Integrated Design and Process Technology, **IDPT-2002**
Printed in the United States of America, June, 2002

USING METAOBJECT PROTOCOL TO IMPLEMENT TAILORING; POSSIBILITIES AND PROBLEMS

Olle Lindeberg, Jeanette Eriksson, Yvonne Dittrich
Blekinge Institute of Technology
Department of Software Engineering and Computer Science
P.O. Box 520, S-37225 Ronneby, Sweden
Fax: +46 457 27125
Phone: +46 457 385000
olle.lindeberg@bth.se, dst98jer@student.bth.se, yvonne.dittrich@bth.se

ABSTRACT

In this article we describe a prototype and how it was used to test if it is possible to use Java reflection API as a means of implementing tailoring. The tailoring capabilities of the prototype make the system configurable during runtime. The system that the prototype was modeled on is an application used by a telecommunication operator as a support system. In such a fast changing area software systems must also change. It is possible to anticipate the type and structure of some of the changing requirements and for them the prototype implements tailoring. Using the metaobject protocol idea, the modest reflection capabilities offered by Java together with a standard Java compiler and the normal Java runtime support are adequate to implement tailoring.

INTRODUCTION

This article is based on an experiment in using the Java reflection API (Sun Java 2, 2001) as a means of implementing a tailorable system. The background and idea behind the experiment was a research project in which we and two industrial partners collaborated. The goal of the project was to investigate a means of developing flexible, adaptable and modifiable software systems. The system that the prototype was modeled on is an application used by one of the research partners that is a telecommunication operator. It was possible to anticipate the type and structure of some of the changing requirements and for them tailoring (Henderson and Kyng, 1991) is a possible way to make the system modifiable. To read more about the project see 'Designing for changing work and business practices' (Dittrich and Lindeberg, 2002). The other partner had developed a meta-model database system. During the research project we developed several prototypes to test how to make a tailorable system using this database. At the same time a normal system development was carried out at the company. The resulting system has only limited tailoring capabilities. The need to make the software adaptable was instead satisfied by making the software easy to modify. This was achieved by making the software in components, which with only a little programming effort can be assembled in new configurations.

There were several reasons why the meta-modeling database tested in the prototypes were not used in the system. Here we will take up only one of these: the system seemed to become too complicated when tailoring was added to all other requirements. This is an example of a general problem; when you add tailoring capabilities to a system this often makes the system more complicated: not only do you have to construct the tailoring interface but the basic program may also become more complicated. To avoid this we constructed the prototype using ideas based on the metaobject protocol (MOP) approach (Kiczales, 1992). The prototype described here is a combination of the MOP approach and the components used in the system development mentioned earlier. This combination results in the meta-programming approach; this prototype is less complex than those implemented earlier on in the project

We start by giving a sketch of what a metaobject protocol is, and more particularly, what it is in Java. We then give a description of the software architecture of the complete system. The prototype implements only part of the system. Following the software architecture are the design and implementation of the prototype. Finally, some conclusions from the prototype are drawn.

THE METAOBJECT PROTOCOL

The metaobject protocol approach originates from the CLOS programming language in which it is possible to

change program behavior by interacting with the runtime system through a metaobject protocol (Kiczales et al., 1991).

The metaobject protocol is based on the idea that one can and must open up programming languages so that the developer is able to adjust the language implementation to fit his or her needs. This idea has subsequently been generalized to systems other than compilers and programming language. In the article "Towards a New Model of Abstraction in the Engineering of Software" (Kiczales, 1992) it is argued that the metaobject protocol concept can be used as a general principle for abstraction in computer science. The idea is that any system that is constructed as a service to be used of client application (as for example an operation system or a database server) should have two interfaces; a base-level interface and a meta-level interface (Kiczales, 1992). The base-level interface gives access to the functionality of the underlying system and through the meta-level interface it is possible to alter special aspects of the underlying implementation of the system so that it suits the needs of the client application. The meta-level interface is called the metaobject protocol (MOP). Simply put, a MOP is a set of rules by which to manipulate and communicate with metaobjects.

A MOP shall consequently:

- Provide extended control over the behavior of the system.
- Have a clear division between the base-level and meta-level interface.

TAILORING AND META MODELING

We have adopted a different approach towards the metaobject protocol. The idea of the metaobject protocol approach has inspired us to transfer the concept to end-user tailorable software. In most systems the end-user has no access to the implementation of the program; in our approach the end-user is given the opportunity to alter or tailor the software should the need arise. Our aim is to give the user the opportunity to add components to the program in a controlled way which does not require any programming. To do this we use a dual-interface: a traditional base-level program and a meta-level program that provides tailoring for the base-level program.

The distinction between a computational base level and a tailoring meta level is a useful one in a tailorable system. In the same way as in a metaobject protocol, the base-level implements what the system normally does. At the meta level you can change what the base level does. The two levels are also often separated in the user interface with a separate tailoring interface. The same separation may exist in the internal design.

Perhaps the obvious way to do this is to let the base-level program be controlled by meta-data which stores the choices the user has made when tailoring. If the tailoring possibilities affect a large part of the program, the base-level program may become littered with tests for the value of the meta-data. If the tailoring is complicated the result may be that the base-level program looks more like an interpreter of the meta-data than a straightforward program. We call this method of implementing tailoring the *meta-data approach*.

The alternative way to implement a tailorable system, the *meta-programming approach*, is closely linked to the metaobject protocol approach. With the meta-programming approach the base-level program is a normal program which performs the normal computation only. When the system is tailored by the meta-level this is implemented by changing the base-level program. To be able to do this we must be able to change (or at least add to) the program during execution. In Java this is possible since new class libraries may be loaded and linked during runtime. Another question is, "where is the meta-level description of the current configuration of the system stored?" In the meta-data approach the meta-level can inspect the meta-data to see how the program is configured;it is the meta-data that will be changed during tailoring. In the meta-programming approach the base-level does not need any meta-data. The radical solution is to take away the meta-data from the meta-level too. This means that it is the base-level program itself that is the meta description of the current configuration. This is the method we have chosen in the prototype.

We have used Java to implement the meta-programming approach. When tailoring changes the program this is implemented by compiling new class libraries (this is done by the compiler in JDK). The new class libraries are loaded and linked in during runtime. For inspection of the program we have used the - rather weak - reflection abilities in Java reflection API.

REFLECTION IN JAVA

Tailoring will change the program; the latter does not know in advance what the changes will look like. To discover what a new class contains, we need reflection capabilities. In a computational system reflection is the capability of an object to, for example, "reason about and act upon itself" (Maes, 1987).

There are two types of reflection: *introspection* and *intercession* (Rivard, 1996). The purpose of introspection is to acquire information about the program itself and to use that information within the program. Intercession goes further. It allows the program to alter its own behavior. Different programming languages have different reflection capabilities. Languages such as Lisp or Smalltalk have both introspection and intercession, while Java is basically introspective only. Java's meta-model is shown in Figure 1. In Java, every class has a meta description which is represented by an object; an instance of the class "Class." This object is a metaobject. While

ordinary objects describe the world, metaobjects describe the ordinary objects. In other words, the metaobject is an object that contains information about the ordinary object (base object) (Golm, 1997). The metaobject may control the execution of the base object (Zimmerman, 1996). The metaobjects together with the ordinary objects are part of a meta-model.

The reflection API in Java provides information about modifiers, methods, instance variables, constructors and the super classes of a particular class. It allows you to create an instance of a class although you do not know the name of the class until runtime. It is also possible to invoke a method on an object without knowing the name of the method during coding.

Accordingly, it is java.lang.reflect which makes it possible to inspect the content of the class (Sun Java 2, 2001). However, in Java 1.3, the Dynamic Proxy API was introduced making it possible to alter the behavior of an object in runtime. We have not used the proxy concept but instead alter the program by adding new classes using the compiler in JDK.

THE SYSTEM ARCHITECTURE

The prototype was produced to test the use of MOP in implementing tailoring. The prototype is a partial implementation of a system that will be described in this section. It is necessary to have some understanding of the whole system to understand the design of the prototype. The system is used for computing certain payments[1]; these payments are triggered by certain events. The receiver of money and how much should be paid are decided by what contract(s) are valid for the event.

The system architecture is described in Figure 2. The system can be regarded as two loosely connected parts: the transaction handler and the contract handler. The transaction handler application manages the actual payments and also produces reports while its database stores data about the triggering events, payments and historical data about past payments. (1)[2] The data describing the triggering events is periodically imported from another system. (2) To compute the payments, the transaction handler calls a stored procedure in the contract handler's database. (3) The event is matched with the contracts; several hits may occur. Some of the contracts cancel others; others are paid out in parallel. We call the process of deciding which contracts to pay 'prioritization'. (4) The result is returned to the transaction handler. (5) The actual payments are made by sending a file to the administrative system.

[1] To protect the business interests of our industrial partner we can only give an abstract description of the system: it is our opinion that this will affect the conclusions we draw.

[2] The numbers refer to figure 2.

An important complication in the data model is the categorization of the values on which some of the conditions are based. The categorization is dependent on other systems, making interaction with the latter essential both when the transaction handler matches events with contracts and when a user wants to use categories in a contract.

The contract handler administrates contracts, or rather formal descriptions of contracts, in a relational database. The interface enables the user to enter new contracts and search for old ones. When entering new contracts, the input is checked to ensure the integrity of the data. The main parts of the contracts are:

- Identification of the contract and version control.
- Some flags controlling who receives the money.
- Conditions determining if the contract is valid for an event or not.
- A payment table deciding the amount to be paid.

The first two parts are common to all contracts; it is the conditions and the payment table that differ.

In order to make the system adaptable to future changes a conceptual model that facilitates a meta-model description of the system is needed. The purpose of the system is to compute payments according to the stored contracts. Each event that triggers a payment has a set of parameters. Today there are only two kinds of events, though several other types of events are under consideration for the future. In the contracts a condition is meaningful only if the transaction handler can evaluate it when payment is due. This leads to the concept of event types: a payment is triggered by an event, and all contracts belong to a particular event type. Each event type has a set of attributes associated with it that limits what conditions a contract belonging to it can have. In the existing system there are a number of contract types that are used for different purposes. From the system's point of view these contract types differ in two significant ways: which conditions you can add to the contract and how the contracts influence each other (if several contracts match the same event one may inhibit the others, or all may be paid out).

In the case study already during the design discussions we constructed a conceptual model with four levels of abstraction (see Figure 3). The actual data that is stored describes the contracts the payments are based on. The contracts are of several *contract types* which form the base level of the abstraction hierarchy. Some contract types has nearly the same parameters but are used for different purpose in the use of the system; this gives the next level, *contract_group*s. At the top level of the abstraction hierarchy are the *event_types* where we group together contract and payments related to the particular event which triggers them.

In an object-oriented implementation the actual contracts would be objects belonging to the concrete

classes in the bottom line. The remainder of the classes would be abstract.

THE PROTOTYPE

The reason for producing the prototype construction was to investigate the feasibility of using Java's meta-programming possibilities to construct a tailorable system. This can be seen as an example of an explorative prototype (Floyd, 1984). We wanted to gain an understanding of the complexities related to this approach. The prototype does not implement the whole system but only the contract handler application. Functionality is reduced, especially the parameters using categorization of values are simplified to simple values, the primary reason for this being that it allowed us to build a prototype without any communication to other systems; in this way development of the prototype was greatly simplified.

The prototype is divided into two levels, the meta-level and the base-level. Two catalogues, one storing contract type and the other parameter classes implement the connection between the two levels. In the meta-level of the prototype, the new contract types are created and stored in the contract type catalogue. In the base-level the same classes are used as part of the program. The parameter class catalogue is used by the meta-level to know which parameters exist and by the base-level as part of the program.

Inheritance, together with the meta representation and the inner structure of the contract types, is essential to the prototype. A simplified model, similar to the conceptual model described in the section about the system architecture – but leaving out the group level – is the basis for the prototype implementation. It resulted in the class hierarchy presented in Figure 4. The events are super classes to the contract types. In the conceptual model an Event has a set of parameters and the contract type is made up of a subset of these parameters. This is not possible in Java; instead there is a specification that defines the set of parameters for the contract types in the Event classes. Some parameters are compulsory for all contract types belonging to an Event; they are put in the Event so that by inheritance they are present in all the contracts, e.g. all contracts must have a contract id.

One problem is that the contracts should be stored for a long time; all contracts ever entered into the system are kept to preserve its history and ensure that old payments are traceable. This is a problem when a change is made in a contract type as the system must still be able to store and display old contracts according to the old type. For this reason, all contract types from the beginning are defined as both an abstract class (e.g. ContractAB) and a concrete subclass (ContractAB_1). When a minor change is made to a contract type this may then be done by making a new concrete class, as ContractAB_2 in Figure 4.

The base-level of the prototype

A contract is essentially a collection of parameters. In the system in use some of the parameters are very complex and some even collect values from other systems. This makes it natural to represent every parameter by an object. Most of the methods in the contracts are implemented using delegation to the parameters. For the contracts' three main methods - checking, storing and displaying themselves - there are corresponding methods in the parameter classes. This is a vertical design where one class takes care of one type of parameter through the whole program instead of the more normal three-layer architecture (interface, logic and storing). This design makes it very easy to add new parameter classes to the system.

When the end-user wants to create a new contract, i.e. create an object from a contract type, all of the concrete classes are fetched from the contract type catalogue, their names are presented and the end-user chooses which contract type to create a contract from. Then a contract is created which has parameter objects without values. The object displays itself by delegating to the parameters. The same principle is used for storing and checking errors. When the user has put values in all slots and wants to store the contract, the error check is delegated to every parameter object. The parameter object checks that the value has the right format and is within the given limits. When a value is incorrect, the slot is marked and the user has to put in a new value. Not until all values are correct, are the values set in the empty contract. The primary problem with the delegation principle is that it is inadequate where parameters are in some way dependent on each other. It is possible for a parameter to access another parameter within the same contract by using a parent reference that all parameters have.

Following is a summary of how to create a new contract:

- The prototype collects the contract types from the contract type catalogue.
- The end-user selects a contract type to make a contract from.
- An empty contract is created from the contract type, the display method of the object is called and the parameters display themselves to the user.
- The user puts in values for the parameters.
- The prototype checks the values; when these are correct they are set in the empty contract. A contract is created.
- The contract is stored in a similar way.

The meta-level of the prototype

The contract types are created in the meta part of the program. When a user wants to create a new contract type all existing contract types are displayed. This is done by collecting all the class files from the contract type catalogue in which they are stored . The end-user chooses

what contract type he wants to have as super class for the new contract type. To make it easier for the user to make a decision as to what contract type is the most suitable, the parameters and the methods of the contract type are also displayed. Java reflection API provides the necessary methods for this.

The next step is to collect all possible parameters for the new contract type. To find the set of all possible parameters the program collect all classes in the catalog dedicated for parameter classes. All parameters may not be used for all Events. This is achieved by putting a filter in the class describing the Event. For example, if a parameter 'xyz' is not valid for Event type B a method that acts as a filter is placed in the abstract class EventB (Figure 4).

Thereafter all possible parameters for this Event type are shown to the end-user for him to select from. The parameters that are inherited are automatically selected and cannot be deselected. To find which parameters are already present in the selected contract type the program looks into the class of the contract type and its super classes with help from java.lang.reflect.

The meta-level of the program is constructed as a meta-model which is implemented as classes. The contract types correspond to objects of the class Metaobject. Our metaobject is in a way the same thing as the classobject in Java. We constructed our own version because a classobject cannot exist without a corresponding class. This means that it is not possible to create a new class from a classobject; as a result we could not use the classobject alone for our purposes. Another factor is that it is important to be able to handle the metamethods in a special way. The relationship between Java's meta-model and our extended meta-model is shown in Figure 6.

In the our extended meta-model a metaobject is an ordinary Java object, but it contains a description of a contract type and thus corresponds to a specific contract type. The MetaobjectClass is the class of metaobjects. The MetaobjectClass is a description of a general class. When the MetaobjectClass is instantiated the fields acquire values. The fields are references to metamethod objects, metafield objects and metaconstructor objects, e.g. the MetaobjectClass has a field of the Metafield type (the metamethods and the metaconstructors are excluded to simplify the example). The MetafieldClass has the field's *name* and *type*. When the MetaobjectClass is instantiated a metabject and a metafield object are created and the fields in Metafield acquire their values. The metaobject has a reference to the metafield object and the latter has a reference to a parameter. If the contract type is to have a parameter named *aCustomer*, the metaobject has a metafield object with an instance variable *name* with value "aCustomer" and an instance variable *type* with the value of"Customer". (Figure 5).

When the user has made his or her choices as to which parameters the contract type is to contain, the class ContractHandlerMOP creates the metaobject according to the input values. From the metaobject the source code for the new class is generated. The java source code is then compiled and a class file is produced. The file is stored in the contract type catalogue.

The ContractHandlerMOP is a class that handles the metaobject. All access to the metaobject goes via the ContractHandlerMOP. The class also restricts what can be done to the metaobject. This can be used to implement business logic controlling what contract types can be created.

Following is a summary of how to create a new contract type:

- The prototype collects the contract types from the contract type catalogue and displays the names of the contract types and their parameters and methods to the end-user.
- The end-user selects a contract type on which the new one is to be built.
- The contract type is inspected and the parameters of the contract type are displayed. All the parameter classes are collected from the parameter catalogue and filtered by the Event type; the result is displayed for the user.
- The user chooses the parameters for the new contract type.
- The program constructs a corresponding metaobject with its metafield, metamethod and metaconstructor objects.
- The metaobject is translated into Java source code.
- The Java source code is compiled and the resulting class file is stored in the contract type catalogue.
- The contract type can be used by the base-level of the prototype.

DISCUSSION

During the project three prototypes were implemented along with the system that is in operation today. The last prototype is the one that is described in this article. The other two prototypes were for the contract handler (with essentially the some functionality as in the prototype described) and for the "compute payment" function respectively. These two prototypes were constructed with the help of the meta-model database that was the starting point of the project. They are examples of the *meta-data approach* mentioned in the section "tailoring and meta modeling" above, for a description of these prototypes see (Lindeberg and Diestelkamp, 2001). There are parallels between using a meta-model database and the MOP prototype. In the meta-level of the program the meta DB structure and the object structure of the program are inspected respectively. The difference comes in the base-level part: the meta DB prototypes were both complicated

and slow since it had to inspect the database to establish the structure of a contract type; it also had to inspect the database to see how a parameter looked.

When we compare the earlier prototypes with the one described in this article we are convinced that the latter is less complex (it has taken less time to develop it). The interesting question is why this is the case and it is important to see if we can draw any general conclusions from this.

One of the reasons why meta-programming was so convenient in the example described here is that it is not the functionality of the program which is changed by the tailoring interface but the model of the data in the program. The base-level program has the same functionality in spite of the alterations. If the tailoring had aimed at extending functionality, for instance, with the aid of macro capabilities, the task would have been complicated in the meta-programming approach. An interesting question is if there is a complementary principle here: when tailoring changes functionality use meta-data approach and when tailoring changes the data model use the meta-programming approach. Our results seem to point in this direction.

Another advantage of the MOP prototype is the loose coupling between the meta and the base part of the program and between the contract types, the parameters and the base-level. This makes the base-level part simpler. By separating the meta- from the base-level we were able to use standard software, which means that at least the base-level is maintainable without any special competence in MOP.

Jet another advantage of the MOP approach is the opportunity it presents to handle unanticipated changes by hand-coding objects. There is always a limit to how far we can get with tailoring since the latter only takes care of anticipated types of changes and there will always be changes in the requirements which cannot be anticipated. In the MOP prototype, hand-coding contracts or new parameter classes can handle some such changes. This goes beyond normal tailoring activity and is part of the maintenance of the system. The advantage of the MOP approach described here is that it is easy to mix hand-coded and automatically constructed objects.

A new contract type can be coded by hand and put in the contract type catalogue. It will be used in the same way as contract types constructed within the program. Such a hand-coded contract type can be modified later by regular tailoring.

One example of this could be a contract type where two parameters depend on each other; if one parameter has a value the other must also have one. We can implement such an example by first using tailoring to let the system construct the contract type without any check between the parameters. Then a programmer can modify the code by adding the constraint between the parameters to the checking method in the contract type. Should we subsequently wish to make a small modification in the contract type, by adding a parameter, for example, this can be done using the normal tailoring interface.

In the same way it is possible to add new parameter objects by simply placing the compiled parameter class in the parameter catalogue. The parameter class is then ready to be used in the usual way by the program; no other code in the system needs to be changed but the new parameter class must obviously be hand-coded by a programmer. The new hand-coded contract type or parameter class must follow the pattern for how a contract type or a parameter class has to be structured. We believe this possibility of mixing hand-coded and automatically generated objects is a general advantage of the meta-programming approach.

One of the reasons that tailoring was not implemented in the real system development that was part of our research project was that the automatically generated user interfaces would not have been of an acceptable quality. This is a problem that occurs whenever tailoring is used to generate user interfaces. The meta-programming approach enables the user to alleviate the problem by making hand-coded interfaces for the contract types that are in the system right from the beginning so that they have good user interfaces. When new contract types are subsequently added by tailoring, less user-friendly interfaces will result; this may be acceptable, and in our case study it would have been an option since the alternative is to handle payments by hand.

CONCLUDING REMARKS

It has been interesting to try out the possibilities in Java for carrying out meta-programming. Our overall conclusion is that the metaobject possibilities available in Java are a convenient way for implementing tailoring in special-purpose applications.

A question we have only touched on in this paper is if it is worth the trouble to make an application tailorable as opposed to being merely “easy to change”. The answer to this question lies in the future: what types of requirement changes will arise? Would the prototype have been able to handle them? After all, the efforts to make software adaptable only pay off if they are used.

ACKNOWLEDGEMENTS

We thank all involved at Europolitan Vodafone AB and Diedata AB for a very interesting and rewarding project. The project was funded to 50% by the industrial partners (Europolitan Vodafon AB and DieData) and to 50% by KKS (The Knowledge Foundation).

REFERENCES

Dittrich, Yvonne & Lindeberg, Olle. 2002: Designing for changing work and business practices. To be published in: Nandish Patel (Ed.) Evolutionary and Adaptive Information Systems. USA, IDEA group publishing, to be published 2002.

Floyd, Christiane 1984: "A Systematic Look at Prototyping", in *Approaches to Prototyping*, Budde, R., Kuhlenkamp, K., Mathiassen, L., Zuellighoven H., eds., Springer-Verlag, Berlin, Heidelberg, New York, Tokio, (1984)

Golm, Michael. 1997: "Design and Implementation of a Meta Architecture for Java", *Diplomarbeit*, der Friedrich-Alexander-Universität, Erlangen-Nürnberg

Henderson, Austin & Kyng, Morten. 1991: "There's No Place Like Home: Continuing Design in Use", in *Design at Work*, GreenBaum, J & Kyng, M., eds., Lawrence Erlbaum, Hillsdale, NJ.

Kiczales, et.al. 1991: "The Art of the MetaObject Protocol", *MIT Press*, England.

Kiczales, Gregor 1992: "Towards a New Model of Abstraction in the Engineering of Software", in *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, Tama City, Tokyo, November 1992.

Kiczales et.al. 1993: "MetaObject Protocols, Why We Want Them and What Else They Can Do", in *Object Oriented Programming: The CLOS Perspective*, Paepcke, A., Massachusetts Institute of Technology, Cambridge.

Lindeberg, Olle & Diestelkamp, Wolfgang 2001: "How Much Adaptability do you Need? Evaluating Meta-modeling Techniques for Adaptable Special-purpose Systems", in *Proceedings of the Fifth Conference on Software Engineering and Applications*, Anaheim, USA, 2001.

Maes, Pattie. 1987: "Computational Reflection", *Technical Report 87_2*, Laboratory for Artificial Intelligence, Vrieje Univeriteit, Brussels, Belgium.

Rivard, Frederic.1996: "Smalltalk: a reflective language", *REFLECTION´96*, San Francisco.

Zimmerman, Chris. 1996: "Reflections on Adaptable Real-Time Metalevel Architecture", in *Journal of Parallel and Distributed Computing* 36, 81-89, 1996.

UNPRINTED SOURCES

Sun "Java™ 2 Platform, Standard Edition, v 1.3, API Specification" <http://java.sun.com/j2se/1.3/docs/api> 7 May 2001

FIGURES

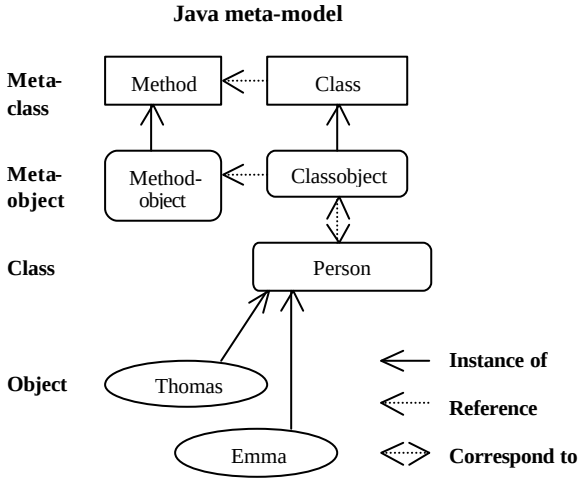

Figure 1 A part of the Java meta-model

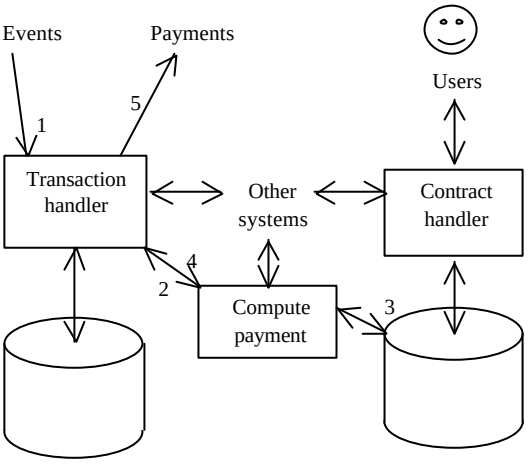

Figure 2 The system architecture

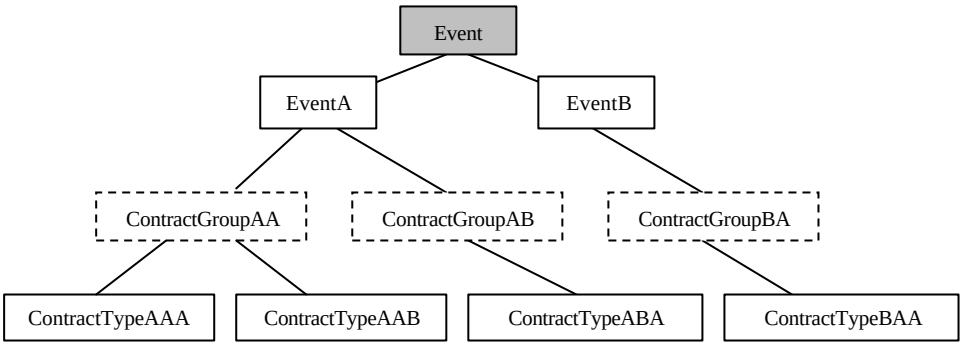

Figure 3 Type Hierarchy

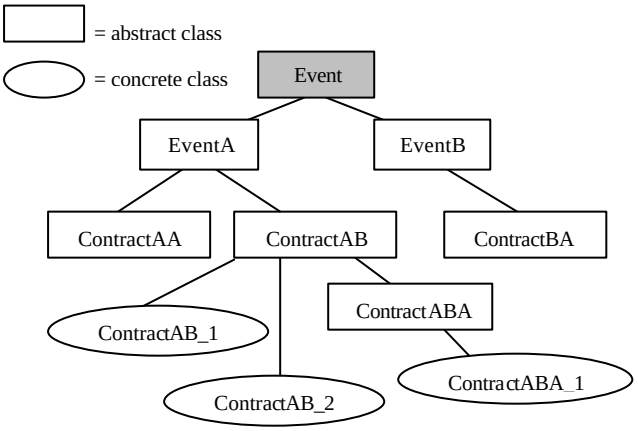

Figure 4 Inheritance hierarchy for the contract types

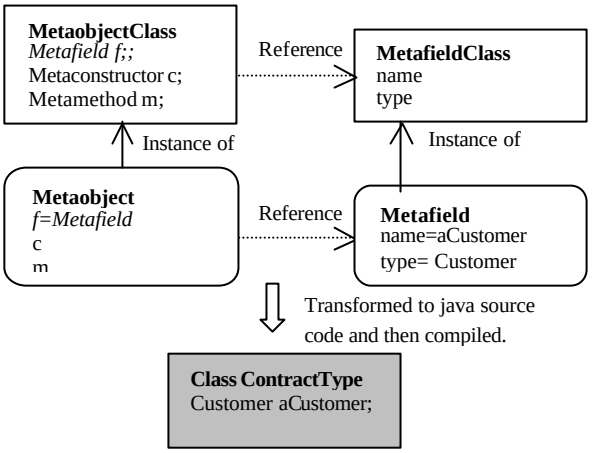

Figure 5 An example

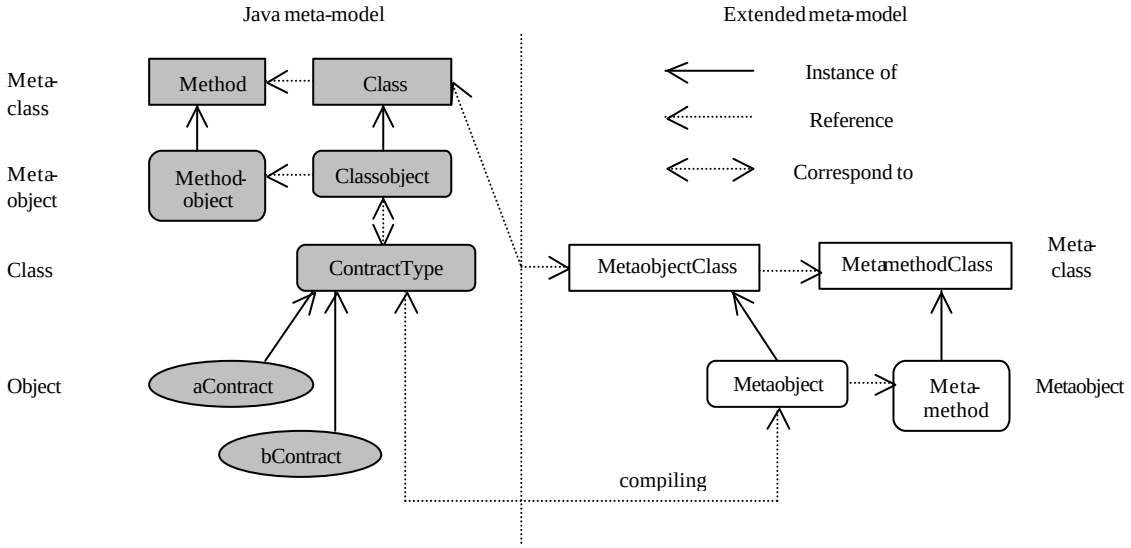

Figure 6 Meta representation