Using UniMod for Maintenance Tasks: An Experimental Assessment in the Context of Model Driven Development

Filippo Ricca, Maurizio Leotta, Gianna Reggio, Alessandro Tiso, Giovanna Guerrini, Marco Torchiano

Abstract:

One of the claimed advantages of Model-driven development is the improvement in maintainability. However, few studies consider this aspect from an empirical point of view.

This paper reports the results of a controlled experiment with 21 bachelor students aimed at investigating the effectiveness of Model-driven development during software maintenance and evolution activities. The tool used in the experiment is UniMod, a specific implementation of executable UML.

Preliminary results indicate a relevant shortening of time with no significant impact on correctness, gained through the use of UniMod instead of conventional programming (i.e., code-centric programming).

Digital Object Identifier (DOI):

`http://dx.doi.org/10.1109/MISE.2012.6226018`

Using UniMod for Maintenance Tasks: An Experimental Assessment in the Context of Model Driven Development

Filippo Ricca[1], Maurizio Leotta[1], Gianna Reggio[1], Alessandro Tiso[1], Giovanna Guerrini[1], Marco Torchiano[2]
[1] Dipartimento di Informatica e Scienze dell'Informazione (DISI) – Università di Genova, Italy
[2] Dipartimento di Automatica e Informatica (DAUIN) – Politecnico di Torino, Italy
{filippo.ricca|maurizio.leotta|gianna.reggio|alessandro.tiso|guerrini}@disi.unige.it, marco.torchiano@polito.it

***Abstract*—One of the claimed advantages of Model-driven development is the improvement in maintainability. However, few studies consider this aspect from an empirical point of view.**

This paper reports the results of a controlled experiment with 21 bachelor students aimed at investigating the effectiveness of Model-driven development during software maintenance and evolution activities. The tool used in the experiment is UniMod, a specific implementation of executable UML.

Preliminary results indicate a relevant shortening of time with no significant impact on correctness, gained through the use of UniMod instead of conventional programming (i.e., code-centric programming).

***Keywords*-Model Driven Development; UniMod; Controlled experiment; Maintenance tasks.**

I. Introduction

Model-driven development (MDD) is a software development methodology based on the concepts of model and model transformation [12]. In practice, higher-level models are transformed into lower-level models until the model can be made executable using either code generation or model interpretation. In this latter case, executable models are directly executed/interpreted by means of specific environments.

The main claimed advantages of MDD are improvements in productivity, portability, maintainability and interoperability [9]. Unfortunately, in the literature empirical studies — in particular, controlled experiments — evaluating MDD and considering these aspects are really rare.

Since we are interested in investigating the effectiveness of MDD, we applied empirical methods to better understand the MDD phenomenon. The first step was an Industrial Italian survey on adoption and maturity of software modelling and model-driven development [13], [14]. The second step consists of controlled experiments with students.

Obviously, not all the development methods and related tools can be experimented with respect to every software characteristic (e.g., portability, maintainability and interoperability). Thus, we had to select an instance among all the possible MDD proposals and to select a characteristic. Finally, after having consulted the literature looking for non proprietary MDD tools, we resort to UniMod [5] and maintainability as main software characteristic to study.

Thus, our main research question can be stated as follows: "*Is UniMod*[1] *effective during software maintenance tasks? And, if yes, how much is UniMod effective?*".

UniMod is a MDD method for designing and implementing object-oriented programs with explicit emphasis on the state concept. We chose it for three reasons: (1) the method relies on UML that is well-known by our students, (2) a MDD free tool[2] (available also as an Eclipse plug-in) based on this method exists, (3) the UniMod tool is simple to use and to install and thus suitable for students.

In this work, we compare UniMod programming (Eclipse IDE + UniMod plug-in) with respect to code-centric programming (Java language with the Eclipse IDE) in the context of maintenance tasks.

The paper is structured as follows. Section II presents UniMod. Section III provides details on the design of the experiment, including the definition of the experiment, context selection, hypotheses formulation, and instrumentation. Sections IV, V and VI present some preliminary results, threats to validity that could affect our results and related work, respectively. Section VII concludes the paper with final remarks and future research directions.

II. UniMod

UniMod is a MDD method for designing and implementing object-oriented programs. The method relies on a specific instance of automata-based programming (SWITCH-technology [11]) and on UML. A tool for the development and execution of UniMod models is available. The tool allows one to create, edit and execute UML class diagrams and state machines, which correspond to the core ingredients of UniMod: *connectivity schemas* and *transition graphs*. We will briefly explain them by using a personalized instance of the "Helloworld" program: it prints the string "Hello, World!" on the console for ten times, then it prints the string "Bye, World!" for ten times. The interested reader can read more about UniMod in [5]. In the remainder of this section we discuss connectivity schema and transition graph, and later we present the UniMod Eclipse plugin.

[1]I.e., our specific instance between all the available MDD proposals.
[2]http://unimod.sourceforge.net/

A. *Connectivity Schema*

A connectivity schema is a class diagram depicting classes with the following stereotypes: *EventProvider* (an event source), *StateMachine* (an automaton) and *ControlledObject* (a controlled object).

Between these stereotyped classes, directed associations of the following three types may exist:

1) from an event source to an automaton;
2) from an automaton to a controlled object;
3) from an automaton to another automaton.

Usually a connectivity schema is depicted as follows: the left part of the schema shows the event providers, the central part depicts one or more automata and the right part contains the controlled objects. The event providers are connected to the automata by means of UML associations and supply them with events. On the contrary, the automata are connected to the objects they control.

The event sources are active: they affect the automata by means of events. By contrast, controlled objects are passive: they perform actions when the automata call them (i.e., they offer some operations). Controlled objects can also be used as input for the automata. In UniMod, by convention, the operations of a controlled object used as input (i.e., returning a value) for the automata are named `xi` (with "i" ranging from `0` to `n`) while the operations used as output (i.e., changing the value of an controlled object's field) are named `zi`.

The automata are activated by the events and, depending on the values of the input variables, controlled objects are affected and/or transitions to new states are executed.

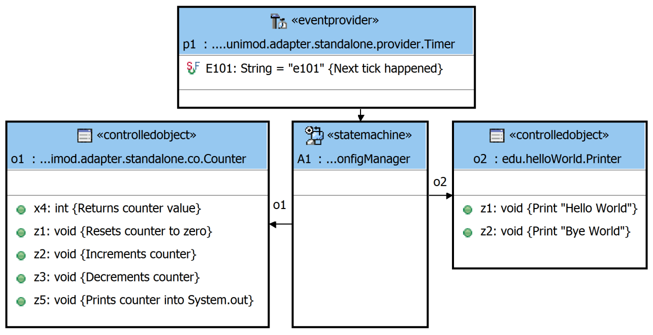
«eventprovider»
p1 : ....unimod.adapter.standalone.provider.Timer
E101: String = "e101" {Next tick happened}
«controlledobject»
o1 : ...imod.adapter.standalone.co.Counter
x4: int {Returns counter value}
z1: void {Resets counter to zero}
z2: void {Increments counter}
z3: void {Decrements counter}
z5: void {Prints counter into System.out}
o1
«statemachine»
A1 : ...onfigManager
o2
«controlledobject»
o2 : edu.helloWorld.Printer
z1: void {Print "Hello World"}
z2: void {Print "Bye World"}

Figure 1. Connectivity Schema for Helloworld

The connectivity schema of our running example is depicted in Figure 1. It contains: (1) an event provider `p1` (a timer) that sends an event (called `e101`) each second (a tick of clock) to the state machine `A1`, (2) a state machine (`A1`) described by the transition graph (see below) and (3) two controlled objects (`o1` and `o2`) used to count the iterations ("Counter") and to print the strings on the console ("Printer"), respectively. For example, `o1` provides five operations: the first (`x4`) is used as input for the automata while the others are used to affect the state of `o1` and/or print some messages on the console. A comment explaining the meaning of the operations is shown in the connectivity schema between brace parentheses.

B. *Transition Graph*

For each automaton, shown in the connectivity schema, a state machine, named transition graph in UniMod, needs to be created. That state machine can have simple and composite states, one initial state and one or more final states. Transitions between states have labels of the form: *event [guard] / actions* (all of them are optional). The square brackets contain a boolean formula, which is the condition of the transition firing (i.e., the guard condition).

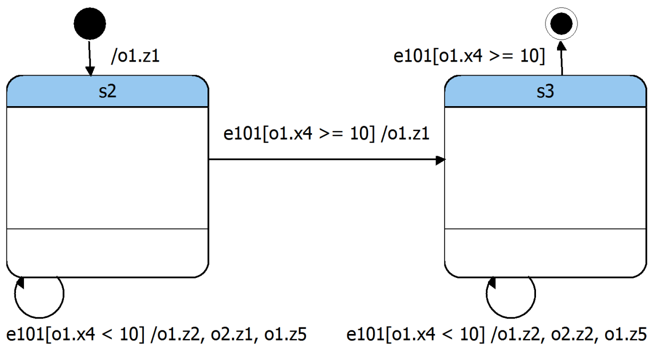
/o1.z1
e101[o1.x4 >= 10]
s2
s3
e101[o1.x4 >= 10] /o1.z1
e101[o1.x4 < 10] /o1.z2, o2.z1, o1.z5
e101[o1.x4 < 10] /o1.z2, o2.z2, o1.z5

Figure 2. Transition Graph for A1

The transition graph for `A1` is shown in Figure 2. It contains one initial state, one final state and two simple states (`s2`, `s3`): the program is in the first state (`s2`) when it is printing the string "Hello, World!" while it is in the second one (`s3`) when it is printing the string "Bye, World!".
For example, the label on the transition between the states `s2` and `s3`:

`e101[o1.x4>=10] /o1.z1`

has the following meaning (see comments between brace parentheses in Figure 1):

when 1 sec elapsed [*if* counter>=10] / reset counter

C. *UniMod Plug-In for Eclipse*

The UniMod tool is available also as Eclipse plug-in. The plug-in provides the following software components:

1) a graphical editor to create/edit the connectivity schema and the transitions graphs (on the contrary, event providers and controlled objects are directly edited with the Eclipse textual editor given that they are pure Java code);
2) an on-the-fly model validator that notifies to the user errors/warnings and suggests possible quick-fixes;
3) a launcher that allows to execute the UniMod model[3] in "one click".

[3]The UniMod model is composed by one connectivity schema and one or more transition graphs, event providers and controlled objects.

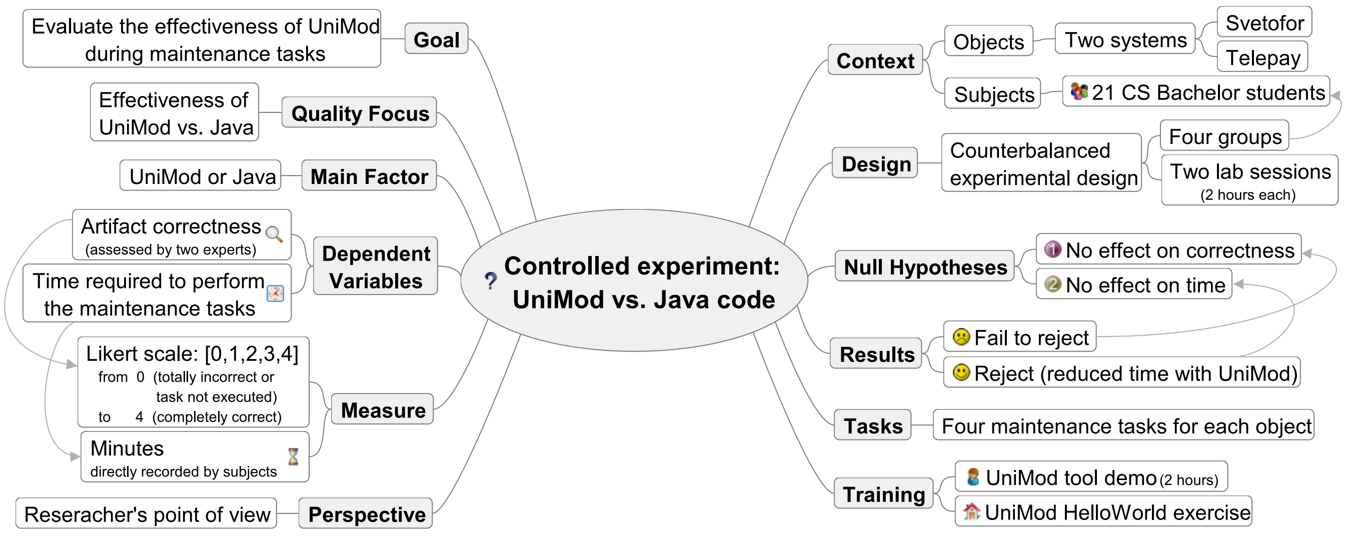
Controlled experiment: UniMod vs. Java code
Goal
Evaluate the effectiveness of UniMod during maintenance tasks
Quality Focus
Effectiveness of UniMod vs. Java
Main Factor
UniMod or Java
Dependent Variables
Artifact correctness (assessed by two experts)
Time required to perform the maintenance tasks
Measure
Likert scale: [0,1,2,3,4] from 0 (totally incorrect or task not executed) to 4 (completely correct)
Minutes directly recorded by subjects
Perspective
Reseracher's point of view
Context
Objects
Two systems
Svetofor
Telepay
Subjects
21 CS Bachelor students
Design
Counterbalanced experimental design
Four groups
Two lab sessions (2 hours each)
Null Hypotheses
No effect on correctness
No effect on time
Results
Fail to reject
Reject (reduced time with UniMod)
Tasks
Four maintenance tasks for each object
Training
UniMod tool demo (2 hours)
UniMod HelloWorld exercise

Figure 3. Mind Map of the experiment

III. Experiment Definition, Design and Settings

We conceived and designed the experiment following the guidelines by Wohlin *et al.* [15]. The mind map shown in Figure 3 summarizes the main "ingredients" of the controlled experiment and the obtained results.

The *goal* of the study is analysing the effectiveness of UniMod (i.e., method + tool) during maintenance tasks with respect to code-centric conventional programming (i.e., directly implementing the change requests in the code using an IDE). The *perspective* is of *researchers*, evaluating how effective is UniMod in terms of (1) correctness of the final artefacts and (2) time required to perform the maintenance tasks. In what follows hypotheses, treatments, objects, subjects, variables and other aspects of the experiment are discussed in detail.

A. Hypotheses

The null hypotheses for the study are the following:

- H_{0a} Using UniMod does not affect the correctness of the maintained artefacts.
- H_{0b} Using UniMod does not affect the time required by the maintenance tasks.

H_{0a} and H_{0b} are two-tailed since we cannot postulate a difference expectation in terms of correctness and time.

B. Treatment and Experimental Design

Two cases can be distinguished: (i) maintenance tasks executed on Java code using the Eclipse IDE and (ii) maintenance tasks executed on the UniMod model using the Eclipse IDE with installed the UniMod plugin. Thus, only one independent variable occurs in our experiment (main factor), which is nominal. It assumes two possible values: "**–**" (Java only) or "**+**" (UniMod).

Table I
Experimental Design (**–** = Java only, **+** = UniMod).

	Group A	**Group B**	**Group C**	**Group D**
Lab 1	Svetofor **+**	Svetofor **–**	Telepay **–**	Telepay **+**
Lab 2	Telepay **–**	Telepay **+**	Svetofor **+**	Svetofor **–**

The experiment adopts a counterbalanced design (see Table I), thus ensuring that each subject works with both the treatments (UniMod and Java only) in the two lab sessions (2-hours each). Subjects were split randomly into four groups, each one working in Lab 1 on the maintenance tasks of a system with a treatment and working on Lab 2 on the other system with a different treatment. This design was chosen because it mitigates possible learning effects between labs (Lab1 and Lab2).

C. Objects

The objects of the study are two software systems: **Svetofor** and **Telepay**. The UniMod versions of these two systems were downloaded from a Website managed by the UniMod's designers[4]. In that Website, several full featured sample UniMod projects are available on-line for download.

Starting from the two downloaded UniMod projects, we completely built in Java two new software systems (Svetofor**–** and Telepay**–**) equivalent to Svetofor**+** and Telepay**+**. During the transformation, we chose to implement the state machines using the nested switch approach [4].

[4]http://is.ifmo.ru/unimod-projects-en/

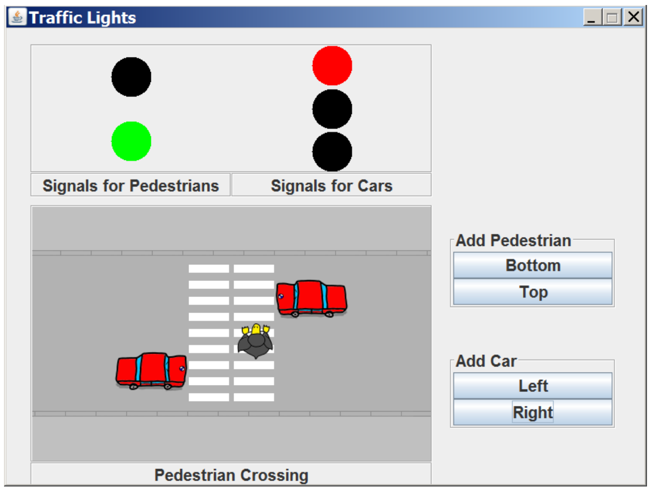

Figure 4. Svetofor GUI

Svetofor[5] is a smart traffic lights simulator equipped with a GUI representing pedestrians and cars (see Figure 4). The user can click the buttons on the right to make pedestrians and cars appear in the GUI. The system simulates a smart traffic light that works as follows. If there are no cars, but some pedestrians are waiting on the crossing, the traffic light indicates green light for pedestrians until at least one car appears, and vice versa. If there are neither cars nor pedestrians, the green light is for cars (they need more time to slow down and speed up). The UniMod version of the Svetofor system consists of three event providers (with 12 events in total), three state machines (A1, A2, A3) and two controlled objects (with 30 operations in total). A1 has two states and 8 transitions, A2 has 11 states and 15 transitions and A3 has four states and 9 transitions. On the contrary, the Java version of the system consists of five Java classes for a total of 1711 LOCs.

TelePay[6] simulates a mobile phone payment terminal. The user can use the keyboard to insert the phone number. Then, she/he can select some banknotes to charge the mobile phone. The UniMod version of the Telepay system consists of three event providers (with 8 events), three state machines (A1, A2, A3) and three controlled objects (with a total of 15 operations). A1 has six states and 13 transitions, A2 has three states and four transitions and A3 has three states and three transitions. By contrast, the Java version of the Telepay system consists of 10 Java classes for a total of 1824 LOCs.

D. Subjects

The subjects were 21 students from the Software Engineering course, in their last year of the bachelor degree in Computer Science at the University of Genova (Italy).

[5]http://is.ifmo.ru/unimod-projects-en/svetofor/
[6]http://is.ifmo.ru/unimod-projects-en/teleplay/

The subjects had a good programming knowledge, in particular Java programming, and an average UML knowledge (which was explained during the course).

Subjects have been trained in UniMod programming with a two hours tool-demo. Moreover, they experimented UniMod on a simple example (HelloWorld, presented in Section II) implementing some simple change requests on the given UniMod model.

E. Variables

The dependent variables to be measured in the experiment are (i) the *artefact correctness* after the execution of the maintenance tasks and (ii) the *time* required to perform the tasks.

Similarly to [10], the correctness of each maintenance task (four for each system) was manually assessed by one of the authors executing an acceptance test suite and giving a score from 0 (totally incorrect or not executed task) to 4 (completely correct) to each task according to the number of test cases passed and failed. Then, another author examined the assessment. In case of disagreement between the two authors on a score, a discussion was undertaken to reach an agreement. The total correctness for each subject is computed summing up the four scores. Thus, the variable *correctness* ranges from zero (no correctly executed tasks) to 16 (four completely correctly executed tasks).

The time was measured by means of time sheets. Students recorded start and stop time for each implemented maintenance task. In this way, we were able to compute the time required to execute each maintenance task simply computing $stop\_time - start\_time$. Finally, the *time* variable is computed for each subject summing up these four values.

F. Material, Procedure and Execution

The development environment was Eclipse (with and without the UniMod plugin, depending on the received treatment).

For each group, we prepared an Eclipse project containing the software (UniMod or Java only). The projects were zipped and made available on a Web server. The experiment was introduced as a laboratory assignment about UniMod.

Every subject received:

- summary description of the application (Svetofor or Telepay) common for both the treatments. We used the original documentation downloaded by the UniMod projects repository, containing also the representation of the state machines;
- instructions to set-up the assignment (how to download the zipped Eclipse project, import it, and execute the application);
- a sheet containing the four maintenance tasks (for Svetofor or Telepay).

For each lab, the subjects had two hours available to complete the four maintenance tasks: CR1-CR4 (Table II

Table II
CHANGE REQUESTS FOR SVETOFOR

CR1 – Reverse the side of appearance of cars. In the original version of the application, when you press the button "Add car left (right)" a new car from the left (right) appears. After the change when you will press the: (1) "Add car left" button the car should appear from the right and (2) "Add car right" button the car should appear from the left.
CR2 – When the light is green for the cars and a pedestrian appears, the green for the car begins to flash (for about 5 seconds) and then goes to yellow. Change the application so that the green light for the car does not flash but remains fixed (again for about 5 seconds) and then move on to the yellow.
CR3 – The modelled traffic light system is "smart" because: *"It has some information about cars on the road and foot-passengers on pedestrian crossing. In presence of some foot-passengers and cars, traffic-light works like a common traffic-light. If there are no cars, but foot-passengers are waiting on pedestrian crossing, it will be indicating green light for them until at least one car appears, and vice versa"*. You have to delete the "smart" behaviour, i.e., after the change the system has to simulate a normal traffic light.
CR4 – When the traffic light for cars goes from green to red, the yellow has to flash (for 5 sec) instead of being fixed (for 5 sec).

shows the English translation of the change requests for the Svetofor system). The first two change requests are very easy to implement, while the third and fourth ones require more work to locate the code and/or the portions of the model to be changed and to implement the change. The maintenance/evolution tasks, for the two different systems, are very similar and of comparable difficulty. The same holds for the two systems.

Between the two laboratory sessions (Lab 1 and Lab 2) a break was given.

For each Lab session, the experiment execution followed the steps reported below:

1) We delivered a sheet containing the description of the system.
2) Subjects were given 10 minutes to read the description of the system and understand it.
3) Subjects had to write their name on the delivered sheet.
4) Subjects had to download from a given URL the Eclipse project and import it.
5) A sheet containing the four change requests was delivered.

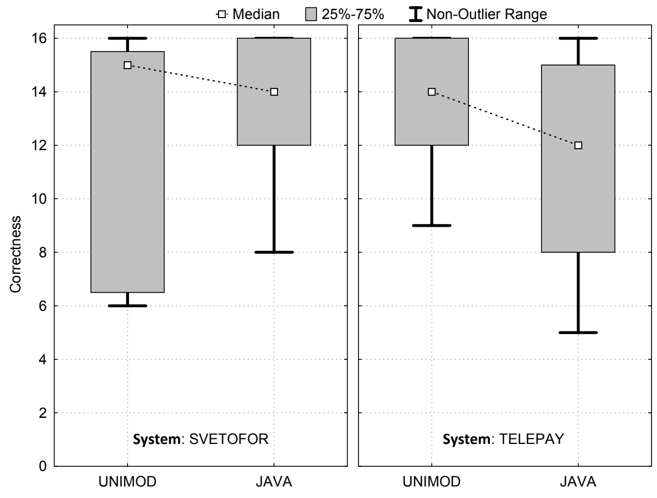

Figure 5. Boxplots of Correctness

6) For each change request (CR1-CR4):
   a) Subjects had to satisfy the change request (for Svetofor or Telepay).
   b) Subjects had to record the time they need to execute the modification (start/stop time).

Finally, subjects were asked to compile a post experiment questionnaire. That questionnaire aimed at both gaining insights about the students' behaviour during the experiment and finding motivations for the quantitative results. It included questions about the tasks and systems complexity, the adequacy of the time allowed to complete the tasks and the perceived usefulness/effectiveness of UniMod.

IV. PRELIMINARY RESULTS

This section reports some preliminary results from the performed experiment, analysing only the effect of the main factor on the dependent variables (correctness and time). More detailed analyses (e.g., analysis of co-factors such as task order or students ability) and the analysis of the post experiment questionnaire are not presented here for space reasons. Results of statistical tests are considered to be significant for a significance level of 95%.

A. Correctness

Figure 5 summarizes the distribution of the correctness variable by means of boxplots. Observations are grouped by treatment (UniMod or Java) and shown partitioning by system (Svetofor and Telepay). The y-axis represents the cumulative correctness of the four maintenance tasks – the cumulative score 16 represents the maximum value of correctness and corresponds to four tasks completely correct. The boxplots show that the subjects achieved a better correctness level when accomplishing the tasks with UniMod. This is true for both the systems but it is more evident

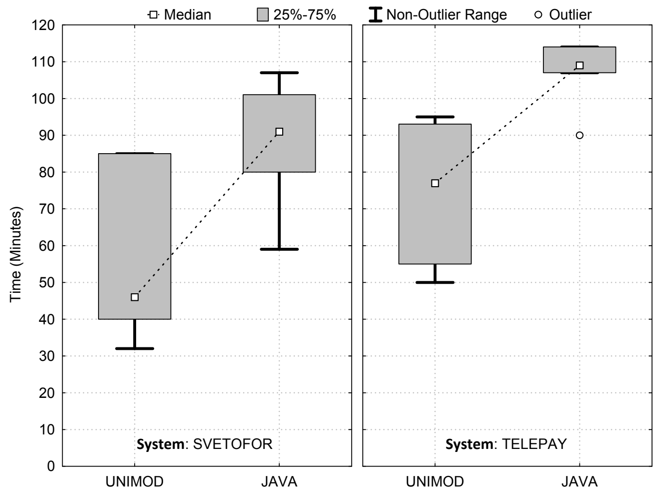

Figure 6. Boxplots of Time

for Telepay (see the slopes of the two lines connecting the medians of the distributions in Figure 5).

To be as much as possible conservative, we test the overall difference (i.e., without partitioning by system) in correctness by applying an unpaired non parametric statistical test[7]. By applying a two-tailed Mann-Whitney U test, we found the difference not to be statistically significant (*p-value=0.53*), therefore we cannot reject the first null hypothesis (H_{0a}).

B. Time

The second hypothesis can be easily tested by looking at the time required to complete all the tasks. For this reason, we analyzed only the time for the students able to complete all the four tasks (25 observations in total). Figure 6 summarizes the distribution of the time variable by means of boxplots. The boxplots show that, for both systems, students using code-centric programming (i.e., Eclipse + Java) employed more time than students using UniMod. The difference between the two treatments is more evident for Svetofor. Overall, the mean time reduction, achieved with UniMod, is 33 minutes (with UniMod the mean is 65 minutes while with conventional programming it is 98 minutes), i.e., approximately 33,7%[8].

To test the second hypothesis, we used a Mann-Whitney U test as done for the first hypothesis. The results of the two-tailed Mann-Whitney test show that the overall difference is statistically significant (*p-value<0.01*). Therefore, we can reject the second null hypothesis H_{0b}.

[7]Given the preliminary nature of this study together with: (1) the limited sample size and (2) the type of the outcome (i.e., a score), we chose to use non parametric tests.

[8]The value is computed using the equation: $98–98x=65$.

V. Threats to Validity

This section discusses the threats to validity that could affect our results: *internal*, *construct*, *conclusion* and *external* validity threats [15].

Internal validity threats concern external factors that may affect a dependent variable (in our case code correctness and time). Since the students had to perform a sequence of four maintenance tasks, a learning effect may intervene. However, the students were previously trained, the tasks were of progressively increasing difficulty, and the chosen experimental design (with a break between the two labs) should limit this effect. Therefore, we expect learning not to have influenced too much the results. Another threat could be due to a possible level of subjectiveness caused by the construction of the Java versions of Svetofor and Telepay that we did starting from the UniMod projects. We applied a standard approach to implement state machines in Java (in practice, we used nested switch [4]) but we cannot be sure that changing the implementation (e.g., using the state pattern) the outcome of the experiment would not be affected. We plan to verify this as part of our future work.

Construct validity threats concern the relationship between theory and observation. It is possible that the used testsuite does not provide an adequate means to measure the quality of the change requests implementation. Moreover, given that the testsuite was manually executed, some errors could have happened in the evaluation process. We mitigated this last possible threat with an independent double check.

Threats to *conclusion validity* can be due to the sample size (only 21 students) that may limit the capability of statistical tests to reveal any effect. We chose to use non parametric tests due to the size of the sample and because we would not safely assume normal distributions.

Threats to *external validity* can be related to: (i) the choice of simple systems as objects and (ii) the use of students as experimental subjects. We do not expect the absolute performance of students being at the same level of professionals, but we expect to be able to observe a similar improvement trend. Further controlled experiments with larger systems and more experienced developers are needed to confirm or contrast the obtained results. Another threat to external validity is that the results are limited to UniMod, and rather different results could be obtained with other MDD approaches.

VI. Related Work

To the best of our knowledge, empirical studies (i.e., industrial surveys, controlled experiments, case studies and systematic reviews) evaluating the relationships between MDD and maintainability are not so frequent. On the contrary, there are many empirical studies evaluating the maintainability of OO software developed in a code-centric conventional style (e.g., [1], [2], [3], [6]). A well-known controlled experiment in this context is [6]. It compares the maintainability of two functionally equivalent systems, one written with a OO language (Objective C) and the other written with a procedural language (C language). That experiment supports the hypothesis that subjects produce more maintainable code with an OO language.

The studies more closely related to ours are: [7] and [8]. Hutchinson et al. [7] present the results of an empirical study on the assessment of MDE (Model Driven Engineering) in industry, having the goal of understanding its success or failure reasons. This work uses three forms of investigation: questionnaires, interviews and on site observations, having as target practitioners, MDE professionals and companies practising MDE respectively. Main results can be summarized as follows: almost all the respondents believe that the use of MDE improved productivity and maintainability.

Kapteijns et al. [8] present a case study in which a small application is developed with and without MDD. In practice, a small middleware application is rebuilt using MDD and the time needed is then compared to the development time of the original application. The authors report that developing the legacy application took 42 days while with MDD the application was rebuilt in only 16 days. In conclusion, the authors report that with MDD there was an increase of productivity of about 2.6 times.

VII. Conclusion and Future Work

In this paper, we have presented a controlled experiment for assessing the effectiveness of the use of UniMod (a specific implementation of executable UML) with respect to code-centric conventional programming. The results indicate a marked reduction in time to accomplish the maintenance tasks (– 33,7%), with no significant impact on correctness, when UniMod is used instead of conventional programming. That result is worth reporting despite the limitations of our experiment because only 2 hours UniMod training were given to the students. We expect that increasing the training a greater time reduction can be obtained.

Future work will be devoted to replicate this experiment in different contexts. We would like to understand whether the maintenance time reduction deriving from the use of UniMod is preserved also for other categories of subjects (e.g., PhD students and professionals) and changing the domain and the complexity of the systems/tasks.

References

[1] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Trans. Softw. Eng.*, 32(6):365–381, June 2006.

[2] L. C. Briand, C. Bunse, J. W. Daly, and C. Differding. An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents. *Empirical Software Engineering*, 2:291–312, 1997.

[3] W. J. Dzidek, E. Arisholm, and L. C. Briand. A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Trans. Softw. Eng.*, 34(3):407–432, May 2008.

[4] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[5] V. Gurov, M. Mazin, A. Narvsky, and A. Shalyto. Tools for support of automata-based programming. *Programming and Computer Software*, 33:343–355, 2007.

[6] S. Henry and M. Humphrey. A Controlled experiment to Evaluate Maintainability of Object Oriented Software. In *Proceedings of International Conference on Software Maintenance (ICSM 1990)*, pages 258–265. IEEE CS, nov 1990.

[7] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, ICSE 2011, pages 471–480, New York, NY, USA, 2011. ACM.

[8] T. Kapteijns, S. Jansen, S. Brinkkemper, H. Houet, and R. Barendse. A Comparative Case Study of Model Driven Development vs Traditional Development: The Tortoise or the Hare. In *4th European Workshop on From code centric to model centric software engineering: Practices, Implications and ROI*, pages 22–33. CTIT, 2009.

[9] A. G. Kleppe, J. Warmer, and et al. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc, 2003.

[10] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, M. Ceccato, and C. A. Visaggio. Are Fit tables really talking?: a series of experiments to understand whether fit tables are useful during evolution tasks. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 361–370, New York, NY, USA, 2008. ACM.

[11] A. A. Shalyto and N. I. Tukkel. SWITCH Technology: An Automated Approach to Developing Software for Reactive Systems. *Programming and Computer Software*, 27:260–276, 2001.

[12] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[13] M. Torchiano, F. Tomassetti, A. Tiso, F. Ricca, and G. Reggio. Preliminary findings from a Survey on the MD* State of the Practice. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM 2011)*, pages 372–375, 2011.

[14] M. Torchiano, F. Tomassetti, A. Tiso, F. Ricca, and G. Reggio. Maturity of Software Modelling and Model Driven Engineering: a Survey in the Italian Industry. In *Proceedings of the International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)*, page (to appear), 2012.

[15] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.