34	02.94%	14/34	41.18%	6/34	17.64%	3/34	08.82%
Seem Equivalent	28/34	82.36%	14/34	41.18%	15/34	44.13%	19/34	55.89%
Text metamodel is a Bit Better	3/34	08.82%	0/34	00.00%	11/34	32.35%	7/34	20.59%
Text metamodel is Much Better	0/34	00.00%	0/34	00.00%	1/34	02.94%	4/34	11.76%

TABLE XII
HYPOTHESES TESTING FOR DIAGRAM MATCHING TASK CORRECT ANSWERS

Property	Value
Null Hypothesis	Difference is Zero;
Alternate Hypothesis	Difference is Negative;
Test type	Exact Wilcoxon Ranked Sign Test;
Test Statistic	143;
p-value	0.690038;

TABLE XIII
HYPOTHESES TESTING FOR DIAGRAM MATCHING TASK TIMING

Property	Value
Null Hypothesis	Difference is Zero;
Alternate Hypothesis	Difference is Positive;
Test type	Paired T-Test;
Estimated Mean	1.590455;
Degrees of Freedom	131;
t-value	0.4151639;
p-value	0.6787011;

The opinion answers for the diagram matching task are also on Table XI. Despite the Java version attaining a better result for the correct answers, only 4 out of 34 of the participants voted Java as a better option. Most participants think they are equivalent, however, 11 votes for text metamodel as better option repeats the result for the relationship counting.

E. Discussion

The purpose of this subsection is to discuss the experiment presented in this section. This experiment was planned to verify if developers could use metamodels as design, as required by MO software development. A textual metamodeling language was used to assist programmers while defining metamodels, with the intent of empowering them to create software to handle models, e.g. models at run-time.

The language was never made public prior to the experiment, which implies that this was the first contact the participants had with it. This suggests that developers could use the language without much training. Indeed, the participants had years of programming experience, they knew the Java Programming Language prior to the study and were able to use textual metamodeling after their first contact with the language. Therefore, it is important to point that this study could also provide insights on how programmers deal with new languages.

In the following task, metamodeling had positive results for relationship identification due to its visible declaration of relationships and compositions. Following the same logic, the explicit usage of "List" in Java also made it clear when Java code had lists. This was not clear for other tasks, which had very high probability for the null hypotheses. Metamodels are not meant to replace Java or any programming language, but they are important in MO development. Their usage is not mandatory, since it is possible to manually write MRT systems without using text metamodels. However, it is hard to assure that the resulting software would be able to handle MRT correctly without proper tools.

It is noteworthy that proper Java code to handle models as data also requires a few model annotations which could pollute the code and cause mistakes. This was not required for this study, which causes further advantages towards the programming language.

Besides these results, the study indicates that developing code for handling models as data could be a feasible task for common programmers. In this manner, another study is being conducted as part of future works to evaluate programming efforts while writing code to develop components for MRT systems.

Regarding the feedback provided as qualitative answers by the participants, there were several participants who praised how the forms were designed and their looks, which could have encouraged them to complete the form and share it to more participants. A few participants stated that the tasks were engaging and fun, making them curious for results. Several participants also wrote criticisms. They have pointed out that it was very cumbersome to answer the diagram matching task, which required them to go back and forth several times to compare the text to diagram. While some participants have found it engaging, others criticized its length and found it boring and tiring.

It is important to point out the critics on the apparent objective of the survey. The participants were led to believe that we created yet another definition language for data structures while adding some semantics that are lost when using Java. This is related to the introduction of Model-Oriented Programming [7], yet, it was not the point of this study. The participants' argument that it could be nonsensical to approximate design to code further confirms assumptions discussed after conducting a systematic mapping in this context [9].

Programmers might dislike learning new languages, which are often created to solve the same problems. The proposition of MOP suggests that programmers should use yet another language. This could be also applied to Model-Driven De-

velopment tools, which might be seen as just another tool for the same needs. Indeed, in the case of Java, annotations could be used to keep the required semantics to generate design models correctly. The annotations are also employed for MRT system coding when using Java for interface and data structures. This is related to the experience bias: the experienced Java programmers could feel more confident using Java, thus affecting their answers.

F. Threats to Validity

Internal validity: Varied experience level of participants could compromise the data. To mitigate this threat, the experiment tasks included a training session. Also, the participants were selected according to their experience. Different computers, devices and network connections could affect the logs. This threat was mitigated by requiring the participants to complete all the tasks without changing their device and connection, allowing to capture the control data in proportion to the treatment data.

Validity by construction: The participants could have expectations that could affect the results. Therefore, we asked them to complete answers in a steady pace regardless of the task. We have concealed the objective of the experiment and their impact on it to avoid them to actively affect their data towards a specific result.

External validity: The exercises might not be accurate for real world applications. The experimental application had simple requirements and random names. To mitigate this threat, the range of the number of objects and relations were based on the case studies. Still, the randomness was used to avoid the memory effect of participants, which would cause another threat to validity.

Conclusion validity: To avoid poor measure reliability, all data was captured automatically as soon as the participants concluded each activity in order to allow better precision; We applied statistical tests to analyze the experiment data, avoiding issues with low statistic power. We also plan to work on replications in the future.

V. Conclusion

In this paper, we discussed MO software and its dependency on metamodeling. Then, we presented an evaluation to verify if developers would be able to comprehend metamodels as required in model-oriented development. We presented an experiment that included professional developers, professors and graduate students as participants. They have analyzed text metamodel and Java artifacts and were able to use both artifacts to understand the semantics required by the study tasks. There were four evaluated activities, which have indicated that each language could be preferred for their specific completion. In most cases, the expected effort was very close in either language, even considering that the developers have never seen text metamodeling languages before and they were experienced Java developers.

These results suggest that textual metamodels could be used by developers to create MRT and related MO software without deep training, which indicates that the language and tools could be benefiting in this specific task. We also provide language specifications and experiment data as part of the study packing[4]. We claim that the presented study suggests how model-oriented development is feasible to be adopted by developers. As future works, we plan to extend the experiment with manual coding activities to evaluate the efforts required to employ model-oriented development methods. We also intend to release the created tools to be used and extended, as well as evaluate their usage outside MOP, e.g. MDE methods.

Acknowledgment

Thank you for reading and thanks the reviewers for all suggestions. Many thanks to all study participants. This work is derivative from projects funded by FAPESP (process 2016/05129-0) and CAPES.

References

[1] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual*, ser. Addison-Wesley Object Technology Series. Addison-Wesley Professional, 2005.

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

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

[4] O. Badreddin, A. Forward, and T. C. Lethbridge, "Model oriented programming: An empirical study of comprehension," in *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '12. Riverton, NJ, USA: IBM Corp., 2012, pp. 73–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=2399776.2399784

[5] T. Gottardi and R. T. V. Braga, "Model-oriented web services," in *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, March 2016, pp. 14–23.

[6] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.14

[7] A. Forward, O. Badreddin, and T. C. Lethbridge, "Umple: Towards combining model driven with prototype driven system development," in *Rapid System Prototyping (RSP, IEEE 2010)*, Fairfax, VA, United states, 2010. [Online]. Available: http://dx.doi.org/10.1109/RSP.2010.5656338

[8] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, February 2006. [Online]. Available: http://www.truststc.org/pubs/30.html

[9] T. Gottardi and R. T. V. Braga, "Understanding the successes and challenges of model-driven software engineering - a comprehensive systematic mapping," in *Proceedings of CLEI 2018*. IEEE., 2018.

[10] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, G. Tortora, M. Risi, and G. Dodero, "Do software models based on the uml aid in source-code comprehensibility? aggregating evidence from 12 controlled experiments," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2695–2733, Oct 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9591-4

[11] H. Störrle, "On the impact of size to the understanding of uml diagrams," *Software & Systems Modeling*, vol. 17, no. 1, pp. 115–134, Feb 2018. [Online]. Available: https://doi.org/10.1007/s10270-016-0529-x

[12] S. Suwisuthikasem and M. Samadzadeh, "Migration from legacy systems to SOA applications: A survey and an evaluation," *Advances in Intelligent Systems and Computing*, vol. 1089, pp. 609–614, 2015.

[13] G. Hinkel and M. Strittmatter, "Predicting the perceived modularity of mof-based metamodels," in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD*, INSTICC. SciTePress, 2018, pp. 48–58.

[4] http://tiny.cc/metamodel-study-pack