

Who are we?

Octopus

Books

Papers/Reports

Research

Info on MDA

Info on OCL

What is OCL?

Current status

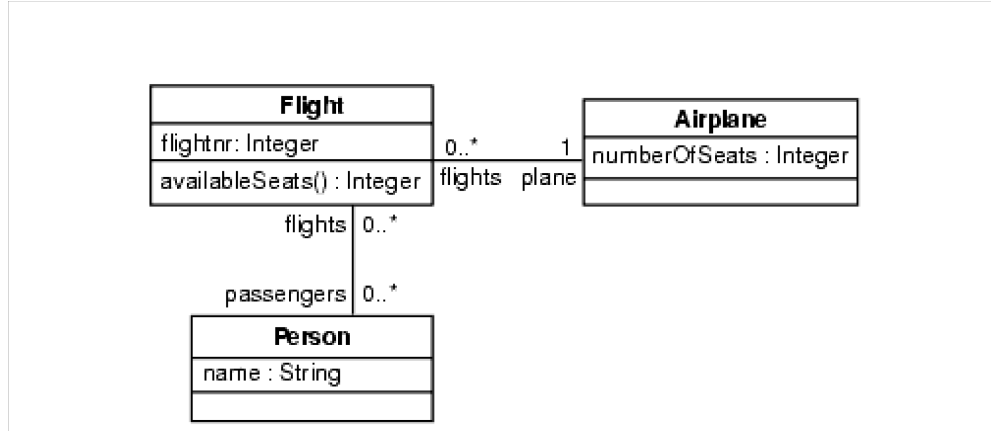
Tools

Info on UML

Downloads

Why Combine UML and OCL?

Modeling, especially software modeling, has traditionally been a synonym for producing diagrams. Most models consist of a number of "bubbles and arrows" pictures and some accompanying text. The information conveyed by such a model has a tendency to be incomplete, informal, imprecise, and sometimes even inconsistent. Many of the flaws in the model are caused by the limitations of the diagrams being used. A diagram simply cannot express the statements that should be part of a thorough specification. For instance, in the UML model shown in Figure 1, an association between class *Flight* and class *Person*, indicating that a certain group of persons are the passengers on a flight, will have multiplicity many (0..*) on the side of the *Person* class. This means that the number of passengers is unlimited. In reality, the number of passengers will be restricted to the number of seats on the airplane that is associated with the flight. It is impossible to express this restriction in the diagram.



In this example, the correct way to specify the multiplicity is to add to the diagram the following OCL constraint:

```

context Flight
inv: passengers->size() <= plane.numberOfSeats
  
```

Expressions written in a precise, mathematically based language like OCL offer a number of benefits over the use of diagrams to specify a (business or software) system. For example, these expressions cannot be interpreted differently by different people, e.g., an analyst and a programmer. They are unambiguous and make the model more precise and more detailed. These expressions can be checked by automated tools to ensure that they are correct and consistent with other elements of the model. Code generation becomes much more powerful. However, a model written in a language that uses an expression representation alone is often not easily understood. For example, while source code can be regarded as the ultimate model of the software, most people prefer a diagrammatic model in their encounters with the system. The good thing about "bubbles and arrows" pictures is that their intended meaning is easy to grasp.

The combination of UML and OCL offers the best of both worlds to the software developer. A large number of different diagrams, together with expressions written in OCL, can be used to specify models. Note that to obtain a complete model, both the diagrams and OCL expressions are necessary. Without OCL expressions, the model would be severely underspecified; without the UML diagrams, the OCL expressions would refer to non-existing model elements, as there is no way in OCL to specify classes and associations. Only when we combine the diagrams and the constraints can we completely specify the model.

Value Added by OCL

Still not convinced that using OCL adds value to the use of UML alone? The diagram in Figure 2 shows another example, which contains three classes: *Person*, *House*, and *Mortgage*, and their associations. Any human reader of the model will undoubtedly assume that a number of rules must apply to this model.

1. A person may have a mortgage on a house only if that house is owned by him- or herself; one cannot obtain a mortgage on the house of one's neighbor or friend.
2. The start date for any mortgage must be before the end date.
3. The social security number of all persons must be unique.
4. A new mortgage will be allowed only when the person's income is sufficient.
5. A new mortgage will be allowed only when the countervalue of the house is sufficient.

The diagram does not show this information; nor is there any way in which the diagrams might express these rules. If these rules are not documented, different readers might make different assumptions, which will lead to an incorrect understanding, and an incorrect implementation of the system. Writing these rules in English, as we have done above, isn't enough either. By definition, English text is ambiguous and very easy to interpret in different ways. The same problem of misunderstanding and incorrect implementation remains. Only by augmenting the model with the OCL expressions for these rules can a complete and precise description of the "mortgage system" be obtained. OCL is unambiguous and the rules cannot be misunderstood. The rules in OCL are as follows:

```

context Mortgage
inv: security.owner = borrower
  
```

context Mortgage

inv: startDate < endDate

context Person

inv: Person::allInstances()->isUnique(socSecNr)

context Person::getMortgage(sum : Money, security : House)

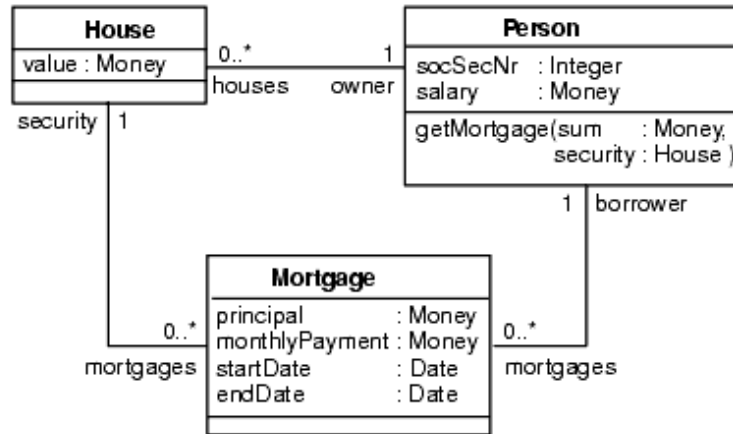
pre: self.mortgages.monthlyPayment->sum() <= self.salary * 0.30

context Person::getMortgage(sum : Money, security : House)

pre: security.value >= security.mortgages.principal->sum()

It is essential to include these rules as OCL expressions in the model for a number of reasons. As stated earlier, no misunderstanding occurs when humans read the model. Errors are therefore found in an early stage of development, when fixing a fault is relatively cheap. The intended meaning of the analyst who builds the model is clear to the programmers who will implement the model.

When the model is not read by humans, but instead is used as input to an automated system, the use of OCL becomes even more important. Tools can be used for generating simulations and tests, for checking consistency, for generating derived models in other languages using MDA transformations, for generating code, and so on. This type of work most people would gladly leave to a computer, if it would and could be done properly.



However, automating this work is only possible when the model itself contains all of the information needed. A computerized tool cannot interpret English rules. The rules written in OCL include all the necessary information for automated MDA tools. This way, implementation is faster and more efficient than by hand, and there is a guaranteed consistency between the model in UML/OCL and the generated artifacts. The level of maturity of the software development process as a whole is raised.

This page was last updated on December 20, 2005

© Copyright Jos Warmer and Anneke Kleppe