Chapter 1

MDA and the Use of OCL

This chapter explains why it is important to create models that contain as much information about the system as possible, especially when working within the Model Driven Architecture. Because the Model Driven Architecture itself is fairly new, a short introduction to this framework is given. Most important, this chapter states why OCL is a vital and necessary element in the Model Driven Architecture.

1.1 INTRODUCING OCL

The *Object Constraint Language* (OCL) is a modeling language with which you can build software models. It is defined as a standard "add-on" to the Unified Modeling Language (UML), the Object Management Group (OMG) standard for object-oriented analysis and design. Every expression written in OCL relies on the types (i.e., the classes, interfaces, and so on) that are defined in the UML diagrams. The use of OCL therefore includes the use of at least some aspects of UML.

Expressions written in OCL add vital information to object-oriented models and other object modeling artifacts. This information often cannot be expressed in a diagram. In UML 1.1, this information was thought to be limited to constraints, where a *constraint* is defined as a restriction on one or more values of (part of) an object-oriented model or system. In UML 2, the understanding is that far more additional information should be included in a model than constraints alone. Defining queries, referencing values, or stating conditions and business rules in a model are all accomplished by writing expressions. OCL is the standard language in which these expressions can be written in a clear and unambiguous manner.

Recently, a new version of OCL, version 2.0, has been formally defined in the *Object Constraint Language Specification* [OCL03] and as such it has been adopted by the OMG. This book explains this version of OCL and how it can be put to use in software development.

OCL.book Page 4 Tuesday, July 22, 2003 9:48 PM

There is a strong relationship between all OMG standards. The most recent OMG initiative is the Model Driven Architecture (MDA). The essence of the MDA approach is that models are the basis for software development. Therefore, models should be good, solid, consistent, and coherent. Using the combination of UML and OCL, you can build such models.

1.2 MODEL DRIVEN ARCHITECTURE

The Model Driven Architecture (MDA) is gradually becoming an important aspect of software development. Many tools are, or at least claim to be, MDA-compliant, but what exactly is MDA?

MDA is a framework being built under supervision of the Object Management Group (OMG). It defines how models defined in one language can be transformed into models in other languages. An example of a transformation is the generation of a database schema from a UML model, UML being the source language and SQL being the target language. Source code is also considered to be a model. Code generation from a UML model is therefore another form of transformation.

This section presents a brief introduction to MDA; it is by no means complete. You can find more information about MDA on the OMG Web site and in a number of books; for example, see *MDA Explained, The Model Driven Architecture: Practice and Promise,* by the same authors [Kleppe03].

1.2.1 PIMs and PSMs

Key to MDA is the importance of models in the software development process. Within MDA, the software development process is driven by the activity of modeling your software system. The MDA process is divided into three steps:

1. Build a model with a high level of abstraction, which is independent of any implementation technology. This is called a *Platform Independent Model* (PIM).
2. Transform the PIM into one or more models that are tailored to specify your system in terms of the implementation constructs that are available in one specific implementation technology; e.g., a database model or an EJB (Enterprise Java Beans) model. These models are called *Platform Specific Models* (PSMs).
3. Transform the PSMs to code.

Because a PSM fits its technology very closely, the last transformation is straightforward. The complex step is the one in which a PIM is transformed to a PSM. The relationships between PIM, PSM, source code, and the transformations between them, are depicted in Figure 1-1.

OCL.book Page 5 Tuesday, July 22, 2003 9:48 PM

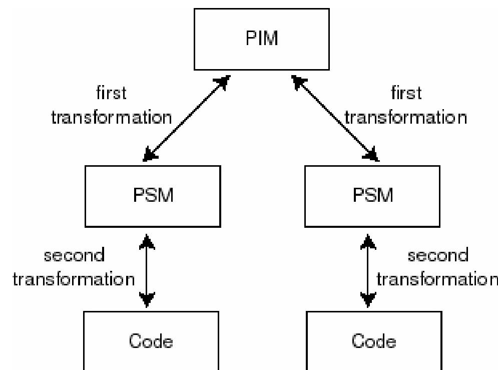

Figure 1-1 *The relationship between PIM, PSM, and code*

1.2.2 Automation of Transformations

Another key element of MDA is that the transformations are executed by tools. Many tools have been able to transform a platform-specific model to code; there is nothing new about that. What's new is that the transformation from PIM to PSM is automated as well. This is where the obvious benefits of MDA lie. Anyone who has been around a while in the business of software development knows how much time is spent on tasks that are more or less routine. For example, building a database model from an object-oriented design, or building a COM (Common Object Model) component model or an EJB component model from another high-level design. The MDA goal is to automate the cumbersome and laborious part of software development.

1.2.3 MDA Building Blocks

The MDA framework consists of a number of highly related parts. To understand the framework, you must understand both the individual parts and their mutual relationships. Therefore, let's take a closer look at each of the parts of the MDA framework: the models, the modeling languages, the transformation tools, and the transformation definitions, which are depicted in Figure 1-2.

Models

The first and foremost element of MDA is formed by models—high-level models (PIMs) and low-level models (PSMs). The whole idea of MDA is that a PIM can be transformed into more than one PSM, each suited for different target technologies. If the PIM were to reflect design decisions made with only one of the target technologies in mind, it could not be transformed into a PSM based on a different

target technology; the PIMs must truly be independent of any implementation technology.

A PSM, conversely, must closely reflect the concepts and constructs used in the corresponding technology. In a PSM targeted at databases, for instance, the table, column, and foreign key concepts should be clearly recognizable. The close relationship between the PSM and its technology ensures that the transformation to code will be efficient and effective.

All models, both PSM and PIM, should be consistent and precise, and contain as much information as possible about the system. This is where OCL can be helpful, because UML diagrams alone do not typically provide enough information.

Modeling Languages

Modeling languages form another element of the MDA framework. Because both PIMs and PSMs are transformed automatically, they should be written in a standard, well-defined modeling language that can be processed by automated tools. Nevertheless, the PIMs are written by hand. Before a system is built, only humans know what it must do. Therefore, PIMs must be written to be understood and corrected by other humans. This places high demands on the modeling language used for PIMs. It must be understood by both humans and machines.

The PSMs, however, will be generated, and the PSM needs to be understood only by automated tools and by experts in that specific technology. The demands on the languages used for specifying PSMs are relatively lower than those on the language for PIMs. Currently, a number of so-called profiles for UML define UML-like languages for specific technologies, e.g., the EJB profile [EJB01].

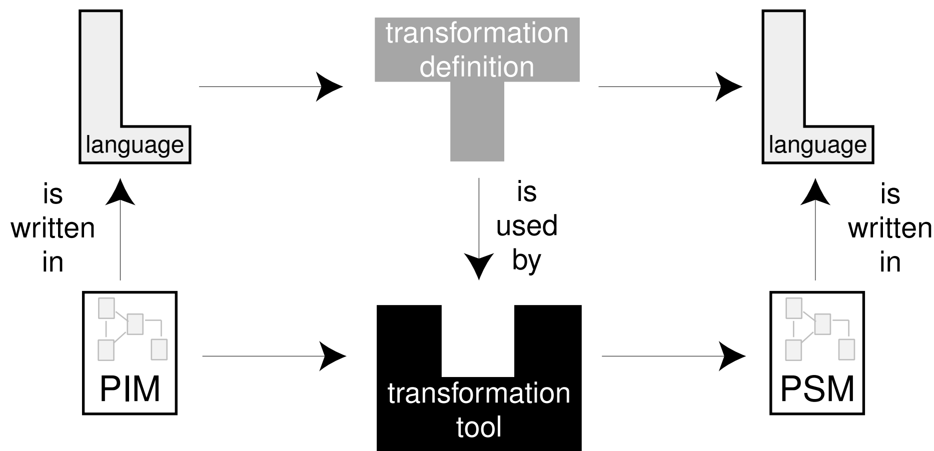

Figure 1-2 *The MDA Framework*

OCL.book Page 7 Tuesday, July 22, 2003 9:48 PM

Transformation Tools

There is a growing market of transformation tools. These tools implement the central part of the MDA approach, thus automating a substantial portion of the software development process. Many tools implement the PSM-to-code transformation. Currently, only a few implement the execution of the transformation definitions from PIM to PSM. Most of the PIM-to-PSM tools are combined with a PSM-to-code component. These tools should offer users the flexibility to tune the transformation to their specific needs.

Transformation Definitions

Another vital part of the MDA framework is formed by the definitions of how a PIM is to be transformed to a specific PSM, and how a PSM is to be transformed into code. Transformation definitions are separated from the tools that will execute them, in order to re-use them, even with different tools. It is not worthwhile to build a transformation definition for onetime use. It is far more effective when a transformation can be executed repeatedly on any PIM or PSM written in a specific language.

Some of the transformation definitions will be user-defined, that is, written by the developers that work according to the MDA process. Preferably, transformation definitions would be in the public domain, perhaps even standardized, and tuneable to the individual needs of its users. Some tool vendors have developed their own transformation definitions, which unfortunately usually cannot be adapted by users because their use is not transparent, but hidden in the functionality of the tools.

1.2.4 MDA Benefits

This section describes some of the advantages of MDA:

1. *Portability*, increasing application re-use and reducing the cost and complexity of application development and management, now and in the future. MDA brings us portability and platform independency because the PIM is indeed platform independent and can be used to generate several PSMs for different platforms.
2. *Productivity*, by enabling developers, designers, and system administrators to use languages and concepts they are comfortable with, while still supporting seamless communication and integration across the teams. A productivity gain can be achieved by using tools that fully automate the generation of code from a PSM, and even more when the generation of a PSM from a PIM is automated as well.
3. *Cross-platform interoperability*, using rigorous methods to guarantee that standards based on multiple implementation technologies all implement identical business functions. The promise of cross-platform interoperability can be ful-

OCL.book Page 8 Tuesday, July 22, 2003 9:48 PM

filled by tools that not only generate PSMs, but also the bridges between them, and possibly to other platforms as well.

4. *Easier maintenance and documentation*, as MDA implies that much of the information about the application must be incorporated in the PIM. It also implies that building a PIM takes less effort than writing code.

1.2.5 The Silver Bullet?

When explaining the MDA to software developers, we often get a skeptical response: "This can never work. You cannot generate a complete working program from a model. You will always need to adjust the code." Is MDA just promising another silver bullet?

We believe that MDA may change the future of software development radically. One argument for this is that although MDA is still in its infancy, you can today achieve great gains in productivity, portability, interoperability, and maintenance efforts by applying MDA using a good transformation tool. Therefore, MDA is, and will continue to be, used. A second argument comes from the history of computing.

In the early 1960s, our industry was in the middle of a revolution. The use of existing assembly languages was replaced with the use of procedural languages. There was a lot of skepticism in those days too, and not without reason. The first compilers were not very good. The Algol programming language, for instance, offered the potential to the programmer to give hints to the compiler about how to translate a piece of code. Many programmers were concerned that the generated assembler code would be far less efficient than handwriting the assembler code themselves. Many could not believe that compilers would become proficient enough to stop worrying about this.

To a certain extent the skeptics were right. You lost efficiency and speed, and you could not program all the assembler tricks in a procedural language. However, the advantages of procedural languages became increasingly obvious. Using higher level languages, you can write more complex software much faster, and the resulting code is much easier to maintain. At the same time, better compiler technology diminished the disadvantages. The generated assembler code became more efficient. Today, we accept the fact that we should not program our systems in assembler. Indeed, anyone planning to write a new customer relationship management system in assembler today would be declared insane.

What MDA brings us is another revolution of the same kind. Eventually, PIMs can be *compiled* (read: transformed) into PSMs, which are *compiled* into procedural language code (which itself is compiled into assembly or raw machine code). The PIM-to-PSM *compilers* (read: transformation tools) will not be very efficient for some years to come. Their users will need to provide hints about how to transform parts of a model. Eventually however, the advantage of working on a higher level of abstraction will become clear to everyone in the business.

OCL.book Page 9 Tuesday, July 22, 2003 9:48 PM

Concluding, we can say that although MDA is still in its infancy, it shows the potential to radically change the way we develop software. We are likely witnessing the birth of a paradigm shift; and in the near future, software development will shift its focus from code to models.

1.3 MODELING MATURITY LEVELS

Within the MDA process, the focus of software development will be on producing a high level model of the system. Currently, many people use UML or another modeling language during software development. However, they all use this standard language in very different ways. To create some order and transparency in working with models, we introduce the modeling maturity levels (MMLs). These levels can be compared to the CMM levels [CMM95]. They indicate what role models play in your software development process, and the direction you need to take to improve this process.

Traditionally, there has been a gap between the model and the system. The model is used as a plan, as brainstorm material, or as documentation, but the system is the real thing. Often, the detailed software code strays a long way from the original model. On every modeling maturity level, this gap is reduced, as shown in Figure 1-3, where the right side represents the actual system and the left side represents the model of the system.

As one climbs to a higher maturity level, the term *programming* gets a new meaning. At a higher maturity level, modeling and programming become almost the same. To be clear in the descriptions of each level, we use the word *coding* to mean the final transformation of all knowledge and decisions about the application—in whatever form that may be—to executable programming language code. Correspondingly, we speak of the *coder* instead of the *programmer*.

1.3.1 Level 0: No Specification

At the lowest level, the specification of the software is in the heads of the developers only. This level is common among nonprofessional software developers. One simply gets an idea about what to develop, and talks about it without ever writing anything down. The characteristics of this level are as follows:

- There are often conflicting views among developers, and between developers and users.
- This manner of working is suitable for small applications; larger and more complex applications need some form of design before coding.
- It is impossible to understand the code if the coders leave (and they always do).
- Many choices are made by the coders in an ad hoc fashion.

OCL.book Page 10 Tuesday, July 22, 2003 9:48 PM

1.3.2 Level 1: Textual

At modeling maturity level 1, the specification of the software is written down in one or more natural language documents. These may be more or less formal, using numbering for every requirement or every system function or not, large or

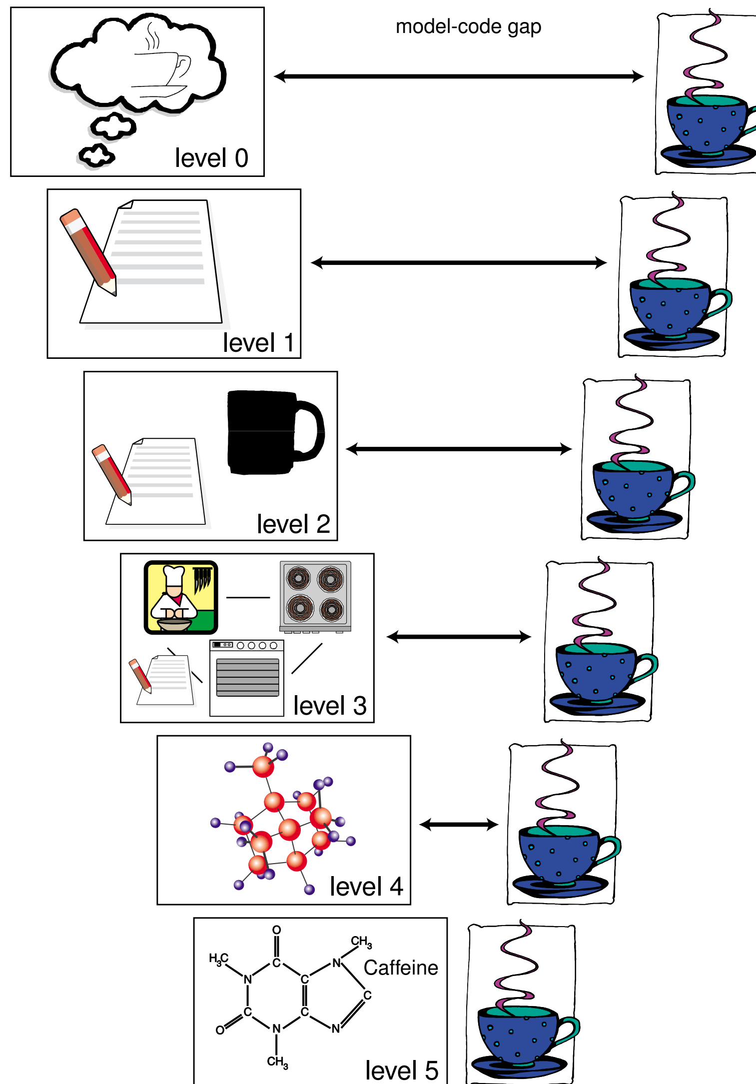

Figure 1-3 *The Modeling Maturity Levels bridging the model-code gap*

OCL.book Page 11 Tuesday, July 22, 2003 9:48 PM

small, an overview or very detailed, depending on taste. This is the lowest level of professional software development. The characteristics of this level are as follows:

- The specification is ambiguous, because natural language inherently is.
- The coder makes business decisions based on his or her personal interpretation of the text(s).
- It is impossible to keep the specification up to date after changing the code.

1.3.3 Level 2: Text with Diagrams

At modeling maturity level 2, the specification of the software is provided by one or more natural language documents augmented with several high-level diagrams to explain the overall architecture and/or some complex details. The characteristics of this level are as follows:

- The text still specifies the system, but it is easier to understand because of the diagrams.
- All characteristics of level 1 are still present.

1.3.4 Level 3: Models with Text

A set of models, i.e., either diagrams or text with a very specific and well-defined meaning, forms the specification of the software at modeling maturity level 3. Additional natural language text explains the background and motivation of the models, and fills in many details, but the models are the most important part of the design/analysis deliverables. The characteristics of this level are as follows:

- The diagrams or formal texts are real representations of the software.
- The transition of model to code is mostly manual.
- It is still impossible (or very difficult) to keep the specification up to date after changing the code.
- The coder still makes business decisions, but these have less influence on the architecture of the system.

1.3.5 Level 4: Precise Models

A model, meaning a consistent and coherent set of texts and/or diagrams with a very specific and well-defined meaning, specifies the software at modeling maturity level 4. Here, too, natural language text is used to add comments that explain the background and motivation of the model. The models at this level are precise enough to have a direct link with the actual code. However, they have a different level of abstraction. A model is more than the concepts of some programming language depicted in diagrams. Level 4 is the level at which the Model Driven Architecture is targeted. At this level:

- Coders do not make business decisions anymore.

OCL.book Page 12 Tuesday, July 22, 2003 9:48 PM

- Keeping models and code up to date is essential and easy.
- Iterative and incremental development are facilitated by the direct transformation from model to code.

1.3.6 Level 5: Models Only

A level 5 model is a complete, consistent, detailed, and precise description of the system. At level 5, the models are good enough to enable complete code-generation. No adjustments need to be made to the resulting code. Software developers can rely on the model-to-code generation in the same way coders today rely on their compilers. The generated code will be invisible to the developer; there is no need to look into it. The language in which the models are written has become the next-generation programming language. Certainly, some text is still present in the model, but its function is equal to comments in source code.

Note that this level has not been realized yet anywhere in the world. This is future technology, unfortunately. Still, it is good to recognize what our ultimate goal is.

1.4 BUILDING BETTER MODELS

In order to apply the MDA process, models on maturity level 4 are neccessary. This is the first level at which a model is more than just paper. At level 4, the models are precise enough to have a direct link with the source code. If you want to be able to transform a model from PIM through a PSM to code, this precision is necessary.

How do you build level 4 models? In Section 1.1, we already mentioned that the best choice for a modeling language for building PIMs is UML in combination with OCL. By specifying your model in a combination of the UML and OCL languages, you can improve the quality of your models.

1.4.1 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 1-4, 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..*)

OCL.book Page 13 Tuesday, July 22, 2003 9:48 PM

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 powerfull.

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.

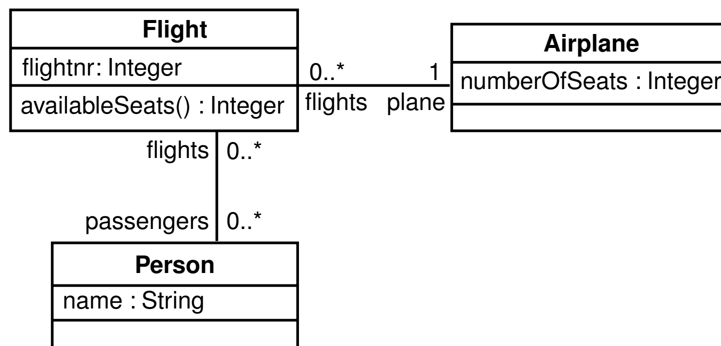

Figure 1-4 *A model expressed in a diagram*

1.4.2 Value Added by OCL

Still not convinced that using OCL adds value to the use of UML alone? The diagram in Figure 1-5 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 unambigous 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

OCL.book Page 15 Tuesday, July 22, 2003 9:48 PM

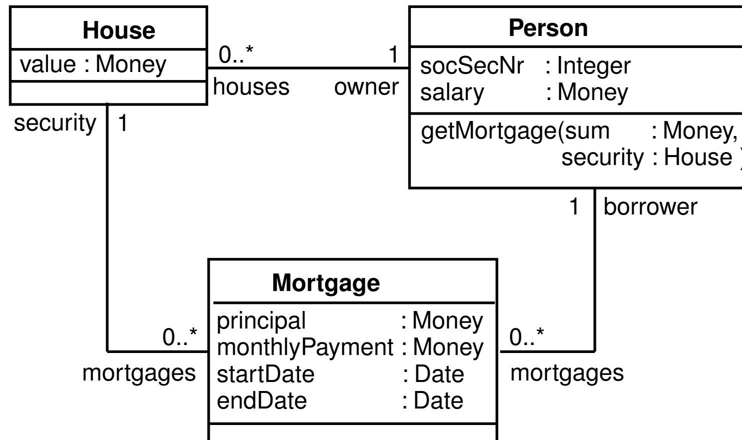

Figure 1-5 *The "mortgage system" expressed in a diagram*

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.

1.5 CHARACTERISTICS OF OCL

In the previous section, we saw that even a simple three-class model needs much additional information in OCL to make it complete, consistent, and unambiguous. If you had only the UML diagram, many open questions would remain. It is very likely that a system built based on the diagram alone will be incorrect. It is clear that OCL is a vital language for building better models. In the next section, we will take a closer look at some of the characteristics of OCL.

OCL.book Page 16 Tuesday, July 22, 2003 9:48 PM

1.5.1 Both Query and Constraint Language

In UML 1.1, OCL was a language to express constraints on the elements given in the diagrams of the model. In the introduction to this book, we already defined a constraint as follows:

> ***A constraint is a restriction on one or more values of (part of) an object-oriented model or system.***

What this means is that although the diagrams in the model state that certain objects or data values may be present in the modeled system, these values are valid only if the condition specified by the constraint holds. For example, the diagram in Figure 1-4 tells us that flights may have any number of passengers, but the constraint restricts the flights to those for which the number of passengers is equal to or less than the number of seats on the plane. Flights for which the condition does not hold are not valid in the specified system.

In UML 2, OCL can be used to write not only constraints, but any expression on the elements in the diagram. Every OCL expression indicates a value or object within the system. For example, the expression *1+3* is a valid OCL expression of type *Integer*, which represents the integer value 4. When the value of an expression is of *Boolean* type, it may be used as a constraint. Therefore, the possibilities that the language offers have grown considerably.

OCL expressions can be used anywhere in the model to indicate a value. A value can be a simple value, such as an integer, but it may also be a reference to an object, a collection of values, or a collection of references to objects. An OCL expression can represent, e.g., a boolean value used as a condition in a statechart, or a message in an interaction diagram. An OCL expression can be used to refer to a specific object in an interaction or object diagram. The next expression, for example, defines the body of the operation *availableSeats()* of the class *Flight*:

```
context Flight::availableSeats() : Integer
body: plane.numberOfSeats - passengers->size()
```

Other examples of using OCL expressions include the definition of the derivation of a derived attribute or association (explained in Section 3.3.8), or the specification of the initial value of attributes or associations (explained in Section 3.3.2).

Because an OCL expression can indicate any value or collection of values in a system, OCL has the same capabilities as SQL, as proved in [Akehurst01]. This is clearly illustrated by the fact that the body of a query operation can be completely specified by a single OCL expression. However, neither SQL nor OCL is a constraint language. OCL is a constraint and query language at the same time.

OCL.book Page 17 Tuesday, July 22, 2003 9:48 PM

1.5.2 Mathematical Foundation, But No Mathematical Symbols

An outstanding characteristic of OCL is its mathematical foundation. It is based on mathematical set theory and predicate logic, and it has a formal mathematical semantics [Richters01]. The notation, however, does not use mathematical symbols. Experience with formal or mathematical notations have led to the following conclusion: The people who can use the notation can express things precisely and unambiguously, but very few people can really understand such a notation. Although it seems a good candidate for a precise, unambiguous notation, a mathematical notation is not suitable for a standard language that is to be widely used.

A modeling language needs the rigor and precision of mathematics, but the ease of use of natural language. These are conflicting requirements, so finding the right balance is essential. In OCL, this balance is found by using the mathematical concepts without the abracadabra of the mathematical notation. Instead of using mathematical symbols, OCL uses plain ascii words that express the same concept.

Especially in the context of MDA, where a model needs to be transformed automatically, the value of an unambiguous mathematical foundation for the PIM language is of great value. There can be no doubt as to what an OCL expression means, and different tools must understand the expressions in the same way.

The result is a precise, unambiguous language that should be easily read and written by all practitioners of object technology and by their customers, i.e., people who are not mathematicians or computer scientists. If you still do not like the syntax of OCL, you may define your own. The OCL specification allows anyone to define his or her own syntax. The only condition is that your syntax can be mapped to the language structures defined in the standard [OCL03]. Appendix C provides an example of a different syntax expressing the same underlying structures. This syntax is directed toward use by business modelers.

1.5.3 Strongly Typed Language

An essential characteristic of OCL is that it is a typed language. OCL expressions are used for modeling and specification purposes. Because most models are not executed directly, most OCL expressions will be written while no executable version of the system exists. However, it must be possible to check an OCL expression without having to produce an executable version of the model. As a typed language, OCL expressions can be checked during modeling, before execution. Thus, errors in the model can be removed at an early stage.

Many popular programming languages are typed languages as well. Java, Eiffel, C#, delphi, and so on all fall into this category.

1.5.4 Declarative Language

Another distinguishing feature is that OCL is a declarative language. In procedural languages, like programming languages, expressions are descriptions of the actions that must be performed. In a declarative language, an expression simply states *what* should be done, but now *how*. To ensure this, OCL expressions have no side effects; that is, evaluating an OCL expression does not change the state of the system. Declarative languages have several advantages over procedural languages.

The modeler can make decisions at a high level of abstraction, without going into detail about how something should be calculated. For example, the body expression of the operation *availableSeats()* in Section 1.5.1 clearly specifies what the operation should calculate. How this should be done is not stated; this will depend on the implementation strategy of the whole system. One option is the representation of associations in the code. How do we find all of the passengers on a *Flight*? A flight object could contain a collection of references to its passengers. Alternatively, a *Flight* has no direct reference to its passengers, but needs to search a *Passenger* table in a database to find its passengers. A third implementation strategy would be to add an additional attribute to *Flight*, containing the number of passengers. Care should be taken to update the value of this attribute whenever a *Passenger* books or cancels a *Flight*. The body-expression in OCL allows for all of these implementations, because it describes only the what, not the how. If a procedural language were used to specify the body of *availableSeats()*, it would have forced a specific implementation style, which is undesirable for a PIM-level model.

In making OCL a declarative language, the expressions in a UML model are lifted fully into the realm of pure modeling, without regard for the nitty-gritty details of the implementation and the implementation language. An expression specifies values at a high level of abstraction, while remaining 100 percent precise.

1.6 SUMMARY

OCL is a language that can express additional and neccesary information about the models and other artifacts used in object-oriented modeling, and should be used in conjunction with UML diagrammatic models.

Much more information can be included in the specification (the model) using the combination of OCL and UML than through the use of UML alone. As you have seen, even in very simple examples, many essential aspects of the system cannot be expressed in a UML diagram. This information can only be expressed in the form of OCL expressions.

The level of maturity of the software process is raised by building a UML/OCL combined model. Tools that simulate a system, generate tests or source code from

OCL.book Page 19 Tuesday, July 22, 2003 9:48 PM

a model, and tools that support MDA need more detailed and more precise models as input. The quality of the output of these tools depends largely on the quality of the model used as input.

Tools that generate tests and source code from UML/OCL make the development process more efficient. The time invested in developing the UML/OCL models is regained during the subsequent stages of development.

OCL is a precise, unambiguous language that is easy for people who are not mathematicians or computer scientists to understand. It doesn't use any mathematical symbols, while maintaining mathematical rigor in its definition. OCL is a typed language, because it must be possible to check an OCL expression included in a specification without having to produce an executable version of the model.

OCL is a declarative, side-effects-free language; that is, the state of a system does not change because of an OCL expression. More importantly, a modeler can specify in OCL exactly what is meant, without restricting the implementation of the system that is being modeled. This enables a UML/OCL model to be completely platform-independent.