Studies and Materials in Applied Computer Science, Vol. 3, No. 5, 2011
pp.21-24

ACTIVE OBJECT DESIGN PATTERN

Łukasz Górski

The student of the 2nd year of the Computer Science
The Faculty of Mathematics and Computer Science
Nicolaus Copernicus University in Toruń
ul. Chopina 12/18
87-100 Toruń
e-mail: lgorski@mat.umk.pl

Abstract: *Abstarct: Parallelization of software plays nowadays a major role in software efficiency increase. The paper aims to present an active object design pattern and to point out its usefulness in parallel programs design. The ProActive system is also roughly presented, together with the implementation of discussed design pattern.*

Keywords: Concurrent programming, functional programming, active object, ProActive, confluence

1. INTRODUCTION

It's a cliche to say that currently – as the processor clock frequency growth has slowed down – the main method for software efficiency increase is its parallelization [1]. However, software parallelization is not easy. A programme using parallelism is not a sequence of individually executed commands, which are relatively easy for analysis, but is comprises of numerous concurrently executed operations.

2. THREADS AND LOCKSA

The most fundamental tools available in programming languages such as threads and locks , are tools of very low level. Moreover, their use results in elimination of the sequential programme properties, such as: comprehensibility, predictability and determinism [2]. When a software developer uses those mechanism, he or she is additionally burden with the necessity to ensure appropriate synchronisation of individual threads. While use of lock mechanisms results in such problems, like [1] [3]:

- correctness of two separately analysed functions in which locks were used does not mean that the code using those functions is correct; to illustrate that problem, one can consider the following example, noted in pseudocode for simplification:

```
global a, b;
function1 () {
   lock (a); /* 1 */ lock(b);
   /* operations... */
      unlock(b); unlock(a);
    }
function2 () {
   lock(b); lock(a);
   /* operations... */
   unlock(a); unlock (b);
}
```

Obviously, when analysed separately, functions behave as expected. However, if they are called successively in the following way: function1(); function2(), this may cause potential deadlock. The problem is caused by (potentially possible) expropriation of the function function1 at place marked (1) as well as by calling the code of the function function2, and as consequence: causing the situation, when

function1 waits for release of the object b, while function2 – of the object a,

- use of locks assumes that the developer shall always obey the discipline, i.e. shall: observe the convention assuming that access to resource shared by threads shall be synchronized each time, i.e. appropriate locks shall be applied and released each time such resource is read or written. Maintaining such convention, particularly in case of groups, can be very difficult,
- the last problem of design nature is the fact that locks are used globally, i.e. each code fragment using shared resource should comply with appropriate access protocol; as a consequence, it is impossible to specify exact code fragment responsible for blocking access as it is distributed over the entire programme – which obviously makes the analysis even more difficult.

3. ASYNCHRONOUS CALLS, FUTURES

As a result of the aforementioned problems, it is necessary to introduce higher level programming abstractions [1] [4]. Threads represent only "sequential processes that share memory" [5]. They neither force use of good practises or prevent use of bad practices thus causing aforementioned problems. Therefore it is recommended to use higher level abstractions using mechanisms such as: asynchronous calls and futures.

Both concepts are used in the active object pattern, so it is purposeful to describe them roughly here.

Asynchronous call of a function (method) assumes that it does not block operation of the calling thread. It simply continues operation and the method works concurrently in a separate execution thread.

Whereas the result of asynchronous operation can be achieved using future object. Generally it gives access to one operation – get – which gets the result of the asynchronous call; if it is impossible to get the result (because the asynchronous operation has not been finished yet), the thread getting the result is locked until the result is accessible. It is worth noting that, retrieving the future object does not have locking character itself.

Use of the aforementioned structures is presented in the following programme developed in Java language:

```
class Foo implements Callable<Integer>
{
    public Integer call() {
        Thread.sleep(3000);
/* simulation of the load operation */
        return 42;
    }
}

public class Main {
    public static void main(String[]
args)
{
        FutureTask task = new
FutureTask(new Foo());
/* creating future object */

Executors.newSingleThreadExecutor().
submit(task);
/* execution of the operation in a
separate thread, main thread is not
locked and can execute other operations
*/

System.out.println (task.get());
/* (locking) retrieval of the future
object value */

}
}
```

4. ACTIVE OBJECT

First of all, active object pattern assumes that method execution shall be separated from its calling. The intention of that separation is to facilitate synchronous access to shared resources by methods called in different execution threads [6] [7]. Active object has its own execution thread as well as a message queue. Method is called asynchronously: i.e. it does not lock the calling thread but places appropriate message in the message queue of the active object. They are handled sequentially and managed by the scheduler, so messages do not have to be handles in order of their placement in the queue but use of different handling policies is also possible. Whereas the value returned by (asynchronous) calling of the method can be retrieved using futures objects.

The chart of classes implementing the active object pattern was shown in the diagram UML (acc. to [8] [6]).

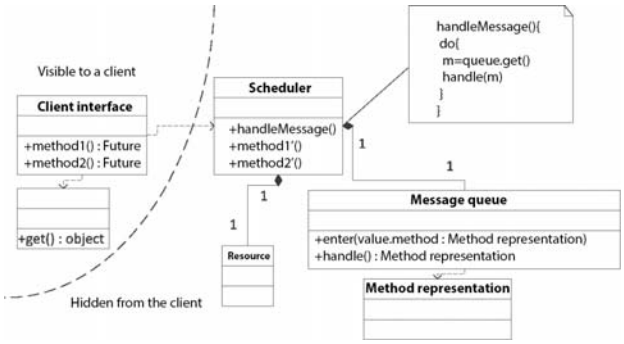

Figure 1 UML diagram of the active object pattern.

Thus the pattern includes the following components:

- Client interface – operated on the client side; creates an object representing the call of an appropriate method and placing it in an appropriate message queue of the active object; returns future object enabling access to the value returned by the method; it constitutes implementation of the design pattern proxy [9],
- Method representation – the component constructed by the client interface, constituting the abstraction of the method call, which is placed in the message queue of the active object,
- Message queue – includes all calls of methods for a given active object.
- Scheduler – calls individual methods represented in the message queue, according to the assumed policy,
- Resource – represents resource, access to which is modelled using active object pattern,
- Future – gives access to the value returned by the method call.

5. FUNCTIONAL LANGUAGES, PROACTIVE AND CONFLUENCE

Functional languages are characterized by high parallelization potential [1] [10]. Programmes developed in such languages like Haskell or OCaml may be, de facto, parallelized totally without the software developer interference. It results from the fact that those languages assume use of immutable objects, while operation performed on those objects are not associated with any side effects. . Thanks to absence of side effects, operations making up the programme may be executed in any order and may be freely interleaved. Hence there is no need to synchronize individual execution threads as they cannot interfere to each other at all. That property is called confluence . It is no doubt that such property significantly facilitates design and analysis of concurrently operating programmes.

This part of the paper includes some remarks regarding theoretical implications of some implementation of the active object pattern, i.e. that used in ProActive system [9]. ProActive is a Java language library, that facilitates concurrent and distributed programming, provides access to elements associated with data protection and migration [11]. It has evolved from the library which is the implementation of the active object pattern described in theoretical studies of Caromel i Henrio [10]. Currently it is categorized as middleware used while working with computational grids.

Theoretical grounds for active object implementation in ProActive is provided by ASP calculus, which constitutes formalization and enables accurate studies on active object properties. More detailed description of it can be found in [10], or a brief description in [12].

First of all, it should be noted that implementation of the active object pattern used in ProActive is characterized by certain differences from the pattern presented above. It assumes that the application is structured into so called subsystems. A subsystem consists of a single active object and some (≥ 0) passive objects (in practice: "common" Java objects without own execution threads and message queues). Only active objects are visible beyond the subsystem. Passive objects belong to certain subsystems, but if they are moved to other subsystems (by calling active objects' methods from another subsystem, within which they are transferred as arguments), deep copy mechanism is used. Only active objects are transferred using reference.

The consequence of such solution is strict separation of individual subsystems. And – in consequence – ensuring confluence properties in appropriate conditions. So the order of calling different methods for different active objects does not influence programme operation, which results from isolation of the methods. That order has little influence on the client side as well – as methods are called asynchronously and they return future objects, hence do not lock client threads. On the other hand, of course, that property is not maintained when methods called for the purpose of the active object modify its state (change values of the object fields). In such a case, results obtained as a result of the call of methods of a given active object depend on the order they were handled by the scheduler.

6. CONCLUSION

Sutter [3] presented very picturesque analogy. He stated that programming using locks is similar to structural programming using goto command. However, the kind of abstraction we need for concurrent programming should correspond to relation between object-oriented programming and structural programming.

It seems that the active object pattern can be regarded as such a solution. Use of the discussed pattern facilitates programming of parallel applications by elimination of the necessity to synchronize the access to shared resources (on the client side). It also enables quite easy use of multi-processor and multi-core architecture of computer systems – all you need is to assign a single active object to a single processing unit [6]. Analogous method is used also in development of distributed processing software (active object mapping – processing node – is sufficient here, taking into account multi-processor character of such nodes) [11]. Active object patterns can be applied in fields where applications of high responsiveness are required, and the architecture capable of supporting multiple independent tasks is desired: e.g. when creating graphic user interfaces or web services [13].

Java developers can use its implementation within advanced project, such as ProActive. They can also use sample code presented in [6]. The paper [7] includes the discussed pattern implemented in C++ programming language. Developers using other languages may use diagrams and assumptions described in the study herein as well as information available in [8] and [6].

References

1. H. Sutter i J. Larus, „Software and the Concurrency Revolution," [Online]. Available: http://queue.acm.org/detail.cfm?id=1095421.
2. E. A. Lee, „The Problem with Threads," [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf.
3. H. Sutter, „The Trouble with Locks," [Online]. Available: http://drdobbs.com/cpp/184401930.
4. H. Sutter, „The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm.
5. H. Sutter, „Use Threads Correctly = Isolation + Asynchronous Messages," [Online]. Available: http://drdobbs.com/high-performance-computing/215900465.
6. R. G. Lavender i D. C. Schmidt, „Active Object: an Object Behavioral Pattern for Concurrent Programming," w Proceedings of the Second Pattern Languages of Programs conference in Monticello, Illinois, September 6-8, 1995.
7. H. Sutter, „Prefer Using Active Objects Instead Of Naked Threads," [Online]. Available: http://drdobbs.com/go-parallel/article/showArticle.jhtml?articleID=225700095.
8. Strona główna systemu ProActive," [Online]. Available: proactive.inria.fr.
9. E. Gamma, R. Helm, R. Johnson i J. Vissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
10. D. Caromel i L. Henrio, A Theory of Distributed Objects, Springer, 2005.
11. D. Caromel, D. Christian, A. di Constanzo i M. Leyton, „ProActive: an Integrated platform for programming nad running applications on Grids and P2P systems," Computational Methods in Science and Technology, nr 12, 2006.
12. D. Caromel, H. Ludovic i B. P. Serpette, „Asynchronous and Deterministic Objects," w Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Nowy Jork, ACM New York, 2004, pp. 123-134.
13. H. Sutter, „The Pillars of Concurrency," [Online]. Available: http://drdobbs.com/architecture-and-design/200001985.
14. International Standard ISO/IEC 14882:2011. Programming Languages – C++.
15. F. Baader i T. Nipkow, Term Rewriting and All That, Cambridge: Cambride University Press, 1998.