JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Constraints Specification Via Tool Support: A Controlled Experiment

Azzam Maraee[*,†], **Eliran Nachmani**[*], **and Arnon Sturm**[*]
[*]Ben-Gurion University of the Negev, Israel
[†]Achva Academic College, Israel

ABSTRACT Models can be used for various purposes such as communication, documentation, design means, and for the generation of various artifacts including code. Nevertheless, as some ambiguities still exist in models, additional languages are required. To address this need, in the context of object-oriented modeling, the Object Constraint Language (OCL) was devised. Alternately, other languages including programming languages can be used for constraint specification. In this work we conducted a controlled experiment using USE and a Java framework we developed for that purpose, and compare the effectiveness of developing model-based constraints with respect to quality, time, and confidence. The results indicate that as Java is more familiar to the subjects than OCL, the time to develop the constraints utilizing the developed framework was shorter whilst the confidence was higher. However, despite the greater familiarity with Java, the constraints quality was better when using OCL and USE.

KEYWORDS OCL, USE, Java, Model-based Constraints, Controlled Experiment.

1. Introduction

Model-Based Engineering (MBE) places models as the core artifacts of software development. Models can be used for various purposes, including communication, documentation, design means, and for the generation of various artifacts including code. Nevertheless, due to the lack of formalization and ambiguities that still exist in models, additional languages are required. To address this need, and in particular for designing object-oriented systems, the Object Constraint Language (OCL) was devised (Warmer & Kleppe 2003). OCL was developed as a language that could be attached to an existing (diagrammatic) modeling language lacking in its expressiveness. A language that works thus reduces the likelihood of misunderstanding when humans read models, and facilitates the detection of errors at an early stage of the development process. Alternately, other languages, including programming languages, can be used for constraint specification. However, limited attention has been devoted to testing whether OCL is the best option for specifying constraints in various contexts. Considerations under this rubric include the impact of OCL on maintenance (Briand et al. 2004), the effect of OCL constraint quality on understanding (Correa et al. 2007), and the effect of improving OCL querying language to increase its usage (Störrle 2013). Only recently, an evaluation of alternatives constraint specification languages was carried out (Yue & Ali 2016). It was found that in general, the quality of constraints specified in Java and OCL were similar. In a previous work (Maraee & Sturm 2019), we also examined the effectiveness of understanding and developing constraints, using Java (represents programming languages and imperative approaches) and OCL (represents declarative approaches). We found out that when applying these languages in "dry" mode, i.e., without means to execute and check these constraints, the results are similar. We determined that when referring to understanding and developing simple constraints, using Java resulted in more correct answers than when using OCL. For complex constraints, the situation was reversed; using OCL led to more correct answers. In a follow-up experiment (Maraee & Sturm 2020), we were able to show that OCL outperformed Java with regard to both understanding and developing constraints. In this paper, we examine whether using tools for developing and checking

JOT reference format:
Azzam Maraee, Eliran Nachmani, and Arnon Sturm. *Constraints Specification Via Tool Support: : A Controlled Experiment* . Journal of Object Technology. Vol. 19, No. 3, 2020. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2020.19.3.a16

constraints affects the effectiveness of developing constraints in OCL and in Java, and further examine the differences between these two languages. For this purpose, we conducted a controlled experiment using USE (Gogolla et al. 2007) and a Java framework developed specifically for this purpose, comparing the effectiveness of developing model-based constraints, with respect to time, quality, and confidence. We further reflect upon the subjects' perception regarding the usage of the languages and the tools.

The paper is organized as follows. In Section 2, we review existing studies focusing on those that assess tool effectiveness. In Section 3, we introduce the tools which we used in the present experiment. In Section 4, we present the experiment design and its execution, and in Section 5 we report on the results. In Section 6 we discuss the results, and identify threats to validity in Section 7. Finally, in Section 8 we conclude and outline plans for future research.

2. Related Work

Textual modeling languages such as OCL, Alloy, and FOML are used in MBE for a variety of purposes, such as model querying and specifying model constraints (Balaban et al. 2016). OCL has become a widely adopted constraint language in model-based engineering (Cabot & Gogolla 2012). In the context of its practical use, Ali et al. (Ali et al. 2014) showed the successful use of OCL in six industrial case studies. They also suggested that a subset of OCL should be sufficient for these applications. OCL has attracted increasing attention from both academia and industry. A variety of OCL tools and verification/validation/testing techniques around OCL are currently available (Gogolla et al. 2013; Pérez & Porres 2019; Gogolla, Hilken, & Doan 2018; Cabot et al. 2014; Ali et al. 2013; Queralt et al. 2012; Wille et al. 2012; Gogolla et al. 2005; Brucker & Wolff 2008; Briand et al. 2004). Gogolla et al. (Gogolla et al. 2013) indicated the need for a set of OCL benchmarks to help evaluate and compare OCL tools, and to this end initialized an OCL repository (Gogolla et al. 2014).

They observed that such benchmarks would encourage the development of new OCL tools. Similar repositories have since appeared in (Noten et al. 2017; Mengerink et al. 2019).

There are a number of tools for Java, UML, and OCL, including industry UML modeling tools which support the specification of OCL (Portal 2014; NoMagic 2020), or support both OCL and Java for specifying constraints such as IBM RSA (IBM 2019) and Papyrus (IBM 2020). However, there are relatively few studies which evaluate the effectiveness of using tools for writing constraints. While the work of Yue and Ali in (Yue & Ali 2016) and our last study (Maraee & Sturm 2020) evaluate the usage of Java and OCL as alternatives constraint specifications, we are not aware of any work in the literature that evaluates the impact of using tools for this purpose.

The remainder of this section focuses on tools that support the specification of OCL and/or Java, studies on students' experiences with software modeling tools and their usability, and studies on teaching modeling and OCL using these tools.

2.1. Modeling Tools

Since the introduction of OCL, a variety of software modeling tools have been developed in academia and in industry (Portal 2014; NoMagic 2020; IBM 2019, 2020). One of the first OCL tools to be introduced was Dresden OCL, which was developed at the Technische Universität Dresden. The most recent version of the toolkit provides an OCL Parser, an OCL Interpreter, and an OCL-to-Java Code Generator, and supports constraints specification and verification (Demuth & Wilke 2009; Software Technology Chair 2004). HOL-OCL is a rich theorem-proving environment, which makes it possible to reason over UML class models annotated with OCL specifications (Brucker & Wolff 2008). Recently, (Hammad et al. 2017) presented iOCL, an interactive tool for specifying, validating, and evaluating OCL constraints. The tool guides the modeler to specify OCL constraints interactively by presenting only the relevant details for selection at any given step in the specification, and automatically correcting syntax errors. iOCL was evaluated using a real-world case study, and was found to be useful in facilitating the process of OCL constraint specification.

OCL has several competing textual constraint languages, such as Alloy and FOML (Balaban et al. 2016; Jackson 2002; Balaban & Kifer 2011). FOML is an expressive logic rule language that provides an intentional and executable formal basis for software models (Balaban & Kifer 2011). FOML supports model-level activities such as constraints transformation, analysis, and reasoning about models and model testing (Balaban & Kifer 2011; Khitron et al. 2016). Alloy is a textual modeling language based on relational logic, that employs SAT solver for the bounded validation of user assertions (Jackson 2002). Nakajima showed the benefit of using Alloy in teaching formal methods in software engineering courses (Nakajima 2014). They used Alloy to encode the core concepts, allowing for swift feedback via the automatic analysis tool. The translation of class models/OCL to Alloy (Anastasakis et al. 2010), and the work of Moas et al. (Anastasakis et al. 2010) provide a UML/OCL analysis tool that supports instance generation and completion.

In addition to tools developed within the academia, the industry also recognizes the importance of OCL. As mentioned before, MagicDraw, IBM Rational Software Architect (IBM RSA), and Papyrus are tools that support the specification of OCL constraints (NoMagic 2020; IBM 2019, 2020). IBM RSA (IBM 2019) and Papyrus (IBM 2020) also support the constraint specification in Java. These also enable the validation of the specified constraints to some extent. MagicDraw IBM RSA and Papyrus are widely-used tools, both in industry and for teaching modeling (Agner et al. 2019).

2.2. Experience with software modeling tools

Several studies have explored the usability of UML modeling tools, mainly by comparing their different features (Auer et al. 2007; Khaled 2009; Safdar et al. 2015; Agner et al. 2019; Planas & Cabot 2020). Usability is a core issue in Human-Computer Interaction (HCI). It is defined as "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use." (Dix 2009). Most studies employed empirical

approaches, analyzing usability issues in a number of ways such as comparing usage between two different tools and assessing the productivity of the software engineers when modeling with these tools. For example, a recent study of Planas and Cabot examined the usability of two modeling tools, MagicDraw and Papyrus, by analyzing 12 hours of video recordings of students using the tools (Planas & Cabot 2020). They analyzed the usability of the tools from three perspectives: the modeling process, the modeling effort, and the modeling obstacles that the students encountered during the process. The study found that there were no notable differences regarding the usability of MagicDraw and Papyrus. They also presented several recommendations for improving the usability of the two modeling tools. Furthermore, (Planas & Cabot 2020) provides a comprehensive review of usability studies related to modeling tools. Whilst the studies under review compared the usability of UML modeling tools, they overlooked the quality or correctness of the developed models. We are not aware of studies comparing different tools when the modeling languages are different. But certainly, these studies indicate the effect of a tool on the performance of the subjects using them. Tool usability is beyond the scope of this paper. Nevertheless, we do refer to a few studies that took the usability aspect into consideration, and that are relevant to the current study.

Safdar et al. (Safdar et al. 2015) conducted a controlled experiment comparing the productivity and the quality of the work produced by software engineers while modeling with RSA, MagicDraw, and Papyrus. They measured the productivity of the engineers in terms of modeling effort required to correctly complete a task; learnability; time and number of clicks required; and memory load required for a software engineer to complete a task. They measured completeness as the percentage of modeling elements modeled correctly by the user for a particular task, compared to the total number of modeling elements in the reference model for that task, where correctness refers to choosing the right model elements. The results showed that there was no significant difference between RSA-MagicDraw and MagicDraw-Papyrus in terms of completeness for modeling class diagrams, although RSA was significantly better than Papyrus in this regard.

Anger et al. surveyed the use of modeling tools by 117 students in software engineering courses in different countries (Agner et al. 2019). The result showed that students considered features such as code generation and model validation as a key benefit. Tools such as MagicDraw, Papyrus, and UMPLE yielded the greatest perceived benefits from the students' perspective in terms of code generation, while USE stood out from the other tools for users who were motivated by the validation and "support to edit models textually" features. In addition, students considered the lack of feedback about models, the slowness, and the difficulty experienced in drawing diagrams, as drawbacks.

Burgueño et al. discussed some of the issues we also faced when teaching modeling to software engineers, presenting a case study modeled with UML and OCL/USE that has been used successfully to teach modeling in class (Burgueño et al. 2018). Students specified a system and its views, verified their relations, and performed several analyses on the overall system specifications. Even though during the course students were exposed to three modeling tools—MagicDraw, Papyrus, and USE—for OCL validation, students only used USE. The paper noted that many students highly appreciated the object generation and model validity checking features in USE.

3. Background

3.1. The UML Specification Environment (USE) tool

The UML-based Specification Environment (USE) tool is a plugin modeling tool that supports the specification of a subset of UML models augmented with OCL constraints (Gogolla et al. 2007; Database Systems Group 2020). The development of USE began with the work of Richters and Gogolla on defining the formal semantics of OCL based on the OCL metamodel (the first version appeared in 1998) (Richters & Gogolla 2002). USE versions were developed further by diploma theses and other student projects (Gogolla et al. 2007). The latest version is 5.2 (Database Systems Group 2020), which is also the version used in this study. This version enables working with class, object, sequence, statechart, and communication models. USE supports developers in analyzing model structure and behavior, and in exploring properties of models (Gogolla et al. 2007). It supports the validation of UML/OCL models; verification tasks such as proving satisfiability of OCL invariants by constructing a positive test; checking independence of invariants, which means that no single invariant can be concluded from other stated invariants; and checking consequences (Gogolla et al. 2005, 2009, 2010; Gogolla, Burgueno, & Vallecillo 2018; Gogolla, Hilken, & Doan 2018). For more detail, we refer the readers to (Gogolla, Hilken, & Doan 2018).

USE is a textual modeling tool. It offers a simple language to describe a class model extended with the OCL constraints. Using USE, one can render a diagrammatic version of the textual model. However, USE is not intended for use in visualizing the models; rather it aims to help designers develop high-quality models using the validation and verification capabilities described earlier.

One of the most important features of USE is its capacity to verify whether a given instance is legal or not. This feature enables the validation of the model, determining whether it conforms to the requirements ("Are we building the right product?") or not. Creating an instance can be done (1) directly using the graphical user interface, or (2) using the Simple OCL-based Imperative Language (SOIL) in the command-line version of USE, or defining an operation that includes SOIL instructions in the USE environment. SOIL is the imperative programming language of USE (Büttner & Gogolla 2014). It enables creating and modifying system states of a USE specification (i.e., creating and modifying an instance), and specifying the behavior of operations in a USE environment. This enables executing the system by providing a sequence of SOIL commands that create the initial objects of the system, and their links (Burgueño et al. 2018).

Figure 1 demonstrates the use of USE for checking the validity of an instance of a class model extended with an OCL

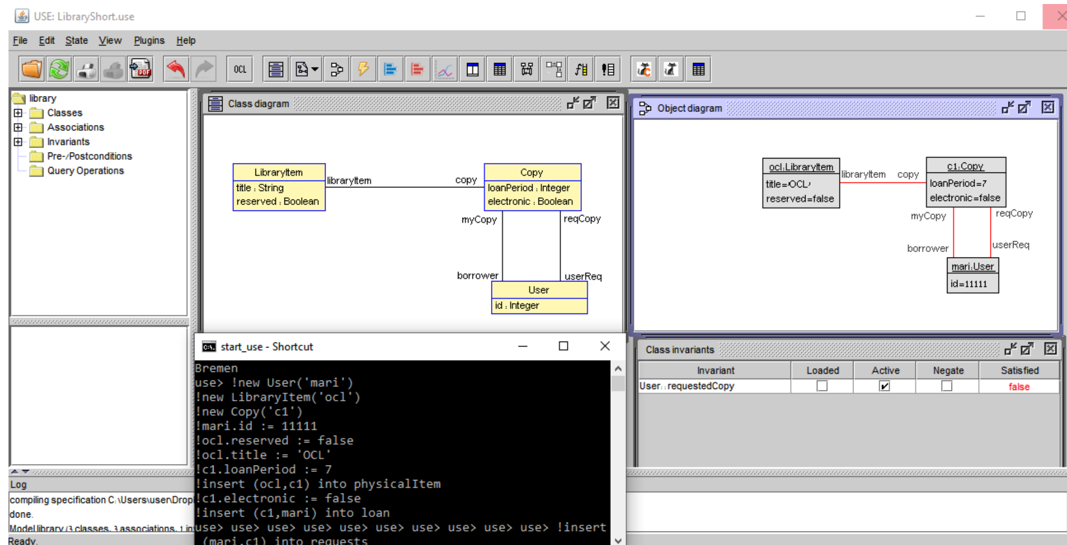
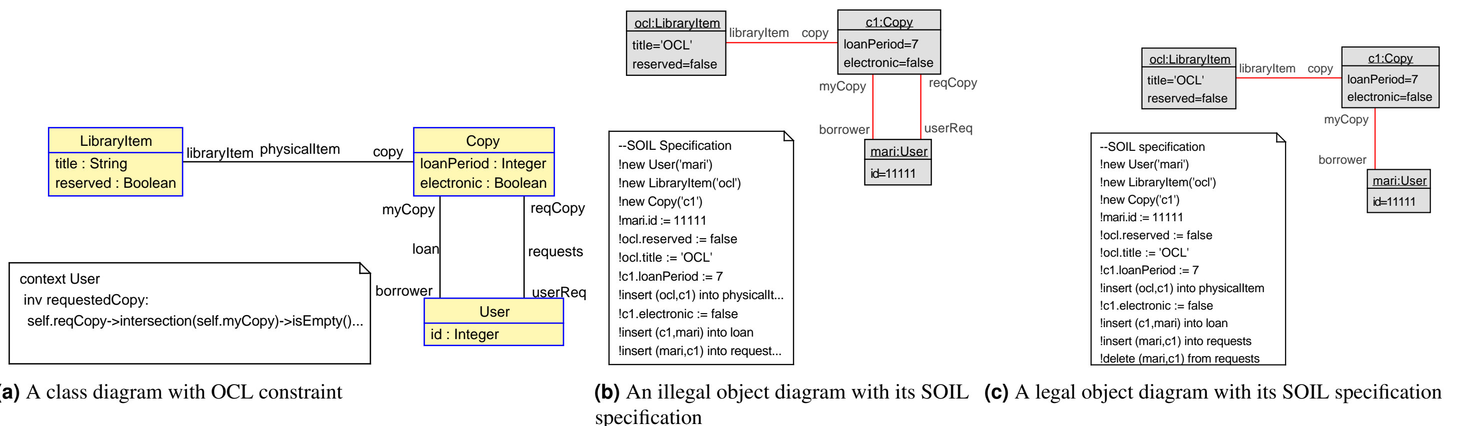


Figure 1 A demonstration of using USE for the validation of a class model extended with OCL constraints



(a) A class diagram with OCL constraint **(b)** An illegal object diagram with its SOIL specification **(c)** A legal object diagram with its SOIL specification

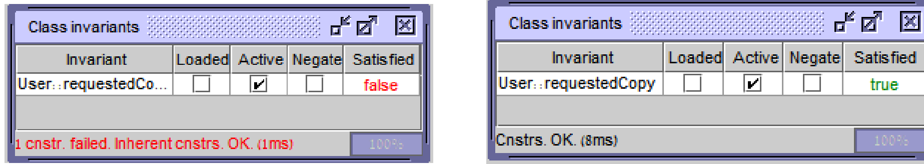
Figure 2 A class model with an OCL constraint, two instances, illegal and legal with their SOIL specifications

constraint. The middle-upper view introduces the class model; the middle-lower view introduces a SOIL specification of a legal instance; the left-upper view presents a graphical version of the same instance using the *Create object diagram view* feature of the GUI version; and the left-lower view presents the OCL invariants, with an indication for each constraint of whether it satisfied by the instance or not.

To further demonstrate the USE capabilities, Figure 2 presents a small class model extracted from a Library Management System model. The OCL constraint states that "*For each User, the currently borrowed copies (myCopy) could not appear in his ordered copies (reqCopy)*". Figures 2b and 2c present two instances with their **SOIL** specifications included in the comment boxes. The instance in Figure 2b is illegal since it violates the OCL constraint. The intersection between the object set resulting from the *myCopy* property and the object set resulting from the *reqCopy* property yields $\{c1\}$. Using USE to check the validity of the instance (via the *Create class invariants view* feature) results with the message shown in Figure 3a, showing that the constraint has not been satisfied. The SOIL specification of Figure 2c updates the SOIL specification of Figure 2b with the operation $!delete\ (mari, c1)\ from\ requests$, which appears in the last line, removing the link between the association *requests* and turning the instance in Figures 2b into a legal one. Indeed, verifying the validity of this instance yields the message presented in Figure 3b, which shows that the OCL constraint has been satisfied.

3.2. The Java Framework - JavaCL

Java is a general-purpose object-oriented programming language. It is considered one of the most popular language for developers and programmers to learn (IEEE Spectrum 2020; RedMonk 2020; Carbonnelle 2020; Ezenwoye 2019). Furthermore, Java has many libraries with rich API, which makes the development process faster and easier (Schildt 2014). Java is supported by numerous integrated development environments



(a) A USE message that show the OCL constraint is not satisfied **(b)** A USE message that show the OCL constraint is satisfied

Figure 3 Use Messages

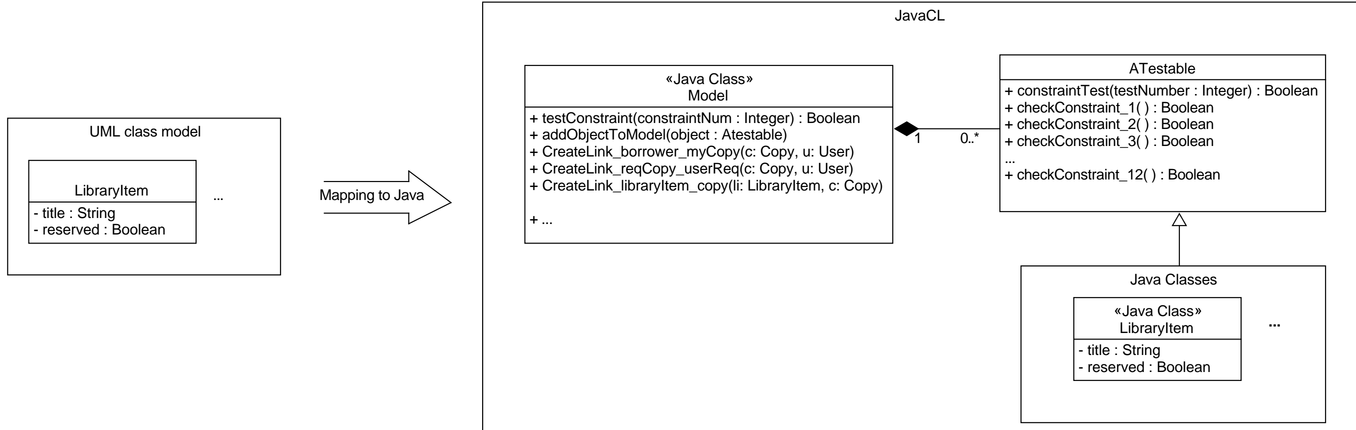


Figure 4 The JavaCL Framework

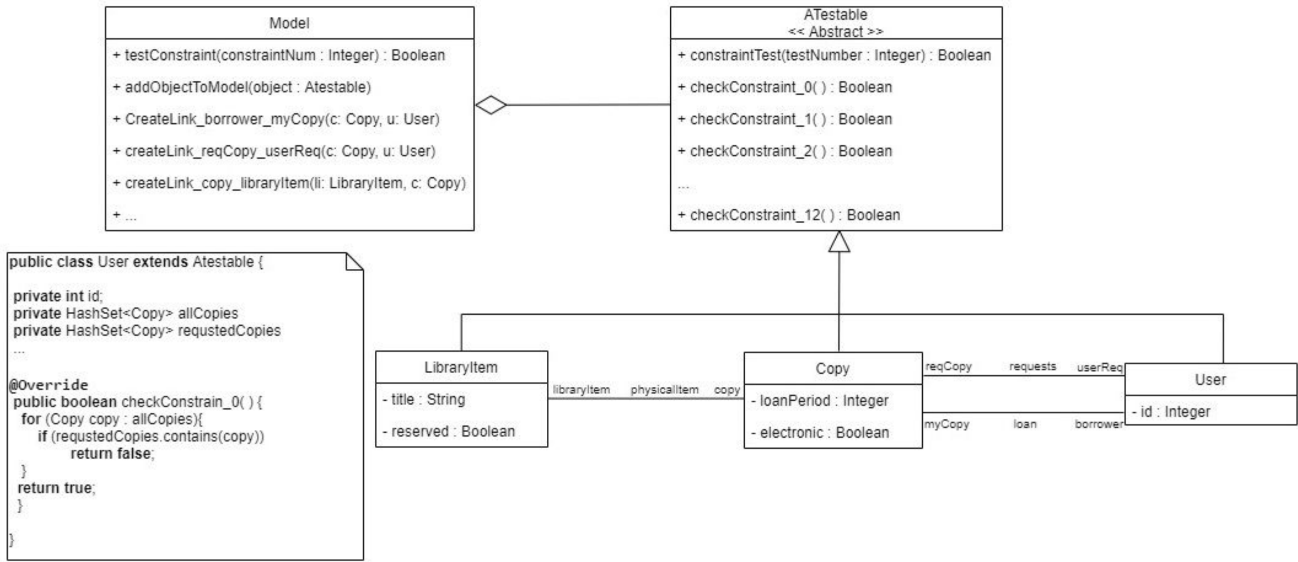
(IDEs), such as IntelliJ, Eclipse, and Netbeans.

However, Java is not designed to develop model-based constraints. To enable developing model-based constraints using Java similar to USE,

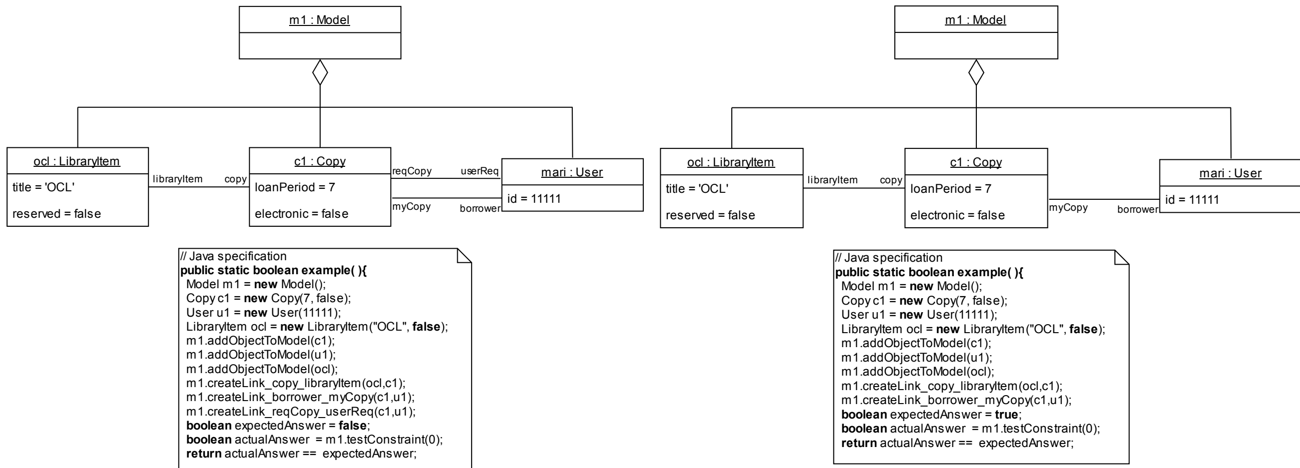
we have implemented an infrastructure that allows for constraint definition, instance creation and instance validation with respect to the defined constraints. We term this framework JavaCL. We implemented this infrastructure by defining an abstract class *ATestable* that is extended by all Java classes in the model. Java classes are generated from the model classes. The ATestable class contains a method for each model constraint. The default return value for each method is *true* implying that are no enforced constraints (similar to OCL, where an invariant includes only true). A specification of a constraint requires the overriding of the corresponding method in the relevant class (like a context class in OCL). Since each class extends the abstract class, they all inherit the default constraints' methods. Therefore, the non-context classes return true upon checking a specific constraint. In addition, JavaCL includes the class *Model*. This class is a container of the class objects (model instance). It enables the creation and validation of instances, and queries the instances. It has capabilities similar to those described for USE in the previous section, with respect to invariant checking. For running JavaCL, we used *IntelliJ IDEA*. IntelliJ provides rich capabilities such as authoring, modifying, compiling, auto-completion, and refactoring, which improve programmer productivity and the code quality. In this way, JavaCL under IntelliJ provides a "constraint-based Specification Environment" similar to USE. Figure 4 presents the JavaCL framework.

In the following we demonstrate the JavaCL capabilities. Figure 5a presents the same model that appears in Figure 2a with its implementation within the developed framework. As mentioned earlier, each class within the model extends the ATestable abstract class. The constraint states that "*For each User, the currently borrowed copies (myCopy) could not appear in his ordered copies (reqCopy)*". For creating instances, there is a need to explicitly create objects and links. Figures 5b and 5c present two instances, alongside their Java code specifications in the comment boxes. After defining the instances to be checked, each constraint is checked by sending its ID to the model. Then, for each object within the model, the constraint method is executed. The Java code *m1.testConstraint(0)* in the last line in the Java specification parts of Figures 5b and 5c invokes the desired constraint. It returns false in Figures 5b, meaning that the instance is illegal; and true in Figure 5c, meaning that the instance is legal.

Table 1 shows the differences between the two tools, USE and JavaCL + IntelliJ. The advantage of USE as a tool designed for models is clear. It is easier to load models, see them visually and to select a specific constraint for validation. In contrast to this, Java has a significant advantage as a programming tool with all the capabilities of syntax error handling and code completion.



(a) A class diagram with Java constraint



(b) An illegal object diagram with its Java code specification

(c) A legal object diagram with its Java code specification

Figure 5 A class model with a Java constraint, two instances: illegal and legal, with their Java specifications

4. Experiment Design and Execution

This paper examines the effectiveness of developing model-based constraints using two constraints languages, OCL and Java, using their supporting tools. In this section, we elaborate on the experiment design and its execution.

4.1. Hypotheses

By effectiveness, we refer to the quality of the resulting constraints, the time it takes to develop such constraints, and the confidence developers have in these constraints. We further establish a difference between four levels of complexity.

Our conjectures regarding the effectiveness of developing constraints using the two languages using their supporting tools were the following: We believe that there is a trade-off in specifying constraints in OCL or Java. Java is more familiar, yet it requires greater effort (in terms of the text size) to develop constraints; OCL, however, is less familiar, but requires less effort. In addition, working with an Integrated Development Environment (IDE) (as in the case of JavaCL+IntelliJ) is much simpler than working out of the checking environment (as in the case with USE). Thus, there is a trade-off involved in using either of the two languages and their supporting tools. This applies to the three factors we are interested in exploring: quality, time to develop the constraint, and confidence in the developed constraint. For the statistical analysis, we formalized the hypotheses as follows:

$$H_0^{Comp\text{-}Factor} : OCL^{Comp\text{-}Factor} = Java^{Comp\text{-}Factor}$$

Table 1 A Comparison between OCL and / JavaCL + IntelliJ

	USE	JavaCL + IntelliJ
Object visualization	✓	✗
Model loading using GUI	✓	✓ requires embedded the model code manually
Instance loading	✓	✓
Instance creation using GUI	✓	✗
Instance creation textually	✓	✓
Instance manipulation using GUI	✓	✗
Instance validation	✓	✓
Constraint selection for validation	✓	✓
Validation feedback	✓ visual, textual	✓ textual
Constraint evaluation	✓	✗
Syntax error handling	✗	✓
Code completion	✗	✓

$$H_1^{Comp\text{-}Factor} : OCL^{Comp\text{-}Factor} \neq Java^{Comp\text{-}Factor}$$

Where *Comp* refers to the complexity level of either all or of each of the levels we defined, and *Factor* refers to quality, time, and confidence.

4.2. Design

In the following, we describe the variables and their measurements, the subjects, and the tasks.

4.2.1. Independent Variables The first variable is the language according to which the constraints are set. It has two options: OCL and Java. Each of the languages is used with its supporting framework, USE and JavaCL+IntelliJ, respectively. The second variable is the constraint complexity. In this work, we adopted metrics inspired by (Yue & Ali 2016). We illustrate the metrics on the OCL constraint in Listing 1. Note that we carried out the same analysis on the constraints specified in Java, and the complexity level of the constraints were similar to those of OCL.

```
Context Library
    inv:  self.libraryItem->select(not electronic
    and oclIsTypeOf(ReferenceBook)).title->forAll(t
     | self.libraryItem->select(oclIsTypeOf(
    ReferenceBook) and title=t)->collect(oclAsType(
    Book)).numberOfCopy->sum()>=3)
```

Listing 1 An OCL constraint

1. Nesting level (M_{nl}) is defined as the maximum number of nested sub-queries within a main query. This is calculated as follow. The nesting level of the main query is 0; its immediate nested sub-query is 1; and the nesting level of each sub query is the nesting level of the upper sub query plus 1. For the OCL constraint in Listing 1, $M_{nl} = 1$.
2. Composition level (M_{comps}) is the maximum number of applications of OCL operations from a context class on the result of previous inner operations. For the OCL constraint in Listing 1, $M_{comps} = 2$.
3. Number of traversals (M_{tr}) is the number of navigations via the association ends or the operator from a context class to the constraint target class. For example, for the OCL constraint in Listing 1, $M_{tr} = 2$.
4. Maximum number of all OCL complex operators (M_{Op}). For the OCL constraint in Listing 1, $M_{Op} = 9$.
5. Number of different OCL operations (M_{DOP}): For the OCL constraint in Listing 1, $M_{DOP} = 6$.
6. Number of clauses (M_{cl}) is defined as the total number of clauses required in a constraint specification. A clause is an OCL expression supported by a Boolean operator. For the OCL constraint in Listing 1, $M_{cl} = 4$.

Based on these metrics, we ordered the constraints sequentially and classified the constraints according to four complexity levels, similar to our previous study (Maraee & Sturm 2020).

4.2.2. Dependent Variables

- The first dependent variable is the quality of a constraint.
- The second dependent variable is the time it takes to develop a constraint.
- The third dependent variable is the confidence that a subject has in the developed constraint.

We also checked the perception of the subjects with respect to the following:

- The suitability of the language (OCL and Java) for developing constraints;
- The ease of developing the constraints;
- The ease of checking the constraints;
- The efficiency of developing the constraints;
- The efficiency of checking the constraints;
- The use of the tools for finding errors; and
- The overall satisfaction of working with the tools.

4.2.3. Dependent Variables

For checking the quality, we prepared a set of instances that either satisfy (legal instances) or violate the constraints (illegal instances). The number of instances in general and the number of legal and illegal instances in each constraint is different, dependent on the complexity of the constraint. Table 2 shows for each constraint the number of legal and illegal instances. Constraint 3 `self.status=UserStatus::ACTIVE implies self.loan.fine->asSet()->forAll(fine | fine.paid)` includes the *implies* operation. Therefore, there are three possible legal instances: an instance where the two statements of the constraint are satisfied; an instance where the first statement of the constraint is satisfied (i.e., an instance with an active user); and instance where both statements are not satisfied (an inactive user who has not paid his fines). The only illegal instance is an instance with an active user who has not, however, paid his fines. Hence, for this constraint we created three legal instances and one illegal instance with similar consideration as with the other constraints.

Table 2 The number of legal and illegal instances for each constraint

Constraint	#Legal instances	#Ilegal instances
1, 2, 4, 9, 10, 11	1	1
3	3	1
5	2	2
6	3	1
7	1	2
8	3	1
12	3	2

Constraints Evaluation For each constraint, we calculated the ratio of the correct identification of the object diagrams given for that constraint. Thus, we actually measured the *precision* and *recall* used in information retrieval and data science for evaluating classifiers (Manning et al. 2008). In our case, an OCL constraint can be considered as a binary classifier. *Precision* is the fraction of retrieved (accepted) documents (instances) that are relevant (legal instances). *Recall* is the fraction of relevant documents (legal instances) that are retrieved (accepted). These metrics are calculated according to the following

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

Where true positive (TP) is the number of retrieved relevant items (accepted legal instances), false positive (FP) is the number of irrelevant retrieved items (accepted illegal instances), and false negative (FN) is the number of non-retrieved relevant items (rejected legal instances). Table 3 summarizes these notions (Manning et al. 2008).

We demonstrate the *precision* and *recall* metrics with respect to constraint 8 in the experiment: *In case a Library Item is not reserved, then it has at least one non-electronic Copy residing in an open Stack*, whose OCL specification appears in Listing 2.

```
context LibraryItem
    inv constraint_8:
        (not reserved) implies self.copy->select(not
    electronic).stack->select(open)->size()>=1
```

Listing 2 The OCL Specification of Constraint 8

Figure 6 presents instances for checking constraint 8, where Figure 6a and 6b present legal instances, and Figure 6c presents an illegal instance. Indeed, the OCL constraint in Listing 2 accepts the two legal instances in Figure 6 and rejects the illegal one. Hence, $TP = 2$, $FP = 0$, $FN = 0$. Therefore, $Precision = \frac{2}{2+0} = 1$ and $Recall = \frac{2}{2+0} = 1$.

A solution that replaces the operator `implies` in the OCL constraint in Listing 2 by the operator `and` as shown in Listing 3 yields a constraint that still accepts the legal instance in Figure 6a, rejects the illegal instance in Figure 6c, but also rejects the **legal instance** in Figure 6b. Hence, $TP = 1$, $FP = 0$, $FN = 1$, $Precision = \frac{1}{1+0} = 1$ and $Recall = \frac{1}{1+1} = \frac{1}{2}$.

```
context LibraryItem
(not reserved) and self.copy->select(not electronic)
    .stack->select(open)->size()>=1
```

Listing 3 The revised OCL Specification of Constraint 8

We also considered the time it took to specify the constraints, as measured by the subjects themselves and the confidence they had in their specified constraints, using a 7-point Likert scale. For the various perceptions, we used a 5-point Likert scale, and asked the subjects to list negative and positive issues.

4.2.4. Subjects

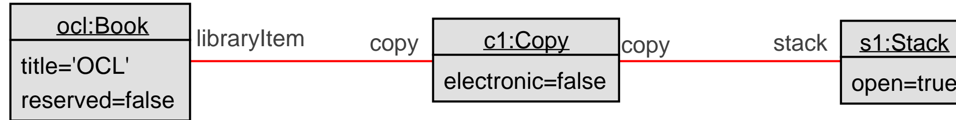
The subjects of the experiment were third year students taking the course "Analysis and Design of Software Systems." The course covers the analysis, design, and implementation of software based on the object-oriented paradigm. During the course, the subjects learned the concept of modeling, and in particular the use of a class diagram. In addition, they learned OCL, emphasizing invariants in class diagrams, and learned how to write constraints in Java. They also practiced class diagrams and writing constraints in both OCL and Java, using the same tools used in the experiment. The students had previous experience with Java and other programming languages, and used these as part of the course. Some of the students already had work experience in the software industry. Recruiting the students was done on a voluntary basis. Nevertheless, they were encouraged to participate in the experiment through the offer of additional bonus points for the course grade, based on their performance. Before recruiting the students, we sought and received approval from the department's ethics committee.

4.2.5. Task

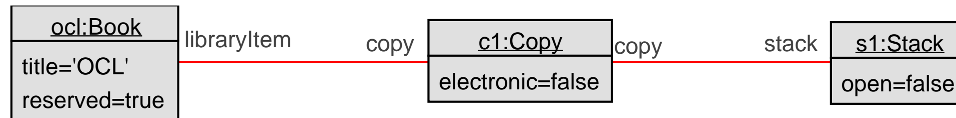
We designed the experiment using one factor—the language (and its supporting tool)—so that each

Table 3 Contingency Table

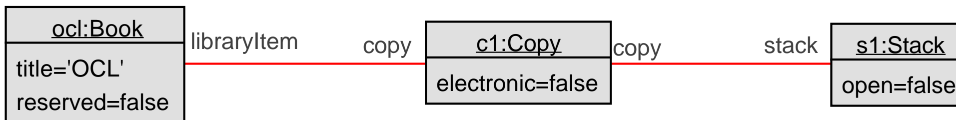
	legal instance	**illegal instance**
accepted by the constraint	true positives (tp)	false positives (fp)
rejected by the constraint	false negatives (fn)	true negatives (tn)



(a) A legal instance that satisfies constraint 8



(b) A legal instance that satisfies constraint 8



(c) An illegal instance

Figure 6 Instances for checking constraint 8

student would have enough time to develop the set of constraints. The experiment form consisted of three parts: (1) a pre-questionnaire, checking the background and knowledge of the subjects; (2) using one language, the subjects received a class diagram of a Library Management system (as appears in Appendix A), and were asked to write 12 constraints based on natural language specification. The constraints were ordered in ascending order of difficulty (see Appendix B). For each constraint, they were asked to mark the development time and their confidence in its correctness; (3) the last part of the form reflected upon the subjects' perception of the language and the tool that they had worked with. As part of the experiment, the subjects received an electronic copy of the model, either in USE or in Java, so that they could work with it during the experiment. They were encouraged to develop tests in the form of object diagrams for both JavaCL and USE. To do so, they either created instances using JavaCL when specifying the constraints in Java, or using SOIL when specifying the constraints in OCL. To check the validity of the instances, they used JavaCL utility and the validate feature in USE.

4.2.6. Execution The execution of the experiment took place in a special session that lasted between 2-3 hours (subjects were free to leave whenever they wished). The session took place in several labs equipped with Lenovo M900 Desktop (ThinkCentre - Type 10FH) with i5-6500 CPU and 16GB RAM. Both Intellij and USE were installed on all computers. A few students worked on their own laptops, which already had Intellij and USE installed. The assignment of the subjects to the groups (i.e., through the different forms) was at random (i.e., following the sitting arrangement in the labs). The distribution of groups was as followed: Form A, in which the subjects were using OCL and USE, was assigned to 49 subjects. Form B, in which the subjects were using JavaCL+IntelliJ, was assigned to 46 subjects. The students downloaded the electronic copy via the course website, and submitted their outcomes via a submission system, as they do with their regular coursework. No communication between the students were allowed, so as to ensure that the experiment was as controlled as possible. It should be noted that submission was not anonymous; nevertheless, the outcomes were checked automatically without referring to the students' details.

5. Experiment Results

In analyzing the results, we first checked the homogeneity between the two groups, using the pre-task questionnaire which referred to the subjects' background and knowledge. The questionnaire consisted of questions related to the subjects' familiarity with class and object diagrams, and with OCL and Java and their supporting tools. We also checked the differences between their GPAs. As all measures except for the GPA are ordinal, we applied the Mann-Whitney test. For the GPA, we checked whether it deviated from the normal distribution, and then applied the T-test. Overall, no significant differences were found among the groups (see Table 4). Nevertheless, subjects who were assigned to OCL/USE reported a lower degree of familiarity with OCL, and the subjects assigned to Java/JavaCL

reported limited knowledge about the specific framework. This was statistically significant.

Before analyzing the results, we examined the descriptive statistics and found that two of the constraints were of low quality. That is to say, all the measures were below 0.7 in both OCL with USE and JavaCL with IntelliJ. We attribute this to the way the constraints were introduced. Thus, we decided to omit these constraints (4 and 9) from our analysis. Table 5 presents the results with regard to quality, time, and confidence. The numbers in bold indicate the best results in each category. The rightmost column indicates the statistical significance of the results. Statistically significance differences are highlighted using an asterix, to denote sig<0.05. We calculated this using the Mann-Whitney test as the data had deviated from the normal distribution (following the Shapiro-Wilk test we performed). In general, time and confidence were in favor of Java, for which only some of the results were statistically significant. On the other hand, the quality measures of precision and recall were in favor of OCL and many of them were of statistical significance.

Based on these results we rejected the following hypotheses:

$H_0^{All\text{-}Time} : OCL^{All\text{-}Time} = Java^{All\text{-}Time}$
$H_0^{All\text{-}Precision} : OCL^{All\text{-}Precision} = Java^{All\text{-}Precision}$
$H_0^{All\text{-}Recall} : OCL^{All\text{-}Recall} = Java^{All\text{-}Recall}$
$H_0^{Level1\text{-}Precision} : OCL^{Level1\text{-}Precision} = Java^{Level1\text{-}Precision}$
$H_0^{Level2\text{-}Time} : OCL^{Level2\text{-}Time} = Java^{Level2\text{-}Time}$
$H_0^{Level2\text{-}Recall} : OCL^{Level2\text{-}Recall} = Java^{Level2\text{-}Recall}$
$H_0^{Level3\text{-}Recall} : OCL^{Level3\text{-}Recall} = Java^{Level3\text{-}Recall}$

The statistical significant differences in the time it took to develop the constraints in favor of Java and JavaCL originate from the time it took to develop the constraints at the second level. The statistical significant differences in the quality in favor of OCL and USE span across all levels, except for the fourth level.

Table 6 presents the results of the post-task questionnaire, which reflected on the subjects' perceptions. The results indicated that both languages fitted the task of specifying constraints. Using JavaCL and IntelliJ was perceived as easier and more effective for specifying the constraints, resulting in higher levels of satisfaction. On the other hand, using OCL and USE was perceived to be easier and more effective for checking the constraints and for finding mistakes.

We further analyzed the comments written by the subjects. For that purpose, we performed a thematic analysis, which is presented in Table 7. The results of that analysis are aligned with the post-questionnaire results. Java and IntelliJ are known to the students; thus they indicated that it was easy to write the constraints. Nevertheless, the utilities for checking the constraints were challenging. OCL and USE were perceived as difficult-to-write constraints; however, they were perceived positively with regard to checking the constraints and identifying semantic errors.

6. Discussion

The results indicate that using OCL and USE achieved constraints of better quality than when using JavaCL and IntelliJ.

```
    public boolean checkConstraint_1(Model model) {
        int count=0;
        for (Loan loan:loans) {
            if(loan.getFine().paid&&loan.getFine()==
    this){
                count++;
            }}
        return count==loans.size();
    }
```

Listing 5 A student Java solution for Constraint 1

This is particularly relevant in accepting legal instances (recall) and is less noticeable in rejecting some of the legal instances (precision). In an attempt to understand these results, we performed a qualitative analysis comparing the types of errors in both settings. For this purpose, we chose three constraints, 1,6, and 11. These are presented in Listing 4.

```
Constraint 1. The number of Loans for which a Fine
    was paid is equal to the number of Loans
    associated with the same Fine.

Constraint 6. Only an Academic can borrow a reserved
     Journal.

Constraint 11. The number of editions of a Book is
    equal to the number of Books with the same
    title.
```

Listing 4 Constraints 1, 6, and 11

Constraint 1. Using OCL and USE achieved better results than when using JavaCL, and the differences in precision were statistically significant. Although there is no statistical significance in favor of OCL with respect to recall, the results in the recall metric are also better than Java. The solution in Java includes one expression `return numberOfLoan==loans.size()`. We therefore expected that there would be no differences between the two languages. However, 21% of the subjects (10 of 46) used unnecessary loop and control expressions to iterate over the loan collection when using JavaCL, compared to OCL where only 6% of the subjects (3 of of 49) used collection operators such as `select` and `forall`. In addition, 8 subjects who applied loops in Java had errors in writing the constraint and each got 0.55 in precision metrics. An example of a Java solution appears in Listing 5.

Constraint 6. In general, developing this constraint using OCL has a notable advantage over JavaCL and it has a recall difference of statistical significance. The constraint requires that *only* ***Academic*** *can borrow a* ***reserved Journal***. Indeed, the constraint is quite complex, as it is required to capture the **implication** relation between a reserved journal and academics: (`reserved implies self.copy.borrower->asSet()->forAll(oclIsTypeOf(Academic)`). Using OCL, the subjects were unable to capture this dependency. Analyzing 19 (39%) such answers showed that the written constraints specified that *students can loan only books* or *a journal can be loaned only by an academic* as shown in the OCL

Table 4 Pre-task questionnaire results. The triplets indicate the number of answers we received, the average of these answers, and the standard deviation. The Sig. column indicates the significance of the statistical test. The numbers in bold font indicate the best results. The asterixes in the Sig. column indicate statistically significant differences.

	Form A - OCL	Form B - Java	Sig
Class Diagram Familiarity	47, 3.85, 0.73	**45, 3.89, 0.78**	0.649
Object Diagram Familiarity	**46, 3.6, 0.77**	45, 3.64, 0.80	0.446
OCL/Java Familiarity	47, 3.40, 0.64	**45, 3.95, 0.74**	0.00*
USE/Java Framework Familiarity	**48, 3.40, 0.64**	42, 3.16, 1.29	0.724
Participation in Tutorial	**44**	43	0.944
Homework Involvement	**47, 4.38, 0.67**	45, 4.29, 0.66	0.453
GPA	47, 81.63, 4.98	**44, 81.66, 5.12**	0.981

Table 5 Constraints Development Results of the Library Management System. The triplets indicate the number of answers we received, the average of these answers, and the standard deviation. The Sig. column indicates the significance of the statistical test. The bold font numbers indicate the best results. The asterixes in the Sig. column indicate statistical significant differences.

Level	Variable	Form A - OCL	Form B - Java	Sig.
Total	**Time**	46, 12.05, 4.11	**46, 10.45, 3.81**	0.028*
	Precision	**49, 0.91, 0.11**	46, 0.84, 0.13	0.004*
	Recall	**49, 0.89, 0.11**	46, 0.80, 0.13	0.001*
	Confidence	46, 5.22, 1.17	**45, 5.42, 0.88**	0.494
Level 1	**Time**	46, 12.14, 6.84	**46, 11.78, 8.25**	0.534
	Precision	**49, 0.98, 0.07**	46, 0.92, 0.16	0.048*
	Recall	**49, 0.98, 0.07**	46, 0.93, 0.13	0.114
	Confidence	46, 5.86, 1.13	**45, 5.74, 1.08**	0.544
Level2	**Time**	46, 12.49, 7.29	**46, 8.77, 4.04**	0.007*
	Precision	**48, 0.90, 0.18**	46, 0.84, 0.27	0.42
	Recall	**48, 0.86, 0.23**	46, 0.75, 0.27	0.01*
	Confidence	46, 5.16, 1.57	**45, 5.63, 1.33**	0.25
Level 3	**Time**	46, 9.20, 4.33	**45, 8.16, 4.42**	0.135
	Precision	**48, 0.92,0.19**	45, 0.86, 0.24	0.191
	Recall	**48, 0.92, 0.20**	45, 0.83, 0.24	0.006
	Confidence	45, 5.06, 1.49	**44, 5.39, 1.28**	0.346
Level 4	**Time**	42, 13.82, 7.58	**46, 11.22, 4.90**	0.239
	Precision	**41, 0.78, 0.29**	46, 0.72, 0.24	0.071
	Recall	**41, 0.75, 0.27**	46, 0.69, 0.24	0.179
	Confidence	41, 4.53, 1.69	**45, 4.95, 1.32**	0.196

constraint in Listing 6. Hence, their constraints rejected legal instances that included students with loaned reserved journals. Therefore, their recall scores were 0.50, while most of the precision scores were 1.

Table 6 Post-task questionnaire results. The triplets indicate the number of answers received, the average of these answers, and the standard deviation. The Sig. column indicates the significance of the statistical test. The numbers in bold font indicate the best results. The asterixes in the Sig. column indicate statistically significant differences.

	Form A - OCL	Form B - Java	sig
The language fits for constraints specification	**47, 4.09, 0.69**	43, 4.07, 0.8	0.993
Using the tool for specifying constraints was easy	46, 2.57, 0.96	**43, 3.74, 0.85**	0*
Using the tool for checking constraints was easy	**47, 3.00, 1.29**	42, 2.71, 1.09	0.274
I manage to write the constraints effectively	46, 3.04, 0.76	**43, 3.70, 0.77**	0*
I manage to check the constraints effectively	**47, 3.04, 1.16**	42, 2.64, 0.98	0.111
The tool helped in identifying mistakes	**47, 3.51, 1.04**	44, 2.83, 0.76	0.001*
I was satisfied while working with the tool	47, 2.55, 1.12	**43, 3.33, 0.97**	0.001*

Table 7 Thematic Analysis of the Subjects' Comments

	Form A - OCL	Form B - Java
Positive	The tool support object visualization (9)	Java familiarity (9)
	Help in error identification (6)	Easy to write constrains (17)
	Eady checking (17)	
Negative	No information on syntax error (12)	Difficult to understand the model (3)
	Cumbersome (13)	Difficult to check the constraint (25)
		Object creation is cumbersome (7)

Using Java, 26 subjects (57%) experienced this kind of error, compared to 19 in OCL. Four subjects received a recall score of 0.33 due to logical errors, whereas in OCL no such errors occurred. Listing 7 presents such a Java solution.

```
OCL:
context Journal inv constraint_6:
self.copy->asSet()->forAll(c: Copy| c.borrower->
    asSet()->forAll(u: User| u.oclIsTypeOf(Academic
    )))
```

Listing 6 OCL student solutions for Constraint 6

Constraint 11. This constraint belongs to Level 4 and there are no statistically significant differences between the two languages. However, it is still noticeable that the subjects were more successful when using OCL. Below, we analyze the performance of this constraint. This is a very complex constraint that requires complex navigation, including using the type operator *oclIsTypeOf*. Using OCL, 22 subjects (45%) received 1 in both metrics; 12 of the 35 subjects (34%) who solved the constraint wrote a constraint that rejected all instances, so they received zero in recall and precision metrics. Most of the mistakes were type mistakes, where the subjects ignored the *oclIsTypeOf* operation or used the operation *oclIAsType* instead.

```
Java:
Class Journal{...
    public boolean checkConstraint_6(Model model) {
        for (Copy copy:this.copies) {
            for (HashMap.Entry<Loan,User> entry:copy
    .getBorrowers().entrySet()) {
                if(! (entry.getValue() instanceof
    Academic) )
                    return false;
            }
    }
    return true;
}
...}
```

Listing 7 Java student solutions for Constraint 6

Analyzing the answers in Java shows that 20 subjects (45%)

```java
class Library{...
    public boolean checkConstraint_11(Model model) {
        boolean good = true;
        HashMap<String,Integer> titels = new HashMap
    <>();
        for(LibraryItem l:libraryItems){
            if(l instanceof Book){
                if(titels.containsKey(l.title)){
                    Integer val = titels.get(l.title
    );
                    titels.replace(l.title ,val,val
    +1);
                }
                else{
                    titels.put(l.title ,1);
                }
            }
        }
        for(LibraryItem l:libraryItems){
            if(l instanceof Book) {
                if(((Book) l).numberOfEditions !=
    titels.get(l.title)){
                    good = false;
                }
            }
        }

        return good;
    }
...
}
```

Listing 8 A student Java solution for constraint 11

received 1 in both metrics; 20 of the 44 subjects (45%) wrote a constraint that rejected all instances and hence received zero in recall and precision metrics. Most of the Java solutions used unnecessarily complex data structures, and unnecessarily nested control expressions. Another issue is that 66% of the answers (29 out of 44) in Java used Library as the context class instead of Book, whereas in OCL 94% of the solutions used Book as the class context. Using Library as a context significantly increased the use of control expressions including nested expressions and complex data structures (such as HashMap); 18 out of the 20 subjects that received zero in the recall and precision metrics used Library as a context. Listing 8 shows the solution of one of the students in Java. In OCL, 94% of solutions did not contain nested expressions (only 6 solutions used Library as context), and 86% of the correct answers used Book as the context class.

Below is a summary of the types of errors we found following the analysis of the students' answers:

1. Using nested loops and unnecessary data structures: Students used unnecessary control expressions and data structures which made their code more complex. A significant proportion of the students did not use the rich API of the collection framework. Table 8 presents the usage of unnecessary control expressions and data structures in several constraints,

2. Obsessive use of temporal variables: Using local variables is not a bad practice per se, but overuse is considered bad design (bad smell) [1]. The overuse of temporal variables by the students increased the complexity of the code and also led to logical errors.

3. Incorrect context (class): Choosing the right context for a constraint is important, and requires considering the object which is responsible for carrying this constraint. Many of the students defined the constraints in the wrong classes which led to the definition of complex constraints (due to long navigation paths or a lot of iterations). The next column in Table 9 presents the right "gold" contexts for the constraints in the first column. The last two columns present the choice of the context in the second column by the students in both languages, Java and OCL. It is notable that there are significant advantages for using OCL. For the rest of the constraints, we found no significant differences, even though OCL had advantages in all.

4. Defensive programming approach: A lot of the students' codes defensively included null checks. Although null checks are essential, they are unnecessary for *non-nullable*[2] reference types. Most reference types are derived from association roles. Association roles whose minimum multiplicity constraints are greater than one define *non-nullable* reference types. Hence, there is no need for the null checking of those references.

Table 8 Percentage of use of unnecessarily control expressions, local variables and data structures in several constraints

Constraint	Java	OCL
Constraint 1	0.21	0.06
Constraint 2	0.36	0.10
Constraint 3	0.47	0.08
Constraint 5	0.32	0.08
Constraint 6	0.32	0.16
Constraint 7	0.32	0.12
Constraint 8	0.36	0.10
Constraint 11	0.68	0.07

In summary, the differences between the two languages (i.e., imperative object-oriented paradigm vs. declarative paradigm) affected the way the subjects solved the constraints. Writing the constraints declaratively, using object navigation, collections, collection operations, and Boolean-valued expressions, shifted the way the subjects formulated (programmed) the constraints. Subjects who wrote the constraints in Java tended to use local variables and control expressions even with constraints that did not require this, such as constraint 1. It seems that the subjects

[1] Indeed, one of the refactoring transformations is *inline a temporary variable* which removes a temporary variable and replaces it with its value instead.
[2] Non-nullable variables must always contain a value and cannot be null

Table 9 Choosing the context in the two languages for the constraints 2, 3, 5 and 11 (have a significant difference of more than 5%)

Constraint	Context	Java	OCL
Constraint 2	Book	0.85	0.98
Constraint 3	Student	0.93	0.99
Constraint 5	ReferenceBook	0.85	0.92
Constraint 11	Book	0.44	0.94

were affected by a culture that values loop and control structures, which led to the unnecessary use of nested loops and to further focus on implementation aspects. We noticed these phenomena in our previous research where subjects wrote the constraints manually without using tools (Maraee & Sturm 2020). By using tools in this study, syntactic errors that manifested in the previous experiment did not appear here, and using the tools to validate the constraints improved the constraint quality. However, Java solutions still included unnecessarily complex control expressions. Object-oriented programming itself does not encourage this writing. On the contrary, it is a paradigm of which an encapsulation is one of its important principles, and it encourages writing in terms of services. But languages like Java enable easy writing of control statements, and the subjects tended to use them. The constraints in Java (JavaCL) were written at the modeling stage, before the development phase, in which design decisions, and architecture decisions are made. Hence, the written constraints were of low quality, and violated principles of the programming language itself as we have indicated.

Comparing the results to those of our previous experiment (Maraee & Sturm 2020), we determined that the two are aligned to a large extent. One noticeable difference is in the confidence of the subjects with respect to the developed constraints, which were increased when using the tools. Another difference manifested in the time it took to develop the constraint. In that sense, using the tool for validating the constraints led to longer development time. The longer development time makes sense, as the subjects used the tools to check their solutions. For the quality of the constraints, there were no consistent differences between the two experiments. However, when referring to OCL, it seems that using USE to develop constraints improved their quality. Although the improvement seemed to be of limited significance, we believe that when models become more complex, the use of tools is inevitable.

7. Threats to Validity

The results of our study need to be considered in view of several threats to validity categorized by (Wohlin et al. 2012) as construct, internal, conclusion, and external validity.

Construct validity threats, which concern the relationships between theory and observation, are mainly due to the method used to assess the outcomes of the tasks. In this experiment, we examined two languages and their supporting tools for specifying constraints. It is difficult to separate the effects of the language and of the tool. The experiment confirms the results of the previous one regarding the language used for developing the constraints. We further refer to the subjects' comments in explaining the differences from the tools' point of view as well.

Internal validity threats, which concern the external factors that might affect the dependent variables, may affect the results due to individual factors, such as familiarity with the domain, the degree of commitment by the subjects, and the training level that the subjects underwent. These factors are neutralized by the experiment design that we chose. Specifically, both groups had similar conditions and performed the same task. Even though participation in the experiment was on a voluntary basis, the compensation of bonus points based on the students' performance increased the motivation and commitment of the subjects as they took advantage of the entire time allocated for the experiment.

Conclusion validity threats concern the relationship between the treatment (the constraint language and supporting tool) and the outcome. We followed the various assumptions of the statistical tests when analyzing the results. As the checking of the quality of the constraints was carried out automatically, the outcomes were calculated objectively. Furthermore, the results of the post-questionnaire also re-confirm the statistical analysis.

External validity concerns the generalization of the results. The effect of using the tools should be taken with caution as we experimented with only two possible tools. Thus, the generalization is limited. Nevertheless, the results indicate that the tools should focus and provide proper support for the main task needed to be performed. In our case, the task was to achieve high-quality constraints, and the tools were required to support that aspect. In particular, they should facilitate the generation of valid and invalid instances, check the constraint validity, and in the case of problems indicate what these problems are. Needless to say, the usability of the tools should be examined; and in our experiment, we overlooked this aspect.

8. Summary

In this study, we examined the effectiveness of using a constraint language with its supporting tool in developing model-based constraints. We found out that in terms of quality, the use of OCL and USE outperforms the use of a Java framework along with an IDE support. We found that the utilities for checking the constraints encouraged the subjects to improve their qualities. Nevertheless, the tool's usability should be further examined; in particular, the means for indicating syntax errors and the working environment. This paper summarizes a series of experiments that we conducted in order to examine the process of working with model-based constraints. All experiments indicated that a declarative language such as OCL better serves the purpose of achieving constraints with proper quality. Unsurprisingly, arriving at these constraints takes much more time and engenders

lower levels of confidence, as such languages are less familiar. Thus, further education on declarative languages should be provided along with a rich set of tools to support their usage. In the future, we plan to look for means to further stimulate modelers to specify constraints by means of recommending or automatically generating the relevant constraints.

References

Agner, L. T., Lethbridge, T. C., & Soares, I. W. (2019). Student experience with software modeling tools. *Software & Systems Modeling*, *18*(5), 3025–3047.

Ali, S., Iqbal, M.-Z., Arcuri, A., & Briand, L.-C. (2013). Generating test data from ocl constraints with search techniques. *IEEE Transactions on Software Engineering*, *39*(10), 1376–1402.

Ali, S., Yue, T., Iqbal, Z., & Panesar-Walawege, R.-K. (2014). Insights on the use of ocl in diverse industrial applications. In *International conference on system analysis and modeling* (pp. 223–238).

Anastasakis, K., Bordbar, B., Georg, G., & Ray, I. (2010). On Challenges of Model Transformation from UML to Alloy. *Software and Systems Modeling*, *9*(1), 69–86.

Auer, M., Meyer, L., & Biffl, S. (2007). Explorative uml modeling-comparing the usability of uml tools. In *9th international conference on enterprise information systems- iceis 2007* (pp. 466–473).

Balaban, M., Bennett, P., Doan, K.-H., Georg, G., Gogolla, M., Khitron, I., & Kifer, M. (2016). A comparison of textual modeling languages: Ocl, alloy, foml. In *16th international workshop on ocl and textual modeling, models (2016).*

Balaban, M., & Kifer, M. (2011). Logic-based model-level software development with f-oml. In *International conference on model driven engineering languages and systems.*

Briand, L. C., Labiche, Y., Yan, H.-D., & Penta, M. D. (2004). A controlled experiment on the impact of the object constraint language in uml-based maintenance. *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 380-389.

Brucker, A. D., & Wolff, B. (2008). Hol-ocl: a formal proof environment for uml/ocl. In *International conference on fundamental approaches to software engineering* (pp. 97–100).

Burgueño, L., Vallecillo, A., & Gogolla, M. (2018). Teaching uml and ocl models and their validation to software engineering students: an experience report. *Computer Science Education*, *28*(1), 23–41.

Büttner, F., & Gogolla, M. (2014). On ocl-based imperative languages. *Science of Computer Programming*, *92*, 162–178.

Cabot, J., Clarisó, R., & Riera, D. (2014). On the verification of uml/ocl class diagrams using constraint programming. *Journal of Systems and Software*, *93*, 1–23.

Cabot, J., & Gogolla, M. (2012). Object constraint language (ocl): a definitive guide. In *International school on formal methods for the design of computer, communication and software systems* (pp. 58–90).

Carbonnelle, P. (2020). Pypl popularity of programming language index [Computer software manual]. (Available at http://pypl.github.io/PYPL.html)

Correa, A., Werner, C., & Barros, M. (2007). An empirical study of the impact of ocl smells and refactorings on the understandability of ocl specifications. In *Proceedings of the 10th international conference on model driven engineering languages and systems* (p. 76–90). Berlin, Heidelberg: Springer-Verlag.

Database Systems Group, B. U. (2020). Use: A uml-based specification environment [Computer software manual]. (Available at http://https://sourceforge.net/projects/useocl/, Accessed: 3-1-2020)

Demuth, B., & Wilke, C. (2009). Model and object verification by using dresden ocl. In *Proceedings of the russian-german workshop innovation information technologies: Theory and practice, ufa, russia* (pp. 687–690).

Dix, A. (2009). Human-computer interaction. In *Encyclopedia of database systems* (pp. 1327–1331). Springer US. Retrieved from https://doi.org/10.1007/978-0-387-39940-9_192 doi: 10.1007/978-0-387-39940-9_192

Ezenwoye, O. (2019). What language? - The choice of an introductory programming language. In *Proceedings - frontiers in education conference, fie* (Vol. 2018-October, pp. 1–8). Institute of Electrical and Electronics Engineers Inc. Retrieved from https://ieeexplore.ieee.org/document/8658592/ doi: 10.1109/FIE.2018.8658592

Gogolla, M., Bohling, J., & Richters, M. (2005). Validating uml and ocl models in use by automatic snapshot generation. *Software & Systems Modeling*, *4*(4), 386–398.

Gogolla, M., Burgueno, L., & Vallecillo, A. (2018). Model finding and model completion with use. In *Models workshops* (pp. 194–200).

Gogolla, M., Büttner, F., & Cabot, J. (2013). Initiating a benchmark for uml and ocl analysis tools. In *International conference on tests and proofs* (pp. 115–132).

Gogolla, M., Büttner, F., & Cabot, J. (2014). Ocl repository [Computer software manual]. (Available at https://github.com/jcabot/ocl-repository, Accessed: 2016)

Gogolla, M., Büttner, F., & Richters, M. (2007). USE: A uml-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, *69*(1-3), 27–34. Retrieved from https://doi.org/10.1016/j.scico.2007.01.013 doi: 10.1016/j.scico.2007.01.013

Gogolla, M., Hamann, L., & Kuhlmann, M. (2010). Proving and visualizing ocl invariant independence by automatically generated test cases. In *International conference on tests and proofs* (pp. 38–54).

Gogolla, M., Hilken, F., & Doan, K.-H. (2018). Achieving model quality through model validation, verification and exploration. *Computer Languages, Systems & Structures*, *54*, 474–511.

Gogolla, M., Kuhlmann, M., & Hamann, L. (2009). Consistency, Independence and Consequences in UML and OCL Models. In *Proceedings of the 3rd international conference on tests and proofs* (pp. 90–104). Springer-Verlag.

Hammad, M., Yue, T., Wang, S., Ali, S., & Nygård, F. (2017).

Iocl: An interactive tool for specifying, validating and evaluating ocl constraints. *Science of Computer Programming*, *149*, 3–8.

IBM. (2019). Ibm rational software architect designer [Computer software manual]. (Available at https://www.ibm.com/developerworks/downloads/r/architect/index.html)

IBM. (2020). Eclipse papyrus [Computer software manual]. (Available at https://www.eclipse.org/papyrus/)

IEEE Spectrum. (2020). The top programming languages 2019 [Computer software manual]. (Available at https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019)

Jackson, D. (2002). Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *11*(2), 256–290.

Khaled, L. (2009). A comparison between uml tools. In *2009 second international conference on environmental and computer science* (pp. 111–114).

Khitron, I., Balaban, M., & Kifer, M. (2016). *The FOML Site.* https://goo.gl/AgxmMc.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge university press.

Maraee, A., & Sturm, A. (2019). The usage of constraint specification languages: A controlled experiment. In *Enterprise, business-process and information systems modeling - 20th international conference, BPMDS 2019, 24th international conference, EMMSAD 2019, held at caise 2019, rome, italy, june 3-4, 2019, proceedings* (pp. 329–343). Retrieved from https://doi.org/10.1007/978-3-030-20618-5_22 doi: 10.1007/978-3-030-20618-5_22

Maraee, A., & Sturm, A. (2020). Imperative vs declarative constraint specification languages: A control experiment. *Software & Systems Modeling*, *in press*.

Mengerink, J.-G., Noten, J., & Serebrenik, A. (2019). Empowering ocl research: a large-scale corpus of open-source data from github. *Empirical Software Engineering*, *24*(3), 1574–1609.

Nakajima, S. (2014). Using alloy in introductory courses of formal methods. In *International workshop on structured object-oriented formal language and method* (pp. 97–110).

NoMagic. (2020). Magicdraw [Computer software manual]. (Available at https://www.nomagic.com/)

Noten, J., Mengerink, J.-G. M., & Serebrenik, A. (2017). A data set of ocl expressions on github. In *Proceedings of the 14th international conference on mining software repositories* (p. 531–534). IEEE Press.

Pérez, B., & Porres, I. (2019). Reasoning about uml/ocl class diagrams using constraint logic programming and formula. *Information Systems*, *81*, 152–177.

Planas, E., & Cabot, J. (2020). How are uml class diagrams built in practice? a usability study of two uml tools: Magicdraw and papyrus. *Computer Standards & Interfaces*, *67*, 103363.

Portal, O. (2014). Ocl tools [Computer software manual]. (Available at http://st.inf.tu-dresden.de/oclportal/index.php?option=com_content&view=category&id=8&Itemid=26)

Queralt, A., Artale, A., Calvanese, D., & Teniente, E. (2012). Ocl-lite: Finite reasoning on uml/ocl conceptual schemas. *Data Knowl. Eng.*, *73*, 1–22.

RedMonk. (2020). The RedMonk Programming Language Rankings: January 2020 – tecosystems [Computer software manual]. (Available at https://redmonk.com/sogrady/2020/02/28/language-rankings-1-20/)

Richters, M., & Gogolla, M. (2002). Ocl: Syntax, semantics, and tools. In *Object modeling with the ocl* (pp. 42–68). Springer.

Safdar, S.-A., Iqbal, M.-Z., & Khan, M.-U. (2015). Empirical evaluation of uml modeling tools–a controlled experiment. In *European conference on modelling foundations and applications* (pp. 33–44).

Schildt, H. (2014). *Java: the complete reference*. McGraw-Hill Education Group.

Software Technology Chair, T. D. (2004). Dresden ocl [Computer software manual]. (Available at https://github.com/dresden-ocl, Accessed:2016)

Störrle, H. (2013). Improving the usability of OCL as an ad-hoc model querying language. In *Proceedings of the MODELS 2013 OCL workshop co-located with the 16th international ACM/IEEE conference on model driven engineering languages and systems (MODELS 2013), miami, usa, september 30, 2013* (pp. 83–92). Retrieved from http://ceur-ws.org/Vol-1092/stoerrle.pdf

Warmer, J., & Kleppe, A. (2003). *The object constraint language: Getting your models ready for mda* (2nd ed.). Addison-Wesley. Retrieved from https://www.safaribooksonline.com/library/view/object-constraint-language/0321179366/

Wille, R., Soeken, M., & Drechsler, R. (2012). Debugging of inconsistent uml/ocl models. In *Proceedings of the conference on design, automation and test in europe* (p. 1078–1083). San Jose, CA, USA: EDA Consortium.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., & Wessln, A. (2012). *Experimentation in software engineering*. Springer Publishing Company, Incorporated.

Yue, T., & Ali, S. (2016). Empirically evaluating ocl and java for specifying constraints on uml models. *Software & Systems Modeling*, *15*(3), 757–781.

A. The Library System Class Diagram

In Figure 7 we show the class diagram of the Library System that was provided to the subjects.

B. The Constraints

In the following, the constraints specified in natural language are ordered by their difficulty and the way they were introduced to the subjects.

Complexity Level 1

Constraint 1. The number of Loans for which a Fine was paid is equal to the number of Loans associated with the same Fine.

Constraint 2. The number of Copies associated with a Book are equal to the number of Copies indicated in the Book itself.

Constraint 3. An active Student paid all her Fines.

Complexity Level 2

Constraint 4. The amount of a Fine over Loans is equal or lower than the sum of all defined Fines (LibraryItem.fine of the copies associated with these Loans).

Constraint 5. For each Reference Book, there is at least one non-electronic.

Constraint 6. Only an Academic can borrow a reserved Journal.

Complexity Level 3

Constraint 7. For each User, the currently borrowed copies (myCopy) will not appear in his ordered copies (requestedCopy) and the list of the currently borrowed copies (myCopy) is included within her entire set of loans (allCopy).

Constraint 8. In case a Library Item is not reserved, then it has at least one non-electronic Copy residing in an open Stack.

Constraint 9. The Copies associated with a Fine includes all Copies associated with the Loans associated with that Fine.

Complexity Level 4

Constraint 10. All the Loans associated with a Fine are of the same User.

Constraint 11. The number of Editions of a Book is equal to the number of Books with the same title.

Constraint 12. A Reference Book has at least non-electronic copies of all its editions.

About the authors

Azzam Maraee is on the faculty of the Information System department at Achva Academic College, and an adjunct faculty of the Computer Science department at Ben-Gurion University of the Negev. His research focuses on software engineering, with emphasis on modeling: model correctness and reasoning, modeling languages, and model patterns. Contact him at mari@cs.bgu.ac.il.

Eliran Nachmani is a Masters degree student in the Software and Information Systems Engineering department at Ben-Gurion University of the Negev. Contact him at nachamni@post.bgu.ac.il.

Arnon Sturm is on the faculty of the Software and Information Systems Engineering department at Ben-Gurion University of the Negev. His research interests focus around models for various purposes including software development ranges from end-user programming, database applications, to complex multi-agent systems, and knowledge representation and management. Much of his work has been on human aspects and the benefits of using models. Contact him at sturm@bgu.ac.il.

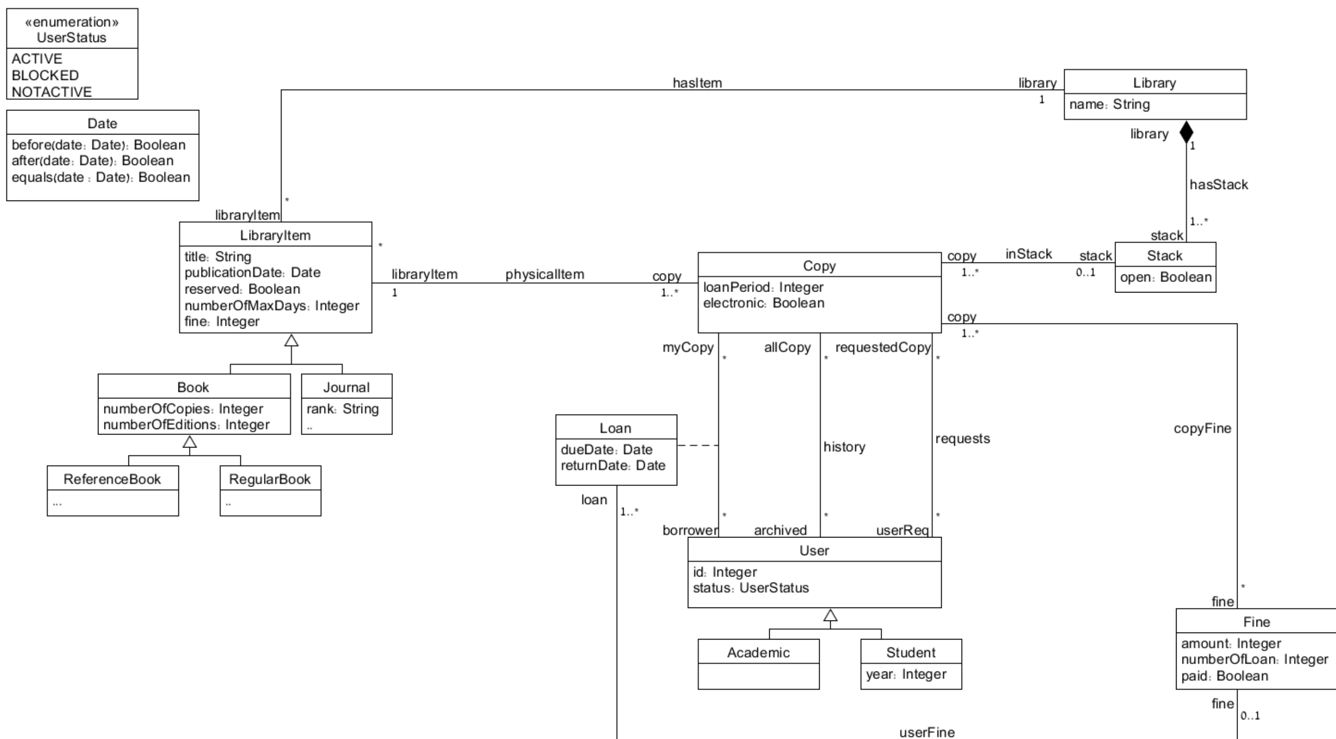


Figure 7 The Library System Class Diagram