POLITECNICO DI TORINO
Repository ISTITUZIONALE

On the Impact of State-based Model-Driven Development on Maintainability: A Family of Experiments using UniMod

Original
On the Impact of State-based Model-Driven Development on Maintainability: A Family of Experiments using UniMod / Ricca, Filippo; Torchiano, Marco; Leotta, Maurizio; Tiso, Alessandro; Guerrini, Giovanna; Reggio, Gianna. - In: EMPIRICAL SOFTWARE ENGINEERING. - ISSN 1382-3256. - STAMPA. - 23:3(2018), pp. 1743-1790. [10.1007/s10664-017-9563-8]

Availability:
This version is available at: 11583/2682971 since: 2018-06-05T10:22:32Z

Publisher:
Springer

Published
DOI:10.1007/s10664-017-9563-8

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
Springer postprint/Author's Accepted Manuscript

This version of the article has been accepted for publication, after peer review (when applicable) and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/s10664-017-9563-8

(Article begins on next page)

12 October 2024

Empirical Software Engineering (EMSE) manuscript No.
(will be inserted by the editor)

On the Impact of State-based Model-Driven Development on Maintainability: A Family of Experiments using UniMod

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

Received: date / Accepted: date

Abstract *Context*: Model-driven approaches are well-known in the academia but one possible showstopper to a wider adoption in the industry is the limited empirical evidence for their claimed advantages and benefits, that could convince the decision makers.

Objective: The aim of this work is gauging one of the claimed benefits of model-driven approaches, namely improvement in maintainability, with respect to a code-centric approach.

Method: We conducted a family of five experiments involving 100 students that possessed different levels of education (64 Bachelor, 25 Master, and 11 PhD students; in groups sized 11 to 26 per individual experiment). In these experiments, UniMod – a State-based tool for Model-Driven Development using the Unified Modeling Language – is compared with Java-based code-centric programming, for a software maintenance scenario, with the goal of analysing the effect on the time to perform the maintenance tasks, the correctness of the modified artefacts, and the efficiency.

Results: The results show a reduction in time to accomplish the tasks and no impact on correctness. The adoption of the UniMod-MDD approach almost doubles the developers'efficiency, and in presence of a higher software engineering experience the efficiency is even three times higher.

Conclusions: We found that the usage of the UniMod-MDD approach in a software maintenance scenario provides benefits over a pure code-centric approach. The benefits deriving from the UniMod-MDD approach are appreciable for all the categories of students, although with differences.

Filippo Ricca, Maurizio Leotta, Alessandro Tiso, Giovanna Guerrini, Gianna Reggio
DIBRIS, Università di Genova, Italy
E-mail: filippo.ricca|maurizio.leotta|alessandro.tiso|giovanna.guerrini|gianna.reggio@unige.it

Marco Torchiano
DAUIN, Politecnico di Torino, Italy
E-mail: marco.torchiano@polito.it

Keywords Model-Driven Development · MDD · UML · Maintainability · UniMod · Family of experiments · Replication and Maintenance Task

1 Introduction

Model-Driven Development (MDD) is a software production approach that uses models as primary artefacts of the software development process [1]. In practice, higher-level models are progressively transformed into lower-level models until the models can be made executable using either code generation or model interpretation.

Although model-driven approaches are popular in academia, their introduction into industry seems to be slow [2,3,4,5]. Apparently, one of the hurdles lies in the difficulty of convincing managers of MDD advantages. The main claimed advantages of MDD are improvements in productivity, portability, maintainability, and interoperability [6]. However, such claims demand for empirical evidence. Unfortunately, in the literature, empirical studies evaluating MDD and considering these aspects [7,8,9,10,11,12] are rare.

In this paper, we focus on maintainability as main quality attribute to study and on controlled experiments – with students as participants – as empirical strategy. Among the above-mentioned claimed advantages of MDD, we chose to evaluate the improvements in maintainability because: (1) software maintenance is recognized as one of the most expensive activities in software development [13] and (2) there are only a few studies (see Section 6.2) assessing the MDD benefits during software maintenance. The main goal of this work is understanding whether MDD is able to reduce software maintenance effort (or not), saving time, improving efficiency, and thus cutting costs. Unfortunately, the goal, stated in that way, is too abstract and ambitious. Indeed, it is not possible to experiment all the MDD approaches and related tools. Thus, we had to select an instance among all the possible MDD proposals. We define the concrete goal of our study as follows: evaluate the difference (if any) of performing software maintenance activities using (1) an abstract executable representation of the code based on UML models or (2) more traditional code-centric programming.

We selected the tool to employ in our experiments among the ones that model software artefacts by means of UML. This is due to the fact that UML is the only language who was certainty known by all the participants to the family of experiments. We analysed several tools such as AndroMDA[1] and BridgePoint[2] but we found them too complex for our experiments (participants could devote only a few hours to the experiment as described in the paper). On the other hand, we found UniMod [14] more simple to install and use and fast to learn.

In this paper, we present a family of controlled experiments planned and conducted using a rigorous approach as described by Wohlin *et al.* [15] and

[1] `http://www.andromda.org/`

[2] `http://www.mentor.com/products/sm/model_development/`

Jedlitschka *et al.* [16]. Five experiments have been designed and conducted involving overall 100 students with different levels of education (in detail 64 BS, 25 MS, and 11 PhD students; from 11 to 26 per experiment). Subjects were asked to perform maintenance/evolution tasks on two different kind of software artefacts — UniMod artefacts and conventional Java code — and their performance has been assessed and compared in terms of tasks correctness, time to complete the tasks and efficiency (i.e., the number of correct tasks per hour).

Specifically, the work presented here compares UniMod development (a mix between models production and coding) with conventional code-centric programming in the context of maintenance/evolution. In UniMod, the maintenance tasks are mainly conducted on the executable models composing the system. The difference between the two approaches consists in the artifacts available to the maintainer and the ensuing working approach, while the working environment (the IDE) is as far as possible similar and close to that of a typical working context.

The paper extends our previous preliminary work [17] in several directions. We provide here the following new contributions: (1) a deeper presentation of the maintenance tasks executed by the participants, (2) results from four further replications, (3) an extended data analysis (e.g., a post-experiment questionnaires analysis was added), and (4) a final discussion on the achieved results, that was absent in our previous workshop paper [17] and some practical implications of our study.

The remainder of the paper is structured as follows. Section 2 gives a brief introduction to UniMod and shows in detail a maintenance task executed by the students during the experiments. Then, Section 3.4 provides details on the design of the family of experiments, including the definition of the experiment, context selection, hypotheses formulation and instrumentation. Experimental results are reported in Section 4 and then discussed in Section 5 together with threats to validity. Finally, related work and conclusions close the paper in Section 6 and Section 7, respectively.

2 UniMod

UniMod is an instance of the State-based Model-Driven Development (SbMDD) tools category. Such tools support the development of software system whose behaviour can be described by an automaton (e.g., Automotive, Avionics, or Aerospace systems). Another well-known tool in the SbMDD category is MathWorks Stateflow[3], an industrial-grade control logic tool used to model and simulate combinatorial and sequential decision logic based on state machines and flow charts. More in detail, UniMod is a MDD approach for designing and implementing object-oriented programs. The approach relies on a specific instance of automata-based programming (SWITCH-technology [18])

[3] `http://www.mathworks.com/products/stateflow/`

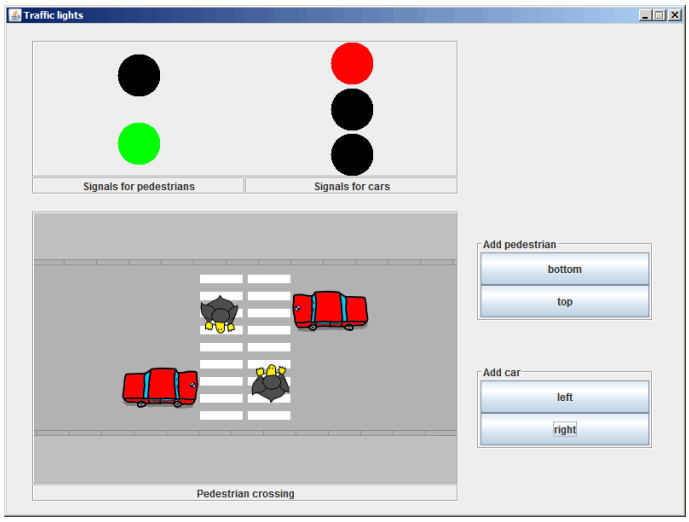

Fig. 1 Svetofor: Graphical User Interface (GUI).

and on UML. A tool for the development and execution of UniMod models is available; it allows creating, editing, and executing UniMod models. A UniMod model is composed by several different artefacts: a *connectivity schema* that depicts the architecture of the system (i.e., how the components of the system are connected) and by its components (i.e., EventProvider, StateMachine, and ControlledObject, see Section 2.2).

We will briefly introduce UniMod using as example Svetofor[4], one of the objects used in our family of experiments (the other one is Telepay[5]). The interested reader can find more information about UniMod in [14].

We chose UniMod due to four main reasons: (1) the approach relies on UML that is well-known by most students and professionals [19,20]; (2) a free software supporting tool exists[6] that, in our opinion, is mature enough in terms of features and usability; (3) the UniMod tool is, in our opinion, simple to use and to install and thus suitable to many students and developers; (4) UniMod did not remain only within the academic context and has been used in various organizations [14].

In the remainder of this section, we first present the Svetofor system. Second, we discuss the UniMod ingredients, and third we show in detail one of the Svetofor maintenance tasks executed by the students. Finally, we give some information about the UniMod tool (an Eclipse plug-in) used in the experiments.

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

[5] Here, for space reasons we will describe more in detail Svetofor than Telepay.

[6] `http://unimod.sourceforge.net/`

2.1 Svetofor

Svetofor is a smart traffic light system (TLS) simulator equipped with a GUI representing pedestrians and cars. It can be downloaded from the UniMod website. It simulates a smart TLS that is capable of detecting incoming pedestrians and cars at a crossroad. The user can press the "add car" and "add pedestrian" buttons to make appear cars and pedestrians, respectively, in the GUI (see Figure 1). When a pedestrian reaches the crossroad, the TLS realizes that a pedestrian is arrived and switches to green for him/her until a car appears. The same happens for cars. For this reason the traffic light system is said smart.

2.2 Connectivity Schema

A connectivity schema is a class diagram depicting classes with the following three stereotypes: *EventProvider*, *StateMachine* and *ControlledObject*.

Event providers are connected to state machines by means of UML associations and supply them with events. State machines are connected to the objects they control. Event providers are active: they affect the state machines by means of events. By contrast, controlled objects are passive: they perform actions when a state machine calls them (i.e., they offer some operations that the connected state machines can call). UniMod adopts the following naming convention for operations of controlled objects [14]: the operations used as input, i.e., returning a value to the state machine, are named `xi` (with `i≥0`), while the operations used as output, i.e., changing the value of a controlled object's field or executing an action (e.g., print("Hello World")), are named `zi`. The state machines 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.

In UniMod, event providers and controlled objects are specified in Java while state machines are expressed in visual form by means of appropriate diagrams, similar to UML state machines (see Section 2.3).

The connectivity schema of Svetofor is depicted in Figure 2. It contains:

- Three state machines `A1`, `A2`, and `A3` (shown in the middle of Figure 2).
- Three event providers `p1`, `p2`, and `p3`. They send several kinds of events to the `A1` state machine. For example, the event `e101` (a tick of clock) is sent every second. Similarly, events `e201`, and `e301` are sent every 5 and 0.1 seconds, respectively. On the contrary, events `e011`, `e012`, `e021`, and `e022` are sent when the buttons of the GUI are pressed (i.e., Add Pedestrian and Add Car).
- Two controlled objects: `o1`, mainly used to display the configuration/state of the TLS on the GUI, and `o2` used for adding and moving pedestrians/cars on the GUI.

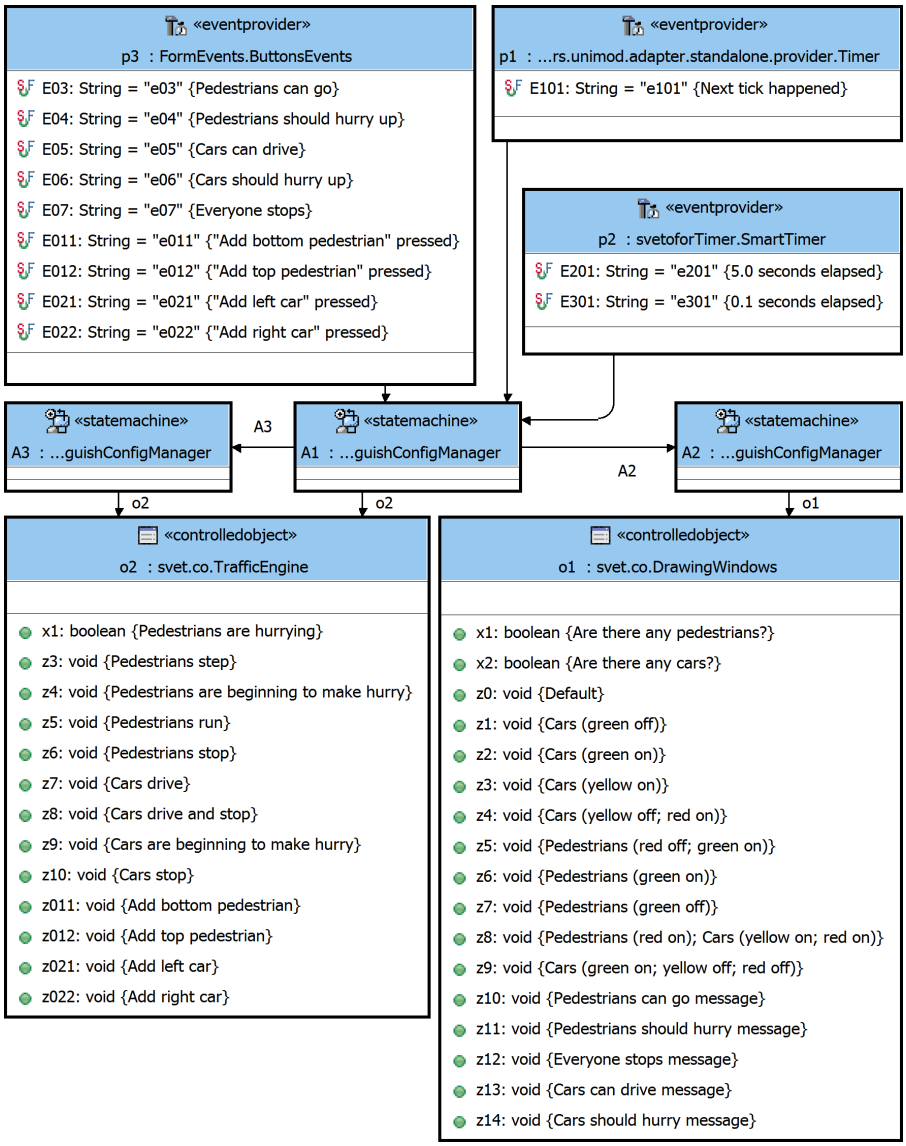

Fig. 2 Svetofor: Connectivity Schema (Class Diagram).

In UniMod, the comments explaining the meaning of events and operations are shown in the connectivity schema between brace parentheses (see Figure 2).

2.3 State Machines

State machines have simple and composite states, one initial state and zero, one or more final states. The transitions between states have labels of the following form:

event [guard] / actions

The square brackets contain a boolean formula (optional), which is the condition of the transition firing (i.e., the guard condition). For example, in the state machine A2 (see Figure 3), the label on the transition between the states s2 and s3:

e201[o1.x1] /o1.z14[7]

has the following meaning (see comments between brace parentheses in Figure 2):

when 5s elapsed [*if* there are pedestrians] / cars should hurry

While working on state machines with the UniMod tool, textual comments like the one immediately above appear as pop-up tooltip when the mouse passes over a transition label (they are generated automatically from the comments inserted in the connectivity schema). Thus, the transition meaning can be easily understood using the detailed documentation included in the experimental-package and the facility provided by the UniMod tool, that translates transition labels in textual comments.

The state machine A1 is the top level state machine. It receives the events from every event provider (i.e., p1, p2, and p3). The state machine A1 has a state (s2) that includes the state machines A2 and A3 (see Figure 3). All events received by the state machine A1 are automatically passed to A2 and A3. Self-transitions in A1 are used to invoke the corresponding methods of the controlled object o2 in response to events related to GUI button pressing.

The state machine A2 is used to control the switching of the TLS. Each state of A2 corresponds to a configuration of the TLS; for example, the state s2 corresponds to green for cars and red for pedestrians while state s6 to the opposite (red for cars and green for pedestrians). The transitions between the states are triggered by the timer events sent by p1 and p2. When the state machine enters in a new state, a method of the controlled object o1 is called to display the right configuration of the TLS on the GUI (see the nodes of A2 in Figure 3; they contain an entry action specified by the keyword enter).

The state machine A3, finally, is used to realize the movement of pedestrians and cars. Specifically, in state s2 both cars and pedestrians are stopped, in state s3 only pedestrians are moving and in state s5 only cars are moving[8].

[7] This odd naming convention (e.g., o1.z14) is a peculiarity of UniMod to reduce the diagrams size.

[8] The complete documentation of Svetofor is available at:
http://is.ifmo.ru/unimod-projects-en/svetofor/

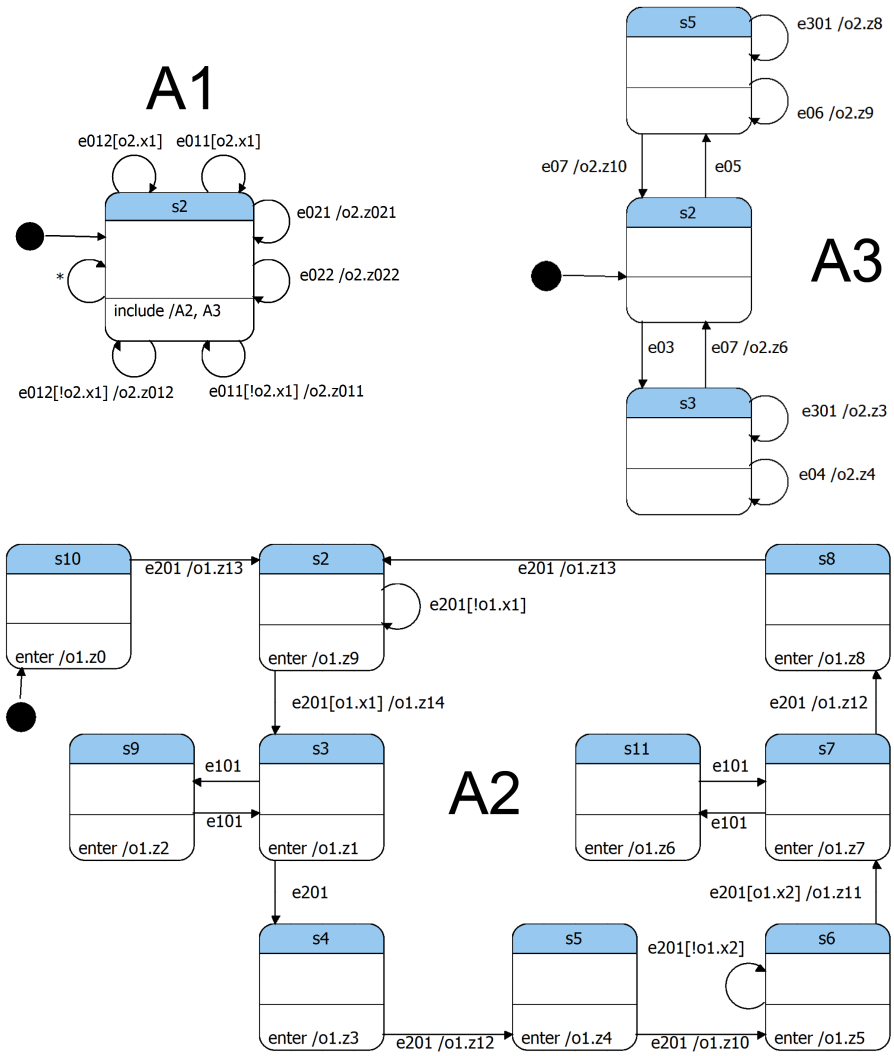

Fig. 3 Svetofor: State Machines A1, A2, and A3.

2.4 Svetofor Maintenance Task Example

Maintenance tasks represent the unit of work in our family of experiments. In order to understand what they consist in, we describe what the execution of a maintenance task – MT3 in Table 2 – entails. First, we will consider the impact of the change on the UniMod version of Svetofor and then on the Java one.

In short, maintenance task MT3 requires to remove the smart behaviour of the TLS. The documentation available to perform the maintenance tasks

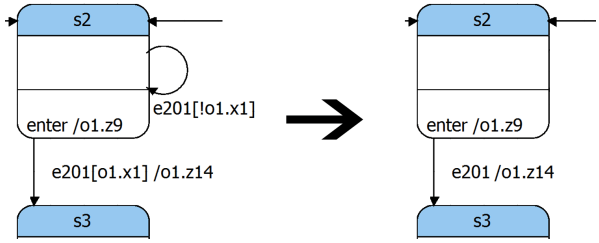

Fig. 4 MT3 in the UniMod Model of Svetofor (before → after).

consists of a *pdf* file with a detailed description of the application (i.e., a textual description, the state machine diagrams, the meaning of each state of the state machines and the connectivity schema, and the descriptions of the event providers and of the controlled objects).

The first step consists of locating the portions of code or model to modify. Using the documentation (which includes the diagrams shown in Figure 2 and Figure 3) it is not difficult to understand that we need to modify the state machine A2; indeed the smart behaviour is implemented in the states s2 and s6 of A2. As previously discussed, state s2 corresponds to green for cars and red for pedestrians and s6 corresponds to red for cars and green for pedestrians. In the following, we analyse the case of s2 (s6 is similar). In a "dumb" TLS the transition outgoing from s2 is only triggered by a predetermined amount of time (e.g., 40 seconds). By contrast, in our smart TLS, the transition outgoing from s2 has a guard (o1.x1) that implements the smart behaviour. That guard is only satisfied if there are pedestrians at the crossroad. Only in this case the TLS switches to the s3 state (i.e., green for cars flashing). Otherwise (i.e., there are no pedestrians) the self-transition s2→s2 (labelled with e201[!o1.x1]) is selected and thus the state machine remains in state s2.

To remove the smart behaviour in the UniMod model of Svetofor, the guard o1.x1 in the transition s2→s3 and the self-transition s2→s2 need to be eliminated. Figure 4 visually shows the required modification.

A similar change has to be done in the Java version of Svetofor. First, starting from the file containing the implementation of the A2 state machine, the portion of code implementing the state s2 and its logic (that portion is shown in Figure 5) needs to be located. Second, the portion of code implementing the self-transition s2→s2 has to be removed. Finally, the code that implements the guard o1.x1 needs to be deleted from the s2→s3 transition. In Figure 5, we have marked with * and highlighted in red the lines of code to delete (or comment) for implementing the maintenance task MT3 (original code comments are marked with // as usual).

```
case s2 :
switch (e) {
 case e201 :
  // s2->s2 e201[!o1.x1]/
        *  if (!(o1_x1)) {
        *   fireTransitionFound(ctx,path,"s2",event,TR_S2_S2);
        *   fireComeToState(ctx,path,"s2");
        *   // s2 [o1.z9]
        *   o1.z9(ctx);
        *   return new StateMachineConfig("s2");
        *  }
  // s2->s3 e201[o1.x1]/o1.z14
        *  if (o1_x1) {
            fireTransitionFound(ctx,path,"s2",event,TR_S2_S3);
            o1.z14(ctx);
            fireComeToState(ctx,path,"s3");
            // s3 [o1.z1]
            o1.z1(ctx);
            return new StateMachineConfig("s3");
        *  }
  // TR not found
        *  return config;
 default :
  // TR not found
  return config;
}
```

Fig. 5 MT3 in the Java Version of Svetofor.

2.5 UniMod Tool

The UniMod tool is available as Eclipse plug-in; it provides the following software components: (1) a graphical editor for creating/editing the connectivity schema and the state machines; (2) an on-the-fly model validator that notifies to the user errors/warnings and suggests possible quick-fixes (similarly to a modern IDE for a traditional programming language)[9]; (3) a launcher that allows to execute the UniMod model in "one click". Note that, starting from the connectivity schema, the tool generates a skeleton Java class for each event provider and controlled object depicted in the connectivity schema. Thus, to implement the behaviour of events and operations, the developer only has to fill the body of the methods in these Java classes.

3 Experimentation Definition and Planning

This section reports the definition, design, and settings of the family of experiments in a structured way, following the GQM approach and guidelines by Wohlin *et al.* [15] and Jedlitschka *et al.* [16].

For replication purposes, the experimental package has been made available[10].

[9] In our study we have not measured if this capability has contributed, or not, to the obtained results.

[10] `http://sepl.dibris.unige.it/UniModVSJava.php`

The *goal* of the study is analysing the difference (if any) of performing software maintenance/evolution activities using (1) an abstract executable representation of the code based on UML models or (2) more traditional code-centric programming *with the purpose* of evaluating possible benefits from adopting the UniMod-MDD approach.

Quality focus are: (1) correctness of the final artefacts, (2) time required to perform the maintenance tasks, and (3) efficiency that measures the number of correct tasks per hour. Results of this study can be interpreted from multiple *perspectives*: a researcher interested to empirically assess the effect of UniMod in maintenance/evolution tasks; a practitioner, willing to understand and quantify possible benefits in the maintenance phase deriving from the introduction of a specific MDD approach/tool in his/her company. The *context* of this study consists of academic students as *participants* playing the role of software maintainers and two small (< 2K LoC) software systems to be modified as *objects*.

3.1 Participants

Participants are university students, either Bachelor, Master or PhD students. The study consists of a family of five experiments involving in total 100 students:

- **UniGE-BS1** was performed with 18 students in their last year of the BSc degree in Computer Science at the University of Genova (academic year 2011/2012 first semester). Preliminary results of this experiment has been reported in a previous paper [17];
- **PoliTO-PhD** with 11 PhD students from Politecnico di Torino (academic year 2011/2012 second semester);
- **UniGE-MS** with 25 students in their last year of the MSc degree in Computer Engineering at the University of Genova (academic year 2011/2012 second semester);
- **UniGE-BS2** with 20 Bachelor students from University of Genova (last year of the BSc degree in Computer Science, academic year 2012/2013 first semester); and
- **UniGE-BS3** with 26 Bachelor students from University of Genova (last year of the BSc degree in Computer Science, academic year 2013/2014 first semester).

Bachelor students in Computer Science at the University of Genova have just basic notions of programming in Java and some initial knowledge of software engineering. UML was explained during the Software Engineering course (i.e., the course in which the experiment was conducted) where a specific attention to UML state machines was given. The experiment was introduced as a normal laboratory assignment. Before it, the students participated in several labs about UML and software engineering tasks.

Master students in Computer Engineering at the University of Genova have an average knowledge about software engineering topics and a good knowledge on Java programming. In fact, they previously developed non-trivial projects using UML as modelling language and Java as implementation language.

Most PhD students held a Master in Information and Communication Technology Engineering; a few were carrying out research in the field of Electronic Engineering.

All participants attended (at least) (a) one Java programming course that made them familiar with that language and with the Eclipse IDE, and (b) one software engineering course where they learned analysis, design and software testing principles.

Before the experiments, all the subjects have been trained in UniMod programming with a two hours session organized as follows: (1) a talk providing an introduction to model-driven techniques, Executable UML and explaining how UniMod works (i.e., details about Connectivity Schema, EventProvider, StateMachine, ControlledObject etc. as described in Section 2); then (2) a practical session where the instructor developed from scratch in UniMod the HelloWorld project [17], i.e., a complete and executable "Helloworld" program that prints the string "Hello, World!" on the console for ten times, then it prints the string "Bye, World!" for ten times. During the session the instructor described: (1) how to implement the various components of an UniMod project; (2) the feature of the graphical editor; (3) how to apply the change requests to the HelloWorld project using UniMod (he highlighted which portions of the Java code can be modified or not since auto-generated by the tool), and (4) how to apply the same changes in case UniMod is not adopted, (e.g., how to manually change the state machines Java implementations). Moreover, to become familiar with UniMod, we asked the students to re-develop from scratch the HelloWorld project and executing some simple maintenance tasks on that system.

3.2 Experimental Objects

In the following, we provide some details on the selection process and construction of the experimental objects together with their description.

We chose to focus on desktop applications since we needed simple self-contained applications. The other possible alternative, i.e. web applications, would have required a web app container (e.g. Tomcat), thus making the experimental setup more complex, the comprehension activity – required by the maintenance tasks – more difficult, and eventually it would have added new confounding factors. Last but not least, the prospective participants included also students without adequate knowledge on web app development.

Selection Process: according to our design (see Section 3.4), we needed two experimental objects, each developed in two different ways (UniMod and Java only). We considered five possible alternative approaches to select/produce the experimental objects: (1) finding two applications for which both versions

(i.e., UniMod and Java only) were available, (2) selecting two existing UniMod applications and then developing by ourself the corresponding Java only versions, (3) selecting two existing Java only applications and then developing by ourself the corresponding UniMod versions, (4) adopting an hybrid approach consisting in having an application developed starting from an existing UniMod implementation and the other developed from a Java implementation, and finally (5) developing both versions from scratch. The first option would have been optimal but, unfortunately, we could not find any application developed in both versions. Thus, since we matured much more development experience in Java than in UniMod, we chose the option number 2 (we believe that if we developed the UniMod versions the result would have been suboptimal with the possibility of influencing the results of the experiment due to low-quality UniMod implementations). As a consequence, we searched the internet and we found only the 22 reference applications available at the website managed by the UniMod's designers[11]. We analysed them; first we discarded those lacking a comprehensive English documentation; then, among the remaining candidates, we selected Telepay and Svetofor for the following reasons: (1) they are comparable in complexity and size, (2) they have an adequate complexity (i.e., they are not trivial) taking into account the time constraints of the experiments and the skills of the participants, (3) their domains are easy to understand (respectively a traffic light system and a mobile phone payment terminal), and (4) they are provided with both a detailed English documentation and a ready to import Eclipse project archive. Clearly, our selection process has been tailored taking into account the limited number of available applications, once we chose option 2.

Java only versions development: starting from the two selected UniMod projects (we refer to them as Svetofor+ and Telepay+ respectively), we built two new software systems (we refer to them as Svetofor− and Telepay− respectively) completely realized in Java and with the same functionality of the original ones. In practice, we decided to implement the UniMod state machines contained in Svetofor+ and Telepay+ completely in Java using the nested switch approach [21]. The rationale behind this choice is: (1) usually students know it very well and adopt it, (2) rarely bachelor students are able to master/command design patterns, such as, e.g., State Pattern [21], and (3) the state machines of the selected systems are quite small. On the contrary, all the other components of the original projects (i.e., GUIs, controlled objects and event providers) were integrally copied in Svetofor− and Telepay− to have two running systems equivalent to Svetofor+ and Telepay+.

Svetofor has already been discussed in Section 2. 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. Overall 701 Java LOCs are used to implement the GUI and the behaviour of the event providers and controlled objects. `A1` has two states and 8 transitions, `A2` has 11 states and 15 transitions and `A3` has four

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

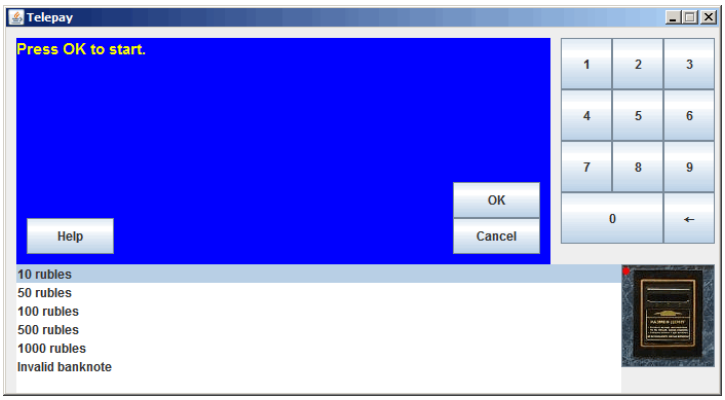

Fig. 6 Telepay GUI

states and 9 transitions. On the other hand, the Java version of the system consists of five Java classes for a total of 1711 LOCs, in this case including the implementation of the state machines. Note that Svetofor is conceptually similar to the example "Modeling an Intersection of Two 1-way Streets using State flow" given on the MathWorks Stateflow's website[12] (see more details about MathWorks Stateflow in Section 6). Indeed, in both cases the behaviour of traffic lights system is provided by means of state machines (see the diagram in the linked web page).

Telepay[13] simulates a mobile phone payment terminal (see Figure 6). The user can use the keyboard to insert the phone number. Then, she/he can select some banknotes to recharge 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. Overall 972 Java LOCs are used to implement the GUI and the behaviour of the event providers and controlled objects. 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.

3.3 Variables and Hypotheses Formulation

The family of experiments has one independent variable (also called "main factor" or Treatment from here on) with two possible levels: *Java only* or *UniMod*. The former means the maintenance tasks are executed on Java code using the Eclipse IDE. The latter indicates the tasks are executed on the UniMod artefacts (i.e., Connectivity Schema, EventProvider, StateMachine, and ControlledObject) using the UniMod-plugin installed on the Eclipse IDE.

[12] https://www.mathworks.com/help/stateflow/examples/modeling-an-intersection-of-two-1-way-streets-using-stateflow.html
[13] http://is.ifmo.ru/unimod-projects-en/teleplay/

Our family have three dependent variables, on which treatments are compared measuring three different constructs: (i) *Correctness*, (ii) *Time* required to perform the maintenance tasks, and (iii) *Efficiency*. Each construct is measured with a variable (respectively *TotalCorrectness*, *TotalTime*, and *TotalEfficiency*) for which we defined the relative metric. Figure 7 graphically shows the relationships between Treatment, Constructs, Variables, and Null Hypotheses.

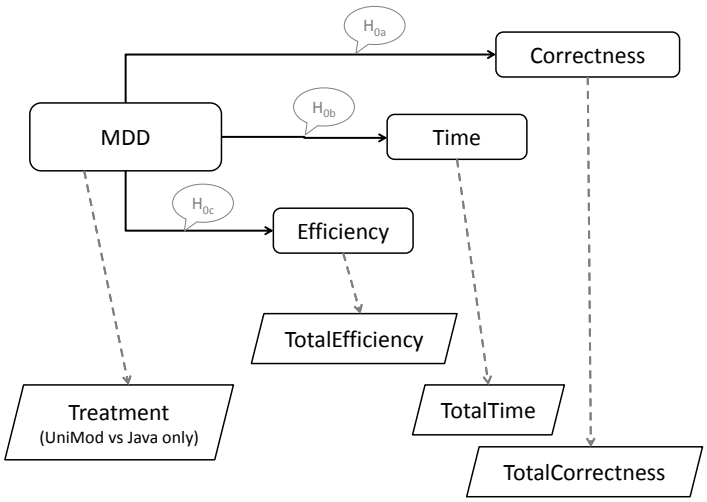

Fig. 7 Relationships between Treatment, Constructs, Variables, and Null Hypotheses

The correctness of each maintenance task was assessed by executing an automated JUnit acceptance test suite and giving a correctness score from 0 (totally incorrect or not executed task) to 4 (completely correct) for each maintenance task, according to the number of test cases passed (we already adopted a similar strategy in previous experiments [22,23]). More in detail, $TS_i = \{T_{i1}, T_{i2}, ..., T_{in}\}$ is a set of n test cases for each MT_i[14] and for each system. We gave correctness $score_i = 0$ if all the testcases in TS_i failed, 4 if all the testcases in TS_i passed. We assigned correctness scores 1, 2, and 3 when the number of passed testcases is respectively included in: $(0, 1/3n]$, $(1/3n, 2/3n]$, $(2/3n, n)$. Finally, the total correctness for each subject (*TotalCorrectness* variable) was computed summing up the four scores. Thus, the *TotalCorrectness* variable ranges from zero (no correct tasks) to 16 (four completely correct tasks).

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

[14] The number of test cases is different for each maintenance task.

compute the time in minutes ($stopTime - startTime$) required to execute each individual maintenance task. Finally, the *TotalTime* variable was computed summing up these four values for each participant.

Efficiency measures the efficiency of a participant in the execution of the maintenance tasks. The efficiency is a derived measure that is computed as the ratio between artefacts correctness and time to perform the four maintenance tasks. To provide an immediately comprehensible measure, we decided to measure the task efficiency measure as *number of correct maintenance tasks per hour*. Considering that the task time is measured in minutes we can define the *TotalEfficiency* variable as[15]:

$$TotalEfficiency = \frac{TotalCorrectness/4}{TotalTime/60} = \frac{TotalCorrectness}{TotalTime} \cdot \frac{60}{4} \qquad (1)$$

Thus, we can state the null hypotheses for the study in this schematic way:

- H_{0a}: *TotalCorrectness* (UniMod) = *TotalCorrectness* (Java only)
- H_{0b}: *TotalTime* (UniMod) = *TotalTime* (Java only)
- H_{0c}: *TotalEfficiency* (UniMod) = *TotalEfficiency* (Java only)

Since we could not find any previous clear empirical evidence that points out a clear advantage of one approach vs. the other, we formulated H_{0a}, H_{0b}, and H_{0c} as non-directional hypotheses. The objective of a statistical analysis is to reject the null hypotheses above, so accepting the corresponding alternative ones: H_{1a}, H_{1b}, and H_{1c}.

In our family of experiments, we also analysed the effect of the participants' profiles on the dependent variables. In particular, we also considered:

- **Experiment**. It indicates the experiment in our family. Therefore, Experiment is a categorical nominal variable that can assume the following values: UniGE-BS1, UniGE-BS2, UniGE-BS3, UniGE-MS, and PoliTO-PhD.
- **Maintainers' SE (Software Engineering) experience**. The experience of students could be a cofactor playing a significant effect. Specifically, we are interested in studying how such factor interacts with the Treatment. We approximated Maintainers' SE experience as their level of education, i.e. bachelor, master or PhD. PhD students were considered to have high SE experience, master students medium, while bachelor students low (we adopted a similar strategy in previous experiments [24,25]). Thus Maintainers' SE experience is a categorical ordinal variable with three levels.

[15] *TotalCorrectness* is divided by 4 since a fully correct task is assigned a score of 4, while *TotalTime* is divided by 60 because we want to measure tasks per hour while the time is measured in minutes.

3.4 Experiment Design

The family of experiments adopts a counterbalanced within subjects design (see Table 1) intended to fit two Lab sessions.

Participants were split into four groups each one working in Lab 1 on a system with a treatment and working in Lab 2 on the other system with a different level of the treatment: Java only (i.e., −) or UniMod artefacts (i.e., +). The design ensures that each subject works on different systems in the two Labs, receiving each time a different treatment.

To equally distribute students in the various groups, we split students in groups balancing as much as possible their software engineering and computer science experience/ability. For BSc and MSc students we used the average score of the previous software engineering lab grades. For PhD students we followed the indications provided by the professor of the PhD course in which the students were enrolled. He subjectively characterized the ability level of the students based on the information he acquired during the course about them.

A within-subjects design was chosen because it allows collecting more data points from a limited number of participants. Moreover, it is well-known that counterbalanced designs limit as much as possible learning effects [26].

3.5 Materials, Procedure, and Execution

This section details the procedure we followed to perform the experiments in our family, and the material employed.

All the experiments of the family took place in a laboratory room equipped with computers. For each group (see Table 1), we prepared a zip file containing two Eclipse projects (one for Lab) and their documentation (common for both the treatments), which consists of: a description of the applications, the state machine diagrams, the meaning of each state of the state machines, the connectivity schema, the descriptions of the event providers and of the controlled objects (as described in Section 2)[16]. Then, the zip files were made available on a Web server. The experiments were introduced as a laboratory assignment about UniMod.

[16] For both projects, we used the original documentation available at: `http://is.ifmo.ru/unimod-projects-en/`

Table 1 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 −

To assess the experimental material and to get an estimate of the time needed to accomplish the tasks, a *pilot experiment* with one BSc student in Computer Science at the University of Genova was accomplished, before the UniGE-BS1 experiment. The student finished the eight maintenance tasks (four for each system) in approximately seven hours and gave us some information on how improving the experimental material. Minor changes were then made to the experimental material and especially to the words used in the text explaining the maintenance tasks to make them clearer.

For each Lab, the subjects had three hours and half to complete the four maintenance tasks: MT1-MT4 (Table 2 and Table 3 show respectively the maintenance tasks for the Svetofor and Telepay). We designed the maintenance tasks for the two systems taking inspiration from the seminal paper [27] that categorise the maintenance tasks on several dimensions. The maintenance tasks, for the two different systems: 1) change the customer-experienced functionality, 2) are very similar, and 3) are of comparable difficulty. In particular, for each system, MT1 and MT2 can be classified as corrective tasks, MT3 as reductive, and MT4 as enhancive [27]. The designed maintenance tasks are simple to fit a one day experiment but not trivial and were mainly designed for bachelor students. We remind the reader that participants using UniMod (treatment +) have to work on Connectivity Schema, StateMachine and ControlledObject, without modifying the generated code (i.e., the implementation of the state machines), while participants using Java (treatment −) have to perform the maintenance tasks by directly changing the provided code. Between the two laboratory sessions (Lab 1 and Lab 2) a 15-minutes break was given.

Each subject received a paper sheet containing some instructions to set-up the assignment (how to download the zip files, how to import an Eclipse project, and how to execute the applications). For each Lab session, the experiment execution steps were as follows:

1) Participants had to download the zip file (corresponding to the group she/he belongs to) containing the Eclipse project from a given URL and import it.
2) Participants were given 15 minutes to read the description of the system and understand it (Svetofor or Telepay).
3) Participants were given five minutes to execute the application trying its functionalities.
4) We delivered a sheet containing the four maintenance tasks.
5) Participants had to write their name on the delivered sheet.
6) For each maintenance task (MT1-MT4) contained in the delivered sheet:
   a) Participants had to record the start time.
   b) Participants had to perform the maintenance task (for Svetofor or Telepay).
   c) Participants had to record the stop time.

Finally, participants were asked to compile a post experiment questionnaire (see Table 4). That questionnaire aimed at both gaining insights about

Table 2 Maintenance Tasks and Impact (not shown to the students) for Svetofor. (J) means that a change to Java code is required also in the case of the UniMod treatment (recall, the behaviour of the operations of the controlled objects is expressed in Java, see Section 2.2).

MT1: Change how cars appear. 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 button "Add car left (right)" the car should appear from the right (left) **Impact**: Modify two transitions **Type**: Corrective
MT2: When the light is green for the cars and a pedestrian reaches the crossroad, the green for the cars blinks (for about 5 seconds) and then the traffic light system switches to yellow for the cars. Change the application so that the green light for the car does not blink but remains fixed (again for about 5 seconds) **Impact**: Delete one state and two transitions and modify one transition **Type**: Corrective
MT3: Delete the "smart" behaviour, i.e., after the change the system has to work as a conventional traffic light system **Impact**: Delete a transition and modify a transition. See Section 2.4 **Type**: Reductive
MT4: When the traffic light system switches from green to red for cars, the yellow has to flash (for 5 sec) instead of being fixed **Impact**: Add one state and two transitions and (J) add one operation to a controlled object **Type**: Enhancive

the students' behaviour during the experiment and finding motivations for the quantitative results. It included questions about: clarity of the maintenance tasks (PQa), ability of participants to understand the provided Java code (PQb) and UniMod models (PQc), clarity of the documentation (PQd), exercise usefulness (PQe), and change localization and modification in the UniMod model vs. bare code (PQf and PQg). Answers were on a Likert scale ranging from one (strongly agree) to five (strongly disagree).

3.6 Analysis Procedure

Different kinds of statistical tests have been used to analyse the results of this family of experiments. All of them have been applied using the R statistical environment [28].

First of all, for each dependent variable we checked if the data set is well-modelled by a normal distribution. For this task, we generated the Quantile-Quantile plots (QQ-plots) for a first visual analysis. Then, we used the Shapiro-

Table 3 Maintenance Tasks and Impact (not shown to the students) for Telepay. (J) means that a change to Java code is required also in the case of the UniMod treatment (recall, the behaviour of the operations of the controlled objects is expressed in Java, see Section 2.2).

MT1: "Reverse" the construction mode of the phone number. When you press a button on the keypad each digit is concatenated in the end of the number in the following way: if for example you press 2, then 3, then 5, then 7 the resulting number will be 2357. The new software (the one obtained after you have performed this change request) will concatenate the digits in the beginning of the number. Considering the previous example the resulting number will be 7532.

Impact: (J) Modify an operation of a controlled object

Type: Corrective

MT2: In the original application the user can make any amount of refills (i.e. there is no limit to the charge). Limit the maximum value of a charge to 3000 rubles (over this amount is no longer allowed to increase the value of the charge).

Impact: Add a transition, modify a transition and (J) add one operation to a controlled object

Type: Corrective

MT3: Disable the behaviour of the application when an invalid bill is inserted. "invalid banknote" will be displayed in the GUI but its insertion into the slot (which corresponds to select it and press the black button at the bottom right) will have no effect.

Impact: Delete one state, modify a transition and (J) add one operation to a controlled object

Type: Reductive

MT4: Add the following functionality: "donating 10 rubles to a charity association after confirming the amount of recharge". Basically, after the confirmation the following question has to appear in the GUI (in the blue zone dedicated to communications): "Do you want to donate 10 rubles to the association XYZ?". The OK button confirms the donation so the final charge will be the reduced by 10 rubles. Instead, CANCEL rejects the donation. To verify that you have successfully implemented this change request is requested to change the final confirmation message (your payment was delivered successfully) so that it will print the actual value of the charge.

Impact: Add one state and two transitions, (J) add two operations to two controlled object and (J) modify a operation of a controlled object

Type: Enhancive

Wilk W test, often used in literature in testing for normality [29]. If the W statistic is significant, the hypothesis that the respective distribution is normal should be rejected, i.e., there is evidence that the data tested are not from a normally distributed population.

Second, we compared the results of the five experiments considering the overall values of correctness, time, and efficiency in performing maintenance tasks (i.e., without partitioning by treatment). This analysis is used to understand whether (or not) there are significant differences among the experiments.

Table 4 Post-experiment Questionnaire

ID	Question
PQa	The maintenance tasks were perfectly clear to me
PQb	I had no problems to understand the provided Java code
PQc	I had no problems to understand the provided UniMod models
PQd	The provided documentation was useful and clear
PQe	I found the exercise useful
PQf	Change localization in the UniMod model is simpler than in the code
PQg	Modifying the UniMod model is simpler than modifying the code

In such case, we have to test the hypotheses separately for each experiment (i.e., we cannot put together all the data and conduct an overall analysis). For this task, we used a non-parametric test because of the non-normality of the data (see next section). In particular, we selected the Kruskal-Wallis test, a non-parametric method for testing whether samples originate from the same distribution [30]. This test is used in literature for comparing two or more samples that are independent, and that may have different sample sizes, and extends the Mann–Whitney U test to more than two groups.

Third, to be as much as possible conservative (because of the sample size and mostly non-normality of the data, see next section), we also used a non-parametric test to test our hypotheses (H_{0a}, H_{0b}, H_{0c}). This choice is in agreement with the suggestions given in [26, Chapter 37]. The unpaired analysis — i.e., an analysis of all data grouped by different treatments of the main factor — was performed using the two-tailed Mann-Whitney (MW) U test [30]. Such a test allows to check whether differences exhibited by subjects with different treatments (Java only and UniMod) over the two labs are significant.

Fourth, we performed the analysis of Maintainers' SE experience co-factor using a two-way Permutation test [31]. The permutation test is a non-parametric alternative to the two-way Analysis of Variance (ANOVA). Differently from ANOVA, it does not require data to be normally distributed. The general idea behind such a test is that the data distributions are built and compared by computing all possible values of the test statistic while rearranging the labels (representing the various factors being considered) of the data points. Since the permutation test samples permutations of combination of factor levels, multiple runs of the test may produce different results. We made sure to choose a high number of iterations (i.e., 500.000) so that results did not vary over multiple executions of the procedure. For the computation of permutation tests we used the package *lmPerm* [32]. This analysis was used first to check the effect of the main cofactors (Maintainers' SE experience and Experiment) alone on the dependent variables, then it was used after the main hypothesis test to confirm the obtained results and to check the effect of the relevant

cofactors. The initial analysis was based on a regression equation with the following structure[17] :

$$D = c_0 + c_1 \cdot \mathrm{CF}_1 + c_2 \cdot \mathrm{CF}_2 + c_{12} \cdot \mathrm{CF}_1 : \mathrm{CF}_2$$

Where the dependent variable (D) can be one of *TotalCorrectness*, *TotalTime*, and *TotalEfficiency*. The right-hand part include an intercept (c_0), two terms for the cofactors (CF_i), and an *interaction* term ($CF_1 : CF_2$). The interaction term models the influence that two terms have on each other. It can be easily understood with a simple example: let us consider the sweetness of a cup of coffee as the dependent variable; the independent variable will be putting sugar (S) and mixing with a stick (M) is the cofactor, both with two possible levels (yes and no). In this case the independent variable alone cannot explain the outcome, though the interaction with the cofactor will. Based on the initial analysis we selected the most relevant cofactor that was used to further analyse the effect of treatment on the outcome taking into account its interplay with the said cofactor. We opted for analysing a single cofactor because of the limited number of data points, too complex a model could easily result into spurious correlations and its interpretation is typically more difficult. The post-test analysis was based on the following equation:

$$\text{Dependent} = c_0 + c_T \cdot \text{Treatment} + c_C \cdot \text{CF} + c_{TC} \cdot \text{Treatment:CF}$$

The analysis of variance shown whether the treatment has a direct effect on the dependent variable after discounting for the effect of the cofactor. In particular the interaction term revealed whether the cofactor influences the effect of the treatment.

Fifth, we computed the effect size for each dependent variable that is a quantitative measure of the strength of a phenomenon. While the Mann-Whitney U test allow for checking the presence of significant differences, it does not provide any information about the magnitude of such a difference. Accordingly, we used the non-parametric Cliff's delta (d) effect size [33]. The effect size is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$. The computation of Cliff's delta was performed using the *effsize* package [34].

Finally, regarding the analysis of post-experiment survey questionnaires, we reported the proportion of negative, neutral, and positive responses in each item. In addition, we applied a Kruskal-Wallis test to check whether a significant difference exists among the experiments of the family. The purpose of this test is checking whether the responses were similar in the different experiments.

In all the performed statistical tests, we decided, as it is customary, to accept a probability of 5% of committing Type-I-error (α) [15], i.e., rejecting the null hypothesis when it is actually true. Since we tested multiple hypotheses

[17] This is a simplified structure, since the variables are nominal they will be represented by $n - 1$ indicator variables, where n is the number of levels

(one per construct) on the same population we ought to counteract the problem of *multiple comparisons* and keep family-wise error rate under control. For this purpose we applied the Bonferroni correction [35]. Such technique prescribes using for the hypothesis rejection a corrected $\alpha_B = \alpha/n$, where n is the number of hypotheses. In addition, the correction was also applied to the analysis of the post-experiment survey questionnaires.

4 Results

This section first shows the analysis we conducted to study the distribution of the dependent variables, then it reports the results from the family of experiments, analysing the effect of the main factor and Maintainers' SE experience cofactor on the dependent variables. Finally, the results from the analysis of post-experiment questionnaires are reported.

4.1 Variables Distribution

Figure 8 reports the *TotalCorrectness* (i.e., sum of the score achieved for each individual MT) of all the participants to the family of experiments without partitioning by experiment and treatment. The distribution is highly skewed (Pearson's moment coefficient of skewness = -2.3). We can observe that in 69.7% of the tasks, the subjects achieved the maximum score (i.e., 16).

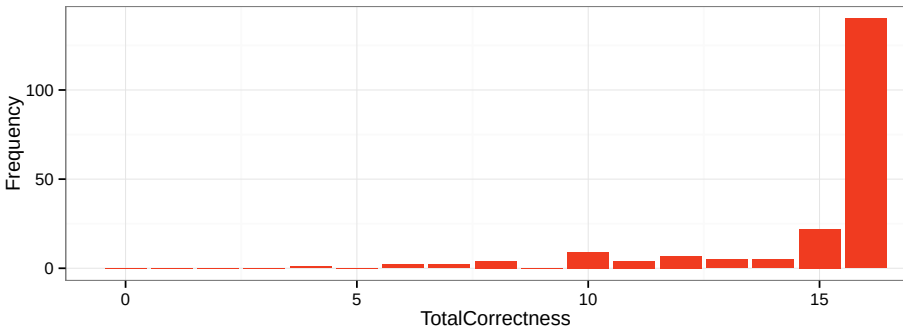

Fig. 8 Histogram of *TotalCorrectness*

Considering the different experiments (but without partitioning by treatment) we observe a significant difference among them (Kruskal-Wallis test $p < 0.01$), as shown in the boxplot of Figure 9. From the boxplot it is evident that UniGE-BS1 students reached the worst *TotalCorrectness* while UniGE-MS and UniGE-BS2 the best ones. Given these differences, we cannot simply merge the data from the five experiments. As a consequence, the five data sets ought to be analysed separately and then we will draw joint conclusions from the results.

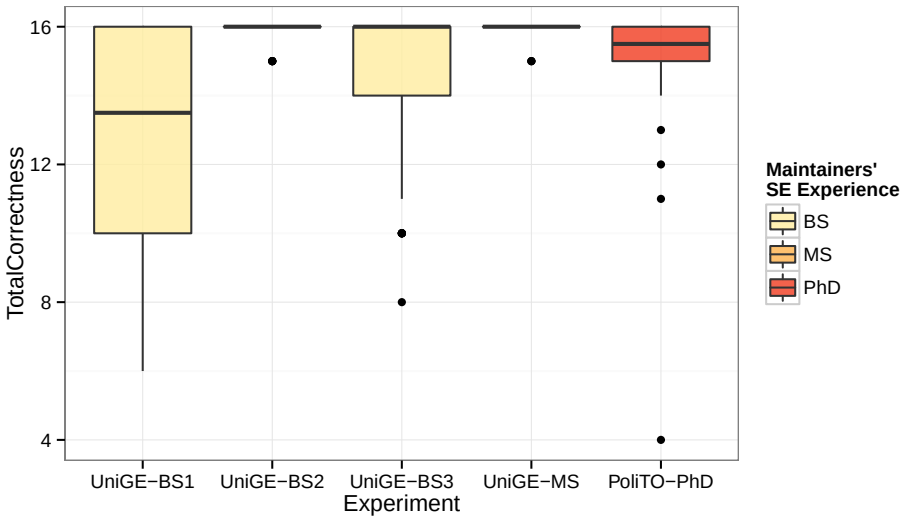

Fig. 9 *TotalCorrectness* per Experiment

The difference among the experiments in terms of *TotalCorrectness* can be explained slightly more by Experiment factor than Maintainers' SE experience factor, as shown by the permutation test reported in Table 5. This can be observed by looking at the sum of squares. Hereafter, values in bold are significant.

Table 5 Permutation test of *TotalCorrectness* vs. Maintainers' SE experience and Experiment

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Maintainers' SE experience	2	123.16	61.58	500000	**<0.01**
Experiment	2	184.83	92.42	500000	**<0.01**
Residuals	196	850.96	4.34		

Looking at the time to complete all the tasks (*TotalTime* variable) for each student without partitioning by experiment and treatment we can assume a log-normal distribution which is visually confirmed by the QQ-plot (not shown in the paper) and by the Shapiro-Wilk test (p-value = 0.37). The same is true considering the five distributions separately (i.e., executing the analysis partitioning by experiment).

Then, we performed the same check for log-normality at individual task time level (i.e., considering the times for each implemented maintenance task in separate way). We can conclude that not all individual MT times are log-normally distributed.

Considering the different experiments (but without partitioning by treatment) we observe a significant difference in *TotalTime* (Kruskal-Wallis $p <$

0.01) among them as shown in the boxplot of Figure 10. From the boxplot it is evident that PoliTO-PhD students were the fastest students to execute the maintenance tasks. This is not surprising since in our family of experiments they are the more skilled category of participants. Thus, similarly to the *TotalCorrectness* variable, we cannot simply merge the data from the five experiments but we have to analyse them separately.

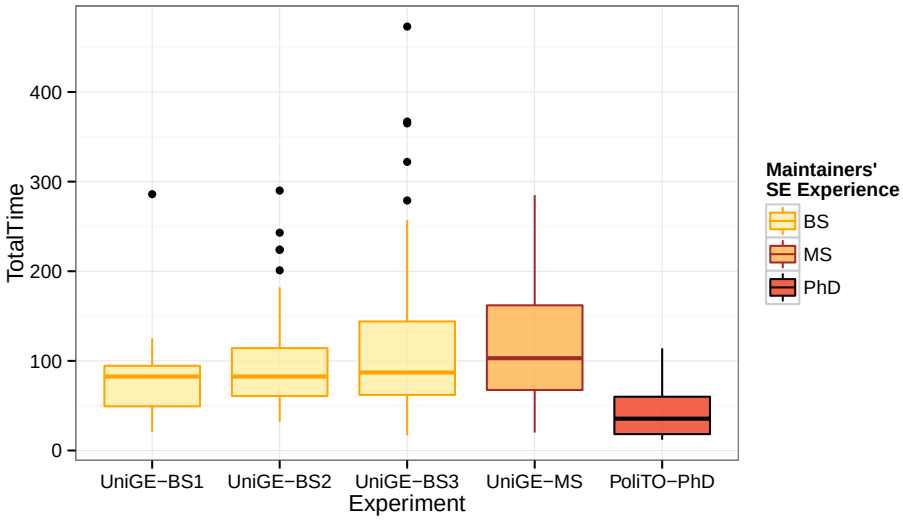

Fig. 10 *TotalTime* per Experiment

The difference among the experiments in terms of *TotalTime* can be explained more by Maintainers' SE experience factor than Experiment factor than, as shown by the permutation test reported in Table 6.

Table 6 Permutation test of *TotalTime* vs. Maintainers' SE experience and Experiment

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Maintainers' SE experience	2	82287.92	41143.96	500000	**<0.01**
Experiment	2	34414.67	17207.33	500000	**0.03**
Residuals	196	943493.95	4813.74		

Similarly to the *TotalTime* variable, for the efficiency of all the tasks (*TotalEfficiency*) without partitioning by experiment and treatment we can assume a log-normal distribution which is visually confirmed by the QQ-plot (not shown in the paper) and by the Shapiro-Wilk test (p-value = 0.34). The same is true considering the five distributions separately.

We then perform the same check for log-normality at the individual task efficiency level. We can conclude that: not all individual MT efficiencies are log-normally distributed.

Considering the different experiments (but without partitioning by treatment) we observe a significant difference (Kruskal-Wallis < 0.01) among them as shown in Figure 11. From the boxplot it is evident that PoliTO-PhD students were the most efficient. Thus, similarly to the other two variables, we cannot simply merge the data from the five experiments but we have to analyse them separately.

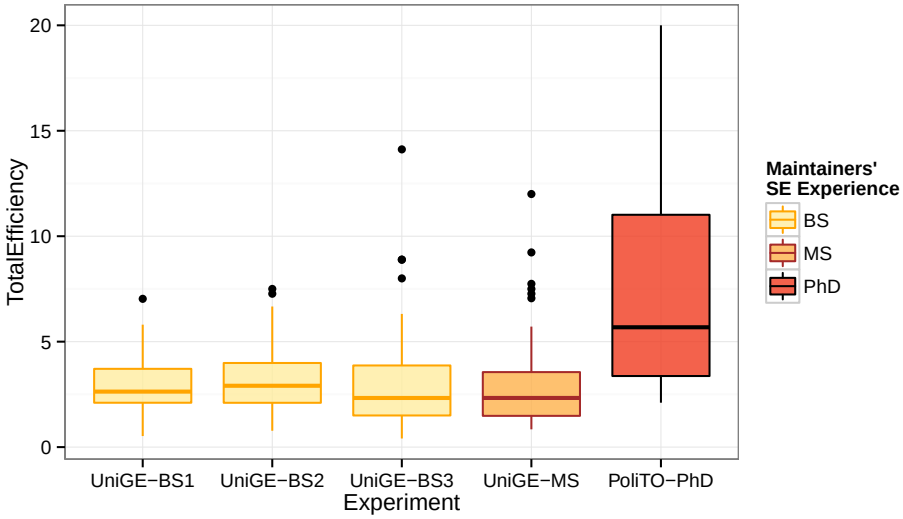

Fig. 11 *TotalEfficiency* per Experiment

The difference among the experiments in terms of *TotalEfficiency* is explained only by the Maintainers' SE experience factor, as shown by the Permutation test reported in Table 7.

Table 7 Permutation test of *TotalEfficiency* vs. Maintainers' SE experience and Experiment

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Maintainers' SE experience	2	117.28	58.64	473454	**<0.01**
Experiment	2	1.62	0.81	5882	0.89
Residuals	196	1559.88	7.96		

4.2 Hypothesis Testing

4.2.1 H_{0a}: Correctness

Table 8 reports the essential descriptive statistics (i.e., number of participants[18], mean, standard deviation) of *TotalCorrectness*, the results of the Mann-Whitney test at the experiment level conducted on data from the five experiments with respect to this dependent variable, and the Cliff's delta effect size (d). The significance level of the test after the Bonferroni correction is 0.016.

Table 8 Summary statistics for *TotalCorrectness* at Experiment level

	Java only			UniMod			MW	Cliff
Exp	n	mean	sd	n	mean	sd	p-value	d
PoliTO-PhD	11	14.18	3.52	11	14.91	1.76	0.62	0.10
UniGE-BS1	18	12.44	3.35	16	13.00	3.60	0.65	-0.05
UniGE-BS2	20	15.90	0.31	20	15.85	0.37	0.09	0.23
UniGE-BS3	26	13.96	2.62	27	15.04	2.08	0.53	0.04
UniGE-MS	25	15.92	0.28	27	15.96	0.19	0.70	0.10

Figure 12 summarizes the distribution of *TotalCorrectness* by means of boxplots for the family of experiments. Observations are grouped by treatment (Java only or UniMod) and shown partitioning by experiment. The y-axis represents the cumulative correctness of the four maintenance tasks: *TotalCorrectness* = 16, as already said, represents the maximum value of correctness and corresponds to four (completely) correct tasks.

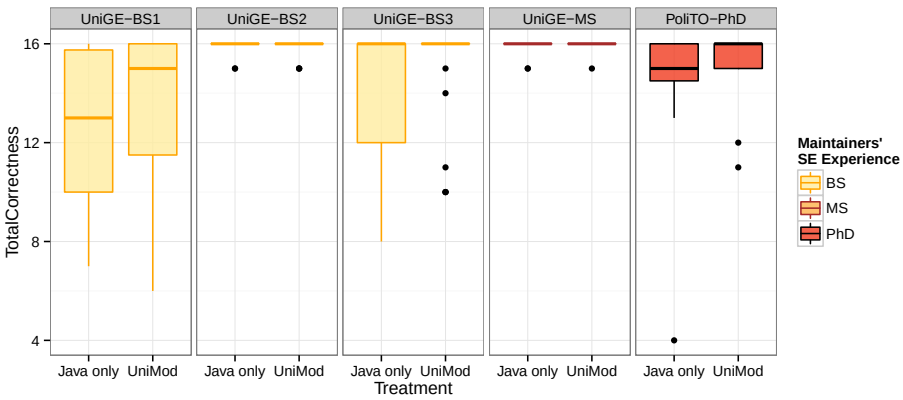

Fig. 12 Boxplot of *TotalCorrectness* vs. Treatment per Experiment

[18] In some experiments of the family, the number of participants is different between the two treatments since some of them took part in a laboratory session only.

The boxplots show that the PoliTO-PhD, UniGE-BS1, and UniGE-BS3 subjects achieved a better correctness level when accomplishing the tasks with UniMod. However, for all the experiments of the family the differences are not statistically significant as indicated in Table 8. Therefore, for all the experiments, we cannot reject the null hypothesis H_{0a}. The Cliff's delta effect size (d) is negligible in all the experiments with the exception of UniGE-BS3 where it is small.

Looking at each maintenance task in each experiment, we cannot find a case out of 20 (5 experiments x 4 MTs) where a statistically significant difference in terms of correctness is observed.

A more comprehensive analysis that takes into consideration also the Maintainers' SE experience as a cofactor can be conducted by means of a permutation test, whose results are shown in Table 9. This analysis highlights the significant effect of Maintainers' SE experience cofactor and confirms the results of the Mann-Whitney test. No interaction is observed between Maintainers' SE experience and treatment.

Table 9 Permutation test of *TotalCorrectness* vs. Treatment and Maintainers' SE experience

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Maintainers' SE experience	2	83.29	41.65	500000	**<0.01**
Treatment	1	6.82	6.82	31240	0.24
Maintainers' SE experience:Treatment	2	3.59	1.80	12002	0.71
Residuals	195	1019.97	5.23		

4.2.2 H_{0b}: Time

Table 10 reports some descriptive statistics of *TotalTime* variable, the results of Mann-Whitney test and the effect size. On the other hand, Figure 13 summarizes the distribution of the *TotalTime* variable by means of boxplots. Observations are grouped by treatment (Java only or UniMod) and shown partitioning by experiment.

Table 10 Summary statistics for *TotalTime* at Experiment level

	Java only			UniMod			MW	Cliff
Exp	n	mean	sd	n	mean	sd	p-value	d
UniGE-BS1	18	82.11	30.59	16	75.00	60.47	0.10	-0.33
UniGE-BS2	20	142.55	66.59	20	60.90	22.75	**<0.01**	-0.79
UniGE-BS3	26	150.42	90.89	27	89.67	89.96	**<0.01**	-0.58
UniGE-MS	25	130.80	65.70	27	101.22	65.52	0.08	-0.28
PoliTO-PhD	11	56.82	33.61	11	31.36	20.88	0.05	-0.51

The boxplots show that students using the code-centric approach consistently employed more time than students using UniMod in all the experiments

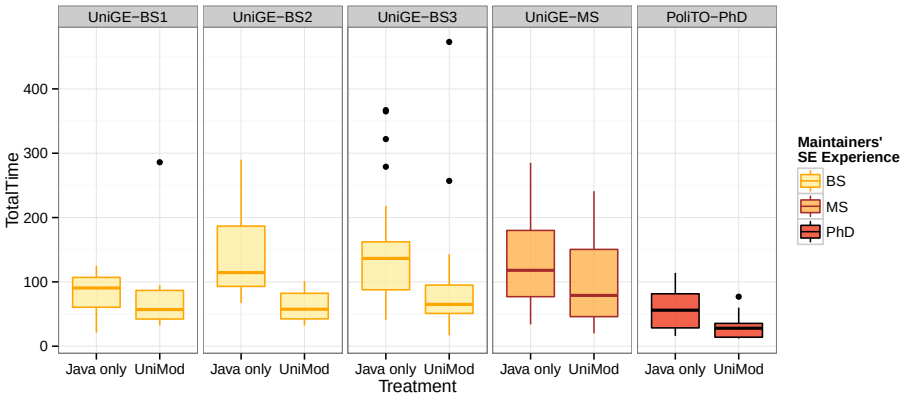

Fig. 13 Boxplot of *TotalTime* vs. Treatment per Experiment

of the family. However, the differences are statistically significant only for UniGE-BS2 and UniGE-BS3 as indicated in bold in Table 10 (we remind that the significance level of the test after the Bonferroni correction is 0.016). Therefore, we cannot reject the null hypothesis H_{0b} for UniGE-BS1 and UniGE-MS only. The effect size is large for the three significant experiments and for PoliTO-PhD, medium for UniGE-BS1, and small for UniGE-MS.

Looking at each maintenance task in each experiment, we can find 4 cases (out of 20) where a statistically significant difference in Time is observed. More in detail, we observed statistically significant differences in MT4 for three experiments (except UniGE-BS1 and PoliTO-PhD), and for MT3 for experiment UniGE-BS3.

A more comprehensive analysis that takes into consideration also the Maintainers' SE experience as a cofactor can be conducted by means of a Permutation test, whose results are shown in Table 11. This analysis highlights the significant effect of Maintainers' SE experience cofactor on the TotalTime variable and the overall effect of the treatment. No interaction is observed between Maintainers' SE experience and treatment.

Table 11 Permutation test of *TotalTime* vs. Treatment and Maintainers' SE experience

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Maintainers' SE experience	2	83049.86	41524.93	500000	**<0.01**
Treatment	1	39411.24	39411.24	500000	**0.01**
Maintainers' SE experience:Treatment	2	6565.97	3282.98	30147	0.49
Residuals	195	877338.56	4499.17		

4.2.3 H_{0c}: Efficiency

Table 12 reports some descriptive statistics of TotalEfficiency variable, the results of Mann-Whitney test, and the effect size. On the other hand, Figure

14 summarizes the distribution of the TotalEfficiency variable by means of boxplots. As for the previous cases, observations are grouped by treatment (Java only or UniMod) and shown partitioning by experiment location.

Table 12 Summary statistics for *TotalEfficiency* at Experiment level

	Java only			UniMod			MW	Cliff
Exp	n	mean	sd	n	mean	sd	p-value	d
UniGE-BS1	18	2.65	1.32	16	3.38	1.62	0.10	0.34
UniGE-BS2	20	2.05	0.91	20	4.48	1.69	**<0.01**	0.82
UniGE-BS3	26	1.96	1.28	27	4.10	3.00	**<0.01**	0.54
UniGE-MS	25	2.44	1.48	27	3.77	2.83	0.08	0.28
PoliTO-PhD	11	5.30	4.10	11	10.43	6.52	0.02	0.58

The boxplots show that students using UniMod outperform in terms of efficiency students using code-centric programming in all the experiments of the family. The differences are statistically significant for UniGE-BS2 and UniGE-BS3 as indicated in bold in Table 12 (the significance level of the test after the Bonferroni correction is 0.016). Therefore, we cannot reject the null hypothesis H_{0c} for PoliTO-PhD, UniGE-BS1, and UniGE-MS only. The effect size is large for the two statistically significant experiments and PoliTO-PhD, medium for UniGE-BS1, and small for UniGE-MS.

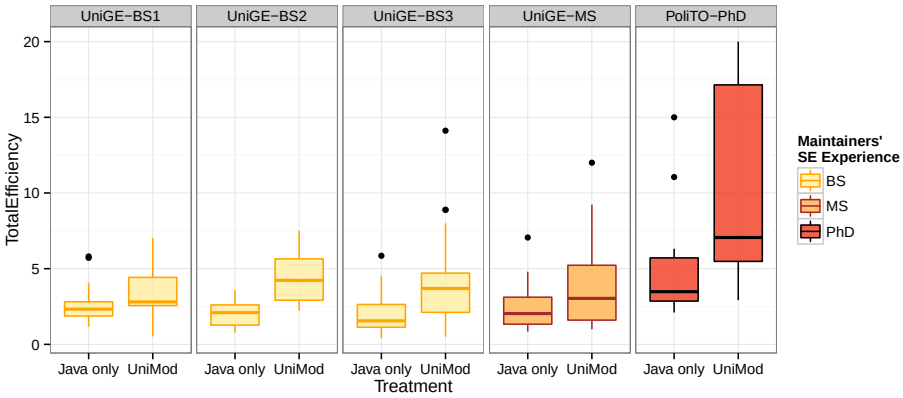

Fig. 14 Boxplot of *TotalEfficiency* vs. Treatment per Experiment

Looking at each maintenance task in each experiment, we can find 4 cases (out of 20) where a statistically significant difference is observed. They are: MT3 for UniGE-BS2 and UniGE-BS3 and MT4 for UniGE-BS2 and UniGE-MS.

A more comprehensive analysis that takes into consideration also the Maintainers' SE experience as a cofactor can be conducted by means of a permutation test, whose results are shown in Table 13. This analysis highlights: (1) the

Table 13 Permutation test of *TotalEfficiency* vs. Treatment and Maintainers' SE experience

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Maintainers' SE experience	2	444.08	222.04	500000	**<0.01**
Treatment	1	238.74	238.74	500000	**<0.01**
Maintainers' SE experience:Treatment	2	59.90	29.95	500000	**0.01**
Residuals	195	1283.93	6.58		

Table 14 Summary statistics (as percentages) for the Post-Questionnaire without partitioning for Experiment

Item	Neg.	Neut.	Pos.	KW p
PQa: The maintenance tasks were perfectly clear to me	5%	13%	82%	0.04
PQb: I had no problems understanding the Java code	21%	32%	46%	**0.005**
PQc: I had no problems understanding the UniMod models	7%	15%	78%	0.10
PQd: The provided documentation was useful and clear	7%	14%	79%	0.11
PQe: I found the exercise useful	5%	12%	83%	0.05
PQf: Change localization in the UniMod model is simpler than in the code	2%	11%	87%	0.88
PQg: Modifying the UniMod model is simpler than modifying the code	1%	15%	84%	0.16

significant effect of Maintainers' SE experience cofactor on the *TotalEfficiency* variable, (2) the overall effect of the treatment, and (3) an interaction between Maintainers' SE experience and treatment.

4.3 Analysis of Post-experiment Survey Questionnaire

The overall results from the post-questionnaires are summarized in Table 14. The Table reports the proportions of negative, neutral, and positive answers. In this summary we do not differentiate between strong and weak positive or negative answers.

Question PQa concerns the overall design of the family of experiments. We observe a high proportion of positive answers, 81.8%. Questions PQb through PQd concern the issues encountered during the execution of the tasks. Here, we observe a relatively low value for PQb – I had no problems to understand the provided Java code – while PQc (77.8% positive answers) and PQd (78.8% positive answers) confirm the clarity of UniMod models and of the provided documentation, respectively. PQe concerns ethical issues, since each experiment of the family represent a lab assignment within the regular schedule of a course. We can confirm a high perceived usefulness of the assignment (82.8%). Questions PQf and PQg focus on the usefulness of the UniMod models: Uni-

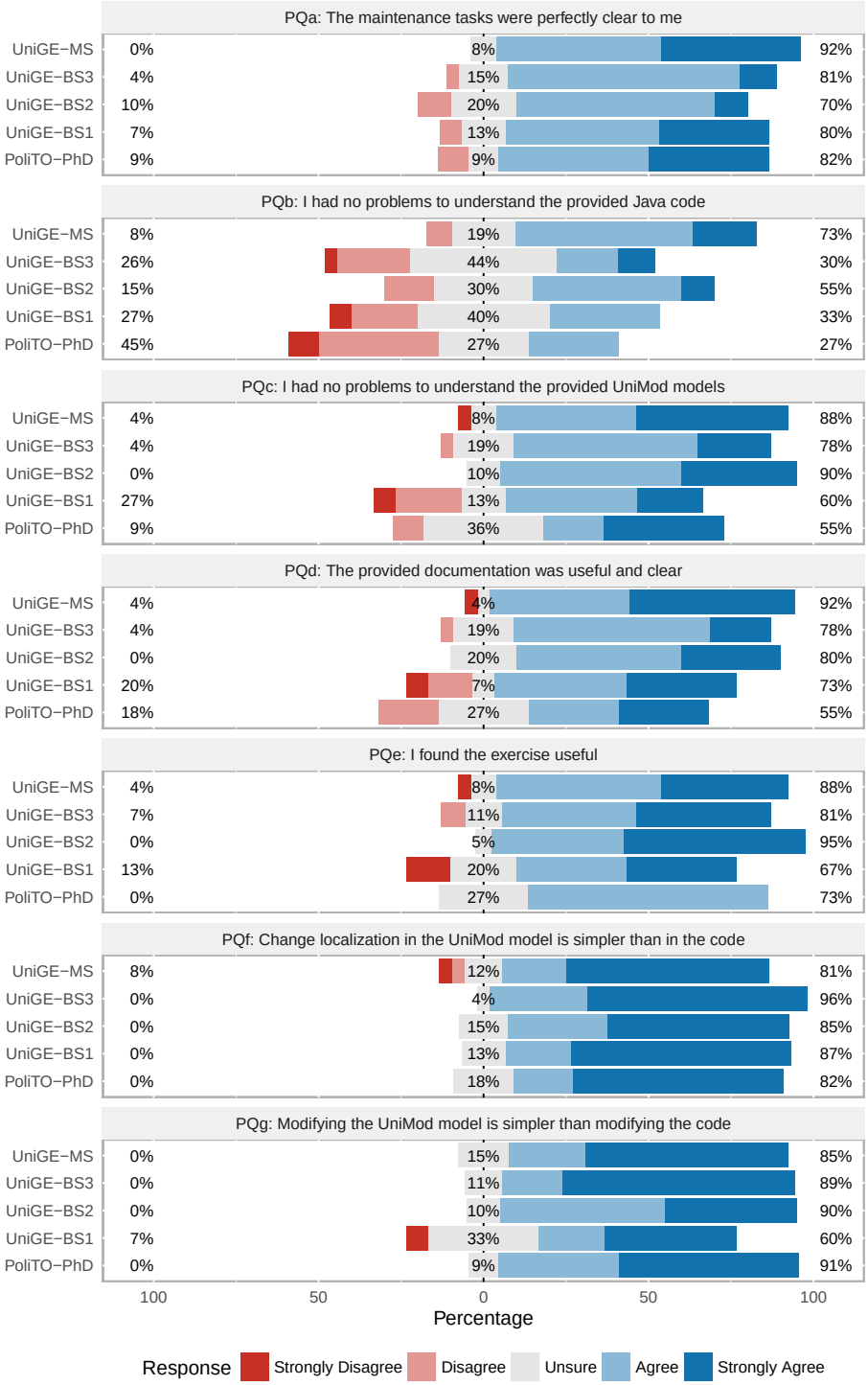

Fig. 15 Detailed post questionnaire responses

Mod makes it easier to both locate the defects (86.9% positive answers) and perform the changes (83.8%).

In addition, we looked at individual level of experiment and relative differences among experiments. The detailed results for each experiment, grouped by question, are reported in Figure 15 using a diverging stacked bar chart [36].

In order to detect statistically significant differences between experiments for any of the questions, we applied the Kruskal-Wallis test. Since the statistical tests are applied to different features of the same participants (multiple comparisons), a Bonferroni correction is necessary to keep the familywise error rate under control. For this reason, we will consider significant a difference only if p-value < 0.0071[19]. The Kruskal-Wallis p-values are reported in the rightmost column of Table 14 with statistically significant results highlighted in bold. We observe that only for PQb the difference is significant.

5 Discussion

5.1 Discussion of Results

Concerning the general distributions of the variables we can observe that:

- *TotalCorrectness* is highly skewed and compressed against the maximum value. This is probably due to the fact that the tasks were easy enough to be completed within the allowed total time (three hours and half for each lab).
- *TotalTime* is log-normal at experiment level, though not at change request level.
- *TotalEfficiency* is log-normal at experiment level, though not at change request level.

All the three dependent variables differ significantly among the experiments of the family, but such difference seems to be also explainable in terms of Maintainers' SE experience. More in details, for *TotalCorrectness*, Experiment apparently explains more variation than the Maintainers' SE experience (see Table 5), on the contrary for *TotalTime* is true the opposite (see Table 6). Finally, for *TotalEfficiency* the Experiment is not able to explain anything that is not already explained by Maintainers' SE experience (see Table 7).

As far as the hypotheses are concerned the results show:

- The treatment has no significant effect on the correctness measured as *TotalCorrectness*. Maintainers' SE experience has a significant effect even when accounting for the other effects.
- The usage of a State-based Model-Driven Development (SbMDD, see Section 6) approach and a tool, such as UniMod, has a significant effect on the time to complete the tasks in two out of five experiments. Maintainers' SE experience also has an effect, independent from the treatment.

[19] 0.05 divided by the number of comparisons, i.e., 7.

– The treatment has a significant effect also on efficiency to conduct the tasks in two out of five experiments. We observe a combined effect (interaction) of Maintainers' SE experience and Treatment as shown in Table 7 and Figure 16.

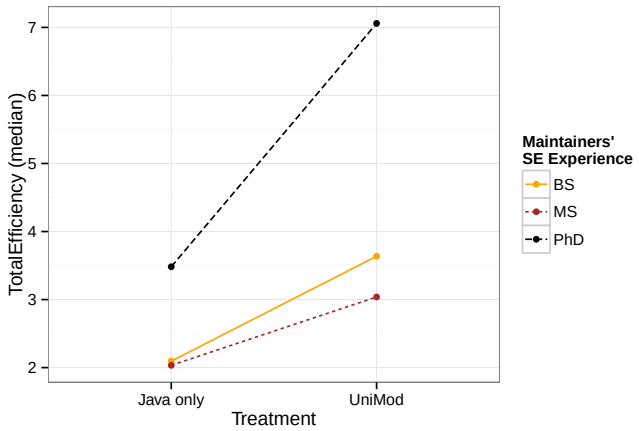

Fig. 16 Interaction plot of *TotalEfficiency* vs. Treatment and Maintainers' SE experience

Focusing on efficiency, it is interesting to quantify the practical effect of the treatment. For this purpose, we fit a linear model, including Maintainers' SE experience and Treatment as predictors (independent variables) and Efficiency as the outcome (dependent variable). As it is customary in this kind of context, the categorical variable Maintainers' SE experience with three levels, has been transformed into a pair of mutually exclusive binary variables: *MS* and *PhD* whose values can be mapped into the levels as shown in Table 15. Concerning the Treatment, it has been mapped to a binary variable *UniMod*; value 0 corresponds to treatment "-" (Java only) and value 1 corresponds to "+" (UniMod).

Table 15 Mapping of *MS* and *PhD* to Maintainers' SE experience levels

		PhD	
		0	1
MS	0	BSc	PhD
	1	MSc	–

The resulting equation is (the coefficients in bold are statistically significant):

$$\begin{aligned} \mathit{Efficiency} = \mathbf{2.18} + (0.25 \cdot \mathit{MS}) + (\mathbf{3.12} \cdot \mathit{PhD}) + (\mathbf{1.86} \cdot \mathit{UniMod}) + \\ (-0.52 \cdot \mathit{UniMod} \cdot \mathit{MS}) + (\mathbf{3.27} \cdot \mathit{UniMod} \cdot \mathit{PhD}) \end{aligned} \quad (2)$$

We observe that:

1. using a SbMDD approach, such as UniMod (vs. code-centric programming) provides an increment of 85% over the mean efficiency — computed considering the intercept (2.18) as base value and the UniMod coefficient as increment (1.86);
2. benefits of a SbMDD approach are relatively small (3/5 circa) compared to those provided by a higher Maintainers' SE experience (i.e., PhD versus bachelor students), that corresponds to an increment of 140% of the mean efficiency;
3. the Maintainers' SE experience has a sort of multiplicative effect; the contemporary presence of a high SE experience (PhD) and of a SbMDD approach brings an additional 150% improvement. In practice, UniMod almost doubles the efficiency, but in presence of a higher experience the efficiency is three times higher.

Concerning the post-questionnaires, the results substantially confirm the results of the family of experiments. UniMod artifacts are easier to understand than Java code (PQc vs. PQb) and UniMod makes it easier to both locate the defects (PQf) and perform the changes (PQg). Moreover, the goodness of the experimental procedure and material is shown by the high percentages of positive answers obtained for PQa and PQd, closed to 80%. Finally, the high percentages of positive answers for PQe highlights the usefulness of the family of experiments from a pedagogical and ethical point of view.

Considering the only post-experiment question showing a significant difference among experiments, PQb, we have to highlight that only the Master students from the UniGE-MS experiment claimed to have had a few problems in understanding the Java code (see Figure 15) while the other students were more pessimistic about this question. The most negative about this question were the PoliTO-PhD students. We speculate that this result is attributable to the following two facts: a) Master students in Computer Engineering had the best level of knowledge on Java programming (see Section 3.1) and were the more trained in development of systems and b) some PoliTO-PhD students were, probably, rusty with Java.

5.2 Implications

We adopted a perspective-based approach to judge the practical implications of our study, taking into consideration the *practitioner/consultant* (simply "practitioner", from here on) and *researcher* perspectives suggested by [37]:

- Using a SbMDD approach, such as for example UniMod, instead of code-centric programming yields, during maintenance activities, an average improvement in term of developers' efficiency of about 85% for all the categories of students. This result is relevant for the practitioner. The question of whether (or not) to adopt SbMDD approaches to reduce maintenance costs has so far been few investigated through controlled experiments.

Thus, the contribution of our family of experiments is one of the first empirical evidence that in this context some benefits exist. This result is relevant from the researcher perspective since our study poses the basis for future work.

- Using a SbMDD approach induces no additional time burdens w.r.t. code centric programming. Indeed, the usage of UniMod reduces the maintenance time for each kind of participants, with no significant impact on correctness of the executed tasks. Also this result is relevant for the practitioner and the researcher.
- The benefits deriving from adopting a SbMDD approach are appreciable for all categories of developers (although to varying degrees depending on SE experience, see next point). This result is practically relevant for the industry: developers with different profiles and different Maintainers' SE experience can all benefit from the usage of a SbMDD approach. This aspect is clearly relevant also for the researcher.
- The Maintainers' SE experience has a sort of multiplicative effect. In practice, using a SbMDD approach almost doubles the developers' efficiency, but in presence of a higher SE experience the efficiency is three times higher. This result is particularly relevant for the practitioner interested to introduce a SbMDD approach in his/her company.
- The effect of adopting a Model-driven approach in maintenance scenarios has been rarely empirically assessed. This is relevant for the researcher, who could be interested in assessing why a SbMDD approach reduces the maintenance time with no significant impact on correctness. Our family of experiments poses the basis for this kind of investigations.
- The study tests only one of claimed benefits of MDD approaches (and as a consequence of the SbMDD approaches), i.e., improvement in maintainability. This is relevant for the researcher, who could be interested in testing other claimed advantages of MDD, such as improvement in productivity, portability, and interoperability.
- The study is focused on desktop applications for simulating a smart traffic light system and a mobile phone payment terminal. From the researcher perspective, the effect of a SbMDD approach on different type of applications represents a possible future direction.
- Even if the maintenance tasks considered in this study belongs to different categories (enhancive, reductive, and corrective tasks [27]) we designed the experiment not having in mind a specific categorization of maintenance tasks, so the individual contribution of each category has not been studied. A work devoted to explicitly study this aspect is needed. This is of particular interest for the researcher.
- We considered here models developed with UniMod, i.e., a SbMDD approach. Using different MDD approaches could lead to different results. This aspect is relevant for practitioners, interested in understanding the best (or most suitable for her/his company) MDD tool, and researchers, interested in investigating if and why a different approach should affect maintainability.

- The software projects selected in this family of experiments are small-sized but realistic. We have no evidence whether and how the achieved results could scale up to large industrial projects. Starting from the evidence provided in our work, further empirical studies devoted to explicitly study this aspect can be planned. This is of interest for the practitioner and the researcher alike.
- The switch to a MDD process requires a complete and radical process change in an interested company. This consideration is extremely relevant for the practitioner.
- The adoption of MDD approaches should also consider the costs (e.g., training and cost of supporting tools) and problems for its introduction and usage in a company. This aspect, not considered in this paper but studied in [2], is of particular interest for the practitioner. In fact, it would be crucial knowing whether the additional cost needed to introduce and use a MDD tool is adequately paid back by the improved maintainability.
- The diffusion of a new technology/approach is made easier when empirical evaluations are performed and their results show that such a technology/approach solves actual issues [38]. This is why the results of our family of experiments could increase the diffusion of State-based MDD in the software industry. This is of particular interest for the practitioner.

5.3 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 factors that may affect a dependent variable (in our case, *TotalCorrectness*, *TotalTime*, and *TotalEfficiency*). Since the students had to participate in two labs (four maintenance tasks each), a learning/fatigue effect may intervene. However, the students were previously trained and the chosen experimental design, with a break between the two labs, should limit this effect. Moreover, a two-way Permutation test was used to study the influence of the Lab order on the dependent variables *TotalTime* and *TotalEfficiency* (see Tables 19 and 20 in the Appendix), for which no significant effect was found in all the experiments of the family. For this reason, we can exclude a learning/fatigue effect. An additional threat concerns the specific applications used for the experiment; we went through the two systems and found no specific features that could bring any advantage to either UniMod or Java only. The possibility of developing the two experimental objects by ourselves was ruled out for several reasons. The main reason was because we wanted to avoid any bias linked to our expectations and skills (we detail it below). As described in Section 3.2, we believe that the best choice would have been finding two applications for which both versions (i.e., UniMod and Java only) were available. Given that (a) this was not possible (we did not find any application available in both versions) and (b) we were much more confident in Java development than in UniMod, we chose to select two

existing UniMod applications and then developing by ourself the corresponding Java only versions. Indeed, starting from existing UniMod applications has allowed us to create the corresponding "Java only" versions that, in our opinion, have a good quality level (provided we judge our own Java programming skills as fairly good); on the other hand, since we were not experts of UniMod development, in the case we would have chosen to develop from scratch the UniMod versions, we would ran the risk of introducing a confounding factor in the experiment due to poor/low quality UniMod implementations (i.e., implementations that cannot be considered at the same level w.r.t. the Java ones).

Another possible threat lies in the two hours training (see Section 3.1) that have been more focused on UniMod rather than Java. However, this is reasonable given that all the participants attended at least a course about Java programming. Finally, another threat to validity concerns the possible different level of expertise of the participants with respect to the two treatments (i.e., UniMod and Java only). Clearly, participants were more familiar with Java since they attended (at least) one Java programming course. However, we believe that with our two-hours training session the participants were able to reach an adequate UniMod proficiency, at least for what concerns the tasks to perform during the experiment.

Construct validity threats concern the relationship between theory and observation. They are mainly due to how we measure the capability of a subject to execute a maintenance task. Thus, this threat is related to how correctness and time were measured; we do not mention efficiency because it is a derived measure. The tasks were chosen to be as representative as possible of realistic maintenance tasks. Also, the measurements we conceived — test cases to assess maintenance tasks correctness — are as objective as possible. The correctness of each maintenance task was assessed by executing a manually created acceptance test suite and giving a score from 0 to 4 (for each maintenance task), according to the number of test cases passed. It is possible that the used test suite does not perfectly measure the quality of the maintenance tasks implementation. It was built to test only the correct execution of the maintenance tasks. For example, for the MT1 of Telepay (see Table 3) we built only a test case testing the reverse order of the phone number. The test cases we used cover only the portions of code we asked to modify in the maintenance tasks. Code not directly impacted by the change has not been tested. The time needed to perform a task was manually recorded by the subjects. This may have led to inaccuracies and therefore to an instrumentation effect. Our previous experiences [39] showed, however, that automated time recording can also lead to inaccuracies. Therefore, for the sake of simplicity, we decided to adopt manual time recording. In addition, we notice that the observed effects could be due to two distinct aspects of MDD: (i) the capability of generating code from the model, or (ii) the documental feature of the model. With the design we adopted it is not possible to discriminate the effects of those two aspects. We speculate that a purely documental non-executable

model is more likely to become quickly out-of-date. In such a case, the model is of little help to the comprehension, if not even it becomes a hurdle.

Threats to *conclusion validity* concern issues that may affect the ability of drawing a correct conclusion. They can be due to the sample size of the family of experiments (100 participants in total) that may limit the capability of statistical tests to reveal any effect and to the chosen statistical tests. In the family of experiments, we chose to use non parametric tests for testing the effect of the main factor (Mann Whitney U test, Kruskal-Wallis test and Cliff's delta effect size) due to the size of the sample and because we could not safely assume normal distributions [26]. Similarly, the analysis of co-factors has been performed using permutation test, which is a non-parametric alternative to ANOVA and does not require data to be normally distributed as ANOVA does. Post experiment questionnaires, mainly intended to get qualitative insights, were designed using standard structure and scales [40].

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. There are two threats concerning the objects of the experiment: — 1) the first is related to the technique used for the construction of the Java versions of Svetofor and Telepay starting from the UniMod projects. We applied a standard approach to implement state machines in Java (in practice, we used the nested switch strategy [21]) but we cannot be sure that changing the implementation (e.g., using the State pattern [21]) would not affect the outcome of the family of experiments. However, since the state machines of our objects are small, we expect that the results would not be changed too much in case one replaces the nested switch implementation with the State pattern. Usually, a difference between the two implementations is appreciated only with large state machines. We plan to verify this as part of our future work. — 2) the second pertains to the size of the state machines contained in the applications used in the experiment. The size is suitable for the time allotted to the experiment but is significantly smaller than most industrial models. It is possible that some phenomena do appear only as the size scales up and thus they are not visible in our experiment. For instance, it is possible that beyond a given size the complexity of the model makes it incomprehensible and the advantage over the code-centric approach is reduced or even reversed. A similar phenomenon has been observed in [24]. We speculate that when state models are concerned that phenomenon is less likely to occur; the reasons being that the state machine formalism is much easier than the Conallen notation used in [24] and the corresponding Java code is fairly complex. Nevertheless, at the current stage we have no evidence about this issue, so further experiments are required.

As far as the participants are concerned, we agree on the fact that it is more interesting to experiment with industrial developers than students, and we are also aware that the expertise of students could be far from that of professionals. However, finding professionals available to conduct a demanding experiment as we designed is not simple. Moreover, this threat was at least mitigated: (a) by considering students with different levels of education and (b) by performing a co-factor analysis by SE experience. Finally, 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 with UniMod (or maybe better whether our speculation about SE experience will be verified) as suggested by studies comparing the performance of students vs. professionals, e.g. [41] and [42]. Further controlled experiments with larger systems and more experienced developers (i.e., industrial developers) are needed to confirm or contrast the obtained results. Another threat to external validity is that the results of our family of experiments are only valid for SbMDD approaches/tools and rather different results could be obtained with other MDD approaches/tools. In particular, this can be true if the chosen MDD tool is not UML based (as for example Portofino[20]) or if it considerably differs from UniMod (as for example BridgePoint or AndroMDA that are not designed for implementing automata-based OO programs). An additional threat to the external validity is represented by the features of the maintenance tasks carried on during the experiment. Although we do not think that the maintenance tasks used in our experiment are not realistic, they pose two different threats: (i) they might cover only a subset of a realistic set of tasks that could happen in industrial settings; (ii) even if representative of a real task, in the context of larger models they could trigger different comprehension and localization processes. In both cases we do not possess any evidence about whether they represent concrete threats to the validity of our conclusions. We definitely need further studies in this direction.

6 Related Work

The idea of using state-based graphical representations of systems as basis for generating the corresponding executable code emerged many years ago. For instance, the visual language Argos [43] has been proposed for programming reactive systems (i.e., systems in which the interactions with an environment are the prominent aspect). It is a language based on automata, with a syntax similar to that of Statecharts [44]. A graphical editor has been developed for Argos programs, as well as, a compiler allowing to generate executable code. Another example from the SbMDD category of tools is MathWorks Stateflow[21], an industrial-grade control logic tool used to model and simulate combinatorial and sequential decision logic based on state machines and flow charts. Stateflow is a mature tool, whose Statecharts-like language semantics has been defined (both denotational [45] and operational [46]) and that has been widely adopted both in the industry and in the academy (on May 2017, Google scholar returns 298 article entries when searching for the specific keyword "MathWorks Stateflow"). With Stateflow, similarly to UniMod, the developer has to define a model containing the state machine(s) representing the behaviour of the software system. Then, the developer can generate auto-

[20] `http://portofino.manydesigns.com/en`
[21] `http://www.mathworks.com/products/stateflow/`

matically from the model the corresponding C or C++ code using MathWorks Simulink Coder[22].

In recent years, several researchers (e.g., [47,48,39,49]) conducted empirical investigations to assess the costs and benefits of using UML documentation during the development of software systems. On the other hand, only a few studies [7,9,10,50,11] faced the problem of assessing the benefits from adopting MDD approaches on software development and maintenance (the lack of industrial evidence of the benefits is still one of the major current problems in model-driven engineering [3]). Finally, we report some recent surveys conducted to investigate the usage of UML and model-driven approaches [2,5,51, 50].

6.1 Effects of the UML Documentation

Arisholm *et al.* [47] reported on two experiments with students aimed to understand whether models help to make quicker and better changes to existing systems. The subjects in the treatment group were provided with UML documentation (in particular a use case diagram, sequence diagrams for each use case, and a class diagram), while the subjects in the control group executed the same tasks without UML documentation. Results show that using UML documentation helps in reducing the time required to make code changes, but when including also the time necessary to modify the diagrams, no savings in effort are visible. On the contrary, in terms of functional correctness of the changes, results indicate that using UML has a significant, positive impact on the most complex tasks.

Dzidek *et al.* [48] presented an experiment with professional developers to investigate the impact of using UML during the maintenance of a real system. The subjects in the treatment group worked with a UML-supported IDE, while the subjects in the control group executed the same tasks without UML. The main result of the experiment is that the subjects in the UML group had, on average, a statistically significant 54% increase in the correctness of the changes without impact on the time.

Both the aforementioned experiments [47,48] treat only the more extreme cases: no UML documents vs. UML documents perfectly aligned with the code. Differently, Leotta *et al.* [39], investigated the effect of two different variants of technical UML documentation during the execution of a maintenance task scenario: one with a better alignment w.r.t. code and the other less accurate. Results indicate a benefit of the 15% in terms of efficiency (computed as number of correct tasks per minute) when a more accurate UML documentation is used.

Table 16 reports a summary of the works: Arisholm *et al.* [47], Dzidek *et al.* [48], and Leotta *et al.* [39].

Anda *et al.* [49] described a qualitative case study aimed to identify the benefits as well as the difficulties (and their causes) resulting from the in-

[22] `http://www.mathworks.com/products/simulink-coder/`

Table 16 Effects of the UML Documentation: Summary of the considered works

Study	Kind	Treatments	Findings
Arisholm *et al.* [47]	Exp	UML vs. No UML	UML ⇒ – time required, + functional correctness
Dzidek *et al.* [48]	Exp	UML vs. No UML	UML ⇒ + 54% correctness, no impact on time
Leotta *et al.* [39]	Exp	UML More Aligned vs. UML Less Aligned	More Aligned ⇒ + 15% efficiency

troduction of the UML-based development in a large international project with 230 system developers, testers, and managers. Case study participants reported several benefits due to the adoption of the UML-based development like, for instance, improvements with traceability from requirements to code and development of test cases as well as in communication and documentation. On the other hand, there were also cons like the need for adequate training.

These works are substantially different from ours, given that all the participants followed a conventional code-based software development/maintenance process and UML models are used only for documentation purposes. However, it is interesting to note that, as we found in our family of experiments, the participants' SE experience consistently has a positive effect when adopting high level documentation (e.g., UML documents). Indeed, if we compare [47] with [48] it emerges that the time spent on updating the UML documents decreases from the 30-35% required by the students to the 13% required by the professional developers. Anda *et al.* [49] found that one of the factors that reduce the positive effects of adopting a UML-based development approach is the lack of an adequate training. Indeed, the participants to the case study, even if well-qualified developers, were almost novices at modelling with UML.

6.2 Benefits of Model-Driven Approaches

On a different front, few empirical studies report on experiences related to the adoption of MDD approaches [7,9,10,11]. In that context, the studies more closely related to ours are [7,9].

Martínez *et al.* [7] studied the impact of Model-driven Engineering (MDE) approaches on the *maintainability* of *web applications* by means of an experimental study conducted with 44 participants. In particular, they compared *WebML* [52], a MDE methodology, with a code-based methodology based on PHP. The dependent variables are: effectiveness, efficiency, perceived ease of use, and perceived usefulness. Results show that subjects performed better with WebML than with PHP in terms of effectiveness and efficiency (i.e., execution time, for which we obtained a similar result), although they showed a preference towards implementing maintenance tasks directly on the source

code. Besides the fact that, in our study, we employed UML diagrams (i.e., class and state machine diagrams) while Martínez *et al.* employed WebML, there are two other notable differences between our study and theirs: (1) they measured the correctness of the maintenance tasks on the modified WebML models while we measured the correctness executing the UniMod models and using a test suite; (2) the context (Desktop in this study vs. Web applications in [7]).

In a subsequent study, Martínez *et al.* [8] further analysed the impact of an MDE approach on the *maintainability* of *web applications* by comparing the model-driven and the code-centric paradigms. In particular, the authors compared the performance and satisfaction of software maintainers adopting two different development approaches, OOH4RIA [53], a MDD approach, and a code-centric approach based on Visual Studio .NET and the Agile Unified Process. The participants of the study were 27 master students. Each subject was asked to perform 10 maintainability tasks, five with the code-centric approach and five with the MDD one. The authors reported that the adoption of OOH4RIA greatly improves the actual performance of subjects carrying out maintainability tasks, while the satisfaction-related variables throws mixed results (i.e., some reveal significant differences among the methods while others not).

Papotti *et al.* [9] performed an experiment aimed at evaluating the usage of mechanisms for code generation from models in the *software development*. As in our family of experiments, the subjects were students (29 from the third and fourth years of Computer Science and Computer Engineering courses). The experiment aimed at analysing the effort required for implementing the CRUD (Create, Retrieve, Update, Delete) functions for a *web application* starting from *UML models* (i.e., class diagrams) of the application entity classes. In particular, the software development was performed: (1) manually (based on the classic software development life-cycle), and (2) by using code generation from models. Results show that the subjects spent less time (about 91%) to complete the implementation when following the code generation approach (on average 11 minutes) than when following the manual coding approach (on average 2 hours). The authors report that such a considerable advantage of the code generation approach is justified by the model-to-code transformations, that are able to automatically generate a substantial amount of source code from the predefined models. The two most relevant differences between our study and theirs are: (1) they focus the study on development activities, we focus on maintenance tasks, and (2) the context (Desktop in this study vs. Web applications).

Hovsepyan *et al.* [10] focused on *aspect-oriented (AO) modelling* and investigated the effect on *maintainability* of two alternatives for generating code from aspect-oriented models. More in detail they compared: (1) an all-aspectual process, preserving the AO paradigm throughout the system development stages (i.e., full adoption of the AO paradigm), where the AspectJ implementations are automatically generated from domain specific models, and (2) an aspect hybrid process, shifting from AO models to OO code (e.g.,

useful for preserving the OO know-how of the development team, if any), where the Java object-oriented implementations are automatically generated from the same models. The experiment involved 10 post-graduate researchers that were asked, similarly to our family of experiments, to perform two maintenance tasks (i.e., functionality addition and improvement) on two selected applications (a toll payment system and a pacemaker system). Results show that the all-aspectual process often provides shorter maintenance cycles. The main difference between our study and the one of Hovsepyan *et al.* [10] is that we compared a model-based software development process with a conventional code-centric one, while they compared two different alternatives for generating code from aspect-oriented models.

Kapteijns et al. [11] presented a case study in which a small application is developed with and without MDD. In practice, a small middleware application was rebuilt using MDD and the time needed was then compared to the development time of the original application. The authors report that approximately the 70% of the required functionality have been generated from models while the remaining parts, such as complex graphical user interfaces, have been manually created after the generation. 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.

Table 17 reports a summary of the works: Martínez *et al.* [7], Martínez *et al.* [8], Papotti *et al.* [9], Hovsepyan *et al.* [10], and Kapteijns et al. [11].

Table 17 Benefits of Model-Driven Approaches: Summary of the considered works

Study	Kind	Treatments	Findings
Martínez *et al.* [7]	Exp	WebML vs. Code	WebML ⇒ + effectiveness, + efficiency
Martínez *et al.* [8]	Exp	OOH4RIA vs. Code	OOH4RIA ⇒ + performance on maintainability tasks
Papotti *et al.* [9]	Exp	Code Generation Yes vs. No	CG Yes ⇒ – 91% time
Hovsepyan *et al.* [10]	Exp	AO paradigm Full vs. Hybrid	Full ⇒ often shorter maintenance cycles
Kapteijns et al. [11]	Case Study	MDD vs. No MDD	MDD ⇒ 2.6x productivity

Table 18 Usage of UML and Model-Driven Approaches: Summary of the considered works

Study	N		Survey Findings
Torchiano *et al.* [2]	155	Modelling ⇒	better: design support, documentation, maintenance, software quality
		MD ⇒	better: standardization, productivity, platform independence
		Problems:	effort, limited usefulness, lack of competencies and supporting tools
Agner *et al.* [5]	209	Facts:	45% of the participants use UML 23.1% of them know/use MD approaches Modelling is used for documentation and design
		MD ⇒	+ productivity/portability
Liebel *et al.* [51]	112	Facts:	Focussed on Embedded Systems Domain Model-based Engineering is widespread Models are the key artifacts
Hutchinson *et al.* [50]	250	Facts: MDE ⇒	85% of respondents make use of UML + productivity and maintainability

6.3 Usage of UML and Model-Driven Approaches

Torchiano *et al.* [2] presented the findings of a survey on the model-driven state of practice in the Italian software industry with 155 software professionals. The survey was performed to investigate: (1) the relevance of software modelling and model driven techniques in the Italian industry, (2) the attainable benefits from such techniques, and (3) the problems affecting the adoption of modelling and model driven techniques. Results show that: (1) software modelling and model driven techniques are very relevant in the Italian industry in terms of adoption spread, (2) the adoption of simple modelling brings common benefits (better design support, documentation improvement, better maintenance, and higher software quality), while model driven techniques make it easier to achieve: improved standardization, higher productivity, and platform independence, (3) among the identified problems, some hinders its adoption (too much effort required and limited usefulness) others prevents it (lack of competencies and supporting tools).

Agner *et al.* [5] presented the findings of a survey aimed at investigating the usage of UML and model-driven approaches in the context of the design of embedded software in Brazil with 209 participants. Concerning modelling, results show that, in general, the value of the modelling is perceived by most of the participants even though little used. UML is the dominant language for modelling since 45% of the 209 participants use UML and the use of modelling tools is widespread. The most relevant problems hindering the adoption of UML includes: the lack of skills, the lack of coherent tools, and the strict time requisites applicable to software development projects. The authors

found that: (1) modelling is primarily used for documentation and design but scarcely used for code generation, (2) model-centric approaches are currently not very popular, and (3) software engineers who work extensively on models are the more experienced ones. Concerning model-driven approaches, results show that 23.1% of the respondents know and use them, partially or not. Model-driven approaches are mainly perceived beneficial in terms of productivity and portability. However, MDE is currently far from achieving a mature level, considering its recent introduction in the software development.

Liebel *et al.* [51] carried out a survey on the use of model-based engineering in the embedded systems domain. The authors collected quantitative data from 112 participants and from the results emerges that model-based engineering is widespread in the embedded domain. Indeed, in this domain, models are the key artifacts of the development processes (e.g., they are used for simulation and code generation) and, thus their usage is not only limited to documentation purposes. In the embedded systems domain, models are also used for behavioural and structural consistency checking, as well as for test case generation, traceability, and timing analysis.

Hutchinson *et al.* [50] presented the results of an empirical study on the assessment of MDE in industry, having the goal of understanding its success or failure reasons. This work used three forms of investigation: questionnaires, interviews, and on site observations, having as target practitioners, MDE professionals, and companies practising MDE, respectively. The main results of that work can be summarized as follows: almost all the respondents believe that the use of MDE improved productivity and maintainability. It is interesting to note that, from the questionnaires emerges that almost 85% of respondents make use of UML and that Class diagram and State Machine (the UML diagrams employed by UniMod) are respectively the first and fifth most used kind of diagrams (Class diagram and State Machine are among the most known/used diagrams also according to two recent studies [19, 20]).

Table 18 reports a summary of the works: Torchiano *et al.* [2], Agner *et al.* [5], Liebel *et al.* [51], and Hutchinson *et al.* [50].

7 Conclusions and Future Work

In this paper, we have presented a family of five experiments, which involved 100 students (64 BS, 25 MS, and 11 PhD students; in groups sized 11 to 26 per experiment) having different levels of education, designed for assessing the use of a specific implementation of executable UML (i.e., UniMod) as an alternative to code-centric conventional programming. In particular, we focused our attention on maintenance activities.

The results given in terms of artefacts correctness and time to accomplish the maintenance tasks indicate a reduction in time with no significant impact on correctness, when UniMod is used instead of code-centric programming. Using the UniMod-MDD approach/tool during maintenance activities provides an increment of in terms of maintainers' efficiency. Moreover, we dis-

covered that the Maintainers' SE experience has a sort of multiplicative effect. In practice, using the UniMod-MDD approach/tool almost doubles the developers' efficiency, but in presence of a higher SE experience the efficiency is three times higher.

Future work will be devoted to replicate this family of experiments in different settings. 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., industrial developers), for other domains (e.g., Web applications) and for larger systems/tasks. In addition, we would like to repeat this empirical study using other MDD tools (e.g., WebRatio or BridgePoint) designed to support other kinds of models different from state machines.

References

1. T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.
2. M. Torchiano, F. C. A. Tomassetti, F. Ricca, A. Tiso, and G. Reggio, "Relevance, benefits, and problems of software modelling and model driven techniques–a survey in the Italian industry," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2110–2126, 2013.
3. G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. Cheng, P. Collet, B. Combemale, R. France, R. Heldal, J. Hill, J. Kienzle, M. Schöttle, F. Steimann, D. Stikkolorum, and J. Whittle, "The relevance of model-driven engineering thirty years from now," in *Proceedings of 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, ser. Lecture Notes in Computer Science, J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, Eds. Springer, 2014, vol. 8767, pp. 183–200. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11653-2_12
4. J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014.
5. L. T. W. Agner, I. W. Soares, P. C. Stadzisz, and J. M. Simão, "A Brazilian survey on UML and model-driven practices for embedded software development," *Journal of Systems and Software*, vol. 86, no. 4, pp. 997–1005, 2013.
6. A. G. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc, 2003.
7. Y. Martínez, C. Cachero, M. Matera, S. Abrahao, and S. Luján, "Impact of MDE approaches on the maintainability of web applications: an experimental evaluation," in *Proceedings of 30th International Conference on Conceptual Modeling (ER 2011)*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 233–246. [Online]. Available: http://dl.acm.org/citation.cfm?id=2075144.2075168
8. Y. Martínez, C. Cachero, and S. Meliá, "Empirical study on the maintainability of web applications: Model-driven engineering vs code-centric," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1887–1920, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9269-5
9. P. Papotti, A. do Prado, W. de Souza, C. Cirilo, and L. Pires, "A quantitative analysis of model-driven code generation through software experimentation," in *Advanced Information Systems Engineering*, ser. Lecture Notes in Computer Science, C. Salinesi, M. Norrie, and Ó. Pastor, Eds. Springer Berlin Heidelberg, 2013, vol. 7908, pp. 321–337. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38709-8_21
10. A. Hovsepyan, R. Scandariato, S. Van Baelen, W. Joosen, and S. Demeyer, "Preserving aspects via automation: A maintainability study," in *Proceedings of 5th International Symposium on Empirical Software Engineering and Measurement (ESEM 2011)*. Washington, USA: IEEE CS, 2011, pp. 315–324. [Online]. Available: http://dx.doi.org/10.1109/ESEM.2011.40

11. 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" (C2M 2009)*. CTIT Workshop Proceedings Series, 2009, pp. 22–33.
12. N. Mellegård, A. Ferwerda, K. Lind, R. Heldal, and M. R. V. Chaudron, "Impact of introducing domain-specific modelling in software maintenance: An industrial case study," *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 245–260, March 2016.
13. I. Sommerville, *Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
14. V. Gurov, M. Mazin, A. Narvsky, and A. Shalyto, "Tools for support of automata-based programming," *Programming and Computer Software*, vol. 33, pp. 343–355, 2007.
15. C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer-Verlag Berlin Heidelberg, 2012.
16. A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008.
17. F. Ricca, M. Leotta, G. Reggio, A. Tiso, G. Guerrini, and M. Torchiano, "Using UniMod for maintenance tasks: An experimental assessment in the context of model driven development," in *Proceedings of 4th International Workshop on Modeling in Software Engineering (MiSE 2012)*. IEEE, 2012, pp. 77–83. [Online]. Available: http://dx.doi.org/10.1109/MISE.2012.6226018
18. A. A. Shalyto and N. I. Tukkel, "SWITCH Technology: An Automated Approach to Developing Software for Reactive Systems," *Programming and Computer Software*, vol. 27, pp. 260–276, 2001.
19. G. Reggio, M. Leotta, and F. Ricca, "Who knows/uses what of the UML: A personal opinion survey," in *Proceedings of 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, ser. Lecture Notes in Computer Science, J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, Eds. Springer, 2014, vol. 8767, pp. 149–165. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11653-2_10
20. G. Reggio, M. Leotta, F. Ricca, and D. Clerissi, "What are the used UML diagram constructs? A document and tool analysis study covering activity and use case diagrams," in *Model-Driven Engineering and Software Development*, ser. Communications in Computer and Information Science, S. Hammoudi, L. F. Pires, J. Filipe, and R. C. das Neves, Eds. Springer, 2015, vol. 506, pp. 66–83. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-25156-1_5
21. M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
22. 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 30th International Conference on Software Engineering (ICSE 2008)*. New York, NY, USA: ACM, 2008, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368138
23. M. Ceccato, M. Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1040–1074, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9248-x
24. F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato, "How developers' experience and ability influence web application comprehension tasks supported by UML stereotypes: A series of four experiments," *IEEE Transactions on Software Engineering*, vol. 36, pp. 96–118, 2010.
25. M. Torchiano, G. Scanniello, F. Ricca, G. Reggio, and M. Leotta, "Do UML object diagrams affect design comprehensibility? Results from a family of four controlled experiments," *Journal of Visual Languages & Computing*, vol. 41, pp. 10–21, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.jvlc.2017.06.002
26. H. Motulsky, *Intuitive biostatistics: a non-mathematical guide to statistical thinking*. Oxford University Press, 2010. [Online]. Available: http://books.google.it/books?id=R477U5bAZs4C

27. N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance*, vol. 13, no. 1, pp. 3–30, Jan. 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=371697.371701
28. R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org/
29. S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 3, no. 52, 1965.
30. D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (4th Ed.)*. Chapman & All, 2007.
31. R. Baker, "Modern permutation test software," *In E. Edgington, editor, Randomization Tests, Marcel Decker*, 1995.
32. B. Wheeler, *lmPerm: Permutation tests for linear models*, r package version 2.0. [Online]. Available: http://CRAN.R-project.org/package=lmPerm
33. R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.
34. M. Torchiano, *effsize: Efficient Effect Size Computation*, 2015, r package version 0.5.5. [Online]. Available: http://CRAN.R-project.org/package=effsize
35. H. Abdi, *Bonferroni and Sidak corrections for multiple comparisons*. Sage, Thousand Oaks, CA, 2007.
36. R. Heiberger and N. Robbins, "Design of diverging stacked bar charts for likert scales and other applications," *Journal of Statistical Software*, vol. 57, no. 5, pp. 1–32, 4 2014. [Online]. Available: http://www.jstatsoft.org/v57/i05
37. B. Kitchenham, H. Al-Khilidar, M. Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu, "Evaluating guidelines for reporting empirical software engineering studies," *Empirical Software Engineering*, vol. 13, pp. 97–121, 2008.
38. S. L. Pfleeger and W. Menezes, "Marketing technology to software practitioners," *IEEE Software*, vol. 17, no. 1, pp. 27–33, 2000.
39. M. Leotta, F. Ricca, G. Antoniol, V. Garousi, J. Zhi, and G. Ruhe, "A pilot experiment to quantify the effect of documentation accuracy on maintenance tasks," in *Proceedings of 29th International Conference on Software Maintenance (ICSM 2013)*. IEEE, 2013, pp. 428–431. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2013.64
40. A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement*. London: Pinter, 1992.
41. M. Svahnberg, A. Aurum, and C. Wohlin, "Using students as subjects-an empirical evaluation," in *Proceedings of 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM 2008)*, October 2008, pp. 288–290.
42. D. G. Feitelson, "Using students as experimental subjects in software engineering research - A review and discussion of the evidence," *CoRR*, vol. abs/1512.08409, 2015. [Online]. Available: http://arxiv.org/abs/1512.08409
43. F. Maraninchi and Y. Rémond, "Argos: an automaton-based synchronous language," *Computer Languages*, vol. 27, no. 1–3, pp. 61 – 92, 2001, visual Formal Methods-VFM'99 Symposium. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0096055101000169
44. D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231 – 274, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167642387900359
45. G. Hamon, "A denotational semantics for stateflow," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 164–172. [Online]. Available: http://doi.acm.org/10.1145/1086228.1086260
46. G. Hamon and J. Rushby, "An operational semantics for stateflow," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5, pp. 447–456, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10009-007-0049-7
47. E. Arisholm, L. Briand, S. Hove, and Y. Labiche, "The impact of uml documentation on software maintenance: an experimental evaluation," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 365–381, June 2006.

48. W. J. Dzidek, E. Arisholm, and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of UML in software maintenance," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 407–432, May 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.15
49. B. Anda, K. Hansen, I. Gullesen, and H. K. Thorsen, "Experiences from introducing UML-based development in a large safety-critical project," *Empirical Software Engineering*, vol. 11, no. 4, pp. 555–581, Dec. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10664-006-9020-6
50. J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proceedings of 33rd International Conference on Software Engineering (ICSE 2011)*. New York, USA: ACM, 2011, pp. 471–480.
51. G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Assessing the state-of-practice of model-based engineering in the embedded systems domain," in *Proceedings of 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, ser. Lecture Notes in Computer Science, J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, Eds. Springer, 2014, vol. 8767, pp. 166–182. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11653-2_11
52. S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera, *Designing Data-Intensive Web Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.
53. S. Melia, J. Gomez, S. Perez, and O. Diaz, "A model-driven development for gwt-based rich internet applications with ooh4ria," in *Proceedings of 8th International Conference on Web Engineering (ICWE 2008)*, July 2008, pp. 13–23.

A Appendix

This appendix reports detailed analysis results. They are not essential to understand the paper, though they provide additional information and complement the results provided in the main body of the papers.

A.1 Detailed Hypotheses Testing

We report here the Heatmap graph concerning the hypotheses tested at the individual maintenance task (MT) level. In particular, Figure 17 reports the p-values of the Mann-Whitney tests; Figure 18 shows the Cliff's delta values.

A.2 Influence of Lab Order

As mentioned in Section 5.3, we performed a set of two-way permutation tests to check the effect of Lab order on the dependent variables. The results of the tests are presented in Table 19 for *TotalEfficiency*, Table 20 for *TotalTime*, and Table 21 for *TotalCorrectness*.

The tests consistently show neither a main effect from the Lab order nor any interaction with the experiment.

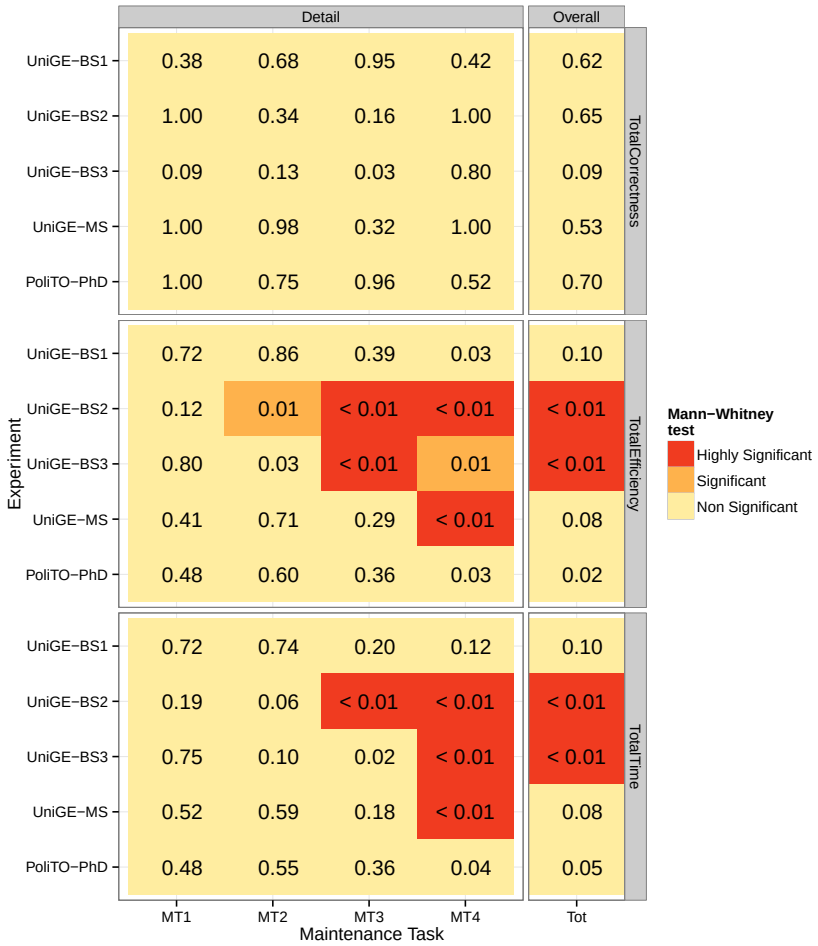

Fig. 17 Heat map of MW p-values

Table 19 Permutation test of *TotalEfficiency* vs. Experiment and Lab order

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Lab	1	6.73	6.73	51	1.00
Experiment	4	210.24	52.56	500000	**<0.01**
Lab:Experiment	4	6.38	1.60	11624	0.78
Residuals	982	3720.85	3.79		

Table 20 Permutation test of *TotalTime* vs. Experiment and Lab order

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Lab	1	1196.21	1196.21	6487	0.61
Experiment	4	61762.18	15440.55	500000	**<0.01**
Lab:Experiment	4	6503.89	1625.97	27078	0.61
Residuals	982	2557659.83	2604.54		

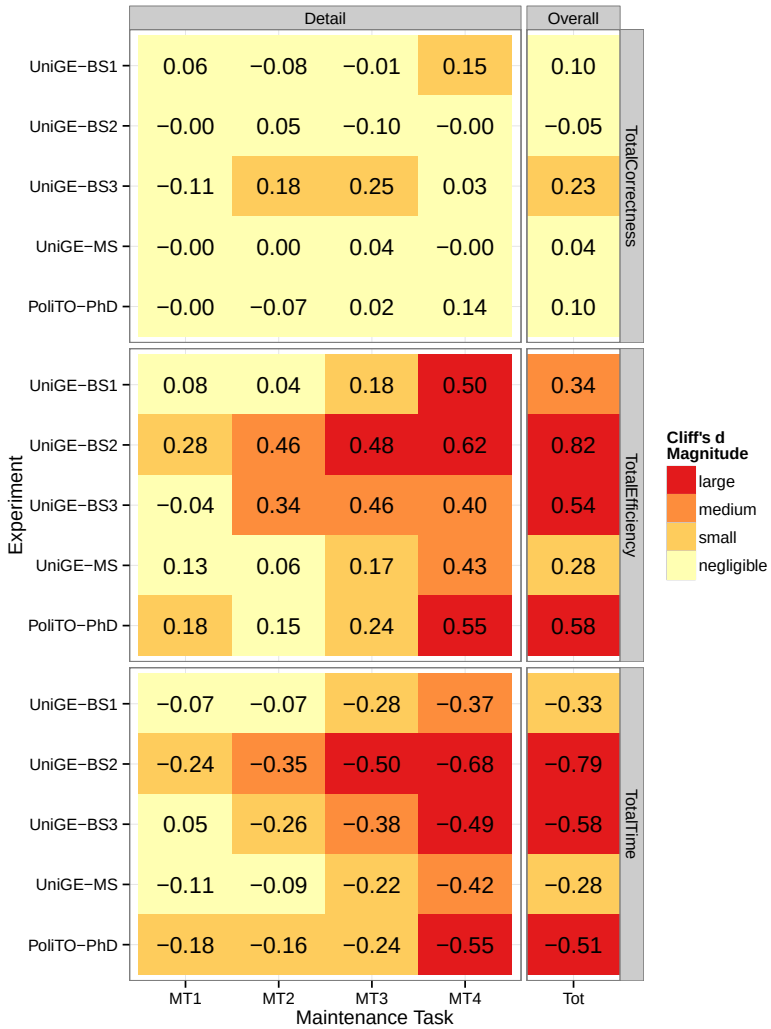

Fig. 18 Heat map of Cliff d

Table 21 Permutation test of *TotalCorrectness* vs. Experiment and Lab order

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Lab	1	6.60	6.60	51	1.00
Experiment	4	192.86	48.21	177642	0.05
Lab:Experiment	4	17.50	4.38	11057	0.94
Residuals	995	21465.51	21.57		