2

# Beyond object-oriented software development

Adam Przybylek
*University of Gdansk*
*Poland*

## 1. Introduction

Dealing with complexity has been one of the fundamental goals of software engineering since its inception. The primary technique for managing the complexity of a software systems is **Separation of Concerns** (SoC) (Chu-Carroll, 2000), (Beltagui, 2003). SoC refers to the ability to decompose and organize the system into manageable concerns, which can be developed and maintained in relative isolation (Przybylek, 2007). A **concern** is a specific requirement or an interest which pertains to the system's development. Concerns can be classified into two categories: **core concerns** and **crosscutting concerns**. The former are usually responsible for the main functionality of a system. They are disjoined by nature and therefore their implementations can be precisely separated from each other. However, a typical system also consists of concerns like authentication, logging, error handling, data persistence, etc., which play a supporting role for core concerns. They capture non-functional requirements or technical-level issues that affect the system as a whole (Przybylek, 2007). Although they can be identified as distinct concerns, their implementations cut across the implementation of some core concerns and cannot be localized using traditional decomposition units such as procedures or classes (Przybylek, 2008). They are called crosscutting concerns and they are damaging to the software architecture.

A number of approaches have been proposed to achieve a better separation of crosscutting concerns. This chapter focuses on two the most prominent among them – aspect-oriented programming (AOP) and composition filters (CFs).

## 2. Background

### 2.1 Modularization

When solving a simple problem, the entire problem can be tackled at once. For solving a complex problem, the basic principle should be divided into easier to comprehend pieces, so that each piece can be conquered separately (Jalote, 2005). Implementation and maintenance costs generally will be decreased when each piece of the system corresponds to exactly one small, well-defined piece of the problem, and each relationship between a system's pieces corresponds only to a relationship between pieces of the problem (Yourdon & Constantine, 1979).

In software engineering, the unit to decompose a system is called a **module**. A module is a lexically contiguous sequence of program statements having a name by which other parts of the system can refere to it (Yourdon & Constantine, 1979), (Stevens et al., 1974). A module consists of two parts: an interface and a module body (implementation). An **interface** presents the services provided by the module. It separate the information needed by a client from the implementation details. A module body is the code that actually realizes the module responsibility. It hides the design decisions and is only accessible from within the module. The parts interface and implementation are also called public and private, respectively. Users of a module need to know only its public part (Riel, 1996). An interface serves as a contract between the module and its clients. This contract allows the programmer to change the implementation without interfering with the rest of the program, so long as the public interface remains the same (Riel, 1996). Designing a module so that the implementation details are hidden from other modules is called **information hiding** (Schach, 2007).

There are several reasons for structuring a program into modules:

- Modularization accelerates implementation by encouraging parallel development of different parts of a system.
- Modularization reduces the propagation of side effects when changes occur. Each change is localized to one specific module.
- Modularization makes the program easier to understand. Instead of trying to keep the entire program in mind at once, it suffices to check that each module meets its specifications under the assumption that all other modules also meet their specifications.
- Well-designed modules can be reused in other programs with no change.

The best known module properties to assess its quality are coupling and cohesion. **Coupling** is a measure of how strongly one module is connected to, has knowledge of, or relies on other modules. If two modules are loosely coupled, they are relatively independent, so changes in one module usually don't affect other modules. **Cohesion** is a measure of how tightly bound the internal elements of a module are to one another (Jalote, 2005). A module has high cohesion if all of its elements are related strongly and are necessary for achieving the functionality required. The goal of software engineer is to design the modules with high cohesion and low coupling. Such modules are easy to analyse, modify, test, and reuse.

### 2.2 From structured to object oriented programming

Various paradigms, that provide different modules to decompose a system, have been studied over years. The oldest decomposition unit are procedures and functions which are the focus of the **structured paradigm**. The structured paradigm merges the ideas proposed by:

- Dijkstra: SoC, layered architecture, structured control constructs, formal verification;
- Wirth: stepwise refinement, modular programming;
- Parnas: information hiding, modular programming;
- Hoare: designing data structures, verification of program correctness;
- Knuth: local variables, literate programming.

In the past, the structured paradigm proved to be very successful. However, as software grew in size, inadequacies of the structured techniques started to become apparent, and the

**object-oriented** (OO) **paradigm** was proposed by Dahl and Nygaard as a better alternative. The OO paradigm, which is currently the most popular, was created from a desire to close correspondence between objects in the real world and their counterparts in software. The object-oriented purism comes from the dogma that everything should be modeled by objects, because human perception of the world is based on objects.

An **object** is a software entity that combines both state and behavior. An object's behavior describes what the object can do and is specified by a set of operations. The implementation of an operation is called a **method**. The way that the methods are carried out is entirely the responsibility of the object itself (Schach, 2007) and is hidden from other parts of the program (Larkin & Wilson 1993). An object performs an operation when it receives a message from a client. A message is a request that specifies which operation is desired. The set of messages to which an object responds is called its message interface (Hopkins & Horan, 1995).

An object's state is described by the values of its attributes (i.e. data) and cannot be directly accessed from the outside. The attributes in each object can be accessed only by its methods. Because of this restriction, an object's state is said to be encapsulated. The advantage of encapsulation is that as long as the external behavior of an object appears to remain the same, the internals of the object can be completely changed (Hunt, 1997). This means that if any modifications are necessary in the implementation, the client of the object need not be affected.

In OO software development, a system is seen as a set of objects that communicate with each other by sending messages to fulfil the system requirements. The object receiving the message may be able to perform the task entirely on its own (i.e. access the data directly or use its other method as an intermediary). Alternatively, it may ask other objects for information, or pass information to other objects (Hopkins & Horan, 1995).

The most popular model of OOP is a classed based model. In this model, an object's implementation is defined by its **class**. The object is said to be an instance of the class from which it was created. A class is a blueprint that specifies the structure and the behaviour of all its instances. Each instance contains the same attributes and methods that are defined in the class, although each instance has its own copy of those attributes.

OO languages offer two primary reuse techniques: inheritance and composition. Software reuse refers to the development of software systems that use previously written modules. **Inheritance** allows for reusing an existing class in the definition of a new class. The new class is called the derived class (also called subclass). The original class from which the new class is being derived is called the base class (also called superclass). All the attributes and methods that belong to the base class automatically become part of the derived class (Cline et al., 1998). The subclass definition specifies only how it differs from the superclass (Larkin & Wilson 1993); it may add new attributes, methods, or redefine (override) methods defined by the superclass.

An object of a derived class can be used in every place that requires a reference to a base class (Cline et al., 1998). It allows for dispatching a message depending not only on the message name but also on the type of the object that receives the message. Thus, the methods that matches the incoming message is not determined when the code is created (compile time), but is selected when the message is actually sent (run time) (Hopkins & Horan, 1995). An object starts searching the methods that matches the incoming message in its class. If the method is found there, then it is bound to the message and executed, and the appropiate response returned. If the appropiate method is not found, then the search is

made in the instance's class's immediate superclass. This process repeats up the class hierarchy until either the methods is located or there are no further superclasses (Hopkins & Horan, 1995). The possibility that the same message, sent to the same reference, may invoke different methods is called **polymorphism**.

A new class can be composed from existing classes by **composition**. Composition is the process of putting an object inside another object (the composite) (Cline et al., 1998). A composite can delegate (re-direct) the requests it receives to its enclosing object. Composition models the has-a relationship. It is claimed that composition is more powerful than inheritance, because (1) composition can simulate inheritance, and (2) composition supports the dynamic evolution of systems, whereas inheritance relations are statically defined relations between classes (Bergmans, 1994).

Inheritance is also called "white box" reuse, because internals of a base class are visible to its extensions. In contrast, composition is called "black box" reuse, because the internals of the enclosed object are not visible to the enclosing object (and vice-versa) (Oprisan, 2008). With composition, an enclosing object can only manipulate its enclosed object through the enclosed object's interface. Because composition introduces looser coupling between classes it is preferable to inheritance.

In the early days of OOP there has been a general agreement that single classes should be the primary unit of organization and reuse. However, over the years it has been recognized that a slice of behavior affecting a set of collaborating classes is a better unit of organization than a single class. In the face of these insights, mainstream programming languages have been equipped with constructs to group sets of related classes, for example name spaces in C++ or packages in Java (Ostermann, 2003).

### 2.3 Weaknesses of object orientation

The OO paradigm has been one of the most important contributions in the history of software development (Clarke & Baniassad, 2005). It was created from a desire to have language constructs for reflecting a natural view of the world. Although OOP improves software reuse and system maintenance, practical experience shows that OOP has not been as successful as expected. The main problem for OOP is that it does not have an efficient way of expressing crosscutting concerns.

As it turned out classes and packages are a powerful way only to modularize core concerns.

Symptoms of implementing crosscutting concerns in OO languages are "code scattering" and "code tangling". **Scattering** occurs when multiple fragments of code that all do the same thing (or that do closely related things) are distributed across multiple modules. Code scattering causes that apparently small changes in requirements usually forces programmers to modify many modules. The term **code tangling** is often used to describe the situation where a class contains logic pertaining to more than one concern. Tangled code makes it difficult to see which code belongs to which concern. Code tangling and scattering negatively affect the software quality. In the result software is difficult to maintain and reuse (Kiczales et al., 1997).

## 3. The tyranny of the dominant decomposition

Every programming paradigm involves some kind of decomposition: starting with a high level depiction of the system's key elements and creating lower level looks at how the

system's features and functions will fit together (Atlee, 2005). The manner in which a system is physically divided into modules can affect significantly the structural complexity and quality of the resulting system (Yourdon & Constantine, 1979), (Parnas, 1972). The most effectively decomposition is usually the one which minimizes dependencies among modules.

In software engineering, there are two common types of decomposition:

- procedural decomposition (in structured programming) - centers on identifying the major system functions and then elaborating and refining them in a top-down manner;
- object oriented decomposition - breaks a large system down into progressively smaller classes that are responsible for some part of the problem domain.

The exact nature of the decomposition differs between the structured and OO paradigm, but it feels comfortable to talk about what is encapsulated as a functional unit of the overall system (Kiczales et al., 1997). Therefore, both decomposition techniques can be generally treated as **functional decomposition**.

There are certain modularity problems that will never be solved satisfactorily with functional decomposition only, how sophisticated they may ever be. The main failure of functional decomposition is that it assumes that real-world concepts have intuitive, mind-independent, preexisting concept hierarchies. However, our perception of the world depends heavily on the context from which it is viewed: There is no conceptual lingua franca (Ostermann, 2003).

To illustrate the problem, consider the example (Fig. 1) presented by Ostermann. Each figure represents a particular concern of a software system. There are three options for organizing this space: by size, by shape, or by color. Each of these decompositions is equally reasonable, but they are not hierarchically related (Ostermann, 2003).

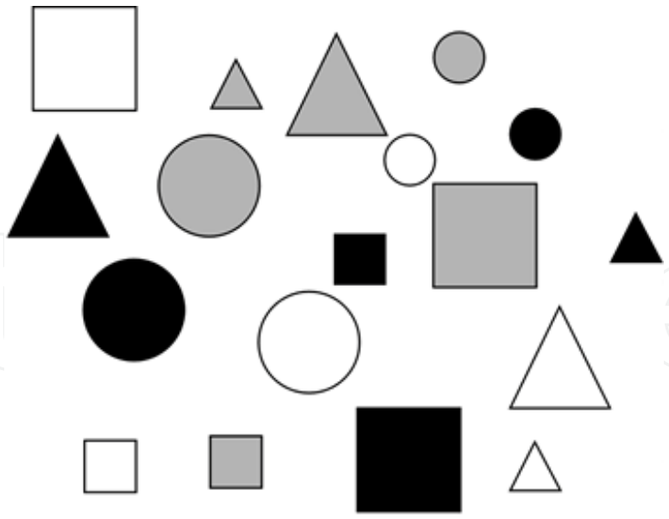

Fig. 1. Abstract concern space

With a mind-independent hierarchical decomposition, one fixed classification sequence has to be choosen. In Fig. 2, the classification sequence is color, shape, size. The problem with such a fixed classification sequence is that only the first element of the list is localized whereas all other concerns are tangled in the resulting hierarchical structure (Mezini & Ostermann, 2004). Fig. 2 illustrates this with the concern "circle", whose definition is scattered around the color-driven decomposition (Ostermann, 2003). Only the color concern is cleanly separated into white, grey, and black, but even this decomposition is not satisfactory because the color concern is still tangled with other concerns (Mezini & Ostermann, 2004).

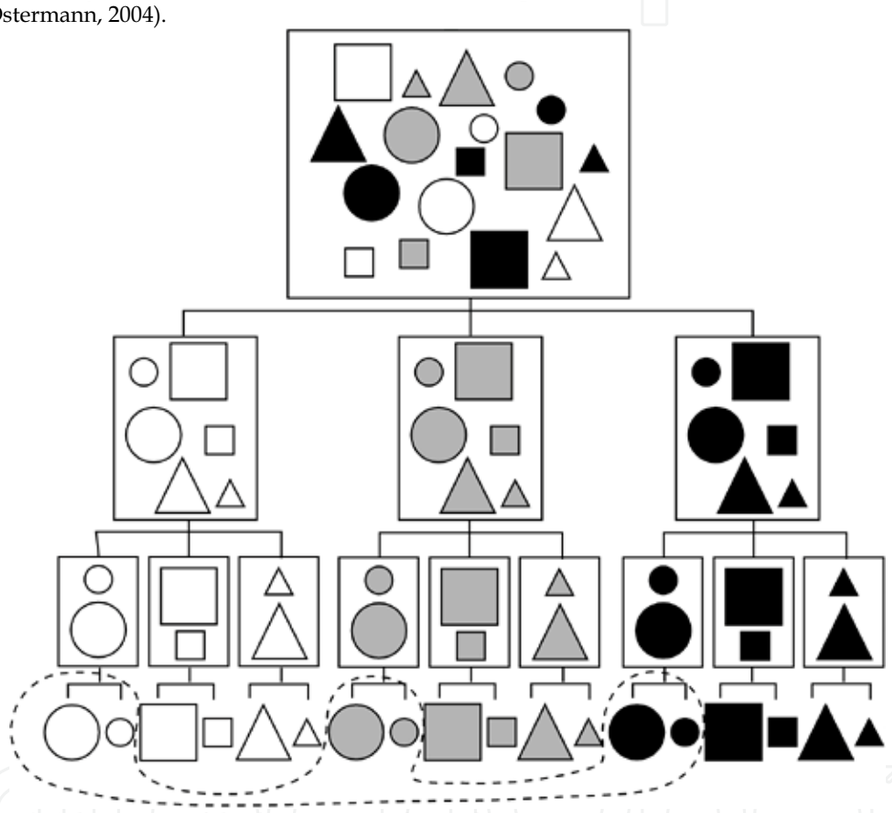

Fig. 2. Arbitrariness of the decomposition hierarchy

The presented problem is known as the "**tyranny of the dominant decomposition**" and it means that existing languages and formalisms generally provide only one, dominant dimension along which to separate e.g., by object or by function (Tarr et al., 1999). In the result, no matter how well a system is decomposed, implementation of crosscutting concerns will be scattered through the whole system and tangled into implementation of core concerns.

Several techniques have been invated to overcome this problem. The most prominent among them are aspect-oriented programming (AOP) and composition filters (CF's). Both

paradigms builds on all the advantages of object orientation, so the term post-OO paradigm has been introduced to refer to them.

## 4. The aspect-oriented paradigm

### 4.1 Basic concepts

AOP grew out of the research work undertaken by Gregor Kiczales (Kiczales et al., 1997) at Xerox PARC (Palo Alto Research Center). It appeared as a reaction to the phenomena of code tangling and scattering. The aim of AOP is to improve SoC by introducing a new unit of decomposition, called aspect. Aspects allow programmers to implement crosscutting concerns in a well-localized way.

The first AOP language was AspectJ, developed by Kiczales and his team. To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an openly-developed Eclipse project in December 2002. Since that time, AspectJ has matured into a stable and complete AOP platform. A program in AspectJ can be thought of being composed of two parts:

- the part implementing the core concerns;
- the part implementing the crosscutting concerns.

This model is called the asymmetric approach, which means that crosscutting concerns are encapsulated in a special kind of module, different from the base units.

AspectJ is an extension to Java. It brings new concepts such as an aspect, a joinpoint, a pointcut, an advice, an introduction, and a parent declaration.

An **aspect** is a module that encapsulates the behaviour and structure of a crosscutting concern. It can, like a class, realize interfaces, extend classes and declare attributes and operations. In addition, it can extend other aspects and declare advices, pointcuts, introductions and parent declarations.

A **joinpoint** is an identifiable location in the program flow where the implementation of a crosscutting concern can be plugged in. Typical examples of joinpoints include a throw of an exception, a call to a method and an object instantiation.

A **pointcut** is a language construct designed to specify a set of join-points and obtain the context (e.g. the target object and the operation arguments) surrounding the join-points as well. The purpose of declaring a pointcut is to share its pointcut expression in many advices or other pointcuts. A pointcut cannot be overloaded. The pointcut signature is follows:

```
[visibility-modifier] pointcut name([parameters]): PointcutExpression;
```

The visibility-modifier defines the visibility of the pointcut. The options for visibility are the same as for other Java artifacts. The PointcutExpression acts as a filter, matching join points that meet its specification. The name, which looks like a method, will be used shortly to handle actions performed when a join point is encountered by the Java runtime.

An **advice** is a method-like construct used to define an additional behaviour that has to be inserted at all joinpoint picked out by the associated pointcut. The body of an advice is the implementation of a crosscutting functionality. The advice is able to access values in the execution context of the pointcut. Depending on the type of advice, whether “before”, “after” or “around,” the body of an advice is executed before, after or in place of the selected joinpoints. An around advice may cacnel the captured call, may wrap it or may execute it with the changed context.

The simplicity format of advice is:

```
advice_type ([parameters]) [returning] : pointcut_name([parameters]) {
  //code to execute
}
```

An **introduction** is used to crosscut the static-type structure of a given class. It allows a programmer to add attributes and methods to the class without having to modify it explicitly. The power of introduction comes from the introduction being able to add methods to the interface.

A **parent declaration** may change the class's super-class or add implemented interfaces by defining an extends/implements relationship.

### 4.2 Examples

**Scenerio 1** - Remote client monitoring

In some server application the need for a new requirement has occurred. IP addresses of remote clients must be logged. Logging is one of the most common crosscutting concern.

Two fundamental classes responsible for network communication are Socket and ServerSocket. ServerSocket runs on the server side and listens for incoming connections using its accept() method. When a client does connect, the ServerSocket::accept() method returns a Socket object, which connects the client and the server. The IP address of the remote client can be gained via the Socket::getInetAddress() method.

The logging concern can be implemented in an OO way by defining a new class that extends ServerSocket and redefines the original accept() method (Listing 1). After the original accept() method returns a new socket, the IP address is retrived from this socket and then logged. Moreover, every place where ServerSocket is instantiated has to be replaced by IPlogServerSocket.

```
class IPlogServerSocket extends ServerSocket {
  public Socket accept() throws IOException {
    Socket s = super.accept();
    String ip = s.getInetAddress().getHostAddress();
    System.out.println( new Date() +" [" + ip + "]" );
    return s;
  }
}
```

Listing 1. The IPlogServerSocket class

The above imlementation tangles the logging concern along with listening for incoming connection. The ancillary logging code doesn't belong to the class at the conceptual level, so it decreases the class cohesion. A better solution can be found when using the AO paradigm. With AspectJ-based logging, there is no need to extend the base class. All programmers need to do is define an aspect (Listing 2).

```
aspect IPMonitoring {
  pointcut clientConnect(): call(public Socket ServerSocket.accept()); //1
  after() returning (Socket s): clientConnect() { //2
    String ip = s.getInetAddress().getHostAddress(); //3
    System.out.println( new Date() +" [" + ip + "]" ); //4
  }
}
```

Listing 2. The IPMonitoring aspect

The most fundamental difference between the conventional and AO solution is a separation of concerns. Code related to logging is encapsulated in the IPMonitoring aspect. The joinpoints at which merging should be made are identified as the places where the ServerSocket.accept() method is called (line 1). The after-returning advice (line 2) runs after a successful return from accept(). Line 4 contains the actual logging code.

**Scenerio 2** – Tracing a method's execution time
Tracing is a special form of logging where the entry and/or exit of selected methods are logged (Laddad, 2003). Suppose, the Matrix class is designed as shown at Fig. 3.

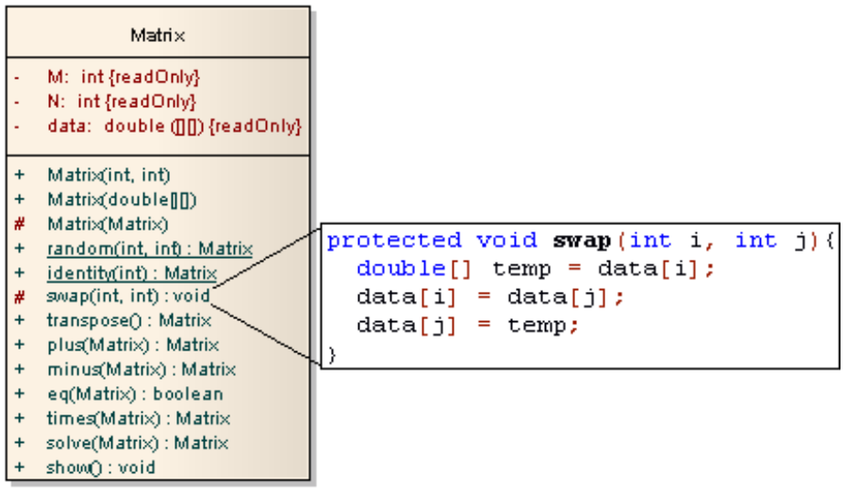

Fig. 3. The Matrix class

There is need to trace how long it takes to execute a method. OO solution requires embedding the tracing code in each method. For example, Listing 4 shows how the swap method is instrumented to mesure its execution time. Such instrumentation is very invasive.

```
public class Matrix {
//...
  protected void swap(int i, int j) {
    long start = System.currentTimeMillis();
    double[] temp = data[i];
    data[i] = data[j];
    data[j] = temp;
    long end = System.currentTimeMillis();
    double time = end - start;
    System.out.println("void Matrix.swap(int,int) - " + time);
  }
}
```
Listing 4. The swap method with tracing

The code needed for tracing is scattered, because almost the same code occures in each method. Alternatively, a new subclass could be created (Listing 5).

```
public class LogMatrix extends Matrix {
//...
  protected void swap(int i, int j) {
    long start = System.currentTimeMillis();
    super.swap(i,j);
    long end = System.currentTimeMillis();
    double time = end - start;
    System.out.println("void Matrix.swap(int,int) - " + time);
  }
}
```

Listing 5. Tracing in the subclass

However, code scattering and tangling are still present. Moreover, each location where Matrix is instantiated would have to be replaced by the new name (LogMatrix). But, what would happen if the tracing concern has to be implemented in several classes of a real system. Imagine how long it would take. Instead of disturbing the existing code it is sufficient to define an aspect that implements the new requirement (Listing 6). The tracing code is injected before and after the execution of each method.

```
public aspect TimeLogging {
  pointcut eachMethod(): (execution(* *.*(..)) && !within(TimeLogging);
  Object around(): eachMethod() {
    long start = System.currentTimeMillis();
    Object tmp = proceed(); //execute the original method
    long end = System.currentTimeMillis();
    double time = end - start;
    Signature sig = thisJoinPointStaticPart.getSignature();
    System.out.println(sig +" - "+ time);
    return tmp;
  }
}
```

Listing 6. The TimeTracing aspect

**Scenerio 3** - Producer-Consumer

This scenerio uses the classical Producer-Consumer problem, where two processes (or threads), one known as the "producer" and the other called the "consumer", run concurrently and share a 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. If the consumer needs to consume an item that the producer has not yet produced, then the consumer must wait until it is notified that the item has been produced. If the buffer is full, the producer will need to wait until the consumer consumes any item.

Assume the existence of the cyclic queue as shown at Fig. 4. The method put(..) stores one object in the queue and get() removes the oldest one. Attribute nextToRemove indicate the location of the oldest object. The location of the new object can be computed using nextToRemove, number of items (numItems) and queue capacity (buf.length).

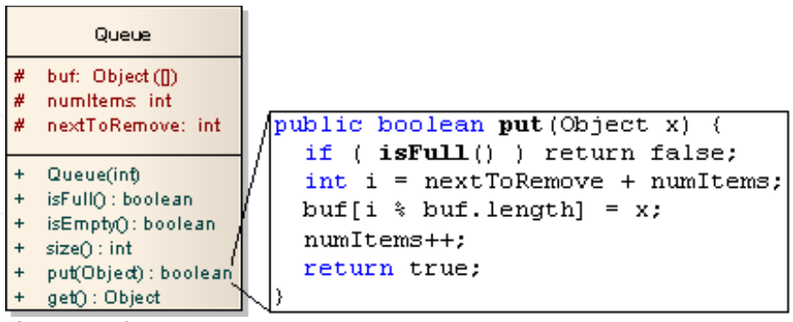


Fig. 4. The queue class

**Stage 1** - Buffer

In order to use the queue as a buffer an adaption to a multi-thread environment is needed. Both put(..) and get() methods have to be executed in mutual exclusion. In addition, the buffer has to block a thread when the thread tries to add an element to a full queue or to remove an element from an empty queue. In Java these methods have to be wraped in synchronization code (Listing 8).

```
public class Buffer extends Queue {
  public Buffer(int capacity) {
    super(capacity);
  }
  public synchronized boolean put(Object x) {
    while ( isFull() ) try {
      wait();
    } catch (InterruptedException e) {}
    super.put(x);
    notifyAll();
    return true;
  }
  public synchronized Object get() {
    while ( isEmpty() ) try {
      wait();
    } catch (InterruptedException e) {}
    Object tmp = super.get();
    notifyAll();
    return tmp;
  }
}
```

Listing 8. The buffer class

The above implementation tangles the synchronization concern with the core logic. Moreover, the synchronisation code is scattered through the methods responsible for accessing the buffer. In the result the put(..) and get() methods contain similar fragments of code for cooperating synchronisation. A separation of synchronization concern can be achieved by using the aspect (Listing 9).

```
public aspect SynchronizedQueue pertarget(target(Queue)) {
  protected pointcut call_get(): call( Object Queue.get() );
  protected pointcut call_put(Object x):
    call( boolean Queue.put(Object) ) && args(x);

  Object around(Queue q): call_get() && target(q) {
    synchronized(this) {
      while( q.isEmpty() ) try {
        wait();
      } catch (InterruptedException e) {}
      Object tmp = proceed(q);
      notifyAll();
      return tmp;
    }
  }
  boolean around(Queue q, Object x): call_put(x) && target(q) {
    synchronized(this) {
      while ( q.isFull() ) try {
        wait();
      } catch (InterruptedException e) {}
      proceed(q,x);
      notifyAll();
      return true;
    }
  }
}
```

Listing 9. The SynchronizedQueue aspect

**Stage 2** – Time buffer

After implementing the buffer a new requirement has occured: the buffer should save current date and time associated with each stored item. A Java programmer may use inheritance and composition as reuse techniques (Listing 10).

```
public class TimeBuffer extends Buffer {
  protected Queue delegateDates;
  public TimeBuffer(int capacity) {
    super(capacity);
    delegateDates = new Queue(capacity);
  }
  public synchronized boolean put(Object x) {
    super.put(x);
    delegateDates.put( new Long(System.currentTimeMillis()) );
    return true;
  }
  public synchronized Object get() {
    Object tmp = super.get();
    Long date = (Long) delegateDates.get();
    long time  = System.currentTimeMillis() - date.longValue();
    System.out.println(time);
    return tmp;
  }
}
```

Listing 10. The TimeBuffer class

The problem is that three different concerns are interwinded within put/get and so these concerns cannot be composed separately. It means that e.g. if a programmer wants a queue with timing he cannot reuse the timing concern from TimeBuffer; he has to reimplement the timing concern in a new class that extends Queue. A slightly better solution seems to be using AOP and implementing the timing as an aspect (Listing 11).

```
public privileged aspect Timing pertarget( instantiation() ) {
  protected Queue delegateDates;

  protected pointcut instantiation():
    target(Queue) && !cflow(within(Timing));
  protected pointcut init(Queue q): execution( Queue.new(..) ) && target(q);
  protected pointcut execution_get(): execution( Object Queue.get() );
  protected pointcut execution_put(): execution(boolean Queue.put(Object));

  after(Queue q): init(q) {
    delegateDates = new Queue(q.buf.length);
  }
  after(): execution_get() {
    Long date = (Long) delegateDates.get();
    long time  = System.currentTimeMillis() - date.longValue();
    System.out.println("<"+time+">");
  }
  after(): execution_put() {
    delegateDates.put( new Long(System.currentTimeMillis()) );
  }
}
```

Listing 11. The Timing aspect

This solution require only one change in the existing code - the instantation pointcut in SynchronizedQueue. It must be the same as in Timing.

## 5. Composition Filters

Composition Filters (CF's) was defined by Aksit and Tripathi in the late 1980s (Aksit & Tripathi, 1988), and originally implemented in the Sina language. The Composition Filters model can be thought of as the conventional OO model in which an object can be surrounded by input and output filters. **Filters** extend the message passing mechanism by manipulating incoming and outgoing messages. Incoming messages have to pass through all the input filters until they are dispatched and the outgoing through the output filters until they are sent outside the object (Bergmans & Aksit, 2001), (Czarnecki & Eisenecker, 2000). Dispatching here means either to start searching of a local method, or to delegate the message to another object. The filters together compose the enhanced behaviour of the object, possibly in terms of other objects. The resulting model and its elements are shown in Fig. 5.

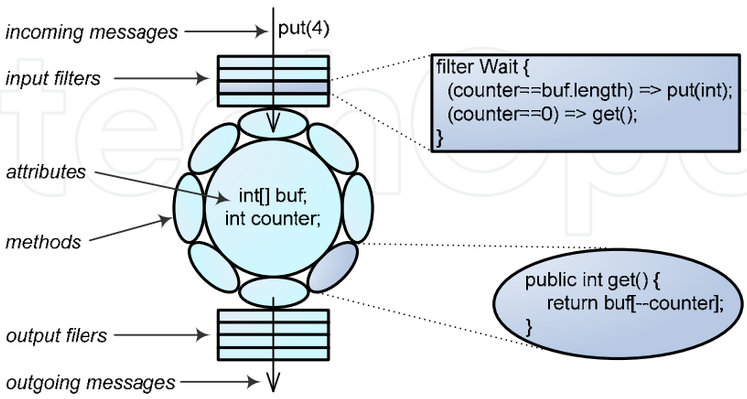


Fig. 5. The Composition Filters model

A filter definition consists of a **filter type** and **filter guards**. It has the following form:

```
filter filterType {
  condition => selector1, selector2, ..., selectorN;       //filter guard 1
  ...                                                      //filter guard 2
}
```

A selector is mainly used for matching messages. In addition it may modify certain parts of messages or indicate the targget object to which the message should be redirected. When the selector on the left hand matches, no further selectors should be considered.

A guard matches the message if (1) the condition evaluates to true, and (2) the message matches one of the selectors. As soon as the first guard is matched, the message is said to be accepted by the filter. A filter rejects a message if none of the filter guards matches the message. The filter type determines the semantics associated with acceptance and rejection of messages (Bergmans, 1994), (Bergmans & Aksit, 2001). In other words, it determines how to handle the messages after the matching process. Examples of predefined filter types are presented in Table 1.

| **filterType** | **semantic** |
|---|---|
| Error | If the message is accepted, an exception is raised, otherwise the message continues to the subsequent filter. |
| Wait | If the message is rejected, it proceeds immediately to the next filter in the set. 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. |
| Dispatch | The last filter in a filter set is always a Dispatch filter. 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 matching selector (Bergmans, 1994). |

Table 1. The predefined filter types

The running example of CF's is shown using scenerio 3 (Producer-Consumer) from Section 4.2. The presented source code is written in a simplified version of CF's.

**Stage 1** - Buffer
The class definition in Listing 12 uses wait filter to provide the synchronization concern.

```
public class BufferCF {
  private Queue delegate;
  public BufferCF(int capacity) {
    delegate = new Queue(capacity);
  }
  filter Wait {
    delegate.isFull() => put(Object),
    delegate.isEmpty() => get()
  };
  filter Dispatch { true => delegate.* };
}
```

Listing 12. The Buffer definition in CF's

An arriving message is evaluated according to the Wait filter specification. The first guard matches the put(Object) 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 Dispatch filter. Otherwise the message is put in a queue, and delayed until the correspondent condition is satisfied. Re-evaluation of 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 Wait filter is specified. However, while a message is blocked, the other messages can be processed.
The wildcard "*" matches all messages which are in the signature of the target. The filter guard "true => delegate.*" declares that all public methods in the delegate object are unconditionally allowed to execute.

**Stage 2** – Time buffer
With CF's adding the timing concern requires to reimplement the synchronization concern. Moreover, the timing concern cannot be separate from the operation's core logic (Listing 13).

```
public class TimeBufferCF {
  private Queue delegate;
  private Queue delegateDates;
  public BufferCF(int capacity) {
    delegate = new Queue(capacity);
    delegateDates = new Queue(capacity);
  }
  public void put(Object x) {
    delegate.put(x);
    delegateDates.put( new Long(System.currentTimeMillis()) );
  }
  public Object get() {
    Long date = (Long) delegateDates.get();
    long time  = System.currentTimeMillis() - date.longValue();
    System.out.println(time);
    return delegate.get();
  }
```

```
  filter Wait {
    delegate.isFull() => put(Object),
    delegate.isEmpty() => get()
  };
  filter Dispatch { true => this.*, true => delegate.* };
}
```

Listing 13. The TimeBuffer definition in CF's

## 6. Discussion

The presented examples illustrate that OOP cannot deal effectively with implementation of crosscuting concerns. In each case the OO solution requires a significant rebuilding of the existing code. Invasive changes break the open-closed principle, which states that modules should be open for extension, but closed for modification. AOP and CF's bring a partial solution. They speed up evolving OO systems to new requirements. AOP usually allows developers to introduce new functionality without making any change to the existing modules. Moreover, it eliminates code tangling and scattering. However, no programming paradigm is without its own set of problems and pitfalls. Applying AO constructs makes the source code hard to understand, because classes are unaware of the presence of aspects. Aspects modify flow control and break encapsulation. Hence, the advantage of AOP over its ancestor is doubtful from the maintenance point of view. Although aspects makes it possible to separate crosscutting concerns, and in the result increase class cohesion, they increase the overall coupling. Aspects are tightly connected with the affected classes, so their reuse is limited.

Implementation of crosscutting concerns in each paradigm provides the following conclusions. AOP is the most beneficial in implementing development concerns such as tracing, debugging, 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 production. AOP is also appropiate for prototyping. Aspects can easily enable and disable the additional functionality when new requirements are explored. AO solutions should also be considered when changes are made only for a moment (e.g. condition checking during the debugging process) and its implementation in an OO way would be invasive.

On the other hand, CF's is less powerful then AOP, but it extends the OO paradigm in a natural way. CF's improves delegation-based reuse and allow the developer to avoid composition anomalies.

## 7. Summary

Some difficulties in software development can't be overcome using the OO paradigm. Although OOP works well at modularizing core concerns, it falls short when it comes to modularize crosscutting concerns. The growing complexity of today's software makes it necessary for developers to deal with more and more crosscuting concerns. The goal of this chapter was to present how post-OO paradigms improves SoC. The principles of AOP and CF's were explained and illustrated by 3 scenerio of adapting software to new requirements. Both AspectJ and CF's solutions have the following advantages over its ancestor:

- the crosscutting concern is explicitly implemented: instead of being embedded in the code of core concerns;

- the evolution of the OO systems is simplified: new language constructs are offered;
- core concerns can be easily reused: crosscutting concern are not interwinded with core concerns.

Although the presented post-OO paradigms introduce new problems to software development, they indicate the direction in which the programming techniques should evolve.

## 8. References


Aksit, M. & Tripathi, A. (1988). Data Abstraction Mechanisms in Sina. *ACM Sigplan Notices*, vol. 23(11), 267-275

Atlee J.M. (2005). *Software engineering: theory and practice,* Prentice Hall

Beltagui F. (2003). Features and Aspects: Exploring feature-oriented and aspect-oriented programming interactions. *Technical Report No: COMP-003-2003,* Computing Department, Lancaster University, Lancaster

Bergmans, L. (1994). Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs. *Ph.D. thesis,* University of Twente

Bergmans, L. & Aksit, M. (2001). Composing crosscutting concerns using composition filters. *Commun. ACM*, vol. 44(10), 51-57

Bergmans, L.; Aksit, M. & Tekinerdogan, B. (1999). Mapping Aspects to Components. University of Twente

Brito, I. & Moreira, A. (2004). Integrating the NFR framework in a RE model, *Proceedings of the 3rd Workshop on Early Aspects, 3rd International Conference on Aspect-Oriented Software Development,* Lancaster

Chu-Carroll M. (2000). Separation of concerns: an organizational approach, *Proceedings of the OOPSLA 2000 Workshop on Advanced Separation of Concerns*

Clarke, S. & Baniassad, E. (2005). *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison Wesley, Boston

Cline, M.; Lomow, G. & Girou, M. (1998). *C++ FAQs,* Addison Wesley

Colyer, A.; Clement, A.; Harley, G. & Webster, M. (2004). *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools,* Addison Wesley, Boston

Czarnecki, K. & Eisenecker, U. (2000). *Generative Programming: Methods, Techniques, and Applications,* Addison-Wesley, Boston

Gradecki, J.D. & Lesiecki, N. (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java,* Wiley, Canada

Hopkins, T. & Horan, B. (1995). *Smalltalk: An Introduction to Application Development Using VisualWorks,* Prentice Hall

Hunt, J. (1997). *Smalltalk and Object Orientation,* Springer

Jalote, P. (2005). *An Integrated Approach to Software Engineering,* Springer, New York

Kiczales, G. et al. (1997). Aspect-Oriented Programming, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. LNCS, vol. 1241, 220-242, Springer

Laddad, R. (2003). *AspectJ in Action,* Manning

Larkin, D. & Wilson, G. (1993). *Object-Oriented Programming and the Objective-C Language,* Addison Wesley

Mezini, M. & Ostermann, K. (2004). Untangling Crosscutting Models with Caesar, In: *Aspect-Oriented Software Development,* Filman, E.E.; Elrad, T.; Clarke, S.; Aksit, M. (Ed.), Addison Wesley, Canada

Nora B.; Fadila A. & Said G. (2006). A Comparative Classification of Aspect Mining Approaches. *Journal of Computer Science*, vol. 2(4), 322-325

Oprisan, A. (2008). Aspect Oriented Implementation of Design Patterns using Metadata. *MSc Thesis,* University of Joensuu

Ostermann, K. (2003). Modules for Hierarchical and Crosscutting Models. *PhD thesis,* Technische Universität Darmstadt

Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM,* vol. 15(12), 1053-1058

Przybylek, A. (2007). Post Object-Oriented Paradigms in Software Development: a Comparative Analysis, *Proceedings of the 1st Workshop on Advances in Programming Languages at IMCSIT'07,* Wisla

Riel, A.J. (1996). *Object-oriented Design Heuristics,* Addison-Wesley, Boston

Schach, S.R. (2007). *Object-Oriented and Classical Software Engineering,* McGraw-Hill, Singapore

Stevens, W.; Myers, G. & Constantine, L. (1974). Structured Design, IBM Systems Journal, 13(2), 115-139

Tarr, P.; Ossher, H.; Harrison, W. & Sutton, S.M. (1999). N degrees of separation: multi-dimensional separation of concerns, *Proceedings of the 21st International Conference on Software Engineering*

Tarr P.; Harrison W.; Ossher H.; Finkelstein A.; Nuseibeh B. & Perry D. (2001). Workshop on Multi-Dimensional Separation of Concerns in Software Engineering. *SIGSOFT Softw. Eng. Notes,* vol. 26(1), 78-81

Yourdon, E. & Constantine, L.L. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and System Design,* Prentice-Hall, New York


**Advances in Computer Science and IT**

Edited by D M Akbar Hussain

ISBN 978-953-7619-51-0
Hard cover, 420 pages
**Publisher** InTech
**Published online** 01, December, 2009
**Published in print edition** December, 2009

The book presents some very interesting and excellent articles for this divergent title. The 22 chapters presented here cover core topics of computer science such as visualization of large databases, security, ontology, user interface, graphs, object oriented software developments, and on the engineering side filtering, motion dynamics, adaptive fuzzy logic, and hyper static mechanical systems. It also covers topics which are combination of computer science and engineering such as meta computing, future mobiles, colour image analysis, relative representation and recognition, and neural networks. The book will serve a unique purpose through these multi-disciplined topics to share different but interesting views on each of these topics.



**InTech Europe**
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

**InTech China**
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen