e-Informatica Software Engineering Journal, Volume 8, Issue 1, 2014, pages: 27–37, DOI 10.5277/e-Inf140102

The Use of Aspects to Simplify Concurrent Programming

Michał Negacz*, Bogumiła Hnatkowska*

*Faculty of Computer Science and Management, Institute of Informatics, Wrocław University of Technology

michal@negacz.net, bogumila.hnatkowska@pwr.edu.pl

Abstract

Developers who create multi-threaded programs must pay attention to ensuring safe implementations that avoid problems and prevent introduction of a system in an inconsistent state. To achieve this objective programming languages offer more and more support for the programmer by syntactic structures and standard libraries. Despite these enhancements, multi-threaded programming is still generally considered to be difficult.

The aim of our study was the analysis of existing aspect oriented solutions, which were designed to simplify concurrent programming, propose improvements to these solutions and examine influence of concurrent aspects on complexity of programs.

Improved solutions were compared with existing by listing differing characteristics. Then we compared classical concurrent applications with their aspect oriented equivalents using metrics.

Values of 2 metrics (from 7 considered) decreased after using aspect oriented solutions. Values of 2 other metrics decreased or remained at the same level. The rest behaved unstably depending on the problem. No metric reported increase of complexity in more than one aspect oriented version of program from set.

Our results indicate that the use of aspects does not increase the complexity of a program and in some cases application of aspects can reduce it.

1. Introduction

Multi-core processors and supporting them systems are widely used at home. It is expected that the number of available cores will continue to grow in the next years [1].

The importance and number of programs that run concurrently has increased with the advance of technology. However, support for multi-core systems forces the use of concurrent programming techniques that are different from those known from single-threaded applications.

Aspect-Oriented Programming is a programming paradigm proposed by Gregor Kiczales. Its purpose is to enable and support a developer in separation of intersecting concerns and their modularization [2]. A costs of development and maintenance of concurrent programs can be reduced if a concurrent behavior is implemented in a modular manner, with minimum changes to an original source code.

The aim of this study is to analyze existing aspects, which solve concurrent programming problems, to propose improvements of existing mechanisms and the construction of a library that implements the existing solutions with the proposed improvements. The created aspect library is available at [3].

The library was used to implement typical programming problems and these implementations were compared with classical non-apsect solutions with the use of metrics. Then we answer the research question: *Does the use of aspects to concurrent programming reduce the complexity of application?*

The remainder of this paper is structured as follows. In section 2 we list the problems that a programmer may encounter when developing a concurrent application. Then, section 3 briefly describes previous research in the field of concurrent programming with aspects. In section 4 we present the use of aspects for concurrent programming and show the introduced improvements. Section 5 presents the results of complexity comparison of the solutions with aspects and without them. After that, in section 6, we present the conclusions and future work.

2. Problems of Concurrent Programming

When designing concurrent programs, in addition to the traditional issues related to the design, one have to deal with a parallel part of an application. That is how tasks are divided between available resources and how to communicate and synchronize them with each other. Typical problems occurring in concurrent programming are:

- Code scattering [4–6].
- Code tangling [4–6].
- Deadlocks [7–9].
- Livelocks [7–9].
- Starvation [7, 9].
- Race conditions [9].
- Synchronizing access to shared resources, which consists of [9]:
  - Restriction of simultaneous access;
  - Visibility of data;
  - Publication of objects.

With these problems, reuse, debug or change the functionality of existing components become a difficult task [4,6,10]. Moreover, because of the concurrent code scattering between the various components, the understanding of the whole structure of concurrency in application is also tough [6]. Costs of developing and maintaining concurrent applications can be reduced if the concurrency is added in a modular manner, with the least possible changes to a code.

3. The Use of Aspects in Concurrent Programming

3.1. Asynchronous Method Execution

Laddad in his book [11] presented the *Worker object creation* pattern. In his solution an aspect is responsible for creating an anonymous class of type *Runnable*, which wraps an original method call. To use his solution, a programmer should define a pointcut in the aspect, which indicate the method for asynchronous execution. For each call the aspect creates an instance, which is passed to a new thread. As a result, instead of a direct synchronous execution, it is moved to a separate thread.

Cunha et al [4] proposed an improved solution. Unlike the previous, the presented mechanism allows threads, which are created in the aspect, to be assigned to a specific group of processes other than the current one. Programmer can optionally define a pointcut, where the current thread waits for spawned threads. It is also possible to define pointcuts for the interruption of thread. In addition, instead of explicitly declare a method as a pointcut, it is possible to give an asynchronous behavior only by marking it with an annotation.

Listing 1. Asynchronous method execution.

```
@Asynchronous
void method() {
    // instructions
}
```

Hohenstein and Gleim also presented their own version of an asynchronous method execution [10] (listing 1). The authors recommended to perform concurrent code in the thread pool instead of creating a new thread for each execution. Concurrent executions are then limited to the upper limit of threads and do not reach the physical limits of the machine. When pool is used, one have to take into account the necessity of closing it. In the paper [10] authors proposed to use an additional annotation that indicates the place where the pool should be closed.

3.2. Asynchronous Method Execution which Returns a Result

A separate mechanism has been proposed for a concurrent execution of the methods that return a result. In the solution proposed by Cunha et al [4], there are two pointcuts. The first pointcut defines place where a calculation method is invoked, while the second indicates location where a result is used. A thread that calls the method will be blocked at the second pointcut as long as the method does not return the result. The authors also mentioned a possibility of creating a fake object as the result, which represents it until it is not available.

Hohenstein et al [10] also created a separate aspect for methods that return a result. They noted that an exception thrown from a Future object requires unwrapping, which is an additional effort imposed on a programmer. They found that it could be possible to solve this problem with an aspect, which uses a generic type to represent the result (listing 2). However, in the examples presented by them one can not see the way in which they achieve this unwrapping behavior. As in the previous case, also in this an annotation can be used.

Listing 2. Asynchronous method execution which return a result.

```
@Asynchronous
Result<Object> method() {
    Object object = // create an object
    return object;
}
```

3.3. Asynchronous Execution of Recursive Methods

A solution proposed for asynchronous execution method with a result works well for recursive calls. Its major disadvantage is that it creates many threads – one for the root call and one for each of recursive method calls. a better solution gives Fork/Join Framework, which is a part of Java since version 7. Hohenstein et al [10] proposed to use this framework with an aspect. To simplify its application one can use an annotation. The aspect uses two pointcuts – the first captures the root call and the second recursive calls. In this case the generic type is also used to obtain the results of calculations.

3.4. Barrier

Cuncha et al [4] proposed an aspect oriented mechanism to implement a barrier. Aspect uses two pointcuts – both define methods where, respectively, the first blocks the thread before and the second after the method execution. The barrier can be added by marking the appropriate method with an annotation. The programmer should specify the number of threads that barrier will stop in parameter of the annotation. Optionally he may provide the name of the thread group, to which stopped threads belong.

3.5. Resource Synchronization

Cuncha et al [4] suggested two ways to simplify a resource synchronization at the method level. The first solution wraps intercepted method call into a Java synchronized block. The aspect provides two possibilities – the first uses a target object as the monitor, while the second uses an aspect object. The second resource synchronization solution allows a thread to only read or write to shared resources. This distinction allows for simultaneous multiple readings, but only one single write to the resource. It is possible to use an annotation for easier determination of synchronized methods.

Hohenstein and Gleim also studied the problem of resource synchronization [10]. They found that blocking can be dangerous and prone to errors due to forgetting to release a lock. An aspect can solve this problem and ensure the final release of any lock. In the proposed solution the following annotation is used *@RWProtect (reads = { "resourceA" }, writes = {"resourceB", "resourceC" })*. Parameters of this annotation are resource identifiers in the aspect. The *@RWProtect* annotation specifies resources to read and write in order to coordinate concurrent access. If different annotated methods reference to the same resource, their access is synchronized – simultaneous reading is allowed at the same time, but writing excludes other writings and readings. Locks at resources are always applied in a specific order to avoid deadlocks. However, in the proposed aspect, despite of use of non-blocking map, there is a race condition. In addition, in certain circumstances a thread starvation may appear, when the thread is waiting for a lock.

3.6. Conditions of Method Execution

Execution of some methods may depend on the state of an object. Cuncha et al [4] proposed *waiting guards* mechanism, which is based on an aspect. When the condition is not met, a thread is blocked until there is an action that changes the state of the object, which will trigger a condition reevaluation. Additionally, the reevaluation may occur after a defined timeout. The concrete aspect defines pointcuts, which indicate methods for which conditions are checked and a method that can change the state of the object, forcing the reevaluation of conditions.

3.7. Active Object

The active object pattern separates method call from its execution. It allows multiple threads to access data which is modeled as a single object. Traditional implementations of the pattern are divided into three layers. The first layer contains a client object, which makes a call, the second layer includes a mechanism to transfer the call to a target object and the third layer is the target active object running in a separate thread, which is still waiting for method calls [12]. The implementation of the active object in an aspect way [4] moves the second and the third layer to aspects. In addition, this solution makes participating classes unaware of their roles in the pattern. To give an object the behavior of the active object one should use specified annotation.

4. Proposed Solution

4.1. Asynchronous Method Execution

To perform an asynchronous method execution, a programmer should mark it with the annotation *@Asynchronous*. By default, the method is performed in a thread pool created by *Executors.newCachedThreadPool()*. All method calls marked with this annotation will be executed in one common pool shared for the entire program. Methods that are annotated with the optional parameter *standalone* = *true* are executed in their own, single threaded, private pool that is immediately closed after the call. The common thread pool can be controlled by the annotation *@Startup*. The pool is created before calling the method marked with this annotation.

Annotation attributes which can be modified are:

- threadPool: ThreadPool – type of pool:
  - FIXED – pool with a fixed number of threads coming from the method *Executors.newFixedThreadPool(...)*. Number of threads is taken from the parameter maxThreads.
  - CACHED – pool with a dynamic number of threads coming from the method *Executors.newCachedThreadPool()*.
  - CUSTOM – pool with the characteristics defined by a programmer.
- maxThreads: int – the number of threads for FIXED type pool and maximum number of threads for CUSTOM type pool. If not specified, it is assumed to be a maximum value from the set {1, the number of available processors - 1}.
- coreThread: int – the working number of threads for CUSTOM type pool. If not specified, the default value is calculated from the formula 1.
- timeout: int – time after an unused thread is killed. Attribute is used exclusively by the CUSTOM type pool and it is measured in seconds. The default value is 60 seconds.

– shutdownAfterMainMethod: boolean – attribute specifies whether to automatically close the pool after leaving a main method of a program.

$$\text{coreThread} = \text{maxThreads}/3 + 1 \qquad (1)$$

The annotation *@Shutdown* is used for closing the common thread pool. After completing marked by this annotation method, the pool will not accept new tasks. The attribute *now = true* results in an immediate closing the pool, calls waiting in a queue will not be executed.

If the method declares an opportunity to throw controlled exceptions, they are softened by an aspect. This facility is dictated by a lack of an exception handling capabilities, which will be thrown in a separate thread. The code placed in the *catch* part of the *try {} catch {}* structure would be unreachable (listings 3 and 4).

Listing 3. Example of an unreachable code.

```
@Asynchronous
void method() throws Exception {
    // ...
}

void callMethod() {
    try {
        method();
    } catch(Exception e) {
        // this code cannot be reached
    }
}
```

To specify where asynchronous method should join to a calling thread, methods can be annotated with *@JoinBefore* or *@JoinAfter* annotations.

Listing 4. Asynchronous method execution in a pool.

```
@Startup(threadPool = ThreadPool.FIXED,
    maxThreads = 3,
    shutdownAfterMainMethod = true)
@Asynchronous
void method() throws Exception {
    // instructions, which we want
    // to call asynchronously
}

void callMethod() {
    method(); // there is no need for
              // handling thrown
              // exception
}
```

Table 1 compares features of previous aspect oriented solutions with our proposal.

4.2. Asynchronous Method Execution which Return a Result

The proposed aspect oriented solution considers two cases. The first case are methods that return an object type, which is not final. As in the case of methods that do not return a result, it is sufficient to mark a method with the annotation *@Asynchronous*. This method will immediately return automatically created *Proxy* object (listing 5). Any call to a method on this object is delegated to the correct result and if it is not yet available, an execution is blocked until it is available. The second case is a situation where the return type is final. In this case a change in a structure of a program is needed. The function result should be wrapped with a generic type. Methods marked with the *@Asynchronous* annotation execute in the same thread pool that methods, which do not return a result. When, during the execution of the method, it will encounter an exceptional situation, an exception will be thrown in its original form when one tries to fetch the result. Aspects are not capable of dynamic declaring new exceptions to methods, so special property of generic type has been used to work around this limitation.

Listing 5. Example of an asynchronous method execution with a proxy as result.

```
@Asynchronous
ExampleObject method() throws
                ExampleException {
    // instructions, which we want
    // to call asynchronously
}

void callMethod() {
    try {
        ExampleObject proxy = method();
                        // asynchronous
                        // method call

        // instructions that you want
        // to do before the result
        // is available

        String something =
                proxy.getSomething();
```

Table 1. Comparison of asynchronous method execution solutions.

Property	Previous solution	Proposed solution
Usage of a thread pool	No [11], No [4], Yes [10]	Yes
The need to handle exceptions in a calling code	Yes	No

```
    } catch (ExampleException e) {
        // exception handling
    }
}
```

In table 2 we presented comparison of features of previous aspect oriented solutions with our proposal.

4.3. Asynchronous Execution of Recursive Methods

A method may be performed recursively in three ways. Each of them requires marking the method with the annotation *@AsynchronousRecursively*. For each recursive call of the marked method an aspect creates a separate Fork/Join pool. It is possible to control the number of threads in the pool by the parameter *threads = 2*. The default number of threads is equal to the number of available processors.

The first possibility is to use generic object *Result*, which wraps an original result returned from the method. In order to better use the *Fork/Join* pool, in the second possibility, one can use the method *Result.scheduleWith(...)* proposed in [10]. Presented in this article method can take only one parameter. We have extended it to any number of parameters. It creates a fork for each result passed, but the result object, on which the method was called, is calculated in a current thread. However, a disadvantage of this solution is the need to change the program source code and adding the call which is not directly related to the application logic. Last, the third possibility is to use auto generated *Proxy* objects (listing 6). This case allows one to make an application completely independent from the library.

Methods, which are performed recursively, use the same concept of exception handling as asynchronous methods that return result. This means that exceptions will be thrown unchanged when one tries to fetch a result.

Listing 6. An aspect oriented calculation of 10th Fibonacci number with a proxy object.

```
void callMethod() {
    Number proxy = fibonacci(10L);

    // instructions that you want to do
    // before the 10th fibonacci number
    // is available

    Long result = proxy.longValue();
}

@AsynchronousRecursively
Number fibonacci(Long n) {
    if (n <= 1) {
        return n;
    } else {
      return fibonacci(n - 1).longValue()
        + fibonacci(n - 2).longValue();
    }
}
```

Comparison of features of previous aspect oriented solutions with our proposal is presented in table 3.

4.4. Barrier

To implement a barrier in an aspect oriented approach it is sufficient to mark a method with annotations *@BarrierBefore* or *@BarrierAfter* with the number of threads that the barrier stops. Barriers can also be named with the *name* parameter of the annotation. The default name of the barrier is *thisMethod*, which means that the barrier is assigned only to the annotated method. If for the one named barrier there are many annotations with different number of threads, then created barrier has a limit indicated in the first method, which is called in a flow of a program.

Table 2. Comparison of asynchronous method execution solutions which return a result.

Property	**Previous solution**	**Proposed solution**
Usage of a thread pool	No	Yes
Usage of a proxy object	No	Yes
The need to unwrap exceptions	Yes [4], No [10]	No

Table 3. Comparison of asynchronous execution of recursive methods solutions.

Property	**Previous solution**	**Proposed solution**
The need to use specific methods (scheduleWith(...))	Yes	No
Usage of a proxy object	No	Yes

The problem may be a situation in which a method uses two or more barriers. Then it is not known which barrier a thread has to consider first. In this case, a developer must determine an order by creating an artificial cascade of methods marked with barrier annotations (listing 7).

Listing 7. A cascade of two methods with two barriers.

```
@BarrierBefore(value = 3,
               name = "firstBarrier")
void method() {
    otherMethod();
}

@BarrierBefore(value = 3,
               name = "secondBarrier")
void otherMethod() {
    // instructions executed after
    // reaching "firstBarrier"
    // and "secondBarrier" by 3 threads
}
```

The solution does not include restrictions for groups of threads, because they are obsolete and it is not recommended to use them [7].

Table 4 compares barrier features of previous aspect oriented solutions with our proposal.

4.5. Resource Synchronization

To synchronize the whole method it is sufficient to mark it with the annotation *@Synchronize.* This will perform a synchronization on a lock assigned to a current object or an object of class *Class* in the case where the method is static. If one wants to synchronize the method using an another lock, then he should specify its identifier. To facilitate the connection of identifiers with resources, they should be marked by *@SharedResource* with a resource name (listing 8), although for proper operation of a program it is not required. Resource identifiers are global to the program. If one wants to synchronize multiple resources, then their identifiers should be listed in the annotation. An aspect acquires locks always in the same order, so the order of identifiers in the annotation is not important. To set up locks, that distinguish between reading and writing to resources, identifiers should be specified in appropriate parameters. Default parameter assumes two types of synchronization.

Following keywords can be also used as a name of identifier in the *@Synchronize* annotation:

- this – a lock is assigned to a current object or an object of class *Class.* The behavior is analogous to precede a method with the word *synchronized.*
- this.name – a lock is assigned as in the case *this*, but also supplemented by the given *name.* The behavior can be understood as embracing the body of a method with the synchronized block with an object field as the argument.
- global – a global lock.

Listing 8. Resource synchronization

```
@SharedResource("sharedResource")
Object sharedResource;

@Synchronize(reads = "sharedResource")
void readResourceMethod() {
    // instructions that read resource
```

Table 4. Comparison of barrier solutions.

Property	Previous solution	Proposed solution
A possibility of sharing barrier between the methods and objects through its naming	No	Yes
A possibility to restrict a barrier only to a select group of threads	Yes	No

```
}

@Synchronize(writes = "sharedResource")
void writeResourceMethod() {
    // instructions that write
    // to resource
}
```

In table 5 we presented comparison of features of previous aspect synchronization solutions with our proposal.

4.6. Conditions of Method Execution

In the proposed aspect oriented solution it is sufficient to mark a method with the annotation *@WaitUntilPreconditions*, then define precondition methods (with the annotation *@Precondition*) and a method for re-evaluation of the conditions (the annotation *@EvaluatePreconditions*). A thread, which tries to execute the method marked with the annotation *@WaitUntilPreconditions* will be slept until all preconditions are not met. Evaluation of conditions can be automatically executed at a time interval set in the annotation parameter *@WaitUntilPreconditions(waitingTime = 1000)* in milliseconds or by calling from a program code the method marked with the annotation *@EvaluatePreconditions.* The precondition can be named and then the annotation *@WaitUntilPreconditions* could specify its identifier (listing 9). If the method is annotated with no parameters, then by default is assumed that all of conditions marked with *@Precondition* must be met in order to execution. As preconditions are considered only methods annotated with *@Precondition* and which return boolean expression.

Listing 9. Method execution after fulfilling preconditions.

```
private boolean state;

@WaitUntilPreconditions({
        "onePrecondition",
        "anotherPrecondition" })
public void method() {
    // instructions executed after
    // fulfilling the preconditions
}

@Precondition("onePrecondition")
public boolean precondition1() {
    return state;
}

@Precondition("anotherPrecondition")
public boolean precondition2() {
    return true;
}

@EvaluatePreconditions
public void notifyMethod() {
    state = true;
}
```

Comparison of features of previous aspect oriented solutions with our proposal is presented in table 6.

4.7. Active Object

In the proposed, aspect oriented solution to implement active object it is sufficient to mark a class with the annotation *@ActiveObject.* If the execution of a method has preconditions, one has to list its identifiers in the annotation *@GuardedBy* and to mark an appropriate predicate method with the annotation *@Precondition.* Marking the class with a parameter *terminateAfterMainMethod = true* will automatically close a thread of the active object after leaving a main method of a program.

Table 5. Comparison of resource synchronization solutions.

Property	Previous solution	Proposed solution
The ability to synchronize static methods	No	Yes
Mark resources with an identifying annotation	No	Yes
Keywords	No	Yes

Table 6. Comparison of precondition solutions.

Property	Previous solution	Proposed solution
Usage only a metadata from a program	No	Yes

Table 7. Comparison of active object solutions.

Property	Previous solution	Proposed solution
Full implementation of the pattern (guard conditions)	No	Yes
Automatically termination of the active object	No	Yes

Table 7 compares features of previous aspect oriented solutions with our proposal.

5. Comparison of the Applications

To be able to compare traditional and proposed solutions we found concurrent programs, which solve the classic problems:

- Dining philosophers problem [13]
- Producer – consumer problem [14]
- Calculation of the n-th Fibonacci number [15]

The next step was to write our own versions of the applications, which solve the above problems, using created aspects. After that we calculated selected metrics with the use of Chechstyle 5.7 and STAN 2.1.2 (see table 8). For all applications following count metrics were calculated:

- LOC/NCSS – Lines Of Code / Non Commenting Source Statements (Checkstyle)
- NOF/NOA – Number Of Fields / Number Of Attributes (STAN)
- NOM – Number Of Methods (STAN)
- TLC – Top Level Classes (STAN)

And the complexity metrics:

- CC – Cyclomatic Complexity (Checkstyle)
- DAC – Data Abstarction Coupling (Checkstyle)
- CFOC – Class Fan Out Complexity (Checkstyle)

Count metrics (LOC/NCSS, NOF/NOA, NOM, TLC) were chosen because of their quantitative representation of the complexity and additive behavior. CC is a classic measure of the complexity of methods. For this metric values below 7 are considered to be acceptable, while above this value metric indicate the need for refactoring. The motivation for choice of DAC and CFOC metrics was, that they measure the complexity of individual classes, they are able to demonstrate differences in relationships of classes. Both are supported by tools. The Checkstyle tool in a default configuration allows 7 for DAC and 20 for CFOC. For all selected metrics, the smaller is the value, the less is the complexity of the examined class.

where

- Pr1 denotes the dining philosophers problem
- Pr2 denotes the producer – consumer problem
- Pr3 denotes the calculation of the n-th Fibonacci number

Table 8. Comparison of programs using metrics

Metric	Pr1 (cla)	Pr1 (asp)	Pr1 (chg)	Pr2 (cla)	Pr2 (asp)	Pr2 (chg)	Pr3 (cla)	Pr3 (asp)	Pr3 (chg)
LOC	69	36	**-33**	86	63	**-23**	39	10	**-29**
NOF	6	4	**-2**	6	6	**0**	5	0	**-5**
NOM	7	7	**0**	11	13	**+2**	7	3	**-4**
TLC	1	2	**+1**	3	3	**0**	3	1	**-2**
CC	1.71	1.71	**0**	1.7	1.17	**-0.53**	1.33	1.5	**+0.17**
DAC	1	1	**0**	1.2	1	**-0.2**	1.33	0	**-1.33**
CFOC	2	1.5	**-0.5**	2.75	1.5	**-1.25**	1.33	1	**-0.33**

- cla denotes a classic version of application (downloaded from the Internet)
- asp denotes an aspect oriented version of application (written by us with the use of aspect library)
- chg denotes a change between an aspect and a traditional version
- Values of LOC, NOF, NOM, TLC were counted as a sum of metrics for all classes in the application
- Values of CC, DAC, CFOC were counted as means of metrics for all classes in the application

For each of three problems the number of lines of code and CFOC values are smaller in the aspect than in the traditional solution. For metrics NOF and DAC two aspect oriented programs are less complex than their traditional counterparts, while both versions of the third program are equally complex. For the remaining metrics (NOM, TLC, CC) in one problem the aspect oriented version is less complex, in the second problem the traditional and in third both versions are equally complex.

Aspect oriented versions are more complex in three cases. In the case where the number of methods is higher in the aspect than in the classical solution the increase is because of the need to create separate predicates method. In found classical dining philosophers solution, *Philosopher* class is nested and not considered by the metric, while in the aspect oriented version *Philosopher* is a separate class. In traditional, concurrent calculation of the n-th Fibonacci number, there are 4 more methods than in the aspect oriented solution. These methods mostly have CC metric value equal to 1, thus they are lowering the average. The maximum CC metric value is equal in both applications.

No metric had shown that in all three cases, the complexity of the aspect oriented solution was higher than a classic application. Also, there was no increase of complexity in more than one aspect oriented version of program per a metric.

In response to the research question, it can be concluded that the use of aspects to the simplification of concurrent programming does not increase complexity of a program and in some cases application of aspects can reduce it.

6. Conclusions

This paper presented an effort to develop an aspect library which simplifies concurrent programming. We improved the previously proposed solutions and presented new features. Then, we conducted research and have shown that the use of aspects may reduce the complexity of concurrent application.

In general, using aspects for the concurrent programming can improve selected maintainability sub-characteristics, i.e. analysability and modifiability. But maintainability also includes testability sub-characteristic. While the proposed aspects may help in understanding and implementing concurrent applications, an open problem is how to test a correctness of the solution.

It should be noted that our research was conducted on a small sample of programs. These programs are small applications and do not come from an industry. In addition, credibility of research is highly influenced by a quality of programs, both those created by the authors and those collected.

Therefore, in the future we are going to repeat the research with bigger number of programs. Moreover, we want explore the use of aspects in Proactor and Reactor concurrent patterns.

References

[1] B. Schauer, "Multicore processors–a necessity," *ProQuest discovery guides*, 2008, pp. 1–14.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming.* Springer, 1997.

[3] "concurrent aspects library," https://github.com/mnegacz/concurrent-aspects.

[4] C. A. Cunha, J. a. L. Sobral, and M. P. Monteiro, "Reusable aspect-oriented implementations of concurrency patterns and mechanisms," in *Proceedings of the 5th international conference on Aspect-oriented software development*, ser. AOSD '06. New York, NY, USA: ACM, 2006, pp. 134–145. [Online]. http://doi.acm.org/10.1145/1119655.1119674

[5] B. Harbulot and J. R. Gurd, "Using aspectj to separate concerns in parallel scientific java code," in *Proceedings of the 3rd international conference on Aspect-oriented software development*, ser. AOSD '04. New York, NY, USA: ACM, 2004, pp. 122–131. [Online]. http://doi.acm.org/10.1145/976270.976286

[6] J. L. Sobral, "Incrementally developing parallel applications with aspectj," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 116–116. [Online]. http://dl.acm.org/citation.cfm?id=1898953.1899048

[7] J. Bloch, *Effective Java (2Nd Edition) (The Java Series)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[8] B. Eckel, *Thinking in Java*, 3rd ed. Prentice Hall Professional Technical Reference, 2006.

[9] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice.* Addison-Wesley Professional, 2005.

[10] U. D. Hohenstein and U. Gleim, "Using aspect-orientation to simplify concurrent programming," in *Proceedings of the tenth international conference on Aspect-oriented software development companion*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 29–40. [Online]. http://doi.acm.org/10.1145/1960314.1960324

[11] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming.* Greenwich, CT, USA: Manning Publications Co., 2003.

[12] R. G. Lavender and D. C. Schmidt, "Active object – an object behavioral pattern for concurrent programming," 1995.

[13] "Dining philosophers problem implmentation," https://github.com/vonhessling/DiningPhilosophers.

[14] "Producer-consumer problem implementation," https://github.com/dcryan/Producer-Consumer.

[15] "Java Fork/Join for Parallel Programming," http://www.javacodegeeks.com/2011/02/java-forkjoin-parallel-programming.html.