Reflective Analysis and Design for Adapting Object Run-Time Behavior

Walter Cazzola[1], Ahmed Ghoneim[2], and Gunter Saake[2]

[1] Department of Informatics and Computer Science,
Università degli Studi di Genova
Via Dodecaneso 35, 16146, Genova, Italy
cazzola@disi.unige.it
[2] Institute für Technische und Betriebliche Informationssysteme,
Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D-39016 Magdeburg, Germany
{ghoneim,saake}@iti.cs.uni-magdeburg.de

Abstract. Today, complex information systems need a simple way for changing the object behavior according with changes that occur in its running environment. We present a reflective architecture which provides the ability to change object behavior at run-time by using design-time information. By integrating reflection with design patterns we get a flexible and easily adaptable architecture. A reflective approach that describes object model, scenarios and statecharts helps to dynamically adapt the software system to environmental changes. The object model, system scenario and many other design information are reified by special meta-objects, named *evolutionary meta-objects*. Evolutionary meta-objects deal with two types of run-time evolution. Structural evolution is carried out by causal connection between evolutionary meta-objects and its referents through changing the structure of these referents by adding or removing objects or relations. Behavioral evolution allows the system to dynamically adapt its behavior to environment changes by itself. Evolutionary meta-objects react to environment changes for adapting the information they have reified and steering the system evolution. They provide a natural liaison between design information and the system based on such information. This paper describes how this liaison can be built and how it can be used for adapting a running system to environment changes.

Keywords: Reflection, Meta-Objects, Design Pattern, UML, Software Evolution.

1 Introduction

Nowadays a topical issue in the software engineering research area consists of producing software systems able to adapt themselves to environment changes by adding new and/or modifying existing functionalities. There are a number of mechanisms for obtaining adaptability. One of these mechanisms are the design patterns [10, 2]. Another mechanism for obtaining adaptability is *reflection* [16, 3]. A non-stopping software system with long life span, has to be able to dynamically adapt itself to changes to its

environment. Two aspects control the evolution of a system of objects: *behavior*, and *dependencies*. Both of them can be involved in system evolution to comply with changes to system requirements.

We present a novel design approach that provides a system with the ability to change object behavior at run-time. This approach integrates reflection and design patterns, by reifying object model, scenario, and statechart of the system. This paper explores how to design a dynamically self-adapting system extending the idea of shifting reflection from linguistic to methodological, that has been presented by Cazzola et al. in [4].

Object oriented methodologies like Objectory [13], and UML [1] statically describe the system's behavior during the design phase: all functions in the system are captured by a use-case model. The dynamic behavior of each use case is described by scenario, and interaction diagrams. Models in the *unified modeling language* (UML) [1] are classified in structural (e.g., class diagram) and behavioral (sequence diagrams and statecharts). Behavioral models deal with the representation of the behavior of the system. Class diagrams show how the system is structured, i.e., the classes composing the system and their relationship. Sequence diagrams and use cases describe the interactions between objects, whereas statecharts [12, 11] describe, as a state machine, the behavior of every object in the system. UML provides a special key, named stereotype, to deal with concepts not well-defined or difficult to model by using only class diagrams. Cazzola et al. [4] adopt a special stereotype, named «`causal-connection`», to model with class diagrams a reflective system. This one has been the basic approach to reflective object-oriented analysis (ROOA).

All these methodologies provide a way to adapt software system to requirement changes as long as the system is under design/development. By subsequent refinement of a prototypal system up to reach its final version. These methodologies don't foresee a way to adapt the system when it is running without stopping it and modifying its design. Hence they are not enough to model a non-stoppable and adaptable software system.

Our approach gets two types of dynamic evolution: *structure*, and *behavior evolution*. To comply with this achievement the system is structured in two levels: base- and meta-level. In the base-level we have the system we want to render self-adapting, whereas in the meta-level there are some meta-objects, called *evolutionary meta-objects*, reifying all the design information related to the base-level. Evolutionary meta-objects deal with both structure and behavior evolution modifying these information. The causal connection is the mechanism which really realizes the dynamic self-adaption reflecting the changes performed by the evolutionary meta-objects in the base-level. The causal connection is achieved by using the pattern *adapter* [10], which allows base-objects to delegate the execution to their meta-objects, and by acting on inter- and intra-object connections through changing systems sequence and collaboration diagrams and applying such a change by using the *state design* pattern [10, 9], which allows an object to change its behavior when its internal state changes.

Saake et al. [17] have proposed a specification framework for modeling evolving objects as basic building blocks of information systems. This is done by dividing object behavior into rigid and dynamic parts at the specification phase, thus we will extend this idea to the design phase.

The rest of the paper is organized as follows: section 2, outlines run-time adaptation approaches. Section 3, illustrates the proposed approach for object behavior evolution. Section 4, briefly points at related works in the discipline of building adaptable software system. Section 5 briefly describes our approach on modeling a banking system. Finally section 6 concludes with an outline of our ongoing research in adapting object behavior.

2 Run-time Software Engineering Adaptation Techniques

The main motivation for using *reflection* and *design patterns* in software engineering is to build applications able to adapt themselves to environmental changes. The continuous evolution in information technologies forces the designer to deal with more and more ambitious requirements. Adaptability is the fundamental property that software systems must have to satisfy their evolution. A system can be considered to be adaptable when it can be modified to satisfy new requirements.

2.1 Design Patterns

The use of patterns is essentially the reuse of well-established good ideas. A pattern is a named, well-understood good solution to a common problem in a specific context.

Collections of design patterns were described in [10,2]. Patterns are used for changing structure and behavior of a software system. Design patterns have been classified into *structural* and *behavioral* patterns. *Strategy* patterns, in [10] on page 315, have been used in the design of dynamically adaptable systems. The design principle underlying the strategy pattern is to delegate the implementation of exported operations of the buffer manager to a replaceable strategy object. Multiple strategies can be derived from an abstract strategy class and compiled into the system.

In this work we will use two type of design patterns: *adapter*, in [10] on page 139, and *state* patterns, in [10] on page 305. An adapter class implements an interface known to its clients and provides access to an instance of a class not known to its clients. Hence it delegates the execution of a method to another object. A state pattern allows an object to change its behavior when its internal state changes.

2.2 Reflection

Reflection is the ability of a system to observe and to modify its computation [16]. An object-oriented reflective system is logically structured in two or more levels, constituting a *reflective tower* [16]. The first level is the *base-level* and describes the computations that the system is supposed to do. The second one is the *meta-level* and describes how the base-level computations have to be carried out. Objects working in the base-level are called *base-objects*, whereas objects working in the other levels (meta-levels) are called *meta-objects*. Each level is *causally connected* to adjacent levels, i.e. objects working into a level have data structures reifying the activities and the structures of the objects working into the underlying level and their actions are reflected into such data structures. Meta-objects supervise the activity of their *referents*, i.e., the base-objects. Trap mechanism explains how supervision takes place. Each base-object action is trapped

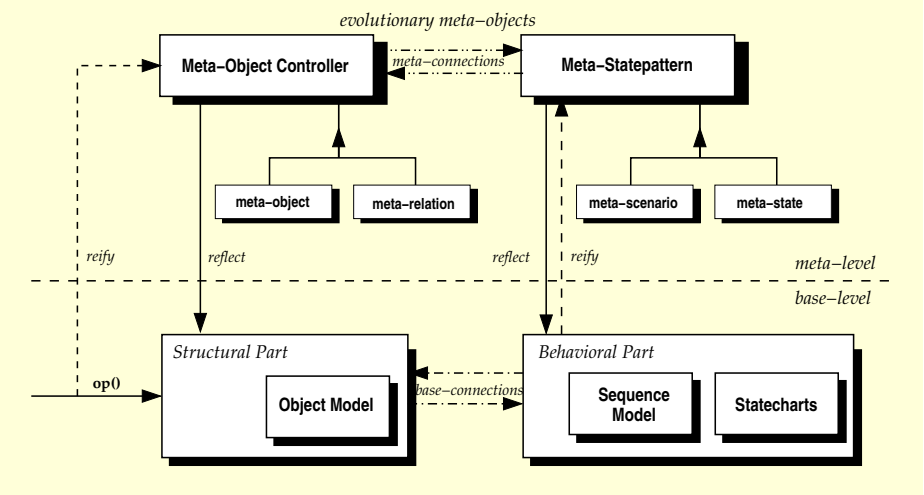

Fig. 1. Structure of an Adaptable System.

by a meta-objects, which performs a meta-computation, then it allows its referent to perform the action.

3 Reflective Dynamic Adaptation of Software Systems

The main purpose of our approach consists of adapting the object behavior at run-time, by using its design information. To do this we are going to apply to the dynamic objects structure offered by Saake et al. [17], a reflective approach as explained by Cazzola et. al. [4].

In our approach the system is divided into its structure (described by its object model) and its behavior (described by statechart and sequence diagrams). Similarly an application is divided into two levels, as shown in Fig. 1: a base-level and a meta-level. The base-level, which implements the real system, is described by three models:

- *Object model*, which describes objects and their relations. This model represents the structural part of the system.
- *Sequence diagrams*, which trace system operations between objects (inter-object connection) for each use case at a time.
- *Statecharts*, which represent different the evolution of the state of each object (intra-object connection) in the system.

The meta-level is responsible of dynamically adapting the base-level and it is composed of some special meta-objects named *evolutionary meta-objects*. The *meta-object controller*, is an evolutionary meta-object which is directly responsible for changing the structure of its referents through adding or removing new objects, methods, and relations. This evolutionary meta-object delegates the adaptation of the behavior of its

referents to its colleague: the *meta-statepattern*. Meta-statepattern is responsible for changing the state of each object, and the behavior of the system in accordance with the work of the meta-object controller.

There are two kinds of meta-object controllers: *meta-object* and *meta-relation*. When the base-level system begins an operation, an evolutionary meta-object adapts the objects involved in such an operation and a meta-relation adapts their behavior in accordance with the preconditions to that operation. After that, we need to determine the states for each referent involved in the execution of that operation. This is done by delegating the control to the meta-statepattern which detects the states of the involved referents. The relationship among the evolutionary meta-objects and its referents is specified by means of a meta-object protocol (MOP) [15]. The MOP works as follows: when an object asks for an operation, the operation and all objects involved in its execution are reified by the evolutionary meta-object controller. Then, it adapts the reified objects and reflect the changes on the base-level. Figure 2 shows, through a sequence diagram, the protocol steering the interactions between base- and meta-level. Basically we have:

- When an object asks for an operation, the meta-object controller sends this operation to the evolutionary meta-object. The evolutionary meta-object reifies the involved base-objects. At the meta-level some meta-computations involving the reified objects occur. Base-objects evolution is carried out by these meta-computations. Evolutionary meta-objects return the result to meta-controller through meta-connection.
- Meta-object controller entrusts the meta-statepattern, to determine the state of each object and the behavior for the system. There are two kinds of meta-statepatterns: the *meta-scenario* which traces the execution of the trapped operation, and the *meta-state* which determines the state of each object involved in the its execution.
- Verified the feasibility of the evolution, the changes are reflected on the base-system via the causal connection relationship among meta-statepattern, meta-object controller and their counterpart in the base-level (the behavioral and structural part).

3.1 Adaptation by Causal Connection and Design Patterns

In the just described architecture the meta-level is causally connected to the base-level, i.e., each event that occur in the base-level is implicitly reified by the meta-level and every action carried out by the meta-level is automatically reflected on the base-level [16]. A system (the base-level) causally connected to another (an active engine supervising the adaptation) is the key to get a system that can evolve.

Evolutionary meta-objects are responsible for supervising and controlling their referents in the base-level. They use the pattern *Adapter* for reifying all objects, and their behavior involved in the execution of an operation. The implicit mechanism that is responsible for changing the behavior of the objects is realized by the evolutionary meta-objects and its MOP. MOP contains *reification categories*, these categories are entities representing objects, methods, and relations. Evolutionary meta-objects work directly with these categories to complete their computations. They also store the changes in the behavior and structure of the reified entities in the corresponding categories. Therefore, the MOP steers the adaptation of the reified entities (objects, methods and so on) without effectively changing the base-level.

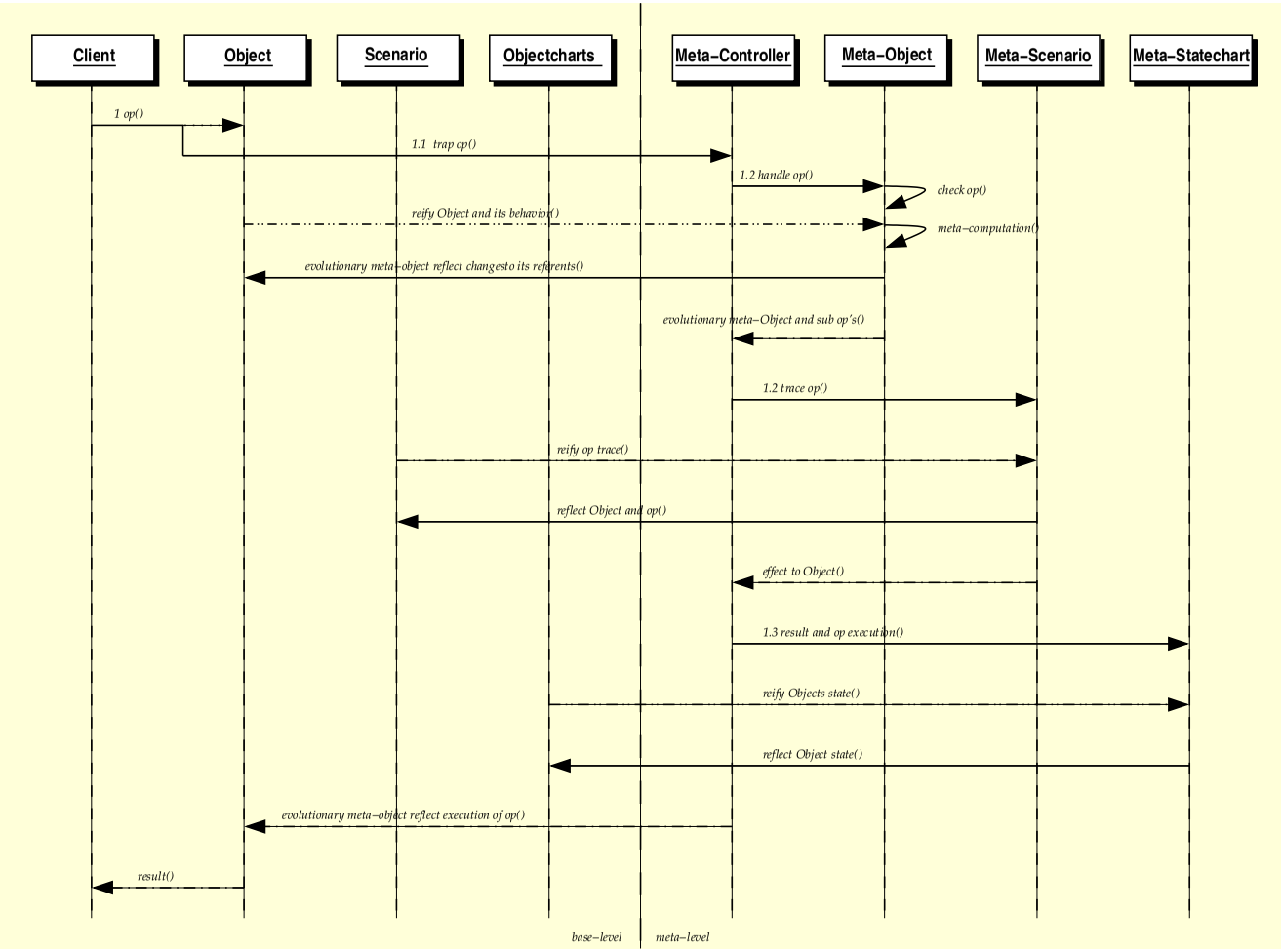

Fig. 2. Reflection interaction system protocol

At the end of its computation, the evolutionary meta-object controller entrusts the changes to the meta-statepattern which has to determine the evolution's feasibility and really evolving the system. The meta-statepattern gets the state of each object involved in the adaptation (thanks to the *Adapter*), builds the categories reifying such states. Then, the meta-statepattern uses the *State* pattern for the evolution of the states in accordance with the changes computed by evolutionary meta-object controller. Finally, meta-scenario evolves the trace by using changes in MOP categories. After the operation completes its execution, then its reification categories are destroyed.

Moreover, the meta-statepattern has to verify the soundness and the consistency of the base-level against the proposed changes. These checks ensure that the implications between evolutionary meta-objects and meta-statepattern are respected. This ensures that the meta-level can carry out the proposed changes without rendering the system inconsistent. Consistency rules are expressed via the VDM formalism [14].

4 Related Work

In the last few years, there has been a growing interest for dynamically evolving object-oriented systems. In the literature there are several approaches related to building adaptive software systems that allow system behavior to evolve after design time.

Dowling and Cahill [8] have proposed a meta-model framework named *K-components*, that realizes a dynamic, self-adaptive architecture. It reifies the features of the system architecture, e.g., configuration graph, components and connectors. This model presents a mechanism for integrating the adaptation code in the system, and a way to deal with system integrity and consistency during dynamic reconfiguration.

Another approach consists of building a reflective architecture by using cooperative object-oriented style [19]. Structural elements of this approach are classes as the basic components, and *CO actions* (*cooperative actions* represent the interactions among objects characterizing the collaborative behavior [7]) as the basic connectors. This approach achieves adaptability by (1) dynamically extending objects behavior using roles, and (2) by selecting at run-time the objects and roles participating in a cooperation.

Seiter et al. [18] have proposed a new relation between classes, named *context relation*. The context relation is meaningful at analysis, design, and implementation levels. It may also impact software testing. As inheritance and dynamic binding modify traditional program flow and subsequent testing models, the context relation may modify them as well.

In the above approaches, the adaptation is achieved in three ways. First, by controlling components and connectors through a reflective architecture. Second, by separating the objects from their interactions and by entrusting their adaptation to meta-objects. Finally, by using design patterns to define a new relation, or providing a mapping between interfaces which are not possible to dynamically change. Our approach exploits all these techniques: we have a reflective architecture, with meta-objects dealing with design information and adapting the system by using design patterns.

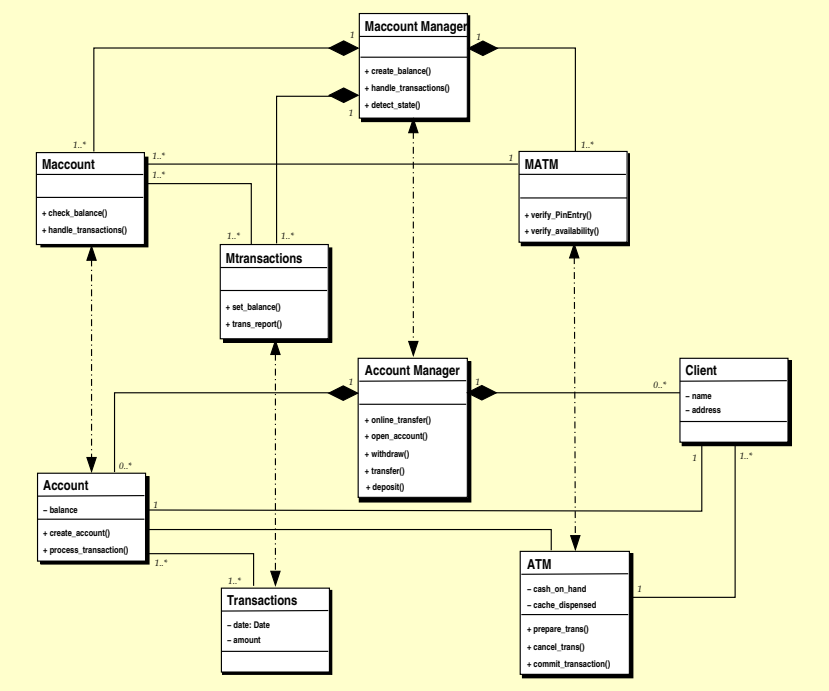

Fig. 3. Reflective object model

5 Case Study: Banking System

Now, we show the benefits and drawbacks of applying our approach to dynamically adapt a system to the environment on a case study.

5.1 Overview of the Case Study

We are considering the classical banking system example with two use cases: open or close account, and transactions (deposit, transfer, withdraw). The bank allows the client (1) to get the balance of his account, (2) to withdraw, and deposit cash, and (3) transfer money from his account to another.

We show the three reflective models of our architecture (see Sect. 3) applied to the case study. The structural part is reified by the reflective object model (Fig. 3), every object is associated to an evolutionary meta-object. The adaptation is done by creating MOP categories, which include objects and their behavior (methods). Meta-objects, such as `Maccount`, `Mtransaction`, `MATM`, `Maccount manager` perform meta-computations. Evolutionary meta-objects are used to reflect changes to objects in the reification categories.

The behavioral part is reified by: meta-scenario and meta-state. The meta-scenario traces bank transactions between meta-object and its referent. The evolutionary meta-

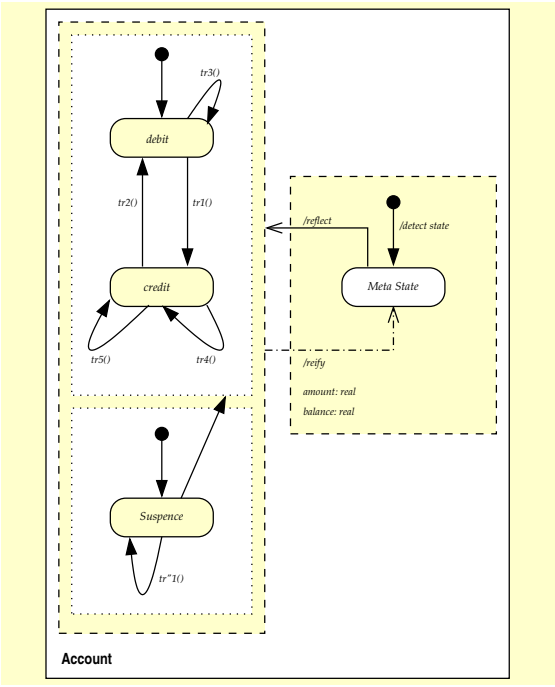

Fig. 4. Reflective objectcharts.

object reifies the following objects (`Client`, `Account`, `Transactions`). The evolutionary meta-objects add new clients, open new accounts and add these instances in the base-level, initialize the balance, and so on.

Moreover, evolutionary meta-objects add transactions such as deposit, transfer, withdraw and so on, to client objects. The purpose of all these transactions consists of adding (or subtracting) money to (from) balance of the client account. Hence, we need only two meta-scenarios for describing the whole system's behavior. The former for describing the open and close operation, the latter for dealing with money movements (Fig. 5). The adaptation of the state of every object is done by delegating it to the meta-statepattern which is responsible for detecting and notifying every change to their state during transactions.

Finally, Meta-statecharts control the state of base-level objects (Fig. 4). Evolutionary meta-object and its MOP categories have the rules to dynamically change the objects. The object states change at run-time by reifying the object states from base-level to state category. Meta-statepattern uses evolutionary meta-objects and its MOP to determine the changes to object states. After that, meta-statepattern reflects the changes to base-state for every object. We use the logic of partial functions from VDM [14], for representing a postcondition for transactions between object states. The transactions of account objects in the bank system are:

– →(suspense):(balance closed no effects)(time=t)

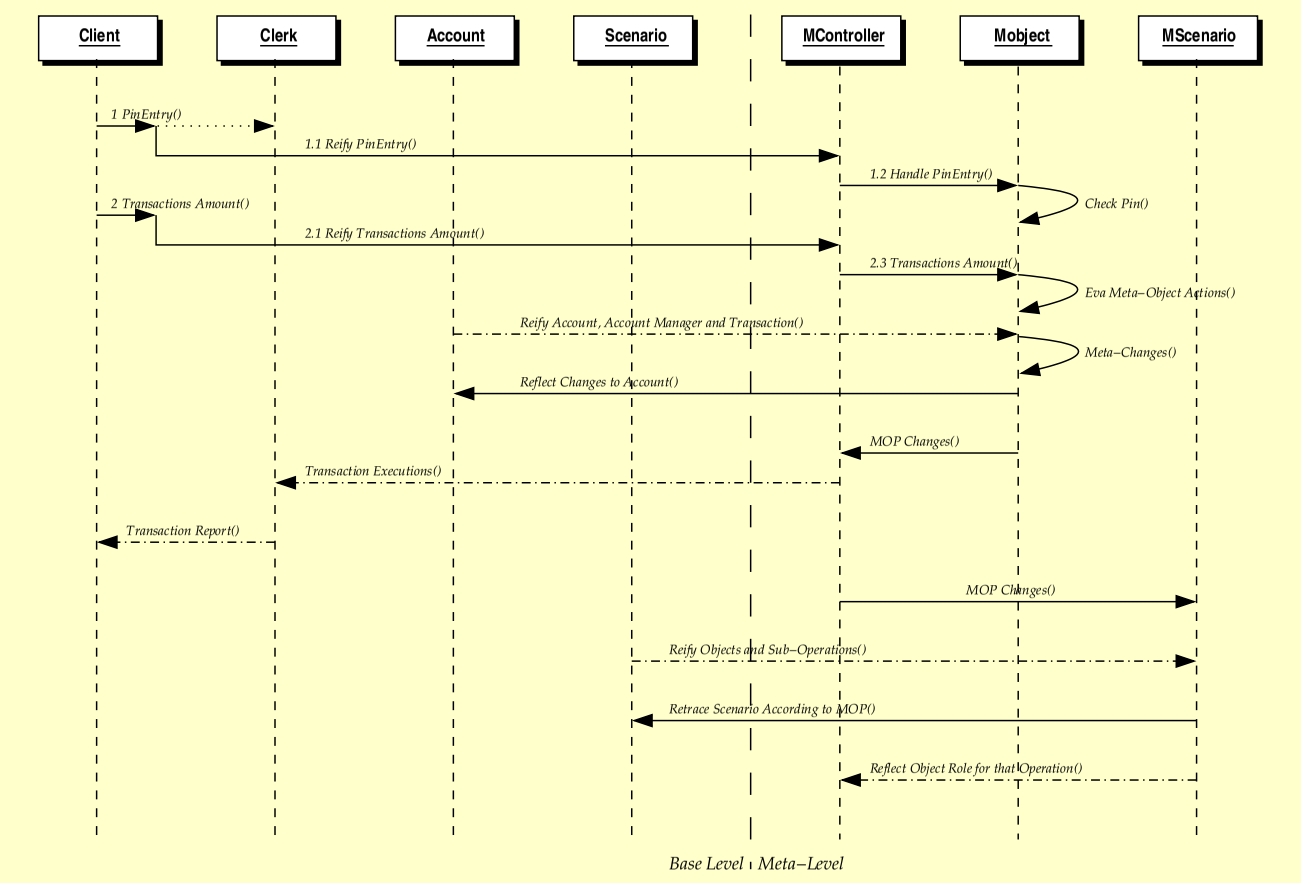

Fig. 5. The sequence diagram describing money movements.

- (suspense)→(suspense):(balance closed no effects)(settime)(time=$\overleftarrow{t}$+t)
- (suspense)→(non-suspense):(true)(balance opened for any transactions)
- →(debit):(true)(open client account)(setbalance)(balance=0)
- (debit)→(credit):(true)[tr1=deposit(amount)](amount>-balance) (setbalance)(balance=$\overleftarrow{balance}$+amount-charge)
- (credit)→(debit):(true)[tr2=withdraw(amount)][amount>balance] (setbalance)(balance=$\overleftarrow{balance}$-amount)
- (debit)→(debit):(true)[tr3=deposit(amount)][amount≤-balance] (setbalance)(balance=$\overleftarrow{balance}$+amount-charge)
- (credit)→(credit):(true)[tr4=withdraw(amount)][amount≤balance] (setbalance)(balance=$\overleftarrow{balance}$-amount)
- (credit)→(credit):(true)[tr5=deposit(amount)](setbalance) (balance=$\overleftarrow{balance}$+amount)

6 Conclusion

In this paper we addressed the problem of dynamically adapting the behavior of a running system. This has been done by building a reflective architecture, which integrates reflection (meta-object) and design pattern (Adapter and State). The proposed reflective architecture was implemented by two levels: meta- and base-level. In the meta-level, we defined new meta-objects named *evolutionary meta-objects*, which are responsible for reifying objects of the base-level and their behavior at run-time. Also, they reflect changes to their referents. The architecture consists of three reflective models: *reflective object model*, *reflective scenario*, *reflective state*. This reflective approach provides a self adaptable information system that extends the initial trial [4] to move reflection from linguistic to methodological.

As future works, we are planning to put this approach on a firm semantic foundation by using object-Z as formal object-oriented language. In our approach, control flow is shifted from base-level to meta-level. So, we can validate the adaptation of objects and their behavior by using *animation*. Finally, we will implement the approach by using a reflective language, like OpenC++ [5] or OpenJava [6].

References

1. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, third edition, February 1999.
2. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd, 1996.
3. Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.

4. Walter Cazzola, Andrea Sosio, and Francesco Tisato. Shifting Up Reflection from the Implementation to the Analysis Level. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 1–20. Springer-Verlag, Heidelberg, Germany, June 2000.
5. 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.
6. Shigeru Chiba, Michiaki Tatsubori, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A Class-based Macro System for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 119–135. Springer-Verlag, Heidelberg, Germany, June 2000.
7. Roger de Lemos and Alexander Romanovsky. Coordinated Atomic Actions in Modelling Object Cooperation. In *Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, volume 30 of *Sigplan Notices*, pages 152–161, Kyoto, Japan, April 1995.
8. Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, LNCS 2192, pages 81–88, Kyoto, Japan, September 2001. Springer-Verlag.
9. Luciance Lamour Ferreira and Cecilia 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.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.
11. David Harel and Eran Gery. Executable Object Modeling with Statecharts. In *Proceedings of 18th International Conference on Software Engineering*, pages 246–257. IEEE Press, March 1996.
12. David Harel and Michael Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.
13. Ivar Jacobson, Magnus Christerson, Patrick Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A use Case Driven Approach*. Addison Wesley, 1992.
14. Cliff B. Jones. *Systematic Software Development Using VDM*. Englewood Cliffs, NJ: Prentice-Hall, second edition, 1990.
15. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.
16. 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.
17. Gunter Saake, Can Türker, and Stefan Conrad. Evolving Objects: Conceptual Description of Adaptive Information Systems. In H. Balsters, B. de Brock, and S. Conrad, editors, *FoMLaDO/DEMM2000, LNCS 2065*, pages 163–181. Springer-Verlag Berlin Heidelberg, 2001.
18. Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of Object Behavior Using Context Relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, 1998.
19. Emiliano Tramontana. Reflective Architecture for Changing Objects. In Walter Cazzola, Shigeru Chiba, and Thomas Ledoux, editors, *On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, June 2000. Available at `http://www.disi.unige.it/RMA2000`.