Shifting Up Reflection from the Implementation to the Analysis Level

Walter Cazzola, Andrea Sosio, and Francesco Tisato

DISCo - Department of Informatics, Systems, and Communication,
University of Milano Bicocca, Milano, Italy
{cazzola|sosio|tisato}@disco.unimib.it

Abstract Traditional methods for object-oriented analysis and modeling focus on the functional specification of software systems, i.e., application domain modeling. Non-functional requirements such as fault-tolerance, distribution, integration with legacy systems, and so on, have no clear collocation within the analysis process, since they are related to the architecture and workings of the system itself rather than the application domain. They are thus addressed in the system's design, based on the partitioning of the system's functionality into classes resulting from analysis. As a consequence, the *smooth transition from analysis to design* that is usually celebrated as one of the main advantages of the object-oriented paradigm does not actually hold for what concerns non-functional issues. A side effect is that functional and non-functional concerns tend to be mixed at the implementation level. We argue that the reflective approach whereby non-functional properties are ascribed to a meta-level of the software system may be extended "back to" analysis. Adopting a reflective approach in object-oriented analysis may support the precise specification of non-functional requirements in analysis and, if used in conjunction with a reflective approach to design, recover the smooth transition from analysis to design in the case of non-functional system's properties.

1 Introduction

Traditional methods for object-oriented analysis and modeling focus on the functional specification of software systems. The relevant concepts from the *application domain* are modeled using concepts (classes, object, operation, attributes, associations between classes, and so on) whose scope hardly includes non-functional requirements such as fault-tolerance, distribution, performance, persistence, security, and so on. These are not related to properties of the entities in the *real world*, but rather to *properties of the software objects that represent those entities*. Such non-functional requirements play a major role in the contract between customer and developer, and are usually included in analysis documents, maybe in the form of labels or *stereotypes* attached to analysis classes. Nevertheless, their treatment lacks a clear collocation in traditional object oriented processes. As a consequence, they tend to be less precisely specified

in analysis, and they do not benefit from the *smooth transition from analysis to design* which is usually regarded as one of the main advantages of the object oriented paradigm. As a consequence, non-functional requirements are only actually dealt with in the design phase. In traditional object-oriented methods, this phase refines the (functional) model produced during analysis. Since non-functional issues have no separate collocation in the decomposition produced by analysis, they tend to be addressed sparsely within the classes resulting from analysis and their refinements. This is not satisfactory if one considers that non-functional requirements are often largely (if not completely) orthogonal to functional ones. It also has the consequence, that code related to non-functional issues is often intertwined with "functional" code, which makes it hard to later modify or adjust the non-functional properties of a system based on changed requirements, versions, and so on.

The main purpose of this paper is that of illustrating in some details the problem described above and proposing a solution. We argue that a *reflective* approach is well suited to address this problem. Reflection is pivoted on the idea of decomposing a system into a base-level and one or more meta-levels that perform computation *on the computation of the lower levels*. Thus, the domain of the meta-level is the base-level, just like the domain of the base-level is the application domain itself. Non-functional requirements, which relate to the workings of the system rather than the application domain, lend themselves to be described as *functional requirements of the meta-level*. In traditional literature on reflection, this approach is only adopted as a *design* or *implementation* solution. We believe that the same ideas can be usefully "shifted up" to the analysis stage. Just as the meta-level may be programmed in the same paradigm (and even language) of the base-level, traditional OO concepts, notations, meta-models, methods, and methodologies used for conventional OO analysis can be employed in the analysis of non-functional properties once these are recast as computation on computation. Finally, as a last advantage, if one also adopts a *reflective approach to design*, the smooth transition from analysis to design is recovered.

Although reflection has gained increasing attention within the last decade, and is now recognized as a relevant and useful feature in OO languages and programming systems, there is still a lack of research efforts devoted to the definition of an OO *process* for reflective systems. As this paper proposes an integrated reflective approach to analysis and design, it can also be regarded as an attempt to shed some light on what such a process might look like.

The outline of the paper is as follows. Section 2 discusses the problems encountered in traditional object oriented analysis for what concerns the treatment of non-functional requirements. Section 3 lists some major concepts from the discipline of (OO) reflection, which provide a basis for solving those problems. Section 4 illustrates our proposal of a *reflective object-oriented analysis*, discussing how non-functional requirements fit into such an approach to OO analysis. Section 5 briefly points at related works in the discipline of reflection, and section 6 draws some conclusions.

2 Non-Functional Requirements and Traditional Object-Oriented Analysis

The fundamental step in traditional OO analysis is related to the modeling of the *application domain*, and results in the definition of the collection (hierarchy) of classes corresponding to the relevant concepts in the domain and of the relationships (associations) between those classes. Following a traditional, consolidated style of software engineering, the authors of those methods insist that implementation details (*how*) should be (as systematically as possible) ignored in analysis (which relates to *what* the system does). Once classes for the relevant entities in the domain are found, as well as their general behavior and relationships, the design begins based on the classes found in the analysis, that are refined and progressively enriched with details about *how* they are going to meet their specification. In this process, new classes may be added (sometimes referred to as *architectural* as opposed to *application* or *business* classes) whose purpose is that of providing a concrete infrastructure for the analysis classes to work. One of the major benefits coming from object-orientation is the smooth transition from analysis to design; the classes that are found in the analysis phase often preserve their original interfaces and overall interrelationships when they are refined into more concrete design classes, although several details are usually added.

This general idea is of course valuable and could hardly be criticized *per se*. Nevertheless, we believe that there is a missing piece, namely, the treatment of non-functional requirements has no clear collocation in the process outlined above. While it is sensible to postpone the treatment of *how issues* such as the choice of algorithms and data structures, the same does not hold for many non-functional requirements. Issues related to fault-tolerance, distribution, performance, persistence, integration with legacy or pre-existing systems, networking, authentication and security, and so on, may be as relevant to the customer as functional ones. (Also relevant may be non-functional requirements on the process, such as limitations to budget or time to market, development resources, or the need to reuse COTS components, although we will not discuss this topic in this paper). These requirements are not captured in analysis if this is conceived as domain modeling alone, since non-functional requirements tend to be expressible only in terms of properties of the *system* rather than real-world entities. Note that many traditional methods explicitly prescribe that, after a first description of the system, the analyst should prune all those elements of the model that are extraneous to the real world [23]; this prescription often leads to suppressing non-functional requirements in analysis. Some object-oriented processes have also been proposed where analysis explicitly addresses some major system-related issues; an example is the *inception* phase of the Rational Object Process, where the overall architecture of a system is defined. Nevertheless, while object-oriented concepts obviously apply well to the description of a system's functionality, we lack a similar vocabulary of concepts for most non-functional issues (i.e., issues about the system's workings). In other words, the problem with non-functional requirements is that they are not easily captured by tra-

ditional OO concepts as employed by traditional OO modeling notations and meta-models.

To illustrate the problem, we shall consider a classical banking system example (of course narrowing our discussion to a very small subset of requirements). We shall refer to UML [3] as a modeling notation, and to a typical object-oriented process employing this notation. (Of course, several different processes have been proposed in the literature; nevertheless, they are all similar with respect to how they deal with non-functional requirements).

The first stage of the process is requirement analysis, which in UML is done with use cases describing the actors interacting with the system and typical uses of the system. For a banking system, for example, the basic actor is the customer, and use cases include obtaining an account and deposits/withdrawals. Use cases represent the primary contract between the software purchaser and developer. They are not related to the system's structure, and do not include detailed information on the application domain, although they are intended to drive, to a great extent, the subsequent stages of the process.

Fig. 1 describes some simple use cases for a banking system.

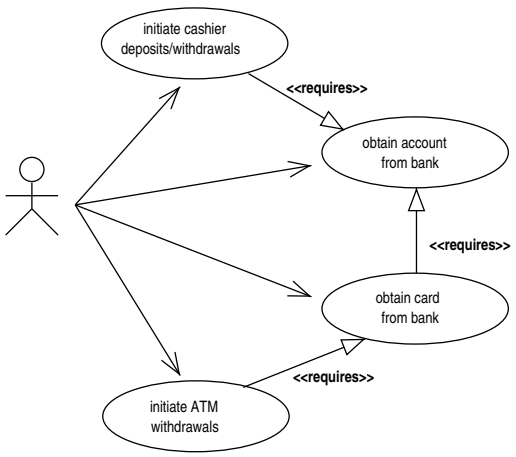

Figure 1. Bank use case.

Requirement analysis is followed by domain analysis (sometimes simply referred to as *analysis*), where the basic real-world entities involved in the system are modeled within one or more classes and related diagrams. This activity is also referred to as *domain modeling*. Fig. 2 describes a class diagram for the banking system.

The classes provided in this diagram should provide the functionality described by use cases, although use-cases are at a higher abstraction level, i.e., a single use case may involve a complex interaction pattern involving a possibly large collection of objects. Sequence diagrams and other dynamic diagrams can

aid in describing the bundle of interactions that correspond to a use case. In this case, obtaining an account is achieved via the `create_account` operation of class `Bank`, and interactions with an ATM are modeled by the operations to prepare, cancel, or commit a transaction (we do not consider cashier transactions).

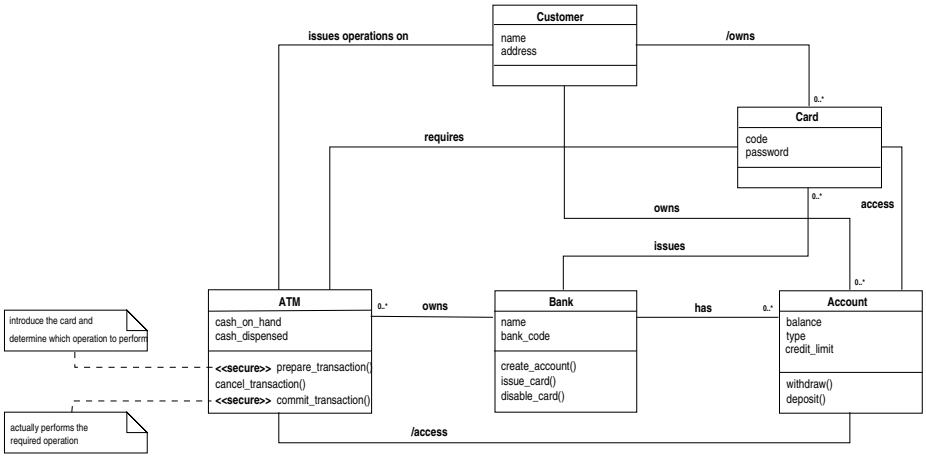

Figure 2. Bank class diagram.

In domain modeling, non-functional issues have no clear collocation. Consider the problem of specifying that operations on the ATM should be secure. While saying that money may be drawn from a bank account may result, say, in the definition of a `withdraw` operation in class `Account`, saying that an operation is secure has no obvious counterpart in traditional object oriented concepts. Nonetheless, this obviously *is* a requirement (and a fundamental one), so that it should clearly mentioned *before* design. Current practice in UML modeling typically approaches this problem using free text specifications or stereotypes. Stereotypes are adopted in UML as a mechanism to extend the expressive power of the basic notation with new semantic concepts. In Fig. 2, the operations `prepare_transaction` and `commit_transaction` of the ATM are marked as «`secure`». Unfortunately, unless the concept of security is elsewhere formally specified, a «`secure`» stereotype is hardly more than an informal, and vague, reminder for the designer/implementor. This non-precise specification of (perhaps essential) system features is of course a non ideal situation, since, for example, it reduces the effectiveness of requirements' analysis as a contract between customer and developer. Note that the primary role ascribed to use cases in UML, and the fact that use cases do not capture non-functional issues, may contribute to this situation.

Related problems occur in the *transition from analysis to design*. Functional requirements are relatively easily translated into design elements (concrete classes, methods, attributes); this is usually regarded as one of the major

(if not the main) benefit of the object-oriented paradigm itself. The classes that were found during analysis are usually kept through to the design stage. They are possibly enriched with new operations and attributes, and their workings may be augmented by "helper" or "ancillary" classes. Non-functional issues, of course, are not tackled as easily. Of course, there is no unique and easy way to map stereotypes (which, by definition, could have any meaning) or free text notes into design. In some cases stereotypes are used to represent non-functional properties which are well-known and have a *standard* design and implementation counterpart. This is the case of the «`persistent`» stereotype. In a typical design, a `Persistent_Object` class is introduced, providing operations to store an object, retrieve it from file, and so on. All classes that were labeled with the «`persistent`» stereotype are then connected to the `Persistent_Object` via inheritance, so that their instances are now persistent objects, among other things. Nevertheless, not all non-functional issues are as standard and well-known. The fact that non-functional properties are tackled *based on the functional partitioning that resulted from analysis* may lead to bad design especially for those non-functional requirements that are not naturally related to any specific object, but rather require a system-wide infrastructure.

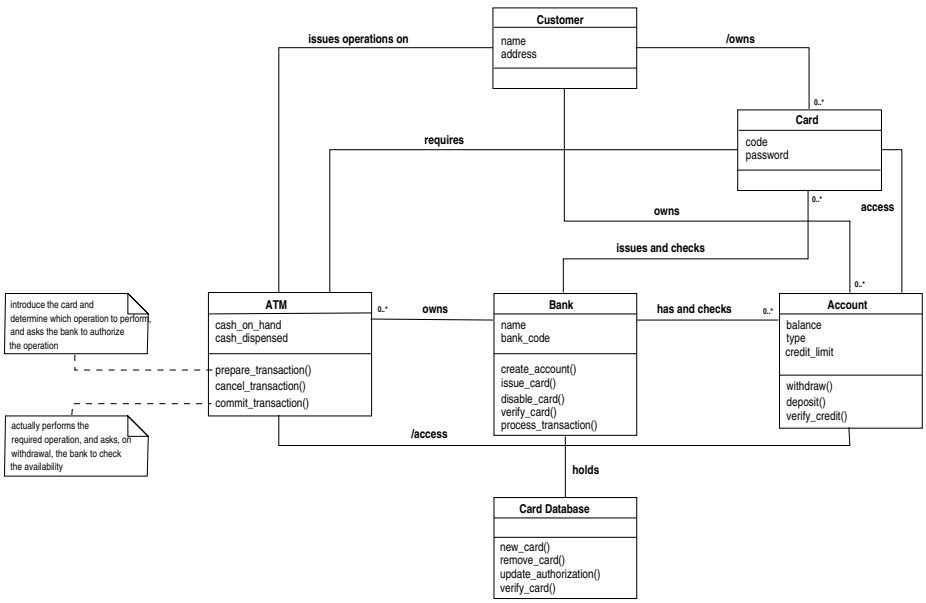

Figure 3. Refined bank class diagram.

A typical outcome of this state of facts is that code related to non-functional issues is often intertwined with *functional* code in the implemented system, thus reducing modifiability and reusability. Consider the case of ATM transactions.

Functionally, an ATM transaction is just a movement of money. Nevertheless, it requires a complex non-functional infrastructure including concurrency control, fault-tolerance support, authentication, and possibly more. In a traditional OO process, the designer may receive, as an outcome of analysis, a class ATM providing one or more operations labeled by stereotypes such as «`atomic`», «`secure`», «`reliable`», and so on. The designer will probably cope with these additional properties refining the ATM operations into a very complex activities, including security checking and so on, and perhaps described by a state diagram with dozens of states and transitions. The resulting implementation is necessarily one where the basic semantics of `withdraw` is obscured and dispersed in the midst of a plethora of additional code that has little to do with the movement of money *per se*, thus making the ATM object harder to reuse and modify. The effects of this problem often span multiple classes. Fig. 3 describes an arrangement for dealing with security. An additional `Card Database` class has been added, managing a database of existing cards and their passwords. Whenever the bank issues a new card, it should notify this to the `Card Database` (via `update_authorization`); the same occurs, conversely, when a card expires. Also, each transaction should involve a call to an operation of `Card Database` (say, `verify_card`) to check whether the authorization may be granted. In the example, the ATM invokes the `verify_card` of `Bank`, which in turn will include a call to the `Card Database` to check the card against the authentication information held in the database itself.

Changes in the analysis documents (e.g., if the customer asks for a higher level of security, e.g., security in network communication between ATMs and banks) provide no hint as to how the system design and implementation should be changed (e.g., changes cannot be traced easily from analysis onto design and implementation). The lack of traceability of non-functional issues from analysis to design is a relevant problem as it is often the case that non-functional properties of a system should be tuned for different versions, products in product families, ports, and so on.

As a final note, we would like to note that many of the considerations above also apply for issues which are not usually considered as non-functional (the boundary between functional and non-functional is of course quite blurred). For example, the problem of blocking withdrawals from an account when the credit limit is reached could be cleanly dealt with in much the same way as authentication, as we shall see below.

3 Object Oriented Reflection

3.1 Basic Concepts

Computational reflection (or *reflection* for short) is defined as the activity performed by an agent when doing computations about itself [19]. The concept applies quite naturally to the OOP paradigm [9,11,19]. Just as objects in conventional OOP are representations of *real world* entities, they can themselves be

represented by other objects, usually referred to as *meta-objects*, whose computation is intended to observe and modify their *referents* (the objects they represent). Meta-computation is often performed by meta-objects by *trapping* the normal computation of their referents; in other words, an action of the referent is trapped by the meta-object, which performs a meta-computation either substituting or encapsulating the referent's actions. Of course, meta-objects themselves can be represented, i.e., they may be the referents of meta-meta-objects, and so on. A reflective system is thus structured in multiple levels, constituting a *reflective tower*. The objects in the base level are termed *base-objects* and perform computation on the entities of the application domain. The objects in the other levels (termed *meta-levels*) perform computation on the objects residing in the lower levels.

There is no need for the association between base-objects and meta-objects to be 1-to-1: several meta-objects may share a single referent, and a single meta-object may have multiple referents. The interface between adjacent levels in the reflective tower is usually termed a *meta-object protocol* (MOP) [16]. Albeit several distinct reflection models have been proposed in the literature (e.g., where meta-objects are coincident with classes, or instances of a special class `MetaObject`, and so on), such a distinction is not relevant for this discussion and will be omitted.

In all reflective models and MOPs, an essential concept is that of *reification*. In order to compute on the lower levels' computation, each level maintains a set of data structures representing (or, in reflection parlance, a *reification of*) such computation. Of course, the aspects of the lower levels' system that are reified depend on the reflective model (e.g., structure, state and behavior, communication). In any case, the data structure comprising a reification are *causally connected* to the aspect(s) of the system being reified; that is, any change to those aspects reflects in the reification, and vice versa. It is a duty of the reflective framework to preserve the causal connection link between the levels (depending on the reflective model, this infrastructure may operate at compile- or at run-time): the designers and programmers of meta-objects are insulated from the details of how causal connection is achieved. Meta-objects can be programmed in exactly the same programming paradigm as conventional computation. It is in fact possible, and most usual, that all levels of the reflective tower be programmed in the same *programming language*. The fact that all the levels of the tower be implemented in a single language qualifies, for some authors, as one of the characterizing features of reflection proper [11].

Another key feature of all reflective models is that of *transparency* [24]. In the context of reflection, this term is used to indicate that the objects in each level are completely *unaware* of the presence and workings of those in the levels above. In other words, each meta-level is added to the base-level without modifying the referent level itself. The virtual machine of the reflective language, in other words, enforces causal connection between a meta-level and its referent level in a way that is transparent *both* to the programmer of the meta-level and to the programmer of the referent level.

3.2 Reflection and non-functional properties

An application of reflection, supported by the feature of *transparency*, is the (non-intrusive) *evolution* of a system: the behavior or structure of the objects in a system can be modified, enriched, and/or substituted without modifying the original system's code. In principle, this may have interesting applications to the evolution of non-stopping systems or systems that are only available in black-box form. Depending on the specific support provided by the reflective language virtual machine, the evolution of a system through the addition of a meta-level may require recompilation or maybe done dynamically.

Another well-known application, which is the one that will be considered in this paper, is that of adopting a reflective approach to separate functional and (possibly several distinct) non-functional features in the *design* of a system (the issue of analysis will be considered in a later section). In a typical approach, the base-level objects may be entrusted to meet the application's functional requirements, while meta-levels augment the base-level functionality ensuring non-functional properties (e.g., fault tolerance, persistence, distribution, and so on). With reference to this partitioning of a system, in the following we will sometimes refer to the base-level objects as *functional objects* and to meta-level objects as *non-functional objects*. While functional objects model entities in the real world (such as `Account`), non-functional objects model properties of functional objects (to reflect this, non-functional classes may have names that correspond to properties, e.g., `Fault_Tolerant_Object`).

There are several reasons why a design could benefit from such an approach. Of course, separation of concerns (in general, hence also in this case) enhances the system's modifiability. Depending on whether a required modification of the system involves functional or non-functional properties, functional objects alone or non-functional objects alone may be modified. If the collection of data comprising an account changes, for example, the (functional) class `Account` will be modified; if, say, a higher level of fault-tolerance is required, the (non-functional) class `Fault_Tolerant_Object` will be changed.

This approach also enhances reusability in two ways: first, the very same functional object (e.g., a `Account` object) can be reused with or without the additional properties implemented by its associated meta-objects, depending on context. Any additional feature of an object (e.g., fault-tolerance, the capability to migrate across platforms, persistence, and so on) has an associated overhead. In a reflective approach, all such features are not hardwired into the code of the object itself but implemented by separated meta-objects; whenever the additional features are not required, the corresponding meta-objects are simply not instantiated. Note that there is a difference between the reflective approach to persistence and that mentioned in the previous section. In the reflective approach a functional object becomes persistent because there is a meta-object that *transparently* modifies its behavior (e.g., intercepts its constructor/destructor and complements them as to load/store an image from/to file). On the other hand, in the non-reflective approach the non-functional object inherits methods related to persistency which appear in *its own* interface, with the same status of

methods such as `withdraw` or `deposit`. These methods must be invoked (probably by the object itself) if the object is to be made persistent. The result is that, even if inheritance may promise some form of separation of concerns, there will necessarily be some intertwining of functional and non-functional code. For example, the reverse operation (giving up the persistency of the `Account` by suppressing the inheritance relation to `Persistent_Object`) is not usually possible. As a second form of reuse, many non-functional properties lend themselves to be implemented in a way that is essentially independent of the specific class of objects for which they are implemented. As an example, support for persistence is usually independent of the specific type of object being made persistent, as demonstrated by the adoption of `persistent` classes in `Java` and other mainstream OO programming languages. In our opinion, this is likely to hold for several typical non-functional properties; some examples are provided by the works on reflective approaches to fault-tolerance [1,10], persistence [17], atomicity [26], and authentication [2,22,25]. Based on this fact, it is reasonable to expect that the same meta-object can be reused to attach the same non-functional property to different functional objects.

In this paper, we are interested in considering whether the idea of computation about computation can be a convenient point of view from which to tackle the analysis of a system's non-functional properties. As stated above, such properties can usually be described as properties of the objects comprising the system rather than real-world entities. It thus seems reasonable to expect that, just as we model properties of entities in the real-world by conventional objects, we could model properties of objects themselves by meta-objects. In a reflective system designed along the lines described above, non-functional properties of the system are actually treated as functionality of the meta-level, whose domain is the software system. This homogeneity of design would yield several benefits if *shifted up* to analysis. This is the subject of the following section.

4 Shifting Up Reflection to Analysis

4.1 General Concepts

Two main points from the considerations of the previous section can be highlighted:

❶ the property of *transparency* of reflection allows for functional and non-functional concerns to be clearly separated in the design of a system, being respectively entrusted to the base-level and to the meta-levels;
❷ the concept of *reification*, and the transparent application of causal connection by the virtual machine of a reflective programming language, allows for meta-levels to be programmed in the same paradigm as the base-level.

Based on these two points we envisioned a novel approach to the treatment of non-functional properties in OO analysis. As we discussed in section 2, non-functional properties have no clear collocation in traditional OO analysis because

they have no counterpart in the vocabulary of OO concepts. Fault-tolerance cannot be represented as a class, an object, an operation, an attribute, an association between classes, and so on. Nevertheless, in a reflective approach, those non-functional properties are represented by meta-objects. Meta-objects are objects themselves, and lend themselves to be described in OO terms. The transition to the *meta*, in a sense, transforms something that is *about* an object (a property of an object) *into an object itself*; it *reifies* a property. As a consequence of this transformation, object properties themselves are absorbed into the scope of notations and meta-models for OO modeling and hence become natural subjects for OO analysis. This can be further illustrated by the following parallelism. Just like the concept of reification allows for meta-levels (that implement *computation on computation*) to be programmed in the same language as used for the base-level (that implements *computation on the domain*), so it allows for the computation performed by the meta-level to be analyzed and modeled using the same concepts and techniques that are used to analyze and model the computation in the base level. When applied to base-level objects, these concepts are used to described properties of the real-world entities that those objects model (e.g., operations model the dynamics of real-world objects, such as drawing money from a bank account). When applied to meta-level objects, the same concepts model properties of the *software objects that represent real-world entities within the system* (e.g., operations model the dynamics of software objects, such as their ability to be saved onto, or restored from, files). This leads us to the main idea presented in this paper.

4.2 Reflective Object-Oriented Analysis

We argue that the considerations made insofar suggest a novel approach to OO analysis, that we shall term *reflective object-oriented analysis* (ROOA). In ROOA, the requirements of the system are partitioned, during the analysis phase, into concepts related to the domain and concepts related to the software system operating in that domain (i.e., into functional and non-functional). The concepts related to the software system may, themselves, be partitioned according to the properties they deal with (fault-tolerance, persistence, distribution, reliability, and so on). Observe that this partitioning is orthogonal to the traditional partitioning of the functional requirements of a system, namely that guided by OO concepts applied to the application domain. It is then natural to speak of additional "levels" of specification, which complement the traditional (functional) level. Note that we are not proposing a *method*; it is not our assumption, as for now, that this partitioning into levels be a step that must be taken *before* or *after* traditional analysis. Our perception is simply that such a partitioning should *complement* the traditional functional partitioning as assumed, explicitly or implicitly, by current OO methods. Let us consider the banking system again. The ROOA approach may affect the very first stage of the process, i.e., the definition of use cases. Non-functional or reflective use cases appear whenever there is some requirement which involves an actor performing

an operation that relates to the system rather than the application domain. An example is a user configuring or reconfiguring the system, or adding an ATM.

ROOA anyway mostly affects class diagrams. These are also separated according to the principle stated above (analysis of the domain vs. analysis of the system). In doing this, we rely on the concept of meta-objects, although with that we simply refer to an object that *models a property of another object*. While we obviously refer to the same concept as found in reflection, please note that in this discipline the term is used in a design/implementation context and not an analysis context. Meta-objects as introduced in this paper may well be implemented with conventional (non-reflective) means. A meta-object provides a model of a software entity, just like an object is a model of a real-world entity. Of course, just as conventional objects only model some aspects of real-world entities (those that are relevant to the system's functionality), in the same way ROOA meta-objects model just some specific aspect of software entities (e.g., persistence, fault-tolerance, and so on). As for traditional use cases and class diagrams, non-functional use cases should be made to correspond to interactions of the external actors with meta-objects. As in reflection, we shall consider meta-objects as comprising a different level of the system, which is largely independent of the base-level where conventional objects reside. We also allow for several independently specified meta-levels to coexist, each describing some particular aspect of the functional objects' properties.

Each system of this partitioning can be analyzed using standard OO analysis techniques, and specified within a traditional OO meta-model (e.g., the UML meta-model). All the concepts from the standard OO vocabulary can be used when modeling meta-levels (class, inheritance, association, attribute, operation, and so on), albeit, as mentioned in the previous subsection, these levels are about properties of the system rather than the domain. Of course, the workings and semantics of the link that binds the meta-level(s) in the analysis diagrams to the analysis base should be defined, i.e., the problem should be solved of how the UML meta-model (or a similar model) can be enriched with a concept corresponding, in abstract terms, to the causal connection relationship between base- and meta-level objects in reflective designs and systems. Given the variety of reflective models, this topic is by itself worth a thorough research. In order to illustrate more of our approach and its consequences, in the next section we shall make a (somewhat simplistic) proposal about how this can done.

4.3 Causal Connection in Analysis

As suggested above, we model non-functional properties in terms of computation performed by meta-objects *on the computation of functional ones*. In general, non-functional objects may perform computation on the *state* and/or the *behavior* of base-level objects. In the following, we will adopt a simple convention, and use the same name for a base-level object and the meta-object that models its non-functional properties. In describing how meta-objects may interact with base-level objects, we will of course rely on the results from reflection, although

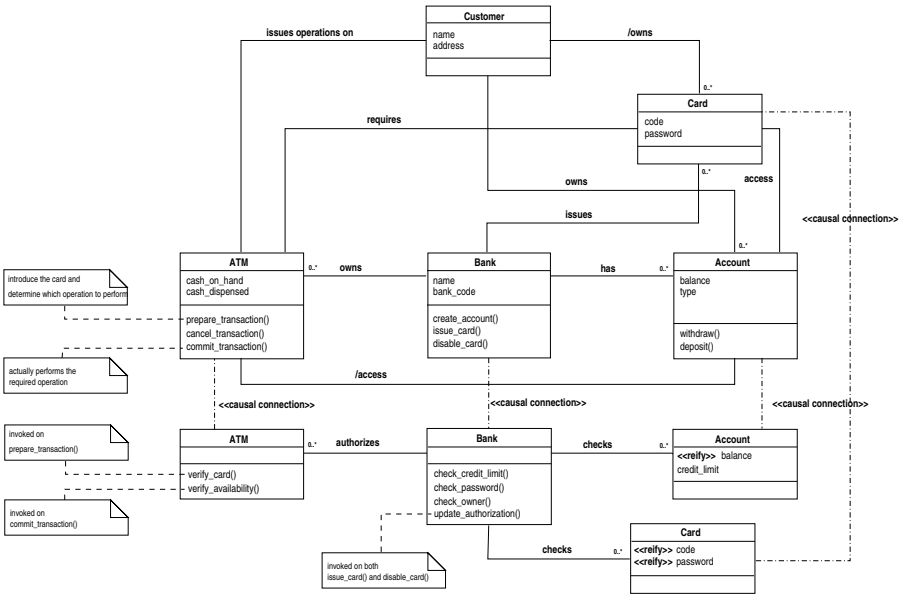

Figure 4. Reflective bank class diagram.

let us note again that we are not necessarily considering a reflective implementation.

State is of course represented by attributes of objects. Meta-objects hence should be entrusted to observe and modify the attributes of (base-level) objects. Thus, the meta-model should be extended to include the possibility of specifying which attributes of an object are observed by the meta-object. For the purpose of this paper, we shall rely on name-matching, i.e., in the examples that follow the meta-object's attribute `x` will be intended to be an image of the corresponding attribute `x` in the referent. Also, we shall mark the reification (i.e., the copy of the attribute held by the meta-object) with stereotype «`reify`». Of course, a more flexible mechanism should be useful in practice, although we won't cover the syntactical problem of how to describe correspondences between attributes in different levels.

Behavior in object systems is usually expressed in terms of invocations of operations (methods). Meta-objects provide operations that integrate and augment those of their functional counterparts by wrapping (i.e., either enriching or substituting) them. In our examples, we shall provide an external specification of mappings between methods using text notes. These compensating operations are assumed to be invoked whenever the referent's operations are invoked, and will usually perform some activity before and/or after invoking the referent's method. That outline above is in fact the general strategy adopted in current reflective languages to augment the base-level computation. Note that there are open problems in the discipline of reflection about the workings of both state and

behavior observation and modification (e.g., the fact that reflection breaks encapsulation, and the problem of multiple meta-levels trapping a single base-level method call). We shall not consider these problems here and let the reader refer to solutions and discussions found in the reflection literature (e.g., [8,13,21]).

Given these premises, consider the statement that ATM transactions are secure and how this would be expressed in ROOA (Fig. 4). This represents the same classes as found in Fig. 3 (for functional objects), complemented by classes for meta-objects performing computation on the functional objects. We use a dotted and dashed line to represent the association between the classes of functional objects and those of the meta-objects (which may be introduced in UML as a «`causal connection`» stereotyped association), and adopt the convention of using the same name for functional classes and their corresponding meta-objects classes. Meta-objects are charged with managing security. A meta-object is introduced for each `ATM`, `Account`, `Card`, and for each `Bank`.

Whenever the `Bank` issues a new card, or an old card expires, this operation is trapped by the `Bank` meta-object, which adds the new card onto (removes the expired card from) its database of cards, together with the associated password. Next, whenever a transaction is prepared in the `ATM` (i.e., the card is inserted and the password introduced), the `ATM`'s meta-object intercepts this operation and checks the card with the `Bank` meta-object. The overall functioning is close to that depicted in Fig. 3 (i.e., the `Bank` class of the meta-level works essentially in the same way as the `Card Database`), except that a reflective approach keeps the two levels clearly separated and avoids intermixing functional and non-functional concerns in the base-level. Note that we can also delegate to the meta-level issues that may be regarded as functional, for example managing the credit limit for accounts. In the example, operation `commit_transaction` is trapped by the `ATM`'s meta-object which executes an operation of its own (`verify_availability`) to check whether the requested operation can be performed, again with the `Bank` meta-object. Since this check is influenced by the current balance and the credit limit of the account, this information also has to be reified at the meta-level (this is why an `Account` meta-object is needed). Note that, if management of credit is done at the meta-level, the credit limit itself becomes a meta-level information (there is no need for a `credit_limit` attribute at the base-level).

Of course, it is up to the analyst to decide what should be modeled in meta-level terms, evaluating pros and cons (just as successful mechanisms such as exception handling are used in different ways by different designers). Even if ROOA is strictly adopted for non-functional issues, it is rather obvious that different analysts and designers should disagree as to what is *functional*. It suffices, here, to point out at the effects: in the system of Fig. 4, it is very likely that many operations at the base-level have a straightforward implementation that directly corresponds to their abstract view as could be found in requirements as produced by the customer (e.g., a withdrawal removes money, and nothing else).

4.4 After ROOA: Design and Implementation

ROOA, as described above, is an *analysis* technique and does not include prescriptions about design and implementation. *Per se*, the approach should yield a

more precise specification of non-functional issues. After such requirements have been analyzed and modeled, it is up to the designer/implementor to decide to what extent should a reflective approach be followed in the subsequent stages of development. As mentioned in section 2, one of the drawbacks of the conventional approaches to analysis is a lack of separation of concerns, in the implementation, between functional and non-functional issues. To solve this problem, a reflective approach could be applied in design as well, that is, the system could actually be *designed* following a decomposition into levels in the reflective sense (i.e., be designed as to have a reflective architecture). If this is done, ROOA provides the additional benefit that *a smooth transition from analysis to design for non-functional requirements* is achieved. Meta-objects are implemented as instances of independent classes and are designed to rule over, and modify, the behavior and state of functional objects. Carrying a reflective approach to design allows the developer to take full advantage of ROOA, and ideally solves all the problems we mentioned in section 2 for what concerns the tackling of non-functional issues in system development.

If the system is designed as to have a reflective architecture, there is a further choice to be taken, namely whether a reflective language should or should not be used for the actual *implementation*. Although many popular languages (first of all Java) include some weak form of reflection, it does not seem reasonable to expect reflective languages to be widely adopted in (industry) development. Nevertheless, of course, the ROOA approach (complemented with reflective design) does not require that a reflective language be used. If a reflective language is used, the link between the non-functional and functional levels is implicitly provided by causal connection. The specific form of reflection we relied upon in this paper (reflection on state and behavior) was explicitly chosen because it is fully supported by many reflective languages, including OpenC++ [7], and OpenJava [29]. On the other hand, if a reflective language is *not* used in the implementation, support for causal connection should be explicitly implemented as a part of the system. It seems reasonable to suggest that the system be first designed assuming causal connection, and then the final design be augmented with a further stage where the causal connection mechanism itself is designed. In this case, the infrastructure for causal connection could be tailored to the specific needs of the application at hand.

Note that the analysis documents are of course the same irrespective of whether the design is reflective and irrespective of the target language, so that they could even be reused for different implementations, some using reflection explicitly, and others using a mainstream (non reflective) language.

5 Related Work

Several authors within the Reflection field have considered the application of reflective techniques to address non-functional software requirements. Hürsch and Videira-Lopes [14] highlight the relevance of an approach that separates multiple concerns (including functionality as one specific concern, as opposed to other

non-functional concerns) both at the conceptual and implementation level. They provide a tentative classification of the concerns that may be separated in general software systems, and encompass the major techniques that may be used to separate them, namely *meta-programming*, *pattern-oriented programming*, and *composition filters*. Their discussion is somewhat less specific than that provided by this paper, as their concept of separation of concerns is not necessarily achieved via the use of reflective techniques (i.e., meta-programming). Most of other related efforts propose *design approaches* (rather than analysis approaches) for structuring a software system in such a way that non-functional requirements are addressed by a system's meta-level(s) and thus cleanly separated from functional (base level) code. As we basically aim at supporting a smooth transition from analysis to design via reflection, reflective design approaches to the enforcement of non-functional properties provide us with some hints as to what the *result* of this process (that is, the resulting design) should look like. Stroud and Wu [27] discuss a reflective approach to the dynamic adaptation of nonfunctional properties of software systems. In their approach, security, persistence and replication properties are transparently added to an existing system (even available in black-box form) and easily tailored to any specific environment onto which the system is downloaded. Their paper explicitly tackles the issue of separating functional and non-functional requirements and reusing meta-objects implementing non-functional properties. Several authors addressed the transparent addition of fault-tolerance features to a software system via reflection, e.g. [1] (that applies channel reification to communication fault-tolerance) [10] (that employs reflective N-version programming and both active and passive replication mechanisms) [20] (that employs reflective server replication) and [15] (that employs reflective checkpointing in concurrent object-oriented systems). As mentioned above, other non-functional issues that were demonstrated to be effectively tackled via reflection include persistence [17], atomicity [26], and authentication [2,22,25].

Also related to the topic of this paper is our work on Architectural Reflection (AR) [4,5,6]. In AR, a reflective approach is adopted for reifying the *software architecture* of a software system. While the definition of the architecture of a system is usually regarded as belonging to (early) design, there are cases where *requirements* on the architecture of a system should be considered from the outset (i.e., the need for integration with legacy systems, the need to reuse COTS components, and so on). Architectural properties of both the whole system and of single objects may be addressed in a reflective approach like that suggested in this paper. Other authors have considered addressing architectural properties of objects using a meta-level; for example, [12] proposes a reflective object-oriented pattern for the separated definition of an object's behavior, and [18] proposes the R-Rio system for system's dynamic reconfiguration through a reflective description of the system's software architecture. Also strictly related to our work is [28], that proposes to use a reflective approach in analysis (within the context of component-based software development), although it is clearly the intention of the authors that reflection be kept as a basic mechanism through design to implementation.

6 Conclusions and Future Work

This paper is intended to suggest how traditional OO analysis could be extended in order to cope with non-functional requirements in a cleaner way than supported by current methods. It suggests that a reflective approach could be taken, whereby a system's specification is partitioned into levels (i.e., in way that is orthogonal to a *functional* partitioning), where the base level includes information on the domain, and the other levels include information on the system. This partitioning into levels could then be mapped easily onto a reflective architecture where requirements related to the system are refined into meta-objects that augment base-objects with non-functional properties. We propose the general lines of a modified object-oriented analysis methodology (which can be applied in any context, e.g., using UML and a method such as Objectory[3]) where non-functional issues are dealt with in a reflective fashion.

While the approach can be used independent of the adoption of reflective principles in *design* (or implementation), we also believe that this paper also provides some useful suggestion for the design of reflective systems themselves. To the best of our knowledge, few efforts have been made to propose extensions or adaptations of OO methods, methodologies, and processes to OO reflective systems. In our view, the best way to design a reflective system is that of considering it in a reflective perspective from the outset, i.e., from analysis. This means that the analysis phase should include a partitioning of the system's requirements into levels as that proposed here.

Of course, this is just a "vision" paper. We plan to continue this work by considering the issues raised in this paper in more detail. In particular, we would like to progress in at least two directions. First, we will study in greater detail how the ROOA may fit with mainstream OO notations, meta-models, methods, and methodologies (starting from the UML meta-model and related processes and methods). As ROOA necessarily leads to a definition of the overall structure of the *system* from the first stages of analysis, it would probably fit best with methods that include an early definition of architecture, such as the already mentioned Rational process. We believe that a very interesting side effect of this study would be that of clarifying, in a sense, the essence of (OO) reflection. In other words, while traditional OO concepts have been deeply understood and clearly formalized in notations such as UML, OO reflective concepts, in our view, are usually perceived as belonging to a lower abstraction level than basic OO concepts such as inheritance, associations, and so on. We believe that reflective concepts will tend to become ever more ubiquitous in object-oriented development, and that a precise understanding of their meaning, at the highest abstraction level possible, is strongly needed.

As a second line of research, we would like to investigate how different reflective models apply within ROOA. We plan to investigate how the different models apply to the specification of traditional non-functional requirements. We believe that this will hardly lead to identifying the *best* model. Rather, it is likely that non-functional issues can be classified according to what reflective model is required to express them most clearly (i.e., no single model applies in all cases).

Acknowledgements

This work has been supported by DISCo (Department of Informatics, Systems and Communication), University of Milano Bicocca.

References

1. Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: a Reflective Approach to Fault-Tolerant Software Development. In *OOPSLA'95 (poster section)*, page 137, Austin, Texas, USA, on 15th-19th October 1995. ACM. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.
2. Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. Reflective Authorization Systems: Possibilities, Benefits and Drawbacks. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, Lecture Notes in Computer Science 1603, pages 35–51. Springer-Verlag, July 1999.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachussetts 01867, 3 edition, February 1999.
4. Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *Proceedings of 6th Reengineering Forum (REF'98)*, pages 12–1–12–6, Firenze, Italia, on 8th-11th March 1998. IEEE.
5. Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Architectural Reflection: Concepts, Design, and Evaluation. Technical Report RI-DSI 234-99, DSI, University degli Studi di Milano, May 1999. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.
6. Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. In *Proceedings of 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pages 263–266, Cocoa Beach, Florida, USA, on 12th-15th October 1999.
7. Shigeru Chiba. A Meta-Object Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *Sigplan Notices*, pages 285–299, Austin, Texas, USA, October 1995. ACM.
8. Scott Danforth and Ira R. Forman. Reflections on Metaclass Programming in SOM. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94)*, volume 29 of *Sigplan Notice*, pages 440–452, Portland, Oregon, USA, October 1994. ACM.
9. François-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, Montréal, Canada, August 1995.
10. Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou, Robert J. Stroud, and Zhixue Wu. Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming. In *Proceedings of FTCS-25 "Silver Jubilee"*, Pasadena, CA USA, June 1995. IEEE.

11. Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of 4th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.
12. Luciane Lamour Ferreira and Cecília M. F. Rubira. The Reflective State Pattern. In Steve Berczuk and Joe Yoder, editors, *Proceedings of the Pattern Languages of Program Design*, TR #WUCS-98-25, Monticello, Illinois - USA, August 1998.
13. Nicolas Graube. Metaclass Compatibility. In Norman K. Meyrowitz, editor, *Proceedings of the 4th Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89)*, volume 24(10) of *Sigplan Notices*, pages 305–316, New Orleans, Louisiana, USA, October 1989. ACM.
14. Walter Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.
15. Mangesh Kasbekar, Chandramouli Narayanan, and Chita R. Das. Using Reflection for Checkpointing Concurrent Object Oriented Programs. In Shigeu Chiba and Jean-Charles Fabre, editors, *Proceedings of the OOPSLA Workshop on Reflection Programming in C++ and Java*, October 1998.
16. Gregor Kickzales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.
17. Arthur H. Lee and Joseph L. Zachary. Using Meta Programming to Add Persistence to CLOS. In *International Conference on Computer Languages*, Los Alamitos, California, 1994. IEEE.
18. Orlando Loques, Julius Leite, Marcelo Lobosco, and Alexandre Sztajnberg. Integrating Meta-Level Programming and Configuration Programming. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Proceedings of the 1st Workshop on Object-Oriented Reflection and Software Engineering (OORaSE'99)*, pages 137–151. University of Milano Bicocca, November 1999.
19. Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.
20. Juan-Carlos Ruiz-Garcia Marc-Olivier Killijian, Jean-Charles Fabre and Shigeru Chiba. A Metaobject Protocol for Fault-Tolerant CORBA Applications. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS'98)*, pages 127–134, 1998.
21. Philippe Mulet, Jacques Malenfant, and Pierre Cointe. Towards a Methodology for Explicit Composition of MetaObjects. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *Sigplan Notice*, pages 316–330, Austin, Texas, USA, October 1995. ACM.
22. Thomas Riechmann and Jürgen Kleinöder. Meta-Objects for Access Control: Role-Based Principals. In Colin Boyd and Ed Dawson, editors, *Lecture Notes in Computer Science*, number 1438 in Proceedings of 3rd Australasian Conference on Information Security and Privacy (ACISP'98), pages 296–307, Brisbane, Australia, July 1998. Springer-Verlag.
23. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
24. Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.

25. Robert J. Stroud and Ian Welch. Dynamic Adaptation of the Security Properties of Application and Components. In *Proceedings of ECOOP Workshop on Distributed Object Security (EWDOS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), pages 41–46, Brussels, Belgium, July 1998. Unité de Recherche INRIA Rhŏne-Alpes.
26. Robert J. Stroud and Zhixue Wu. Using Meta-Object Protocol to Implement Atomic Data Types. In Walter Olthoff, editor, *Proceedings of the 9th Conference on Object-Oriented Programming (ECOOP'95)*, LNCS 952, pages 168–189, Aarhus, Denmark, August 1995. Springer-Verlag.
27. Robert J. Stroud and Zhixue Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In Chris Zimmerman, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, chapter 3, pages 31–52. CRC Press, Inc., 2000 Corporate Blvd.,N.W., Boca Raton, Florida 33431, 1996.
28. Junichi Suzuki and Yoshikazu Yamamoto. Extending UML for Modeling Reflective Software Components. In Robert France and Bernhard Rumpe, editors, *Lecture Notes in Computer Science*, number 1723 in Proceedings of ≪UML≫'99 - The Unified Modeling Language: Beyond the Standard, the Second International Conference, pages 220–235, Fort Collins, CO, USA, October 1999. Springer-Verlag.
29. Michiaki Tatsubori. An Extension Mechanism for the Java Language. Master of engineering dissertation, Graduate School of Engineering, University of Tsukuba, University of Tsukuba, Ibaraki, Japan, February 1999.