STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LV**, Number 3, 2010

# A FRAMEWORK FOR ACTIVE OBJECTS IN .NET

DAN MIRCEA SUCIU AND ALINA CUT

Abstract. Nowadays, the concern of computer science is to find new methodologies that help decomposing large programs and run them efficiently onto new parallel machines. Thus, the popularity of concurrent object-oriented programming has increased proportionally with the market requirements of concurrent and distributed systems that meet simplicity, modularity and code reusability. The purpose of this paper is to define a class library based on Active Object pattern introduced in [3], which has a high level of extensibility. Class library's main objective is to help in building concurrent object-oriented applications with a minimum effort and using a significant amount of already existing code. This approach addresses the problem of integrating concurrency with object-oriented programming and respects the principles imposed by them. In order to present the main features of our model a sample application is presented.

## 1. Introduction

An important motivation behind *concurrent object-oriented programming* (COOP) is to exploit the software reuse potential of object-oriented features in the development of concurrent systems [4]. *Object-oriented programming* (OOP) and *concurrent programming* (CP) unification seems natural if we think that real-world objects are indeed concurrent. On one hand, OOP has been developed having as a model our environment (seen as a set of objects among which several relationships exist and which communicate between them by message transmission). On the other hand, the concurrency between objects led to the normal trend of transposing this into programming. However, the integration of concurrency into OOP languages is not an easy task. The concurrent features of a language may interfere with its object-oriented features making them hard to integrate in a single language or cause many of

Received by the editors: August 6, 2010.

2000 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.3 [**Software**]: Software Engineering – *Coding, Tools and Techniques*; D.2.7 [**Software**]: Software Engineering – *Distribution, Maintenance and Enhancements* .

*Key words and phrases.* object-oriented concurrent programming, active objects.

their benefits to be lost [4]. For this reason, we should carefully choose a proper mechanism for synchronization of concurrent objects.

*Active object pattern* is a high-level abstraction that simplifies CP and works on the object level. This paradigm proposes a new style of programming by decoupling method execution from method invocation in order to simplify synchronized access to an object that resides in its own thread of control [3]. Taking advantages of this pattern, we propose a general active object model which combines the reusability with the elegancy of integrating concurrency into OOP. Reusability of code is an important advantage of OOP that simplifies the development process by reducing the design and the coding. Later, in this paper, we present a sample that uses this specific active objects model for generating code from scalable statecharts [6].

## 2. Active Object model

*Active object pattern* comes to simplify the synchronization access to an object that is running in its own thread. The major difference imposed by this pattern is that it works on the object level not on an object hierarchy like most design patterns. This implies modeling classes as active classes with the implication that their operations are processed asynchronously and inherently thread-safe with respect to each other by processing at most one operation at any given time [2].

An active object has two important particularities: it runs in its own thread of control and the invoked methods don't block the caller but are executed asynchronously. Figure 1 illustrates the components of an active object as they are presented in [3].

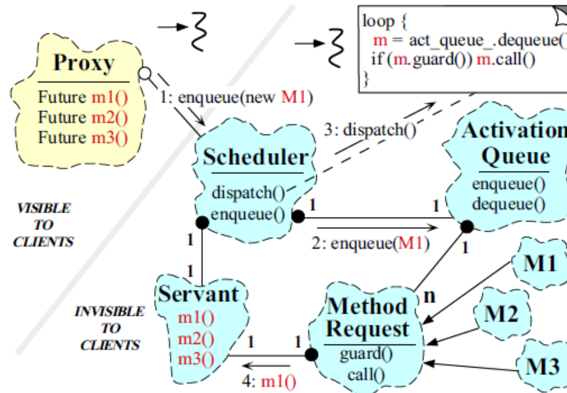


Figure 1. The components of the Active Object pattern [3]

We can see that an active object has two important parts according to the visibility property. The first part, which is visible to the client, contains a *proxy* which represents the public interface of an active object. It is responsible for accepting method calls or requests from clients (other objects that use an active object) and convert them into messages (*method requests*) that will be added into a *message queue*. The second part will contain the components that are hidden from the client. A *scheduler* is a special object which runs in the context of the active object thread. It maintains an *activation queue* with incoming messages. The scheduler, based on some criteria (the order in which the methods are inserted into the activation queue, some *guards*), will decide which message to dequeue in order to process it. After it processes the message, the scheduler will invoke the actual methods of the *servant*. The *servant* represents the private implementation of the active object. It encapsulates the data and defines the behavior and the state of the active object ([2]). In this manner, method invocation and method execution are decoupled and concurrency between objects is introduced.

Our proposed abstraction meets the properties of *active object pattern*: message-based property, asynchronous execution property and thread-safe property. The major concern was to develop a more general active object that allows us reuse as much code as possible and simplifies the development process. Figure 2 presents the components of our active object model and the relationships between them.

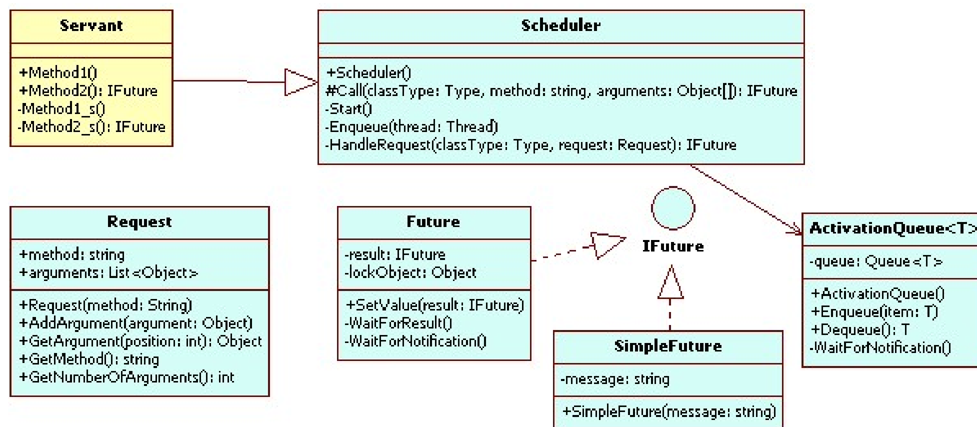


FIGURE 2. Extensible active object model

In order to obtain a model with a high extensibility we have decided to modify the structure of *active object pattern*. The first step in achieving our

goal was to unify the *proxy* and the *servant*. In this case the *servant*, besides its regular activity, will also serve as a public interface for the client. It contains public methods that can be accessed by the clients, but these methods will only be used to forward a request to the scheduler. In this manner, the *servant* covers the behavior of a *proxy* presented in [3]. But, our *servant* also has some private methods (corresponding to the public methods) that represent the actual implementation of the services offered by this active object.

The second step to obtain a high extesibility was to define standard classes for the other components proposed in active object pattern (*scheduler*, *request*, *activation queue* and *future*) that offer us the possibility of reusing the code as many times as we need without any modification inside those classes. Our *scheduler*, as presented in figure 2, contains only four methods (three of them are private and one is protected) plus a public constructor. Although we have modified the structure of the *scheduler* to achieve our aim, it still meets all the functionalities of a scheduler presented in [3]. The protected method Call is the one that the *servant* calls each time a client makes a method call. This call is possible because the *servant* is a subclass of *scheduler*. Otherwise, the *servant* wouldn't be able to access a protected method of *scheduler*. The task of Call is to accept method calls, transform them into requests and add them, using Enqueue, into an activation queue. It will return an object IFuture that represents the place from where the client can read the result of his call. We already know that a *scheduler* runs in the context of the active object thread. The private method Start creates and starts the thread of the scheduler, a thread that will always try to handle the requests from the activation queue. HandleRequest is in fact the private method that will manage the requests. But how can it call the actual methods of the servant and still respect the standard that we want to obtain? Well, it uses reflection in order to call the proper methods of the servant when it knows the names of the methods (as strings) and the list of formal parameters.

Our proposed abstraction model takes into consideration the scenario when a client calls a method and waits a response from that method. We have modeled this by using future objects. A future object is a place where the active methods put their possible results. It can be considered a *rendezvous* for the caller and the active object ([2]). Once the result of a method is computed it will be stored in a future object and the caller can access the result from there. In the case that the caller tries to access the result before the method has computed it, the caller automatically blocks until the result is stored in the related future.

## 3. Sample - Robots Application

In order to prove the efficiency of our active object model, we developed a sample application implementing the behavior of a robot object, which is searching the exit of a maze using *the left-hand rule*. All objects (the robots and the maze components) used in the sample application are active objects: they have their own thread awaiting to receive and process method calls.

The robots are placed in the same table (representing the maze) and are sharing the same tracks. An important remark is that some tracks are blocked, meaning that there is a wall and the robot cannot access that tracks. When a robot meets a blocked track it should bypass it taking into consideration the *left hand side rule*. This rule assures that a robot tries to make left each time it meets an obstacle (maze margins or blocked tracks).

The directions of the robot are the ones corresponding to the four cardinal directions: *south* will be codified as direction 1, *east* will be direction 2, *north* will be codified with direction 3 and *west* will represent direction 4. So, each robot moves in the maze taking into consideration only these four directions.

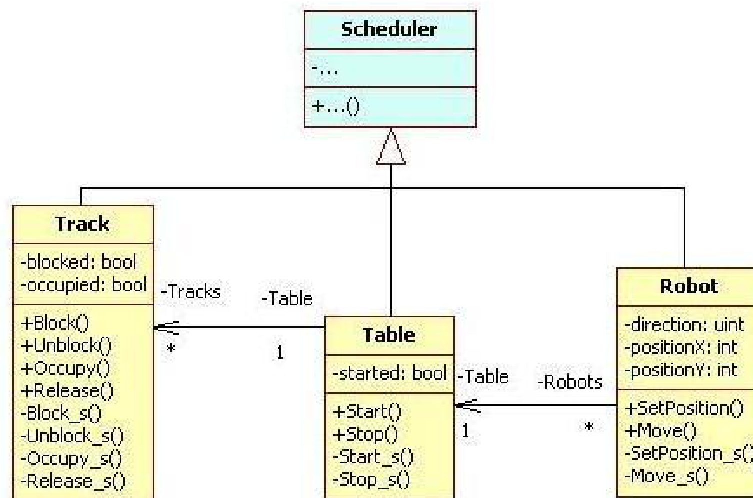


FIGURE 3. *Robots* application class diagram

Figure 3 contains the class diagram describing the structure of classes *Robot*, *Track* and *Table*. All these classes are derived from *Scheduler*, so they implement three distinct types of active objects. An important aspect is that no method implemented at Scheduler class level needs to be rewritten in its descendants. In other words, Track, Table and Robot objects are active objects (have their own execution thread, method queue and synchronization

protocol) without implementing any element specific to concurrent programming, everything being inherited from Scheduler class. The only additions are referring to the implementation of their particular behavior.

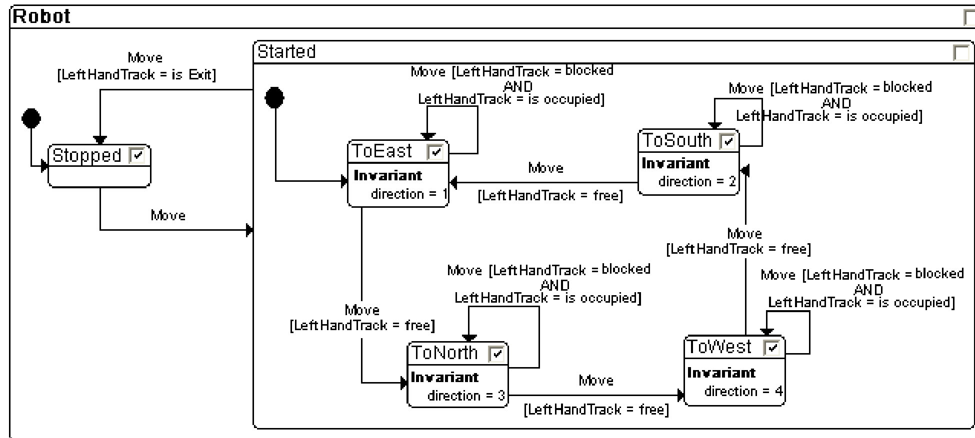


FIGURE 4. *Robot* class behaviour description using scalable statecharts

A robot keeps a direction and its coordinates on the table. It always makes a move to the right when the track is trying to conquer is blocked. In the case that a track is free then when a robot reaches it the track will become occupied.

For situations when two robots try to reach the same track we have considered that each time a robot occupies a cell, that track will also become blocked. From the robot point of view there is no difference between a blocked cell and an occupied cell. Of course that an occupied track will become free as soon as the robot leaves that track while a blocked track will remain blocked forever. When a robot tries to walk into an occupied track its move fails. In this situation the robot will make a move to the right from its position similar to the case in which that track is blocked.

Figure 4 shows the behavior model of robot objects defined with scalable statecharts created using *ActiveCASE* tool ([5]). Moreover, *ActiveCASE* tool was modified accordingly to support the proposed active object model and to generate source code based on it.

Figure 5 shows a screen shot of the *Robots* application.

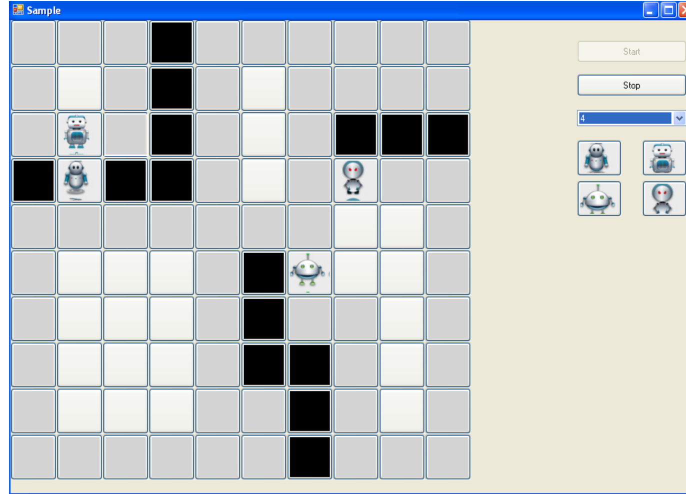

FIGURE 5. *Robots* application screen shot

## 4. CONCLUSIONS

The popularity of COOP is increasing as the concurrency is becoming a required component of ever more types of systems. *Active object pattern* offers us an elegant way of decoupling method invocation from method execution. Based on this pattern we have obtained a more general model by trying to reorganize the structure of a regular active object and keep the functionalities of its components. This type of active object is characterized by a higher extensibility and a larger amount of reusable code while meeting the base properties of the *active object pattern* proposed in .

Starting from the original model of an active object presented in [3] we have focused on developing an abstract C# library for active objects. Our aim was to reduce the number of classes that should be implemented for an active object, in order to simplify the work with this type of objects. For achieving this goal we have made some modifications in the structure of *active object pattern*.

Some important features of our library are:

- the concurrent programming aspects are handled exclusively at framework level. All future descendant classes will take care only about logic implementation of their behavior, without taking into account parallel execution or synchronization of methods,
- the higher extensibility offered by the abstraction of the classes,

- the reusability of code possible because of the inheritance relation introduced between the servant and the scheduler
- the flexibility provided by the easy adaption to the external changes in the implementation of the services.

The features mentioned above help to reduce the effects of inheritance anomalies that characterize COOPL, as described in [1]. We have also introduced the possibility of working with guards when trying to execute a method. A guard is a constraint that should be satisfied before its associated method can be executed.

We have decided to demonstrate the applicability and the efficiency of our framework by developing a sample application based on our model. The sample aplication presents the moves of some robots in a labyrinth with obstacles.

Future improvements of our library may introduce priorities at the level of the services offered by an active object. Taking into consideration this idea, a scheduler will execute the methods according to their priorities. This implies an improvement at the level of the activation queue. In this manner, instead of a simple queue a priority queue may be used.

## References

[1] Jean-Pierre Briot , Akinori Yonezawa, "Inheritance and Synchronization in Concurrent OOP", European Conference on Object-Oriented Programming (ECOOP87), LNCS 276, pp. 3240, 1987.
[2] Tobias Gurok, "Active Objects and Futures: A Concurrency Abstraction Implemented for C# and .NET", Fakultat fur Elektrotechnik, Informatik und Mathematik, Universitat Paderborn, Bachelor Thesis, 2007.
[3] R. Greg Lavender, Douglas C. Schmidt, "Active Object: an object behavioral pattern for concurrent programming", Pattern languages of program design, Addison-Wesley Longman Publishing Co, Boston, USA, pp. 483-499, 1996
[4] Michael Papathomas, "Concurrency Issues in Object-Oriented Programming Languages", in Object Oriented Development, ed. D. Tsichritzis, Centre Universitaire dInformatique, University of Geneva, pp. 207-245, 1989
[5] Dan Mircea Suciu, "ActiveCASE Tool for Design and Simulation of Concurrent Object-Oriented Applications", Studia Universitatis Babes Bolyai, Informatica, Vol. XLVI, No. 2, 2001, pp. 73-80
[6] Dan Mircea Suciu, "Reverse Engineering and Simulation of Active Objects Behavior", Knowledge Engineering, Principles and Techniques - KEPT-2009 Selected Papers, Babes-Bolyai University of Cluj-Napoca, July 2-4 2009, pp. 283-290

Department of Computer Science, "Babeş-Bolyai" University, 1 M. Kogălniceanu St., RO-400084 Cluj-Napoca, Romania
*E-mail address*: tzutzu@cs.ubbcluj.ro, alina_cut@yahoo.com