Proceedings of the International Multiconference on
Computer Science and Information Technology, pp. 753 – 762

ISSN 1896-7094
© 2007 PIPS

Post object-oriented paradigms in software development: a comparative analysis

Adam Przybyłek

Department of Business Informatics, University of Gdańsk,
Piaskowa 9, 81-824 Sopot, Poland
adam.przybylek@univ.gda.pl

Abstract. The object-oriented (OO) paradigm has been popularised as a solution to the problems encountered with the structured paradigm. It facilitates the understandability, extensibility, reusability and maintainability of systems. However, years of experience and analytical studies have shown that this is only partially true, and that there are still issues which have never been successfully resolved in the OO paradigm. These issues arise whenever programmers need to deal with peripheral requirements which spread over a system. A simple modification of these requirements typically involves intensive changes in code. Over the last decade interesting and worthwhile work has been done on the subject of implementing peripheral requirements. Perhaps the most successful outcomes have been obtained by aspect-oriented programming and composition filters. The main goals of the paper are: (1) to present open problems for the OO paradigm; (2) to analyse two post-OO paradigms involved in confronting these problems; (3) to indicate a possible application of these paradigms.

1 Introduction

For many years now theorists have agreed that the best way to manage the complexity of the system is to separate the system concerns [15, 19, 23, 24]. A concern is a specific requirement or matter for consideration that must be addressed in order to satisfy the overall system goal [2, 7, 10, 12, 13, 16, 20, 23]. In this paper concerns are categori sed according to whether they (1) capture the main business requirements or (2) capture peripheral requirements or technical-level issues that affect the system as a whole. The first category of concern is relevant to the application domain and is termed a "core concern". The important property of core concerns is their ability to be separated at implementation level from other concerns of the same kind. On the other hand, "crosscutting concerns" play a supporting role and are required by some other concerns. Although they can be identified as distinct concerns, they are usually spread over the system and cannot be modularised into classes. Typical examples of crosscutting concerns are authentication, logging, synchronisation, transaction processing, failure handling, reliability, memory management, optimisation, performance tracking, real time constraints, storage management, data persistence, resource pooling and remoting. Figure 1 gives a visual depiction of concerns in a hypothetical application.

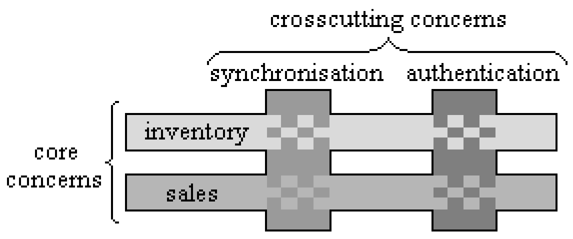

Fig. 1 . Intertwined concerns

"Separation of concerns" (SoC) refers to the ability to identify, encapsulate and manipulate parts of the software in order to deal with one important issue at a time [2, 9, 15]. It reduces software complexity and enhances understandability, adaptability, reuse and traceability throughout the development process [9, 19, 24]. Dijkstra [15] and Parnas [25] have suggested, that the best way to achieve SoC is through modularisation, the process of breaking the system into loosely-coupled modules that hide their implementation from each other. Individual modules can be relatively easily composed so as to meet the system requirements [10, 20, 23, 24]. This process, known as composition, is performed according to explicit statements about how the concerns should be put together.

The evolution of a software development paradigm is driven by the need to achieve a better SoC [15]. In software engineering the term "paradigm" is widely used to refer to the essence of certain software development processes. In spite of the fact that a paradigm refers to all the phases of system development, initial research usually originates in programming languages. New paradigms, therefore, occur in terms of programming, as in aspect-oriented programming (AOP).

Various paradigms have been studied that deal with system complexity and divide the systems into modules. Currently the most popular is the object-oriented (OO) paradigm. This provides techniques for reducing software complexity by introducing concepts such as abstraction, encapsulation, aggregation, inheritance and polymorphism. However, modularisation of each concern in an extensible manner cannot be achieved using the OO paradigm [10, 12, 20, 23]. Attempts to provide a solution to this problem have included AOP and composition filters (CFs). Each of these paradigms builds on all the advantages of the OO paradigm and overcomes some OO weaknesses. The term "post-OO paradigm" has therefore been introduced to refer to them.

The remainder of the article is structured as follows. Section 2 continues analysis of the motivations for post-OO paradigms. Section 3 and 4 introduce the principles of AOP and CFs respectively, both paradigms being illustrated with examples which focus on software reuse. Section 5 compares the paradigms under consideration and presents similarities and differences between them. Section 6 outlines the future work. Section 7 concludes the paper.

2 Weaknesses of the object-oriented paradigm

The OO paradigm was created from a desire to have constructs to represent real world objects. It has been claimed that: (1) OO systems support software evolution, extension, and modification; (2) OO systems are reusable and can be easily constructed from existing components; (3) OO systems are understandable to domain experts and developers [13, 17, 18, 26]. However, years of experience have revealed that the OO paradigm fails to deliver all its claimed benefits [21].

Classes are a powerful way of encapsulating and localising changes to the core concerns, but they fall short in addressing the crosscutting concerns which affect multiple places in the system. Symptoms of implementing crosscutting concerns in an OO environment are "code scattering" and "code tangling" [10, 12, 20, 23]. Scattering occurs when multiple fragments of code that all do the same thing (or that do closely related things) are found in more then one class. The term "code tangling" is often used to describe the situation where a class contains logic pertaining to more than one concern. Code tangling and code scattering together negatively affect software design and development in many ways: poor traceability, low productivity, low code reuse, poor quality and difficulty in evolution [23].

3 Aspect-oriented programming

In the OO paradigm everything is considered as an object, but not everything is an object either in the real world or in system development. For example, synchronisation, transaction processing and reliability are not objects [21]. They are crosscutting concerns that cannot be effectively modularised in classes. AOP was introduced to address these shortcomings. It does not replace the OO but instead complements and builds on what the OO already offers. The AOP allows crosscutting concerns to be implemented separately from core concerns and then merges both into an executable code [10, 11, 12, 20, 22, 23].

Crosscutting concerns, together with weaving rules, are implemented in a new unit of modularity called an "aspect". Aspects centralise functionality, which is distributed over the whole system in the OO model [20]. Weaving rules specify how to integrate core and crosscutting concerns in terms of pointcuts and joinpoints. A pointcut is a language construct used to define a set of joinpoints. Joinpoints are well-defined locations in the program flow at which a crosscutting concern needs to be merged with core concerns. Typical examples of pointcuts could be a method being called, an exception being thrown, or an object being created.

The code that is inserted upon reaching a particular joinpoint is called an advice. An advice represents the body of a crosscutting concern. There are three types of advice: "before", "after", and "around". A before advice executes a designated code just before matched joinpoints. An after advice runs after the execution of matched joinpoints. An around advice executes a defined code instead of the code in the joinpoints [11, 20]. Moreover, an advice is able to access parameters of any method in any class. The actual process of concern merging, also known as "weaving", uses aspects and classes to compose the final system. Core concerns no longer embed the logic of crosscutting concerns, which are isolated into aspects [23].

The AOP has moved into the forefront through the work [22] of Gregor Kiczales and his team at Xerox's PARC. The most mature aspect-oriented implementation available today is AspectJ, which is an extension to Java.

For the purpose of illustration, an implementation of a tracing concern is pr esented. A learning management system (LMS) has been deployed at a college. After some time, the suspicion is formed that unauthorised persons know lecturers' passwords and are modifying data in the system. As a result, it is decided that all data-changing SQL statements have to be watched. In order to do this, it is proposed that the existing application be extended by logging every DML statement that modifies the records, together with the date of the incident and the user name.

The original version of the application uses JDBC to access the database (Fig. 2). The key elements of JDBC API (in terms of the example presented) are the Connection interface, the Statement interface and the DriverManager class. DriverManager manages all the details involved in establishing the connection to a specified database. The established connection is returned by DriverManager::getConnection(..), which is a static method. In a typical application scenario the next step is creation of the Statement object. The Connection::createStatement() method is called upon to do this. The Statement object is associated with an open connection and used to send SQL statements to the DBMS. DML statements such as INSERT, UPDATE, DELETE are usually executed using the method Statement::executeUpdate(..).

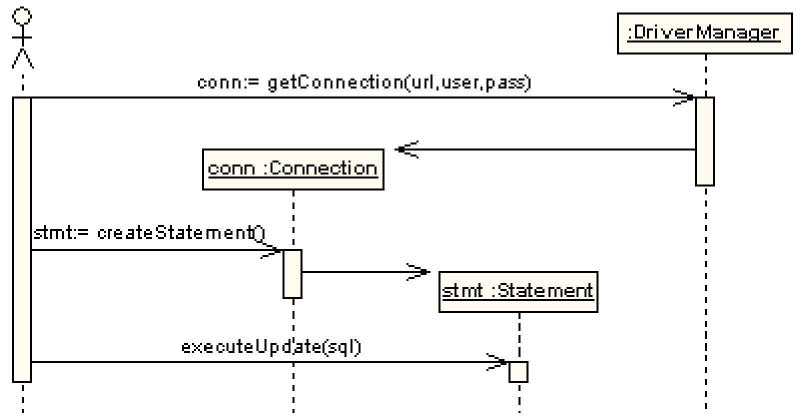

Fig. 2 . A typical usage scenario for accessing a database

In the conventional OO model, the best solution to the problem presented is based on a delegation-based reuse. A LogStatement class has been created to trace queries. This class implements the Statement interface and aggregates the statement object. All messages specified by the Statement interface are delegated to the statement object. Those which are able to modify the records (i.e. execute, executeUpdate) are redefined in order to implement an additional concern (tracing). The definition of the LogStatement class and an example of its application is presented on Listing 1.

```
public class LogStatement implements Statement {
  private Statement delegate;
  publ ic LogStatement(Statement st) {
    delegate = st;
  }
  private boolean isDML(String sql) {
    String tmp = sql.toUpperCase();
    return tmp.indexOf("UPDATE")>=0 ||
     tmp.indexOf("INSERT")>=0 || tmp.indexOf("DELETE")>=0;
  }
  private void log(String sql) throws SQLException {
    String user =
     delegate.getConnection().getMetaData().getUserName();
    System.out.println("["+new Date()+"]; "+user+"; "+sql);
  }
  public void cancel() throws SQLException {
    delegate.cancel();
  }
  public void close() throws SQLException {
    delegate.close();
  }
  public Connection getConnection() throws SQLException {
    return delegate.getConnection();
  }
  public ResultSet executeQuery(String sql) throws SQLException {
    return delegate.executeQuery(sql);
  }
  public boolean execute(String sql) throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql);
  }
  public boolean execute(String sql, int autoGeneratedKeys)
   throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql, autoGeneratedKeys);
  }
  public boolean execute(String sql, int[] columnIndexes)
   throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql, columnIndexes);
  }
  public boolean execute(String sql, String[] columnNames)
   throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql, columnNames);
  }
  public int executeUpdate(String sql) throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.executeUpdate(sql);
  }
  //other methods specified by the Statement interface
}
```

Listing 1. LogStatement class definition

The suggested implementation limits the number of changes in the existing source code. However, in order to fulfil the new requirement, every statement object returned by Connection::createStatement() has to be wrapped with the LogStatement. The LogStatement implements 40 methods. Of these 1 is a constructor, 8 methods perform tracing and delegating, 29 methods are used for delegating only and 2 methods implement tracing only. It is apparent that the tracing concern (marked code) is scattered through and tangled with the business logic. A better solution can be obtained by using AOP and implementing a new functionality as an aspect (Listing 2).

```
public aspect DMLmonitoring {//1
  pointcut DMLexecute(String sql, Statement st)://2
   call( * Statement.execute*(..) )
   && args(sql,..) && target(st);
  after(String sql, Statement st) returning (Object o):
   DMLexecute(sql,st) {//3
    String user = "";
    try {
      user=st.getConnection().getMetaData().getUserName();
    } catch (SQLException e) {};
    String tmp = sql.toUpperCase();
    if( tmp.indexOf("UPDATE")>=0 || tmp.indexOf("INSERT")>=0
     || tmp.indexOf("DELETE")>=0 ) {
      if (o instanceof Integer) {
        int i = ((Integer)o).intValue();
        sql += "; ("+i+")";
      }
      tmp = user + "; " + sql;
      System.out.println("[" + new Date() + "]; " + tmp);
    }
  }
}
```

Listing 2. DMLmonitoring aspect definition

The fundamental difference between a conventional and an aspect-based solution is modularisation of the tracing concern. Instead of mixing concerns in a single class, the aspect is defined (1). The joinpoints at which weaving should be made are identified as the places where the update methods are called (2). An after-returning advice (3) runs after a successful return from the execution of a matched joinpoint. The statements inside this advice implement the tracing concern and are applied whenever the records are updated, deleted or inserted.

4 Composition Filters Object Model

The Composition Filters Object Model (CFOM) was defined in the late 1980s by Aksit and Tripathi [1, 3] as a modular extension to the OO model. It offers composable techniques for specifying the behaviour of the class [4]. The motivation for the CFOM was difficulty in expressing any kind of message co-ordination in the conventional OO model [14].

The CFOM can be presented as two partitions: an interface layer and an impleme ntation layer [4]. The implementation layer can be thought of as a conventional OO model. It can be extended by the interface layer. To achieve this, the interface layer contains an arbitrary number of input and output filters. Incoming messages pass through the input filters and the outgoing through the output filters [5, 8, 14]. Each message has to pass through all the filters defined until it is dispatched. The filters are able to inspect and manipulate messages; messages may, for example, be discarded or redirected to other objects. A filter definition specifies a filter type and filter guards. A filter accepts a message if any of the filter guards matches the message. If not, the message is said to be rejected. Each guard has the form:

condition => selector_expression

A guard is matched if (1) the condition evaluates to true, and (2) the selector ex pression matches the message. The filter type determines the semantics associated with acceptance and rejection of messages [4], [5].

This paper presents the CFOM on a classical producer-consumer problem. In a producer-consumer dilemma two processes (or threads), one known as the “producer” and the other called the “consumer”, run concurrently and share a common, fixed-size buffer. The producer generates items and places them in the buffer. The consumer removes items from the buffer and consumes them. However, the producer must not place an item into the buffer if the buffer is full, and the consumer cannot retrieve an item from the buffer if the buffer is empty. Nor may the two processes access the buffer at the same time to avoid race conditions. The producer-consumer problem can be generalised to have multiple producers and consumers. The UML's sequence diagram in Fig. 3 presents a typical interaction between producer, consumer and buffer.

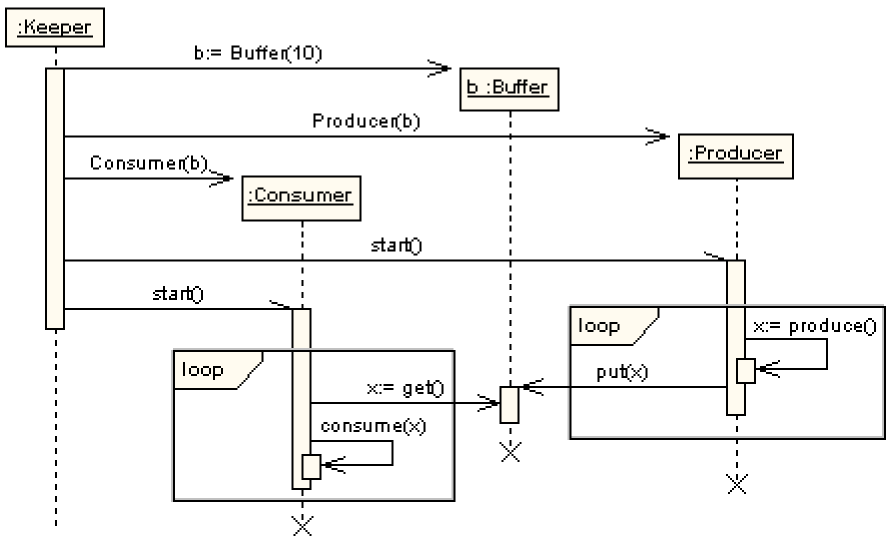

Fig. 3 . The producer-consumer interaction

Listing 3 gives a definition of a buffer class in Java. The OO solution tangles business logic together with the synchronisation concern. Moreover, the synchronisation code is scattered through the methods responsible for accessing the buffer. In the result the put(..) and get() methods contain fragments of code for cooperation synchronisation.

```
public class Buffer {
  private int[] buf;
  private int counter;
  public Buffer(int capacity) {
    buf = new int[capacity];
  }
  public synchronised void put(int x) {
    while (counter == buf.length) try {
      wait();
    } catch (InterruptedException e) {
      System.err.println( e.getMessage() );
    }
    buf[counter++] = x;
    notifyAll();
  }
  public synchronised int get() {
    while (counter == 0) try {
      wait();
    } catch (InterruptedException e) {
      System.err.println( e.getMessage() );
    }
    int tmp = buf[--counter];
    notifyAll();
    return tmp;
  }
}
```

Listing 3. Buffer definition in the OO model

The alternative implementation is shown on Listing 4. The buffer is defined in pseudo-Java code using a simplified version of CFs.

```
public class Buffer {
  private int[] buf;
  private int counter;
  public Buffer(int capacity) {
    buf = new int[capacity];
  }
  public void put(int x) {
    buf[counter++] = x;
  }
  public int get() {
    int tmp = buf[--counter];
    return tmp;
  }
  filter Wait {
    (counter==buf.length) => put(int),
    (counter==0) => get()
  };
  filter Dispatch { true => this.* };
}
```

Listing 4. Buffer definition in the CFOM

A message arriving at a Wait filter is evaluated according to the filter specification. The first guard matches the put(int) message if the buffer is full (i.e. the current number of elements in the buffer is equal to the capacity of the buffer). The second guard matches the get() message if the buffer is empty. When the message is rejected, it proceeds immediately to a Dispatch filter. Otherwise the message is put in a queue, and delayed until the correspondent condition is satisfied. Re-evaluation of a Wait filter occurs at least at every change of the object state. In order to provide mutual exclusion, the pre-processor puts a synchronised block around every method for which a Wait filter is specified. However, while a message is blocked, the other messages can be processed.

The last filter in a filter set is always a Dispatch filter [4, 5]. When a Dispatch filter rejects a message, an exception is raised. In case of acceptance, the message is dispatched to the object that corresponds to the target of the first matching guard [4]. The pseudovariable "this" refers to the implementation layer. The wildcard "*" matches all messages which are in the signature of the target [5]. The definition "true => this.*" declares that all public methods in the implementation layer are unconditionally allowed to execute.

The next example illustrates that, CFs are also more effective than the OO paradigm, in adapting software to changes in requirements. The class definition in Listing 5 uses CFs to provide the tracing concern described in Section 3. LogStatementCF implements 11 methods. Of these, 1 is a constructor, 8 methods perform tracing and delegating, and 2 methods implement tracing only. No methods are used for delegating only, because delegation is achieved by the Dispatch filter. However, it should be noticed that in this example CFs implementation mixes the concerns.

```
public class LogStatementCF implements Statement {
  private Statement delegate;
  public LogStatementCF(Statement st) { delegate = st; }
  private boolean isDML(String sql) {
    String tmp = sql.toUpperCase();
    return tmp.indexOf("UPDATE")>=0 ||
     tmp.indexOf("INSERT")>=0 || tmp.indexOf("DELETE")>=0;
  }
  private void log(String sql) throws SQLException {
    String user =
     delegate.getConnection().getMetaData().getUserName();
    System.out.println("["+new Date()+"]; "+user+"; "+sql);
  }
  public boolean execute(String sql) throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.execute(sql);
  }
  //other execute(..) methods
  public int executeUpdate(String sql) throws SQLException {
    if (isDML(sql)) log(sql);
    return delegate.executeUpdate(sql);
  }
  //other executeUpdate(..) methods
  filter Dispatch { true => this.*, true => delegate.* };
}
```

Listing 5. LogStatementCF definition in the CFOM

5 Comparative analysis of AOP and CFs

For the purpose of this paper an evaluation study of two post-OO paradigms was performed. The examined paradigms share the same goal, which is to modularise crosscutting concerns for better reusability and easier system maintenance. In order to achieve this goal AOP introduces new units of modularity, whereas CFs enrich the class module. The guidelines for determining which paradigm is more effective for which issue are outlined in Table 1.

Table 1 . AOP vs. CFs.

paradigm / issue	**AOP**	**CFs**
implementing core concerns		√√
implementing crosscutting concerns when they: a) cut across a single core concern b) cut across multiple core concerns	 √√ √√	 √√
adjusting a system to new requirements	√√	√
prototyping	√√√	√
tool availability	√√	

Although AOP is more powerful th an CFs, it is slightly controversial. It breaks encapsulation and modifies flow control, making the source code hard to understand. AOP should therefore be used only for maintaining the system, rather than being the choice of convenience for developing the initial version of the system. On the other hand, CFs extend the OO paradigm in a natural way and so can be used like the classical OO mechanism. CFs also improve delegation-based reuse and allow the developer to avoid composition anomalies. In contrast to CFs, AOP is a completely non-invasive technique, allowing developers to introduce the new functionality without making any change to the core concerns, and is therefore an ideal prototyping tool because new concerns are easily added or removed as new requirements are explored. AOP is also appropriate for implementing development concerns such as tracing, profiling and testing. Development concerns are of interest only to the development team and have to be removed from a system prior to its release into the preproduction environment. Moreover, AOP is invaluable for implementing those crosscutting concerns which cut across multiple core concerns. On the other hand, crosscutting concerns that cut across a single core concern can be implemented by using AOP as well as by CFs. CFs are, however, still a theoretical concept unsupported by mainstream programming languages.

6 Future work

CFs potentially have the ability to work well together with the OO paradigm. However, the benefits of CFs are still in the realm of speculation, and more research is required to evaluate the practical usability of this approach. In particular the current OO languages should be enhanced to support CFs.

Table 1 shows that the presented post-OO paradigms are complementary in some areas. Combining the ideas behind AOP and CFs might be a valuable contribution to both paradigms. Therefore it is worth considering the development of tools that support the integration of these paradigms. Moreover, the way in which the post-OO paradigms support modularising crosscutting concerns at analysis and design level is an area that has still not been thoroughly explored.

7 Summary

The essential problem with the OO paradigm is the lack of proper mechanisms to separate the implementation of crosscutting concerns from the implementation of core concerns. Although it is possible to implement the crosscutting concerns in the conventional OO fashion, the presented examples show that such implementation produces classes with low cohesion and high coupling. Poor software quality causes that slight changes in requirements result in invasive modification of the source code.

The l imitations of the OO paradigm can be overcome by AOP and CFs, which, as outlined, indicate the direction in which software development should evolve. Although both post-OO paradigms form suitable candidates for implementing crosscutting concerns, one of them may be more suitable for some issues than the other. Moreover, these paradigms complement each other so that their respective weaknesses can be addressed, while maintaining their desirable properties. AOP modularises crosscutting concerns into separate aspects, making complete SoC possible. However, it violates encapsulation and should be used only as a last resort where other mechanisms introduce code tangling or scattering. On the other hand, the CFOM is consistent with the OO model and can improve its quality, making CFs appropriate for developing the initial version of the system. CFs are also useful in reusing modules without unnecessary redefinitions.

The main contribution of this paper lies in the guidelines for determining which paradigm is appropriate for which software development issue. Moreover, the author has presented the problems he encountered in OO system development and, in accordance with good practice, has explained how to solve these problems with the post-OO paradigms.

References

1. Aksit M.: *On the Design of an Object-Oriented Language Sina,* Dept. of Computer Science, University of Twente (1989).
2. Aksit M.: Composition and Separation of Concerns in the Object-Oriented Model. ACM Computing Surveys, vol. 28(4), ACM Press, New York (1996).
3. Aksit M., Tripathi A.: *Data Abstraction Mechanisms in Sina,* ACM Sigplan Notices, vol. 23(11). ACM Press, New York (1988) 267–275.
4. Bergmans L.: *Composing Concurrent Objects – Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs,* Ph.D. thesis, University of Twente (1994).
5. Bergmans L., Aksit M.: *Composing crosscutting concerns using composition filters.* Commun. ACM, vol. 44(10). ACM Press, New York (2001) 51–57.
6. Bergmans L., Aksit M.: *Analyzing Multi-Dimensional Programming in AOP and Composition Filters,* In Proceedings of the OOPSLA'99 workshop on Multi-Dimensional Separation of Concerns (1999).
7. Bergmans L., Aksit M.: *Composing Crosscutting Concerns Using Composition Filters,* Communications of the ACM, vol. 44(10). ACM Press, New York (2001) 51–57.
8. Bergmans L., Aksit M., Tekinerdogan B.: *Mapping Aspects to Components,* University of Twente (1999).
9. Brito I., Moreira A.: *Integrating the NFR framework in a RE model,* In Proceedings of the 3rd Workshop on Early Aspects, 3rd International Conference on Aspect-Oriented Software Development, Lancaster (2004).
10. Clarke S., Baniassad E.: *Aspect-Oriented Analysis and Design: The Theme Approach,* Addison Wesley, Boston (2005).
11. Colyer A., Clement A.: *Aspect-oriented programming with AspectJ,* IBM Systems Journal, vol. 44(2). IBM Corp., Riverton (2005) 301–308.
12. Colyer A., Clement A., Harley G., Webster M.: *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools,* Addison Wesley, Boston (2004).
13. Constantinides C., Skotiniotis T.: *Reasoning about a classification of crosscutting concerns in object-oriented systems,* Second Workshop on Aspect-Oriented Software Development of the German Information Society. Bonn (2002).
14. Czarnecki K., Eisenecker U.: *Generative Programming: Methods, Techniques, and Applications,* Addison-Wesley, Boston (2000).
15. Dijkstra E. W.: *A Discipline of Programming,* Prentice Hall, Englewood Cliffs (1976).
16. Filman E. E., Elrad T., Clarke S., Aksit M.: *Aspect-Oriented Software Development,* Addison Wesley, Canada (2004).
17. Findler R. B., Flatt M.: *Modular object-oriented programming with units and mixins,* ACM SIGPLAN Notices, vol. 34(1). ACM Press, New York (1999) 94–104.
18. Fischer G., Redmiles D., Williams L., Puhr G., Aoki A., Nakakoji K.: *Beyond Object-Oriented Technology: Where Current Object-Oriented Approaches Fall Short,* Human-Computer Interaction, vol. 10. (1995) 79–119.
19. France R., Georg G, Ray I.: *Supporting Multi-Dimensional Separation of Design Concerns,* AOSD Workshop on AOM: Aspect-Oriented Modeling with UML, (2003).
20. Gradecki J. D., Lesiecki N.: *Mastering AspectJ: Aspect-Oriented Programming in Java,* Wiley, Canada (2003).
21. Harrison W., Ossher H.: *Subject-Oriented Programming: A Critique of Pure Objects,* ACM SIGPLAN Notices, vol. 28(10). ACM Press, New York (1993) 411–428.
22. Kiczales G. et. al.: *Aspect-Oriented Programming,* In Proceedings of the European Conference on Object-Oriented Programming (ECOOP). LNCS, vol. 1241. Springer, New York (1997) 220–242.
23. Laddad R.: *AspectJ in Action,* Manning (2003).
24. Mili H., Elkharraz A., Mcheick H.: *Understanding separation of concerns,* In Proceedings of the 3rd Workshop on Early Aspects, 3rd International Conference on Aspect-Oriented Software Development. Lancaster (2004).
25. Parnas D. L.: *On the criteria to be used in decomposing systems into modules,* Communications of the ACM, vol. 15(12). ACM Press, New York (1972) 1053–1058.
26. Wrycza S., Marcinkowski B., Wyrzykowski K.: *UML 2.0 in information systems modelling,* Helion, Warsaw (2005).