SPECIAL ISSUE

Teaching UML and OCL Models and their Validation to Software Engineering Students: An Experience Report

Loli Burgueño[a] and Antonio Vallecillo[a] and Martin Gogolla[b]

[a]Universidad de Málaga, Spain; [b]University of Bremen, Germany.

ARTICLE HISTORY
Compiled April 2, 2018

ABSTRACT
Models are expanding their use for many different purposes in the field of software engineering and, due to their importance, universities have started incorporating modeling courses into their programs. Being a relatively new discipline, teaching modeling concepts brings in new challenges. Our contribution in this paper is threefold. First, we list and describe the main issues we have come across when teaching modeling in a dedicated Software Engineering course. We then present a simple case study that we have developed and successfully used in class, which permits students specify a system and its views, simulate them, check their relations and perform several kinds of analyses on the overall system specifications. For this, we use a combination of UML and OCL. Finally, we report on the results of a survey we conducted among the students of the last two years to evaluate our proposal, and the lessons we have learned.

KEYWORDS
Computer science education; teaching modeling; UML; OCL; model views.

1. Introduction

Model-Driven Engineering (MDE) is a prominent area in the software engineering field. Over the years, a new trend of approaches has emerged advocating languages, standards, tools and well defined practices. These include UML (Booch, Rumbaugh, & Jacobson, 2005) and OCL (OMG, 2000). MDE raises the level of abstraction in software utilizing *models* that focus only on the features of interest alleviating the complexity. These models are applied as a means of communication as well as for code generation and checking the correctness of a solution. The application of MDE increases the automation of software development and permits the detection of problems in early stages of the development cycle. MDE also permits the application of an agile methodology as they both can be combined in what is called Agile Modeling.

Despite their growing relevance and increasing acceptance in industry to build software at the right level of abstraction, with less effort and errors, models are not yet properly appreciated by many software engineering students. In particular, based on our experience, which is founded on in-class conversations, tutorship sessions, students' questions and exam mistakes, we identified three main problems with our students when teaching the use of models.

CONTACT Loli Burgueño. Email: loli@lcc.uma.es

First, students normally tend to see models as drawings, mostly used for communication among humans, and not as software artifacts that need to be processed by computers. Hence they tend to be less precise and formal than when developing programs.

The second problem has to do with the use of model views. Any non-trivial system specification needs to be decomposed in various models (views), each one focusing on a particular set of concerns and described in a particular notation. For example, UML defines a set of diagrams for modeling a system that permit describing its structure (e.g., with class diagrams), the behavior of its individual objects (e.g., with state machines), the collective behavior of collections of objects (e.g., with communication and sequence diagrams). However, when dealing with multi-viewpoint models, students may be able to understand each one of these diagrams, but they have problems understanding how they are related, and how the overall system specifications actually work when composed of a set of views.

Finally, tools are essential elements in any engineering discipline. Students from these disciplines work better with hands-on experiences, because they need to be able to build and to manipulate the corresponding software artifacts if they actually want to comprehend the related concepts (Offutt, 2013; Vincenti, 1990). In this respect, modeling tools directly influence the way they learn, develop and use models. However, most modeling tools do not provide support for developing precise and correct models, or tend to be used as mere diagrammatic instruments, at most with program (skeleton) generation capabilities. The variety of existing tools and their different characteristics does not help in this respect, either.

To address these issues, the authors of this paper designed a change in the way modeling was taught in one of the subjects of the Software Engineering degree at the University of Málaga. First, we emphasized the nature of models as software artifacts, and treated them as such. The role of models as communication means among human users was minimized, stressing their abilities to synthesize, analyze, prototype and simulate systems. The fact that tools, and not humans, had to perform these tasks over the models, implied a change in the students' mindsets: like any other program they develop, software models have to be correct and precise enough to be processed by tools in order to conduct an analysis or run simulations.

Secondly, we developed an explanatory example of a system which permits students to create several views of a system, simulate them, and understand their relationships. We use a combination of UML and OCL and the possibilities that the OCL/USE tool (Gogolla, Büttner, & Richters, 2007; Gogolla & Hilken, 2016) provides to simulate and analyze the models, as proper software artifacts.

To tackle the tooling issue we ask the students to use three different modeling tools during the course: MagicDraw,[1] Papyrus[2] and USE. In this way they are able to learn the different capabilities of each tool and the possibilities each one offers for developing correct models and work with them.

This paper describes the example that we have used, and reports on our experience after implementing the changes in our modeling course. The survey we conducted on the students after two consecutive years shows the progress made so far and suggests some ideas for further improvements.

This work is an extended version of the paper we originally presented at the MODELS 2017 Educators Symposium (Burgueño, Vallecillo, & Gogolla, 2017), where we

[1] `https://www.nomagic.com/products/magicdraw`

[2] `https://www.eclipse.org/papyrus/`

described only the example. Here we also present the overall educational context, the problems to address, the survey that tries to assess our proposal, and the results of such evaluation and its conclusions.

The rest of the paper is structured as follows. After this introduction, Section 2 describes the students teaching environment and the modeling course. Then, Section 3 describes the example and shows some relevant and interesting properties provided by USE. Section 4 presents the survey we conducted to evaluate the impact of our changes, its results, as well as the lessons we have learned. Finally, Section 5 presents our conclusions and outlines some lines of future work.

2. Teaching Environment

The University of Málaga offers a 4-year degree on Software Engineering. Its contents are mostly based on the SWEBOK Guide (Bourque & Fairley, 2014), covering all principal areas of Software Engineering. The first two years are dedicated to the common topics that any computer scientist or software engineer should learn, including a general and introductory course on Software Engineering in the fourth semester that provides a global view on the discipline, and teaches its basic concepts. UML models are taught by the first time in that course. The last two years cover essential knowledge areas such as Requirements, Software Testing, Maintenance, or Project Management. The degree also includes a capstone project that students should individually develop during the last year, in order to make them put in practice all knowledge acquired during the degree.

One third-year course is "Software Modeling and Design". It is a six ETCS credits course, taught during the first semester (October-January). The main goal of this course is to teach students the key software modeling concepts and mechanisms, the construction and use of high-level models in software development, and the basis for software design from the system's high-level models (including, e.g., design patterns).

In the modeling part, traditionally this course covered the basic UML diagrams, and used MagicDraw as the primary modeling tool. From the beginning, one distinguishing feature of this course is that it presented the modeling concepts and mechanisms first, independently from any notation, and then it showed how to realize them with UML—following the approach described in the book by Olivé (2007). Evaluation was conducted by developing some basic models of a given system, in order to check whether the students have properly understood the basic UML concepts. OCL was briefly mentioned but not used for defining integrity constraints or model well-formed rules, and the produced models were not checked for precision, correctness, or completeness. The focus was more on the communication aspects of the models, with some basic notions at the end about code generation from models using any of available tools for that (such as IBM Rational Rhapsody[3]). Issues such as model instantiability (i.e., satisfiability), prototyping, simulation or validation were not part of the course.

As mentioned in the introduction, one of the problems with this approach is that it implicitly conveyed the message that software models did not need to be as precise as computer programs, and that ambiguity, imprecision and lack of formality was perfectly permitted. In particular, the transformation from high-level models to design models, ready to be implemented, still required a human-centered and non-automatable process, and the high-level models were not really made for computer processing. Besides, (UML) model views remained at the decorative level, i.e., they

[3] `http://www-03.ibm.com/software/products/en/ratirhap`

represented different views of a system, but their relationships were not made explicit nor manipulable by tools. More precisely, views seemed independent and not closely related, and issues such as change propagation and viewpoint synchronization were not even addressed.

Two years ago we decided to implement some changes in this course, with the goal of addressing the issues mentioned in the introduction of this paper.

The first year (2016/17) we introduced the use of three tools (MagicDraw, Papyrus and USE) and asked students to develop all models with the three of them. The goal was to teach the students master multiple tools, each one with different characteristics, and let them, at first hand, compare their features, functionality and capabilities. MagicDraw and Papyrus are widely used tools in industry, and also for teaching modeling (Agner & Lethbridge, 2017). The former is a commercial tool and Papyrus is open-source and developed as an Eclipse project. USE is a free academic modeling tool with full support of OCL, and also with some simulation and object model generation facilities.

The second change was the use of a running example to illustrate some of the main concepts and mechanisms taught in the course for teaching model views. This is the example described in detail in Section 3.

Finally, we decided to put more emphasis on the correct and precise specification of all models, so that they could be manipulated by tools. The use of OCL was encouraged, as an essential language to complement the UML diagrams if they have to constitute correct and complete system specifications. The basic elements and features of OCL were explained, and the OCL/USE facilities for validating models and for executing the specifications were introduced, to help students develop correct models and test them. MagicDraw and Papyrus OCL validation facilities were also explained, although students did not use them much (students considered them quite cumbersome and not naturally integrated into these two modeling tools, contrarily to what happened with USE).

Given that students responded very well to the use of OCL, and they fully understood its need and advantages, the second year (2017/18) we explained the full OCL language and made it compulsory for expressing the models integrity constraints and the behavior of operations.

This first part of the course covers the main concepts and mechanisms of conceptual modeling. The second part of the course is dedicated to design models, and we only used one of the tools (MagicDraw). The focus of this second part is on how to transform the high-level models of the system, which were expressed in the *problem domain*, into design models that are expressed in the *solution domain*. We realized how the students appreciated very much the fact that the conceptual models were correct and complete in the first place, because it significantly simplifies the design task: it then mainly becomes an application of a set of skeletons and design decisions, rules and patterns, something that can be easily automated.

3. A UML and OCL Model for a Production Line System

The example we present here—originally defined in (Rivera, Guerra, de Lara, & Vallecillo, 2008)—is based on a production line of a plant that produces hammers. It is modeled using a combination of UML and OCL. We show how these notations permit modeling the system in a formal manner using different views, and allow performing several interesting analyses on the system.

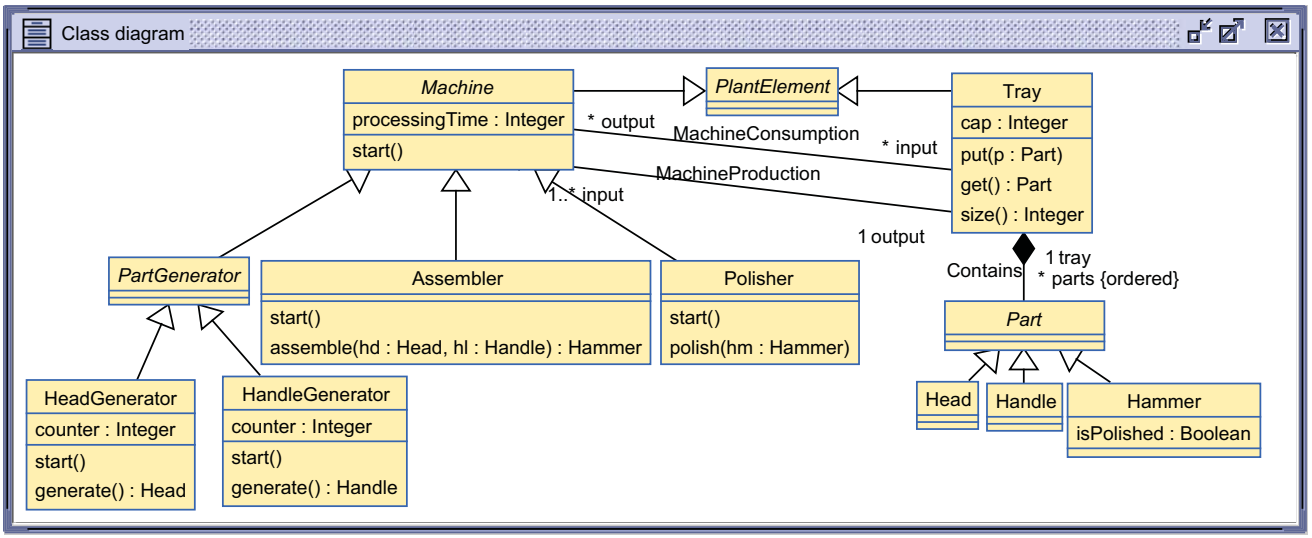


Figure 1. Class diagram for the production line example.

The production line is composed of four types of machines: one that generates hammer heads, another that generates handles, one that assembles heads and handles, and finally one that polishes the hammers. There are also trays that are used to collect the goods in production.

The plant operates as follows: each generating machine has an associated tray, in which it places the heads or handles as soon as they are produced. From these trays, the pieces are taken by an assembler machine, which puts together one head and one handle to produce a hammer, which is placed in a different tray. From this tray, the polisher takes hammers, works on them and leaves them in a different tray.

3.1. Structural Elements

Figure 1 shows a UML class diagram with the structural elements of the system. We can see how each `Tray` has a capacity (`cap`) and each `Machine` keeps record of the time that its job takes (`processingTime`). Futhermore, the head and handle generators (`HeadGenerator` and `HandleGenerator`) keep a counter of the number of elements that have produced (`counter`). In order to perform the actions of placing or removing pieces from the trays, class `Tray` provides `get()` and `put()` operations.

3.2. Behavioral Elements

3.2.1. Pre- and post-conditions

The behavior of the system can be expressed in UML and OCL by different means. One of the most common ways is by adding the specification of pre- and post-conditions to the operations, defining their behavior. This is shown below for `get()` and `put()` operations of class `Tray`.

```
put(p:Part)
  pre  notFull: self.parts->size() < cap
  post ElementAdded: self.parts = self.parts@pre->append(p)

get():Part
  pre  notEmpty: self.parts->size()>0
  post FirstElemRemoved: result = self.parts@pre->at(1) and
                  self.parts@pre = self.parts->prepend(result)
```

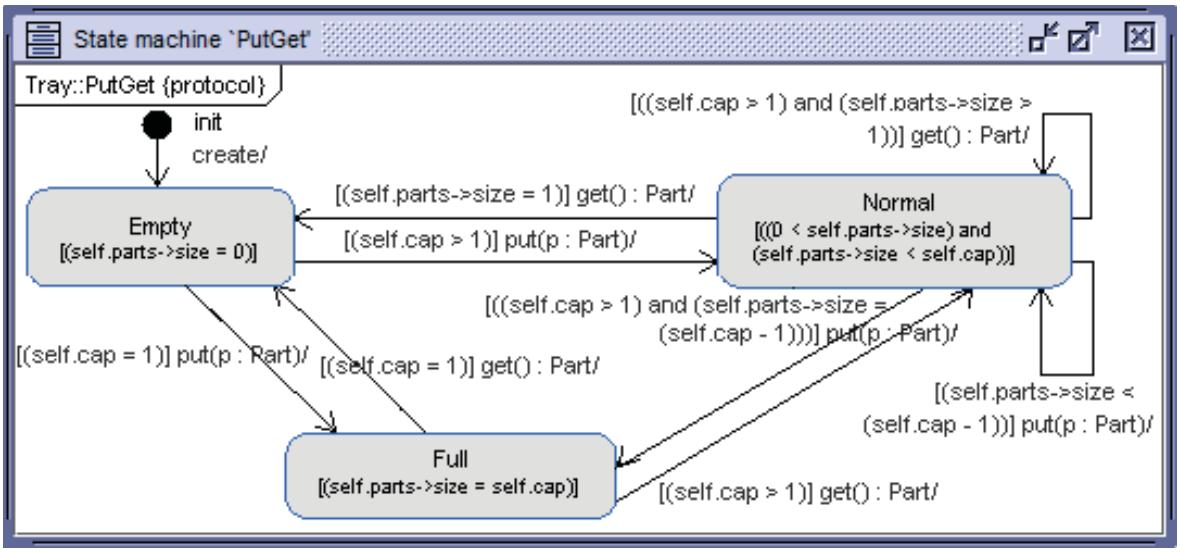


Figure 2. State Machine for Tray objects.

3.2.2. State machines

UML permits specifying the behavior of individual objects by means of State Machines that determine how their internal state changes as a result of the invocation of their provided operations.

One of the advantages of USE/OCL is that it permits formally specifying state machines for individual objects, and then automatically deriving the corresponding UML diagram from them. For example, the following listing shows the specification of the state machine of a `Tray`. The corresponding UML diagram is shown in Figure 2.

```
psm PutGet
  states
    init: initial
    Empty  [self.parts->size()=0]
    Normal [0<self.parts->size() and self.parts->size()<self.cap]
    Full   [self.parts->size()=self.cap]
  transitions
    init -> Empty { create }
    Empty -> Normal { [self.cap>1] put() }
    Normal -> Normal { [self.parts->size()<cap-1] put() }
    Normal -> Full { [self.cap>1 and self.parts->size()=cap-1] put() }
    Empty -> Full { [self.cap=1] put() }
    Full -> Empty { [self.cap=1] get() }
    Full -> Normal { [self.cap>1] get() }
    Normal -> Normal { [self.cap>1 and self.parts->size()>1] get() }
    Normal -> Empty { [self.parts->size()=1] get() }
end
```

3.2.3. Behavior of operations

USE/OCL provides some interesting features on top of the standard UML and OCL capabilities. We decided to make use of some of them, particularly those that permit prototyping the models and executing them. We discovered that students appreciate very much the possibility of *running* their models, simulating their behavior. On the one hand, this helps them understand their specifications much better. On the other hand, this forces them to be more rigorous and precise, because they have to provide enough information for the models to the complete, and also correct if their behavior should be as expected. One important issue is that this information is provided at the same level of abstraction than the models.

Thus, in addition to the pre- and post-conditions, USE also permits to specify the behavior of operations using a simple executable language called SOIL (Büttner & Gogolla, 2014). For instance, the behavior of `Assembler::start()` and `Assembler::assemble()` operations can be specified as follows.

```
start()
  begin
    declare hd:Part, hl:Part, hm:Hammer;
    hd:=self.input->select(t|t.parts->size>0 and
          t.parts->forAll(oclIsTypeOf(Head))) -> single().get();
    hl:=self.input->select(t|t.parts->size>0 and
          t.parts->forAll(oclIsTypeOf(Handle))) -> single().get();
    hm:=self.assemble(hd.oclAsType(Head), hl.oclAsType(Handle));
    self.output.put(hm);
  end
```

```
assemble(hd:Head,hl:Handle):Hammer
  begin
   destroy hd,hl;
   result:=new Hammer;
   result.isPolished:=false;
  end
```

3.2.4. Executing the specifications

Once we have the behavior of the operations, in USE/OCL we can also execute the system by providing a sequence of SOIL commands that create the initial objects of the system, their links, and invokes the operations. For example, we can show the students how to simulate the system by creating an initial model of the system and invoking the `start()` operation on the four machines.

```
-- Machines
!new HandleGenerator('hag')
!new HeadGenerator('heg')
!new Assembler('asm')
!new Polisher('pol')
-- Trays
!new Tray('Handle2Assem')
!Handle2Assem.cap:=4;
!new Tray('Head2Assem')
!Head2Assem.cap:=4;
!new Tray('Assem2Polish')
!Assem2Polish.cap:=4;
!new Tray('Polish2Out')
!Polish2Out.cap:=4;
-- Production Line Connections
!insert (hag,Handle2Assem) into MachineProduction
!insert (heg,Head2Assem) into MachineProduction
!insert (Handle2Assem,asm) into MachineConsumption
!insert (Head2Assem,asm) into MachineConsumption
!insert (asm,Assem2Polish) into MachineProduction
!insert (Assem2Polish,pol) into MachineConsumption
!insert (pol,Polish2Out) into MachineProduction
-- Process
!heg.start()
!hag.start()
!asm.start()
!pol.start()
```

Figure 3 pictorially shows a filmstrip of the behavior of the system as a sequence of snapshots after every operation execution. Aggregation associations are used to visualize 'ownership' between objects (e.g. a part is placed on a tray). We have also

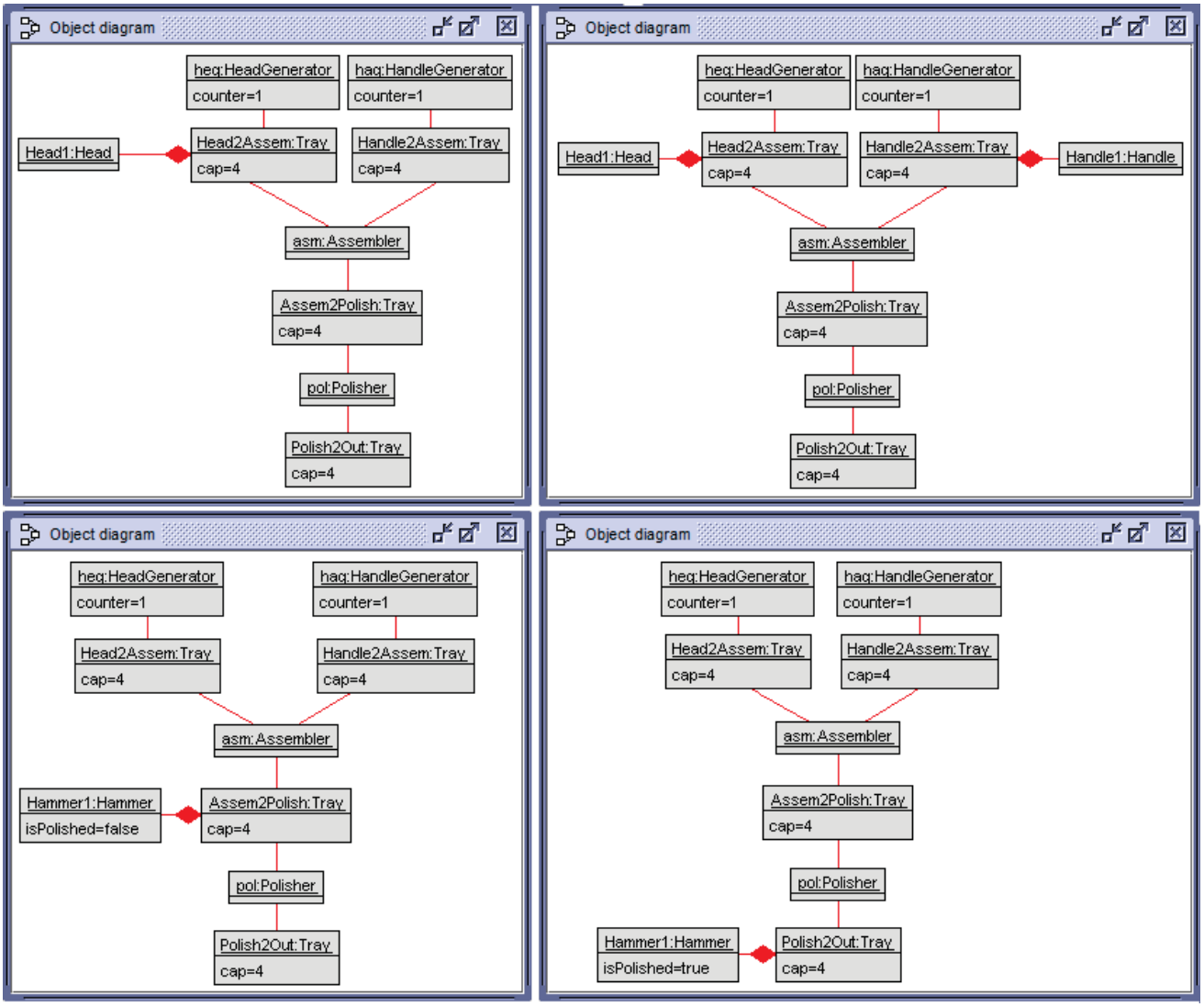


Figure 3. Object diagram sequence displaying the behavior of the system.

developed a video with the complete filmstrip.[4]

Note that both MagicDraw and Papyrus tools have some extensions and modules to check user-defined integrity constraints and to simulate the models. In our course we decided to employ only USE/OCL because the plugins and modules of the other two tools were either not free (MagicDraw) or required more complex installation and operation procedures (Papyrus). The easy and free access to these features in USE/OCL made us decide to teach them just with this tool.

3.2.5. Object interactions

So far we have focused on the structural view of the system and the behavior of individual objects. UML also permits representing the interactions between objects by means of Sequence and Communication diagrams. Figures 4 and 5 show the sequence and communication diagrams for the production plant system.

With MagicDraw and Papyrus, these diagrams are independently produced, and their relationship with the conceptual model described in the class diagram is based on the types of the elements used, and the operations invoked by the objects—which should be explicitly specified in the class diagram. Given the fact that enforcing these

[4] `http://atenea.lcc.uma.es/Descargas/EduSymp17/3Hammers.mp4?dl=0`

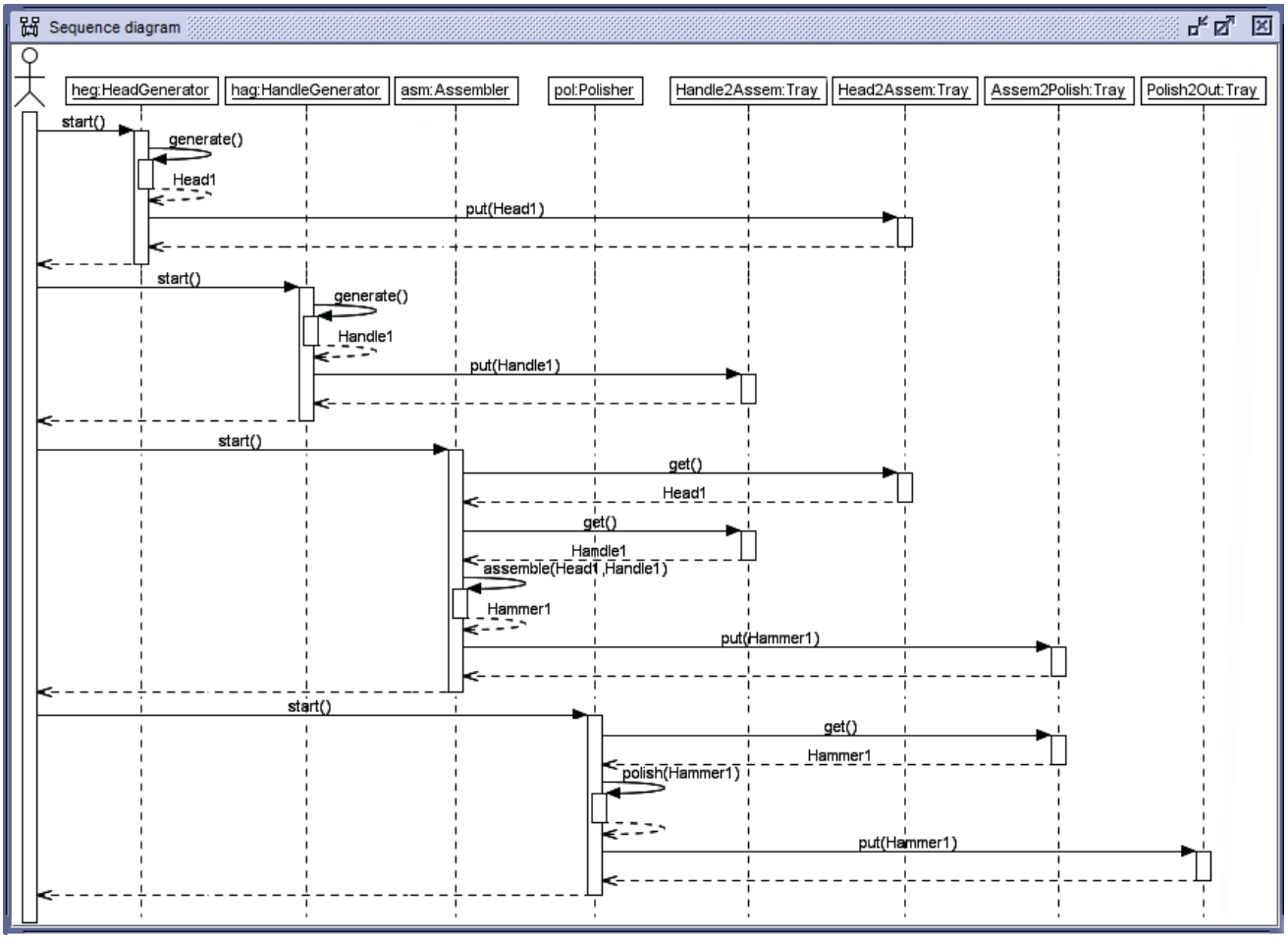


Figure 4. Sequence diagram.

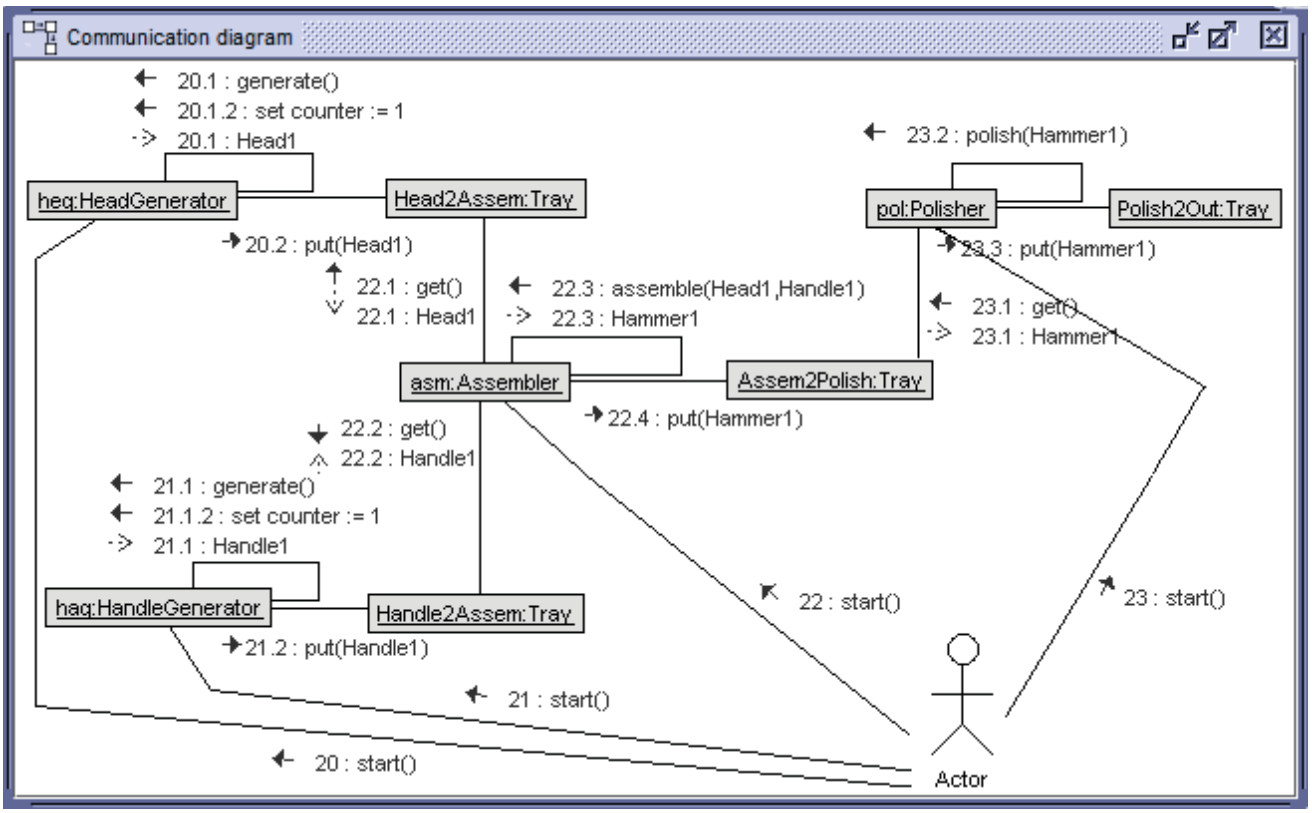


Figure 5. Communication diagram.

relationships is not compulsory,[5] many students (and also modelers, in general) tend to use elements in the sequence and communication diagrams which do not match the ones in the class diagrams. However, this is not desirable in a teaching context like ours, where we want all elements in all diagrams to be typed, and related to the structural model that defines all valid entity and relationships types.

With USE/OCL, we are able to automatically derive the interaction diagrams of a system from the simulation of the system's behavior described above, in terms of sequences of operations invocations. For example, the UML sequence diagram shown in Fig. 4 is automatically constructed by USE/OCL.

The behavior can also be displayed as a communication diagram as in Fig. 5. Interestingly, for every step we can represent the current states of all the state machines of the `Tray` objects—not shown here for space reasons, although they are similar to the one depicted in Fig. 2 but with the current state highlighted; the interested reader can see the presentation[6] we have developed to show the state changes. Also, all the code developed for this case study is available.[7]

The relationships between the views can also be explored with this approach. One of the most interesting examples happens when one of the views is changed, and the effects of this change on the rest of the views. For example, when an invariant is added or a class is changed or removed; when an element changes its name; or when a guard for a state change is altered. Every kind of change in a model has a different impact on the validity of another model, and therefore it requires the other model view to be updated. Students can learn from the effects of each change, and understand how views are related.

In this way, students perceive these *views* as projections of an underlying model, and not as independent and possibly unrelated views. Using USE/OCL, if they want to create a sequence or interaction diagram, they have to describe the precise behavior of the system and then they can represent their executions using these diagrammatic views. Although this provides a first and basic *view* on views, it is important they understand them as *projections* first, before they can later learn other viewpoint-based modeling systems in which views are not projections but independent models related by correspondences (Romero, Jaen, & Vallecillo, 2009), for instance.

3.3. Further analysis

Once we have the specifications, there are different kinds of analyses that we can perform on the system that show some of the potential advantages of developing model-driven system descriptions with UML and OCL. In particular: visualization of complex structures and processes; execution and simulation of scenarios (operation call sequences); checking structural properties of the system within states by OCL queries (e.g. calculating the number of finally produced parts); checking behavioral properties (e.g., testing the executability of an operation by testing its preconditions); checking for weakening or strengthening model properties (invariants, contracts, guards) by executing a scenario with modified constraints; proving general properties within the finite search space with the USE model validator, such as structural consistency (i.e., all classes are instantiable); behavioral consistency (i.e., all operations can be executed); deadlocks (e.g., construction of deadlock scenarios due to inadequate buffer capacities), etc.

---

[5] There are reasons in UML for this, but we will not discuss them here.

[6] `http://atenea.lcc.uma.es/Descargas/EduSymp17/snapshots-buffer.pdf?dl=0`

[7] `http://atenea.lcc.uma.es/Descargas/EduSymp17/sources.zip?dl=0`

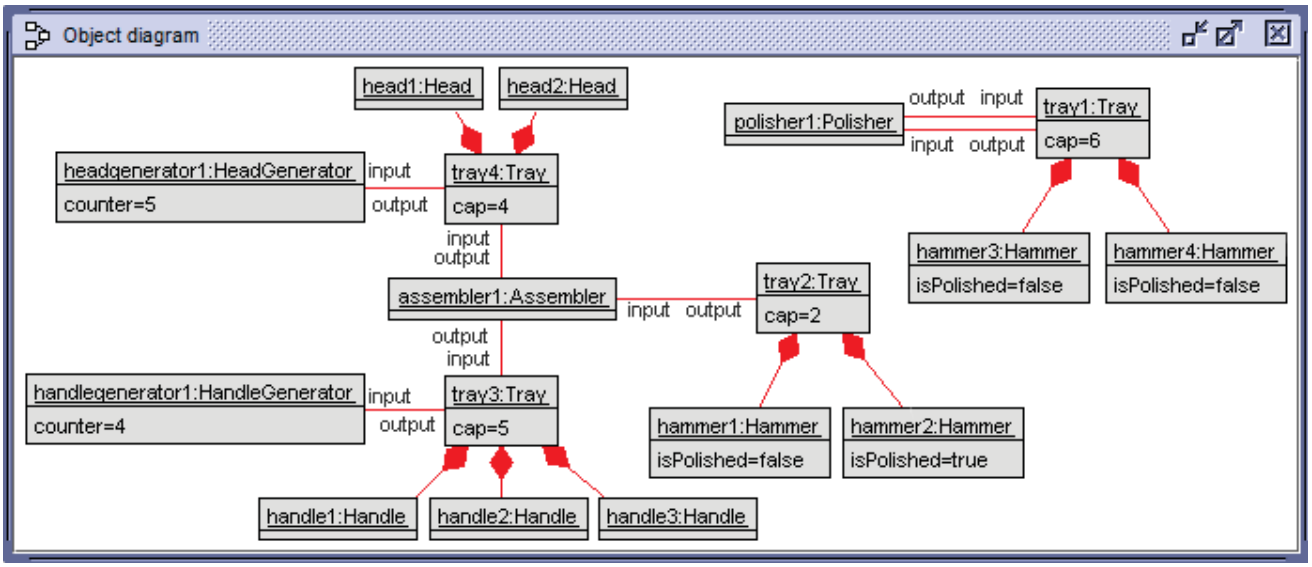


Figure 6. Generated test case for Producer-Consumer-Tray configuration.

Finally, we want to highlight the importance of running structural tests on the metamodels. One of them concerns their instantiability and their ability to faithfully represent the application domain. For example, we decided to ask the USE model validator (Gogolla & Hilken, 2016) to generate a plant configuration using the system metamodel. The resulting object diagram is shown in Fig. 6. Interestingly, the produced system is wrong. The polisher is not connected to any tray that provides it with hammers. This motivates the need to develop additional, currently missing invariants (on the structural system level) and demonstrates the potential usefulness of this approach for validating these kinds of properties which are normally overlooked for being considered obvious.

Many students appreciate very much the possibility of generating objects from the models—i.e., instantiating the models. This allows them to check whether their models are satisfiable or not, and whether the generated object models are correct. Due to time limitations (the *conceptual modeling* part of the course runs for 8 weeks only, before we start with the *design* part) we just mention briefly the features of the USE model validator and give some examples. Expanding this part in future editions of the course is something we are currently considering.

4. Evaluation and Lessons Learned

4.1. Evaluation

After the first two years, we decided to evaluate our approach and the impact on the learning process. For that we developed a survey, and we asked the students who were enrolled in this subject in the academic years 2016/17 and 2017/18 to voluntarily participate in the study.

In the survey,[8] which is available in English and Spanish, the students were asked to rank the importance of different modeling aspects according to their own opinion. We divided the survey in two main groups: "Concepts and techniques" and "Tools". Concepts and techniques were divided into five different subcategories: Structure, Be-

[8] `https://encuestas.uma.es/93737/lang-en`

Q: In your view, how important for modeling is/are...

STRUCTURE		
Q1	Object properties (e.g., by means of attributes, references)	4.14
Q2	Object relationships (ordinary, whole-part)	4.72
Q3	Factoring out properties (i.e., building abstractions/generalizations)	4.21
Q4	Specifying complex structural restrictions (i.e., integrity constraints)	4.26
BEHAVIOR		
Q5	Actions and processes	4.26
Q6	Factoring out actions (i.e., building abstractions/generalizations)	3.99
Q7	Life cycles (e.g., state machines)	3.92
Q8	Specifying complex behavioral restrictions	3.96
Q9	Process and Action Realizations (i.e., procedural specification of their behavior, in addition to their declarative specification)	3.82
DEVELOPMENT		
Q10	Building test cases/scenarios	4.17
Q11	Obtaining feedback on test cases	4.13
Q12	Verifying/validating emergent/derived model properties by test cases and/or proof techniques	4.23
Q13	Textual notation (in addition to graphical)	3.35
MODEL PROPERTIES: A good model should be...		
Q14	Abstract	3.60
Q15	Structured/Correctly organized	4.71
Q16	Understandable	4.75
Q17	Traceable (to/from requirements)	4.34
Q18	Reusable	4.25
Q19	Executable	4.06
Q20	Verifiable (w.r.t., e.g., correctness, instantiability, consistency)	4.50
MODEL VIEWS		
Q21	Structuring a Model into Views	3.90
Q22	Explicit relationships between the views	3.85
Q23	Defining derived views (e.g., from scenarios)	3.54

Table 1. Questions about 'Modeling Concepts and Techniques'.

havior, Development, Model properties, and Model views. These subcategories and the questions for each of them are presented in the second column of Table 1 (Q1-Q23). To answer them, the student could choose among the following options: *Not important at all* (with a weight of 1), *Low importance* (weighed 2), *Moderate importance* (weighed 3), *Slightly important* (weighed 4), *Very important* (weighed 5), or *No opinion*.

In the second part of the survey, for each of the tools (MagicDraw, Papyrus and USE), we asked the questions shown in the second column of Table 2 (Q24-Q31). For each of them, participants had to rate the support that the tools offer in their opinion by selecting one of the following possible answers: *Bad/No support* (with a weight of 1), *Improvable support* (weighed 2), *Average support* (weighed 3), *Good support* (weighed 4), *Excellent support* (weighed 5), and *No opinion*.

A total number of 65 students took part in the survey, 21 of them were enrolled in the modeling course in the academic year 2016/17, and 44 in the academic year 2017/18. Their average ages were 22.41 and 21.57 years, respectively.

Q: How well do the tools Papyrus, MagicDraw and USE support...

TOOLS		MDraw	Papyrus	USE
Q24	Precision (being able to represent the system and their elements with the required level of details)	4.33	2.47	3.93
Q25	Support for test case development	3.39	1.82	3.47
Q26	Execution/Simulation	2.74	1.91	4.03
Q27	Validation/Verification (being able to check and infer properties/characteristics about the system being modeled)	2.69	1.84	4.21
Q28	Viewpoint consistency	3.71	2.09	3.67
Q29	Realization (at the model-level—i.e., execution without generating code)	3.45	1.93	3.56
Q30	Code generation (for programming platforms/languages)	3.19	1.73	2.68
Q31	Usability (i.e. ease of use)	4.31	1.96	2.88
	Average	3.48	1.97	3.56

Table 2. Questions about 'Tools'.

The third column of Table 1 presents the average of the answers given by students for questions Q1-Q23, while Figure 7 shows graphically the average of the answers taking both groups of students separately. We can highlight that there is no value below 3.0 (moderate importance), and that the feature they consider more important is that models are understandable (Q16), while the least important for them is the fact that they count with textual notation in addition to the graphical notation (Q13). In addition to the averages shown in Table 1 and Figure 7, for each question we calculated the standard deviation and the percentage of 'No opinion' answers. For year 2017/18, the average of the standard deviations is 0.84 and the percentage of 'No opinion' is 0.02; for year 2016/17 these are 0.92 and 0.04, respectively.

We can see that the trends for the two years in Figure 7 are basically the same. The only noticeable difference is the importance given to state machines (Q7). Year 2016/17

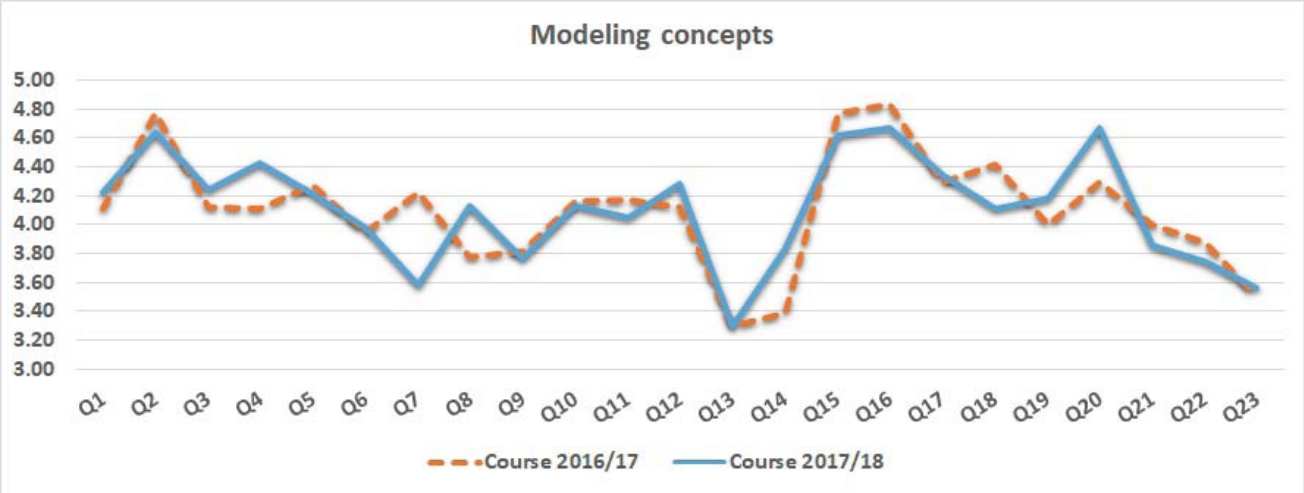
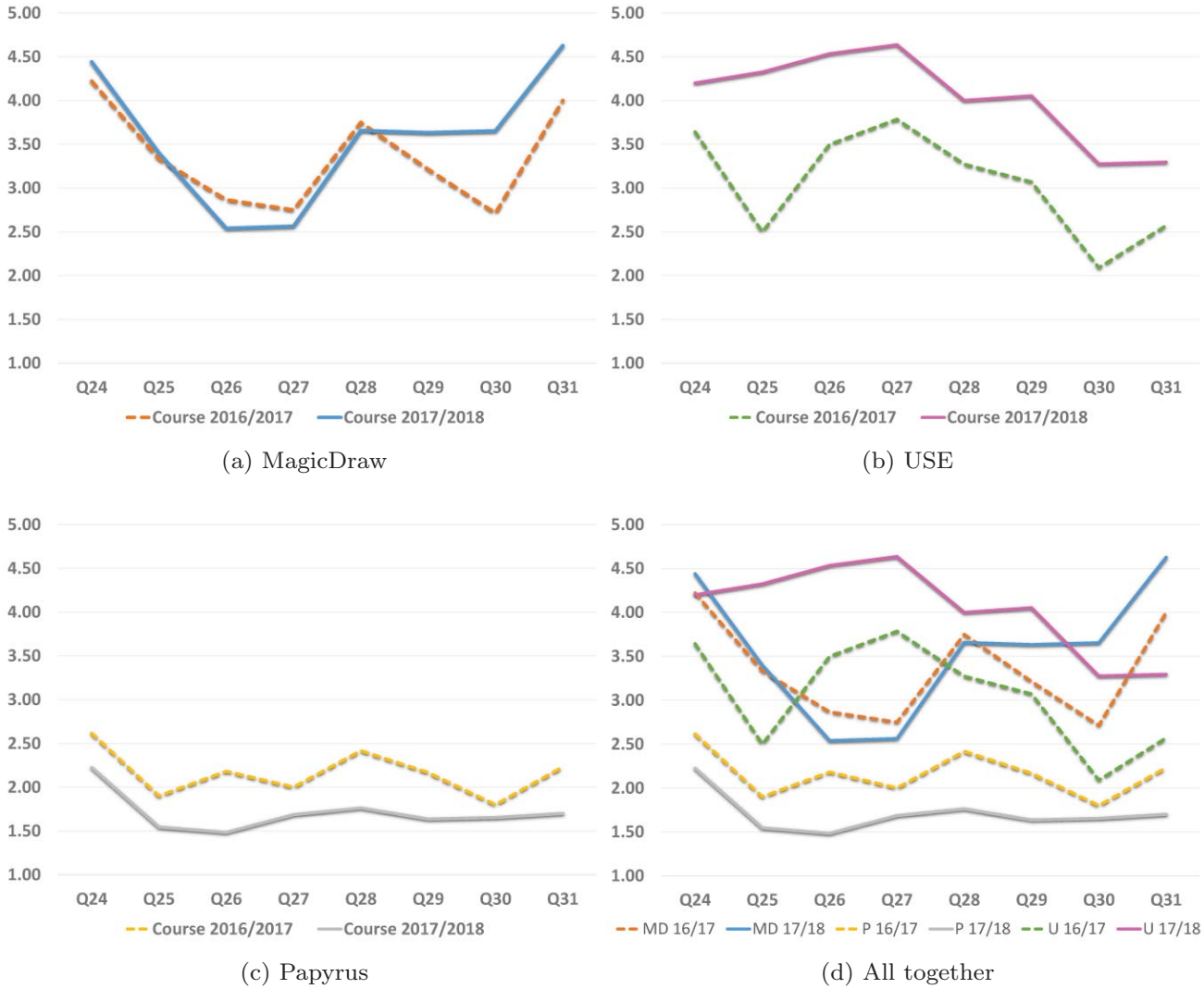


Figure 7. Modeling concepts



(a) MagicDraw

(b) USE

(c) Papyrus

(d) All together

Figure 8. Modeling tools

students give more importance to them. This is probably because these students have gone through more subjects, and some of them (e.g., Testing) make heavy use of state machines. Year 2017/18 students are not fully aware of the importance of these kinds of models yet.

The third, fourth and fifth columns of Table 2 present the average rate given by the students for each tool for questions Q24-Q31. The last row is the average of all the other values. We observe that their best rated tool is USE closely followed by MagicDraw. Figure 8 presents, for each tool and each question, the average of the answers distinguishing between the two groups of students.

Table 3 presents more details for the tools questions. For each course and each tool, the average rate, the standard deviation and the percentage of 'No opinion' answers are shown. The standard deviation is around 1 point in all cases, while the percentage of 'No opinion' is significantly higher when compared with questions Q1-Q23.

Regarding the average rates, we can conclude that, while Papyrus is always the least liked tool, USE is the favorite for students enrolled in year 2017/2018, and MagicDraw for students enrolled in 2016/2017. We believe this issue is due to the fact that in the second year the specification of the models' constraints had to be done in OCL, while being optional the first year. Note that this decision had a beneficial effect not only in USE (which scored above 4, almost 1 more point than the previous year) but also for MagicDraw, which increased its score too. In general, all students learned the

	Course 2016/2017			Course 2017/2018		
	MagicDraw	Papyrus	USE	MagicDraw	Papyrus	USE
Average	3.39	2.23	3.07	3.56	1.71	4.04
Standard Deviation	0.93	0.97	1.08	1.02	0.95	0.95
% of 'No opinion'	0.14	0.61	0.41	0.27	0.31	0.13

Table 3. Summary of tooling support.

importance of being formal when specifying models, and its benefits.

In informal conversations with the students, they admitted that they liked very much being sure that their models were correct, as well as the possibility of being able to check the invariants, simulate the models, and even generate object models. This was very rewarding for them, similar to the feeling they have when they developed computer programs, executed them, and tested them until proven correct. Once they knew how to check their models, they wanted to have the same confidence they had about their programs. After they learned USE/OCL, most students drew their models first with USE and, once they felt confident about their correctness, drew them with MagicDraw and with Papyrus (in this order).

4.2. Lessons learned

Our experience of teaching modeling concepts with UML and OCL has been very positive. At the beginning the students feel uneasy with OCL because they are used to build UML models that are always right—as Bertran Meyer (1997) once said, "Bubbles and arrows never crash." But they progressively discover the advantages of being able to check that the models they are building are correct. This is similar to what happens when you are used to untyped programming languages and you use for the first time a strongly typed language; then you benefit from the compiler and an execution platform.

Furthermore, by using a tool such as USE that permits building the views automatically, they discover the meaning of *model views*, the relationships between them, and how changes in the model manifest in the views.

Another interesting benefit of our approach is that we can follow an incremental development process for building the models. Starting from a simple class diagram we can incrementally add classes, associations, attributes, operations, invariants, contracts, SOIL operation implementations and protocol state machines. Another feature that students like very much is creating object models. They then discover that models are more than *pictures*, but assertions on the objects that conform a system (and their relationships). They enjoy developing growing sets of scenarios, defined by means of SOIL command sequences that build sequences of system states (object diagrams), and then the corresponding sequence and communication diagrams that graphically describe the interactions.

This incremental development process supports direct feedback and model improvements. They discover that modeling is similar to programming, in the sense that you write a program (formalize an artifact and develop assumptions about its properties) and execute it to check whether its behavior is as you expected (make experiments to check the assumptions).

Students can check resulting system state sequences of the scenarios they develop by 'looking at' the object, sequence or communication diagrams. This gives them direct

feedback, and they can compare the results with their expectations.

Regarding validation, students can check their models using various functionalities available in USE: the diagrams, the evaluation browser, the class extent, the single object window, the class invariants window, etc. And they also learn that views are not completely independent. On the contrary, they are all *projections* of an underlying model.

The results of the survey show that in general the students appreciate the importance of the different modeling concepts and techniques.

Another important decision is the use of multiple tools, each one with different characteristics. Students realize about this diversity, and get prepared to work with any of them. Special mention deserves the use of a formal tool that permits them checking the correctness of their models. Even when later given a less formal tool, they have learned the importance of being precise, and try to be as formal as possible.

5. Conclusions

In this paper we have discussed some of the issues we have faced when teaching modeling to Software Engineers, and presented a case study modeled with UML and OCL/USE that has been successfully used to teach modeling in class. It does not only focus on the different views of the system but also on the relationships the different models have among them. Furthermore, we disclose several of the advantages of modeling with multiple UML/OCL tools, and specially with USE.

We look forward to other interested modeling educators to use our example. We decided to publish this illustrative example and make it available to the modeling community in case our experience can be used to improve the modeling experience in other institutions. The replication of our survey in other centers could also help comparing teaching approaches.

We are currently working on preparing more teaching material on how to generate objects models from the models, and make it part of the basic contents of the course. This part is optional now, but we have seen how the students become really interested in it. This would require some more in-depth explanations and further exercises, and an introduction to the complex problem of generating objects from models using constraint solving techniques. However, the use of the USE model validator, and the application of classifying terms (Gogolla, Vallecillo, Burgueño, & Hilken, 2015) can provide interesting benefits and greatly simplify both the use and the learning processes of these techniques.

Acknowledgments. This work was partially supported by the Spanish Government under Grant TIN2014-52034-R. We would like to thank the reviewers of this paper for their valuable comments and suggestions.

References

Agner, L. T. W., & Lethbridge, T. C. (2017). A Survey of Tool Use in Modeling Education. In *Proc. of MODELS'17* (pp. 303–311). IEEE Computer Society.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *Unified modeling language user guide (2nd edition)*. Addison-Wesley Professional.

Bourque, P., & Fairley, R. E. (Eds.). (2014). Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), version 3.0 [Computer software manual].

Burgueño, L., Vallecillo, A., & Gogolla, M. (2017). Teaching model views with UML and OCL. In *Proc. of models 2017 satellite events* (pp. 529–534). CEUR Proceedings.

Büttner, F., & Gogolla, M. (2014). On OCL-based imperative languages. *Sci. Comput. Program.*, *92*, 162–178.

Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Sci. Comp. Prog.*, *69*, 27–34.

Gogolla, M., & Hilken, F. (2016). Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In A. Oberweis & R. Reussner (Eds.), *Proc. Modellierung (MODELLIERUNG'2016)* (p. 203-218). GI, LNI 254.

Gogolla, M., Vallecillo, A., Burgueño, L., & Hilken, F. (2015). Employing classifying terms for testing model transformations. In *Proc. of MODELS'15* (pp. 312–321). IEEE Computer Society.

Meyer, B. (1997). UML: The positive spin. *American Programmer*, *10*(3), 37–41. Retrieved from `https://archive.eiffel.com/doc/manuals/technology/bmarticles/uml/page.html`

Offutt, J. (2013). Putting the Engineering into Software Engineering Education. *IEEE Software*, *30*(1), 94–96.

Olivé, A. (2007). *Conceptual modeling of information systems*. Springer.

OMG. (2000). *Object constraint language specification, chapter 7* (No. OMG ptc/08-06-08).

Rivera, J. E., Guerra, E., de Lara, J., & Vallecillo, A. (2008). Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. In *Proc. of SLE'08* (Vol. 5452, pp. 54–73). Springer.

Romero, J. R., Jaen, J. I., & Vallecillo, A. (2009). Realizing correspondences in multi-viewpoint specifications. In *Proc. of EDOC'09* (pp. 163–172). IEEE Computer Society.

Vincenti, W. (1990). *What engineers know and how they know it*. The Johns Hopkins University Press.