CARLOS AUGUSTO DA SILVA CUNHA

Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms

Dissertação de Mestrado em Informática

Dissertação submetida à Universidade do Minho, para a obtenção do grau de Mestre, elaborada sob a orientação do Professor Doutor João Luís Ferreira Sobral

Universidade do Minho

Escola de Engenharia

Departamento de Informática

Braga, Setembro 2006

Abstract

Concurrent programming is more complex than sequential programming, which requires from programmers an additional effort to manage such increase of complexity in software.

Concurrency requires the specification of actions that can occur simultaneously and the prevention of undesirable interactions between these concurrent actions. The lack of suitable software abstractions to specify concurrent behaviour makes concurrent software harder to develop and maintain as well as reduces their reuse potential.

Several high level concurrency constructs have been proposed in the last fifteen years in Concurrent Object Oriented Languages (COOLs). These constructs can help to structure concurrent programs to manage this increase in complexity. However, current mainstream object oriented languages, such as Java or C#, do not include many of these high level abstractions. More recently, some of these constructs were revisited as a set of patterns, which represent recurring solutions to frequently occurring concurrency problems.

Use of high-level concurrency patterns and mechanisms in object oriented languages can lead to implementations where concurrency concerns are tangled and scattered over domain-specific code. This harms modularity, maintainability and reuse of patterns code.

Aspect oriented programming promises to improve modularisation of crosscutting concerns which can help to manage complexity and to enable code reuse. This dissertation explores the suitability of aspects as an alternative to object oriented implementations of concurrency patterns and mechanisms with the intention of achieving a better modularity, reuse and unpluggability of concurrency-related code. To this effect, this dissertation presents AspectJ implementations of well-known high-level concurrency patterns and mechanisms. These implementations are based on abstract aspects and aims to improve the reuse of concurrency constructs. A pointcut model is used to introduce concurrent behaviour in domain-specific code and the use of annotations is proposed in the most cases as an alternative to explicit pointcut specification.

Keywords: Aspect Oriented Programming, Concurrency Patterns, Reusable Aspects.

Resumo

A programação concorrente impõe um nível de complexidade superior ao da programação sequencial. A concorrência é responsável pela especificação de acções que podem decorrer em simultâneo, assim como pela prevenção de interacções indesejáveis entre essas acções. A falta de abstracções adequadas à especificação de concorrência dificulta o desenvolvimento e manutenção de software e reduz o seu potencial de reutilização.

Nos últimos quinze anos foram propostas várias abstracções de concorrência, apresentadas sob a forma de linguagens orientadas a objectos para concorrência. As abstracções presentes nessas linguagens tinham como objectivo a estruturação de programas concorrentes, de forma a gerir o aumento de complexidade introduzido pela concorrência nas aplicações. No entanto, as linguagens orientadas a objectos mais populares, tais como, o Java e o C# não incluem muitas dessas abstracções de alto nível. Mais recentemente, algumas delas foram publicadas como padrões de concorrência que representam um conjunto de soluções recorrentes para problemas comuns.

A utilização de padrões e mecanismos de concorrência nas linguagens orientadas a objectos podem levar a implementações onde o código associado às facetas de concorrência está emaranhado e disperso no código específico do domínio. Como consequência, a modularidade, a manutenção e a reutilização do código dos padrões são prejudicadas.

A programação orientada a aspectos tem como motivação principal a modularização de facetas transversais nas aplicações, que contribuem para o aumento da complexidade e para a redução da reutilização do código das aplicações. Esta dissertação pretende explorar a adequação da programação orientada a aspectos para substituir as implementações dos padrões e mecanismos de concorrência orientadas a objectos do ponto de vista da modularidade, reutilização e desacoplamento do código de concorrência das aplicações. Nesta dissertação é apresentado um conjunto de implementações de padrões e mecanismos de concorrência em AspectJ. As implementações encontram-se estruturadas sob a forma de aspectos abstractos e pretendem aumentar o potencial de reutilização das implementações dos vários padrões. É utilizado um modelo de composição baseado em *pointcuts* para acoplar o código da concorrência ao código específico de domínio. A utilização de anotações é considerada na maior parte dos casos como alternativa à definição explícita de *pointcuts*.

Palavras chave: Programação Orientada a Aspectos, Padrões de Concorrência, Aspectos Reutilizáveis.

Acknowledgments

I wish to express my gratitude to everyone who contributed, directly or not, to make this dissertation. I would like to acknowledge particularly the valuable insights and observations contributed by the following: Professors João Luís Sobral, and Miguel Pessoa Monteiro.

This work was partially supported by PPC-VM project POSI/CHS/47158/2002, funded by Portuguese Fundação para a Ciência e Tecnologia (FCT) and by European funds (FEDER).

To my wife Michelle and my little boy Ivo

Contents

Figures

Tables

Acronyms

AJDT	AspectJ Development Tool
AO	Aspect Oriented
AOP	Aspect Oriented Programming
AOSD	Aspect Oriented Software Development
CMP	Chip-level Multiprocessing
FIFO	First In First Out
IDE	Integrated Development Environment
JDK	Java Development Kit
JGF	Java Grand Forum
LIFO	Last In First Out
OO	Object Oriented
OOP	Object Oriented Programming
PD	Pointcut Designator
PM	Patterns and Mechanisms
RW	Readers-Writer
SMP	Symmetric Multiprocessing
TLP	Thread-level Multiprocessing

Chapter 1. Introduction

Concurrent programs define actions that may be performed simultaneously. Contrasted with sequential programs, where only one activity is running at a given time, concurrent programs specify two or more sequential activities that may be executed concurrently as parallel processes (or threads) [2][1].

Concurrent programming is gaining importance as built-in support in recent object-oriented (OO) languages, such as C# and Java [29][38] and is essential to leverage the fast growing multi-core CPU market [45]. Until now, concurrent applications have been widely used primarily on systems that do a lot of processing – e.g., Web servers. However, with the advent of multi-core processors, concurrency is becoming mainstream for conventional applications (e.g., desktop applications), as long as sequential applications do not benefit from multiple cores. Notwithstanding inevitability of preparing applications to run concurrently, programming with concurrency using traditional languages is a complex task, usually left to experts. Debugging concurrent applications is equally complex, mainly due to the execution unpredictability introduced by concurrency. It would be often hard to trace incorrect behaviour to its underlying cause – i.e., trace the defect to concurrent or base functionality.

Concurrency related concerns do not align well with class decomposition, and therefore source code related to such concerns suffers from the well-known negative phenomena of code scattering and tangling [24]. Examples of concurrency-related concerns are: specification of tasks that can run in separated threads; definition of sections of behaviour that must be subject to synchronised access in order to avoid race conditions – and applying the right scheduling policies to those parts –; and creation of barriers where threads must synchronise. In addition, concurrency constructs are the cause of the widely-known inheritance anomaly [34][35], caused by the presence of conflicting decompositions [36] – the core functionality and concurrency-related concerns. Various concern-specific languages [3][36][30][41] were also proposed to avoid the inheritance anomaly by making the concurrency control less tightly coupled to the core functionality language.

Various efforts have been carried out to improve the development of concurrent applications. Specialised libraries such as those provided by Java 1.5 Tiger [21] help to reduce the number of lines of code needed to add concurrent behaviour to applications.

However, specialised libraries fail to address the problems associated with crosscutting. New languages have been proposed that provide alternative abstractions, which incorporate high-level concurrency constructs [17][30]. ABCL [48] is an early example using active objects, one-way calls and futures to model concurrency.

Aspect-Oriented Programming (AOP) was proposed to deal with crosscutting concerns [24]. AOP promises to bring to concurrency-related concerns the usual benefits of modularisation, namely, improved code readability and analysability, and more independent development, testing and configurability. Modularisation is a necessary condition to attain reusability and (un)pluggability.

In [15], Hannemann and Kiczales present implementations in Java and AspectJ of the 23 Gang-of-Four patterns and provide an analysis. In their concluding remarks, they mention several directions for further experimentation, including applying AspectJ to more patterns. This dissertation provides a contribution in that direction, by presenting an Aspect Oriented (AO) collection of reusable high-level concurrency patterns and mechanisms that revenue from the aforementioned AOP benefits. Such collection includes One-way and Future calls, Waiting Guards, Readers-Writer locks (RW-Locks), Barriers, Scheduler, *Synchronized* and Active Objects [29][40][48][28]. The collection is coded in AspectJ [46] and built on top of Java's concurrency mechanisms. It does not include mechanisms such as semaphores and monitors, as these are not usually provided in concurrent object oriented languages [47]. Developing the collection gives rise to the following questions – this work provides a contribution to answering them:

1. What are the main benefits and drawbacks from going to a modern OO concurrency pattern implementation to an AO implementation?
2. Can we replace concurrent OO approaches using this collection?

Support for Annotations whenever possible is considered. Annotations can simplify the quantification process and describe the programmer intentionality [25].

This dissertation does not dictate how to design applications to attain a separation of concurrency from domain specific code. A deep study about composition of pattern implementations and application-level benchmarks are also issues that are not addressed in this dissertation. The code used in this dissertation can be subject to further optimisations – e.g., by using Java 5 constructs –, although, optimisation is not considered an essential concern to achieve the objectives proposed for this dissertation.

This dissertation is structured as follows. The next chapter (Chapter 2) introduces concurrent programming, describing several concepts that represent the basics of parallelisation and synchronisation. Chapter 3 presents an overview of well known patterns and mechanisms for parallelism and synchronisation and illustrating them with Java implementations. The core Aspect Oriented Programming concepts are presented in Chapter 4, as well as an introduction to AspectJ language, which includes the annotations meta-data facility. AspectJ implementations of concurrency patterns and mechanisms are described in Chapter 5 and the corresponding validation with several illustrated examples is exposed in Chapter 6. Performance issues are addressed in Chapter 7. Discussion about AOP pattern implementations, the comparison with equivalent object oriented implementations and the analysis of benefits and drawbacks of each approach is addressed in Chapter 8. Chapter 9 concludes this dissertation.

Chapter 2. Concurrent programming

This chapter introduces the fundamentals of object oriented concurrent programming. An overview of basic principles of creation of parallelism, synchronisation and reusability of concurrency in object oriented applications is provided.

Concurrent programs contain independent processing steps – at block, statement and expression level – that may be executed in parallel. When concurrent behaviour is specified by the program designer it can be termed *explicit concurrency*. Detection of concurrency implies identification of sequences of independent operations that can be executed in parallel. Parallel execution of code aims to improve performance and responsiveness.

A distinction between concurrent, parallel and distributed programs should be made at this point. Precise meanings for these terms are not given regularly. A concurrent program defines actions that may be performed simultaneously. Concurrency refers to devising a program in terms of a set of activities that may overlap in time, whilst preserving safety and liveness properties – e.g. by avoiding race conditions, live-locks, starvation and deadlocks. A parallel program is a concurrent program that is designed to be executed in a parallel computer, by more than one physical processor (or more than one processor core). A distributed program is a program designed for an execution on a network of non-shared memory processors [5][8].

2.1 Parallel Systems

Flynn's taxonomy [10] classifies parallel computers in terms of kinds of interconnection between processors:

- SISD (Single Instruction Single Data) – sequential computer which a single stream of instructions is applied to a single stream of data in a sequential manner; it exploit no parallelism – e.g., traditional uni-processor machines.
- SIMD (Single Instruction Multiple Data) – the same instruction is applied to multiple data in parallel.
- MISD (Multiple Instruction Single Data) – multiple instruction streams operate on a single data stream – e.g., to achieve redundant parallelism.

- MIMD (Multiple Instructions Multiple Data) – multiple instruction streams simultaneously processing different data.

MIMD computers enable multiple processors simultaneously executing different instructions on different data, accordingly to the following classifications:

- Symmetric Multiprocessing (SMP) : several processors sharing the physical memory (Figure 1);
- Distributed Memory Systems: each processor has a different address space where the communication between processors is done usually through message-passing;
- Distributed Shared Memory: physically distributed applications with a simulated shared memory. Applications share the same memory logical space.

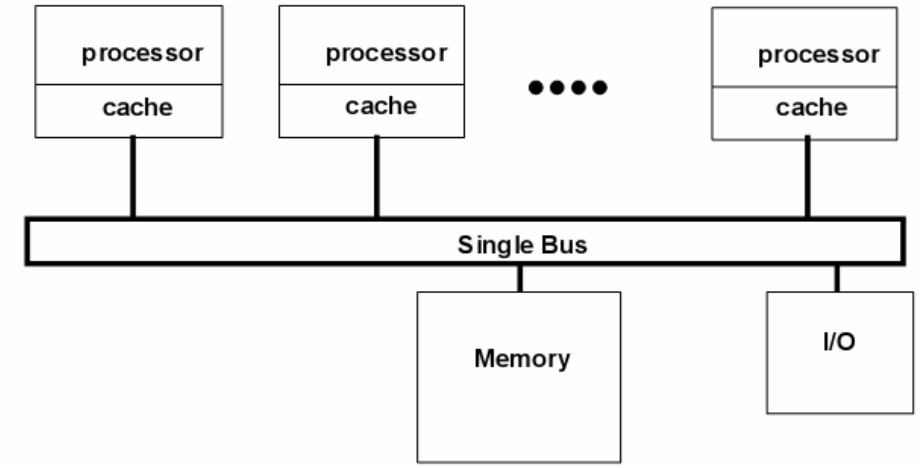

Figure 1 – MIMD SMP architecture

Recently a MIMD type of machines, CMP (Chip-level Multiprocessing) combines two or more processor cores into a single package – i.e., inside a single integrated circuit. CMP is a specific implementation of a SMP. Figure 2 presents a CMP architecture with shared cache. CMP can take advantage of Thread-level Parallelism (TLP) by running multiple threads in parallel. Thus, applications are able to tolerate I/O intensive tasks and high memory access latency.

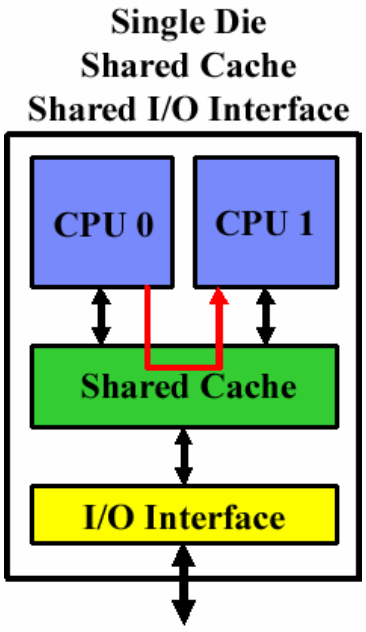

Figure 2 – MIMD CMP architecture

2.1.1 Software-level parallelism

The boost of CMPs acceptance in domestic computers enhances the importance to study new ways to develop parallel applications, in order to leverage processor resources. To implement TLP, applications should be prepared to parallelism – e.g., by dividing the program execution into independent tasks and synchronise access to shared data to avoid race conditions[1]. Traditional programming languages offer little support to the development of parallel applications. Some efforts have been done to evolve languages with the purpose of simplifying the development of concurrent applications – e.g., the Java 1.5 Tiger concurrency library.

Parallelism aims to improve applications performance by increasing the number of simultaneously concurrent activities and/or improve application responsiveness. Patterns for parallelism do not provide new constructs for parallelisation. Instead, they support parallelisation by providing a flexible and structured way to include the functionality required to create and manage concurrent activities.

[1] A race condition represents an anomalous behaviour due to unexpected critical dependence on the relative timing of events.

PMs granularity is three-fold: (1) Object level granularity, (2) Method level granularity and (3) Field level granularity.

Active Object pattern (section 3.1.1) implements object level parallelism, through the creation of a dedicated concurrent activity per each object, which means that each object has its own thread. One-way (section 3.1.2), Futures (section 3.1.3), Barrier (section 3.2.3), *Synchronized* (section 3.2.1), Waiting Guards (section 3.2.5), RW-Lock (section 3.2.2) and Scheduler (section 3.2.4) support method level granularity. Field level granularity is supported by Barrier, *Synchronized* and Scheduler.

2.2 Synchronisation

Synchronisation concerns coordination of simultaneous activities in order to complete a task, by preserving the correct runtime order and avoiding race conditions. Synchronisation specification is two-fold: (1) *exclusion synchronisation* and (2) *condition synchronisation*. Exclusion synchronisation restricts concurrent activities in critical sections to protect them against data inconsistency due to simultaneous access for writing. Condition synchronisation delays concurrent activities until some precondition is met.

2.2.1 Exclusion Synchronisation

In object oriented multithread programming an object could be accessed by many threads simultaneously. In consequence, *data races*[2] can occur if applications are not prepared to deal with concurrency. Such applications should be *thread safe*. *Thread safe* property ensures that all objects always maintain consistent states. An object is in a consistent state if its fields and the fields of other objects on which it depends have legal and meaningful values. Two types of conflicts were identified in [29]:

- **Read/Write conflicts -** One thread reads a field value while another thread changes it. The value read is hard to predict if such value was not atomically updated;
- **Write/Write conflicts** - Two threads are accessing a shared field to write. Resulting value is unpredictably determined if each update was not done atomically.

[2] *data races* occur when two or more threads attempt to access a shared variable and at least one of them change the value in a non atomic way.

Exclusion synchronisation restricts execution within critical regions in two alternative ways:

1. By allowing only one activity at a time to be running on critical regions (*mutual exclusion*) – e.g., Scheduler (section 3.2.4) and Java *synchronized* mechanism;
2. By stratifying the activity types with the purpose of achieving a less restrictive form of synchronisation, allowing many concurrent activities that do not interfere between each other inside a critical region, e.g., the Readers-Writer (section 3.2.2) which allows multiple readers or one writer.

2.2.2 Condition Synchronisation

Condition synchronisation delays activities until some condition is met. When such a condition involves the state of data, we term it *data synchronisation*. Waiting guards (section 3.2.5) fall in this category. Whenever the condition reflects the state or progress of an activity, we term it *activity synchronisation* [19] – e.g., Barrier (section 3.2.3).

2.3 Safety and Liveness

Many properties particular to concurrent applications can be classified as either a safety or a liveness property [27]. Concurrent applications should ensure both properties.

Safe applications would never transit to an inconsistent state. An object is consistent if all fields obey their invariants. Usually, safety is assured by exclusion synchronisation mechanisms. Though exclusion mechanisms preserve safety, they can be the cause of liveness problems. In live systems, every activity eventually completes and every invoked method eventually executes.

Deadlocks and starvation are side-effects of exclusion and the main cause of liveness problems. A typical deadlock situation occurs when one thread acquires an object lock, another thread acquires a second lock, and then both wait forever for the other lock to be released (Figure 3). Starvation occurs when an activity never gets enough resources to run to completion. One possible cause is the existence of higher priority threads with precedence over lower-priority threads. Consequently, lower-priority threads could never execute.

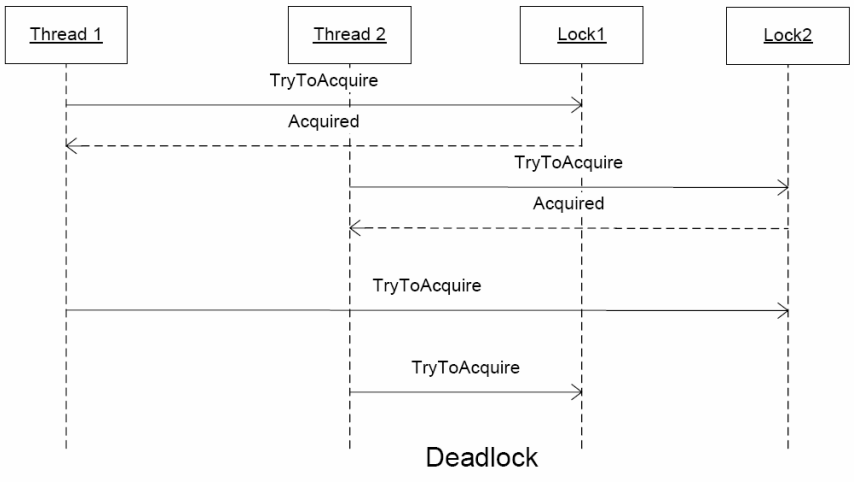

Figure 3 – Deadlock occurrence

Development of concurrent applications represents a constant compromise between safety and liveness, as one affects the other. It is desirable to reduce liveness, though maintaining safety. Several alternative design principles are suggested by Lea [29]:

- **Immutability** – creation of immutable objects that are substituted when a state change is required. Such objects will never be changed – as they are substituted by an new object – and consequently do not need synchronisation;
- **Confinement** – use of encapsulation techniques to structurally guarantee that only one activity can access a given object at a time.

Other important techniques can be used to relax synchronisation and continue keeping the safety property, as the following:

- **Readers-Writer lock** – differentiates access type, by distinguishing threads that read object state from the ones that change state;
- **Splitting synchronisation** – divides class behaviour into non-conflicting subsets to use fine-granularity helper objects which implements the synchronised code.

In addition, more specific techniques as Read-only adapters, Copy-on-Write and Open Containers can be used to reduce synchronisation. Further information can be found in [29].

2.4 Reusability

Classes are potentially reusable units that should be used across several contexts. Protecting classes to be safe across multiple contexts requires the use of concurrency constructs. Notwithstanding, programs which use safe objects can have liveness problems, as long as threads can be blocked inside objects, and thus originating deadlocks.

Classes would be reused either as black-box – where only interfaces are exposed – or white-box components – by having their implementations accessible to programmers. Black-box components that implement concurrency can have a limited reusability if appropriate documentation is not available, for the reason that, usually, synchronisation information does not appear on component interfaces. The lack of documentation can also be a problem when synchronisation is relaxed to improve liveness, seeing that components are secure only in restricted usage contexts.

Inheritance anomaly [34] is caused by the presence of conflicting decompositions: base functionality and concurrency-related concerns. To circumvent inheritance anomaly issues, modularisation of concurrency code in applications is required to enable the reuse of domain specific code without concurrency.

Design patterns offer flexible solutions that help to structure applications to solve common software development problems. Chapter 3 presents implementations of concurrency patterns used recurrently in concurrent object oriented programs.

Chapter 3. Concurrency patterns and mechanisms

This chapter presents an overview of some well known patterns and mechanisms used in object oriented concurrent applications. Lea [29] and Schmidt [40] describe the patterns and mechanisms covered by this dissertation. Such concurrency constructs have been used to model concurrency for years. Sections 3.1 and 3.2 present an overview of each construct.

Design patterns are general repeatable solutions to commonly-occurring problems in software design [12]. A design pattern describes one solution to a problem that can occur in different contexts, through the definition of relationships and interactions between classes or objects, without specifying their final implementation [40]. Mechanisms differ from patterns as they solve computational specific problems rather than design problems.

The classic patterns and mechanisms described in this chapter are classified into two categories: parallelism and synchronisation.

Parallelism related patterns cover the creation and management of parallelism associated to tasks which are to be executed in separate processes or threads. We provide one reference implementation of each pattern and mechanism. One-*way* (section 3.1.2), Futures (section 3.1.3) and Active Objects (section 3.1.1) fall into this category.

Synchronisation relates to the control of interactions of concurrent activities in order to avoid undesirable execution sequences and the coordination of threads by delaying concurrent activities. Examples of synchronisation patterns are *Synchronized* (section 3.2.1), RW-Lock (section 3.2.2), Barrier (section 3.2.3), Scheduler (section 3.2.4) and Waiting Guards (section 3.2.5).

3.1 Parallelism patterns and mechanisms

This section presents patterns and mechanisms for parallelism. An outline of Active Object, One-way call and Future is presented with reference to their characteristics, architecture and implementation.

3.1.1 Active Object

Active object decouples the invocation of methods from their execution [28][29]. Each object runs into its own thread of control. Whenever a client object invokes a method from an active object, the thread associated with the active object carries out the execution. Active object merges the concept of object with the concept of process as in ABCL [48].

Several applications can benefit from the use of active objects. Client applications such as windowing systems and network browsers can employ active objects to simplify concurrent, asynchronous network operations. Multi-threaded servers, producer/consumer and reader/writer applications are also well suited for the use of active objects. Two main advantages were identified from this approach:

- Synchronisation on active objects is straightforward to program. Object synchronisation is carried out on the pattern, through the sequentially execution of method invocations by the active object thread;
- Regular client-side asynchronous calls impose the overhead of spawn one thread per each invocation. By contrast, active objects execute threads asynchronously from the client point of view, by enabling non-blocking calls using the active object thread of control. Therefore, clients perform non-blocking calls without supporting the cost of creating one thread per invocation.

Traditional implementations of active objects are structured into three layers [40] (Figure 4). The first layer includes the object that makes the call, the second layer comprises the mechanisms that forward the call to the target object and the third layer has the target object running in a dedicated thread that is continuously waiting for method invocations. In Figure 4, the method names colored as grey are executed by the active object dedicated thread.

A new subclass of *MethodRequest* (Figure 5) should be created per each method implemented by the *Server Object*. Such class – represented in Figure 5 as *RequestMethodX* – should define the inherited abstract method *call*, which invokes the target method on *Server Object*. An instance of *Proxy* and reusable classes – i.e., *Scheduler*, *ActivationList*, *MethodRequest* and *Future* – are created per each active object instance.

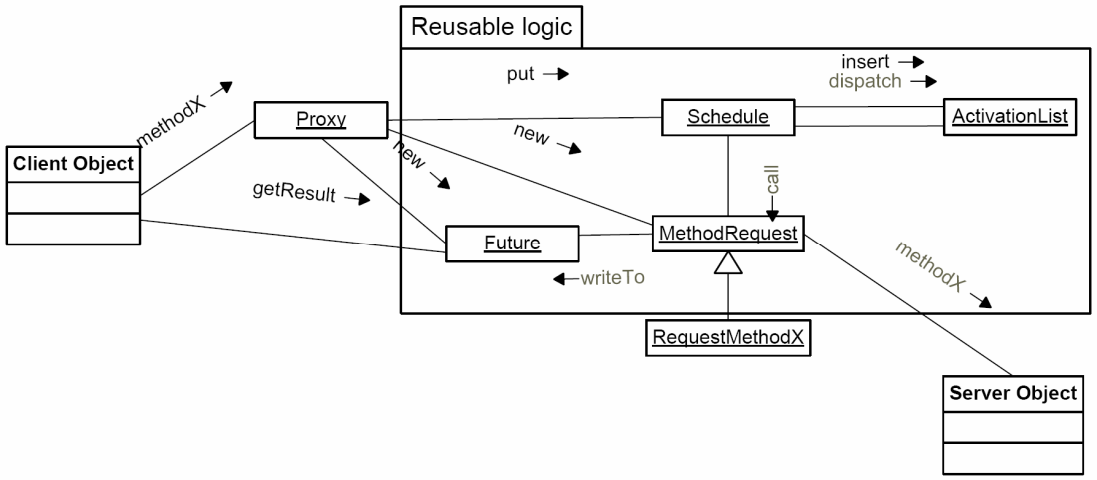

Figure 4 – Active Object Pattern

Client objects perform method invocations on the respective *Proxy* object, which instantiates the corresponding subclass of *RequestMethod* and calls the method *put* on *Schedule* object. In its turn, the *Schedule* (Figure 6) inserts the request object into *ActivationList*. The execution of those steps is performed by the *Client Object* thread. The *Server Object* dedicated thread – i.e., the active object thread – invokes the *dispatch* method on *Schedule*, which picks the request from the *ActivationList*, carrying out its execution. Optionally, a *Future* object may be used to enable client asynchronous invocations of non-void methods on the active object.

Active object implementation is heavy as, in addition to reusable related objects, a new *Proxy* class must be created per each active object and a new *MethodRequest* subclass for each method of that active object class.

```
public abstract class MethodRequest {
   ...// fields

   public void setParameters(Object serv, Object args[], Message res){
      servant = serv;
      if(args != null){
         param = new Object[args.length];
         for(int i = 0 ; i < args.length ; i++) param[i] = args[i];
      }
      result = res;
   }

  public Message getResult(){  return result; }

   public abstract void call();
}
```

Figure 5 – MethodRequest class definition

```
public class Scheduler implements Runnable{
  private ActivationList actList;

  public Scheduler(int size){
    actList = new ActivationList(size);
  }

  public void run(){
    try{
      dispatch();
    }catch(InterruptedException ie){ ...// exception handling
    }catch(TimeOverException ie){  ...// exception handling
    }
  }

  public synchronized void insert(MethodRequest mr){
    actList.insert(mr);
  }

  public void dispatch() throws
       InterruptedException,TimeOverException{

    for(;;){
      actList.getFirst().call();
    }
  }
}
```

Figure 6 – Scheduler implementation

3.1.2 One-way

One-way mechanism applies to methods that run on a thread of their own: the client never blocks, waiting for some result. One-way [29] pertains only to asynchronous void method calls – when the method does return a value, a future (section 3.1.3) should be used.

One-way calls can improve throughput in cases in which parallel tasks can run faster than the purely sequential counterparts – e.g., when there are more available execution units than concurrent activities. Furthermore, it contributes to improve responsiveness, as applications can serve many requests concurrently and, consequently, if one of them stalls waiting for some resource, another request can be processed in the meanwhile.

Figure 7 shows the sequence diagram representing the interaction between the client object, the spawned thread and the target object. Whenever the asynchronous method is invoked, a new thread is created to execute that method. Figure 8 shows the implementation of One-way: a *Runnable* class – represented by *OnewayCall* – ought to be created for each asynchronous method. The invoker method creates a new thread, passing the *Runnable* object as parameter of Thread constructor and, afterwards, starts the thread.

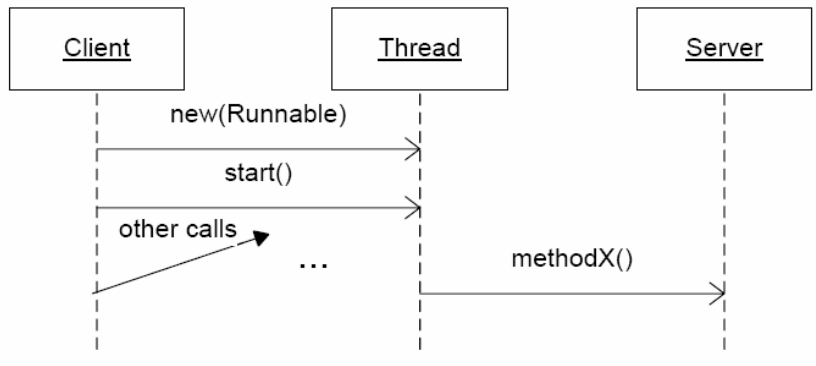

Figure 7 – Sequence diagram of One-way method invocation

```
class RunEverything{
   ...//fields and other methods

   public void execute(){
     OnewayCall obj = new OnewayCall(...//parameters);
     Thread th  = new Thread(obj);
     th.start();
   }
}

class OnewayCall implements Runnable {
   ...//fields

   public OnewayCall (...//parameters) {
      ...//constructor code
   }

   public void run() {
      ...// code executed by the new thread
   }
}
```

Figure 8 – One-way implementation

3.1.3 Future

Futures [29] allow two-way asynchronous invocations of methods that return a value to the client. Futures are join-based mechanisms based on data objects that automatically block when clients try to use their values before the corresponding computation is complete. During the execution of methods, the Future is a placeholder for the value that has not yet materialized.

In typical situations, futures are used when a variable stores the result of a computation, which will not be used immediately. Consider the following code:

```
a = someobject.compute();
...// other statements
a.doSomething();
```

The *compute* method is executed by the spawned thread. Instead of blocking at the computation phase – i.e., during the execution of *compute* – the client thread blocks when the variable is actually accessed – i.e., when the method *doSomething* is executed.

Figure 9 shows the sequence of operations occurring when a future is used. Firstly, a new thread is created in a similar way to a one-way call. The client continues executing and only blocks when it needs to access the real value returned by the called method. At that time it accesses the *Future* object (Figure 10) – which assumes the role of mediator between the client and the method executed by the new thread – and blocks if the value is not available yet.

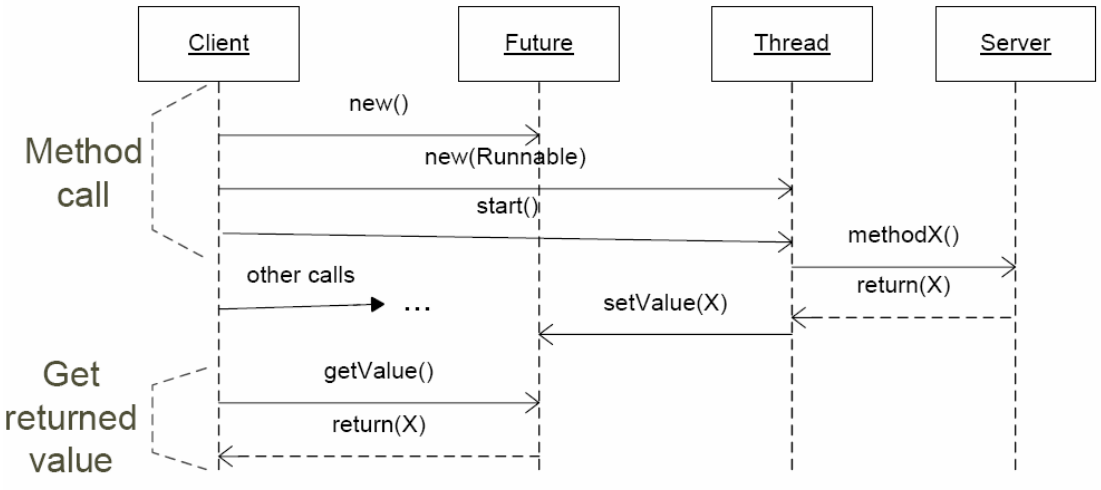

Figure 9 – Sequence diagram of Future mechanism

```
public class Future {
   Object value = null;

   Future(){ }

   public synchronized Object getValue(){
      try{
        while(value == null) wait();
      }catch(InterruptedException ie){
        ...// Exception handling
      }
      return value;
   }

   public synchronized void setValue(Object obj){
      value = obj;
      notifyAll();
   }
}
```

Figure 10 – Future class definition

3.2 Synchronisation patterns and mechanisms

This section presents patterns and mechanisms used to implement synchronisation in concurrent applications, namely *Synchronized*, Readers-Writer, Barrier, Scheduler, and Waiting Guards. Each pattern is characterised with a short implementation in Java language.

3.2.1 *Synchronized*

In Java each instance of class *Object* or its subclasses holds a lock. Object locking obeys to an acquire-release protocol controlled by the *synchronized* keyword. A lock is acquired on entry to a *synchronized* method or block and released on exit. Threads, before acquiring a lock on a *synchronized* construct, should wait until the current lock-owner thread releases it. As a lock operates on a per-thread basis, the lock-owner thread would execute other methods in the same object without acquiring the lock again – i.e., lock is re-entrant.

Synchronisation is implemented by using the *synchronized* modifier at method level or the *synchronized(Object){ }* construct at instruction or block level (Figure 11). Whilst in the former the critical region is the method code and the associated lock is the object lock, in the latter, we can enclose a set of instructions in a critical region associated with the lock of other objects. This can be useful to synchronise code scattered throughout many methods defined in one or more classes, using the same lock.

```
public class Point {
   //...
   public synchronized void moveX(int delta) {  x+=delta; }
   public synchronized void moveY(int delta) {  y+=delta; }
   public void restoreOriginalSettings(){
      synchronized(this){
         x = 0;
         y = 0;
      }
      setColor(0);
   }
}
```

Figure 11 – Example of the use of *synchronized* construct

3.2.2 Readers-Writer lock

The synchronisation mechanism allows a single thread to enter a critical section of code for reading or writing. RW-Lock differentiate accesses that change object state from the ones that just read state, allowing multiple simultaneous readers or one writer, exclusively. Access for reading is allowed when no writers are executing or waiting to access object state whereas writers can access state when there are no writing or reading operations executing (Figure 12).

Readers-Writer lock fits well when access is predominantly for reading purposes. This assumes that methods in a class can be semantically separated into those that read variables values – i.e., *Readers* – and those that change such values – i.e., *Writers*. In such situations we may have multiple simultaneous readers, contrasted with traditional pure synchronisation where only one reader/writer is allowed. RW-Lock requires a more complex synchronisation scheme which introduces more overhead, compared with traditional *synchronized* mechanism, when accesses are mostly for writing.

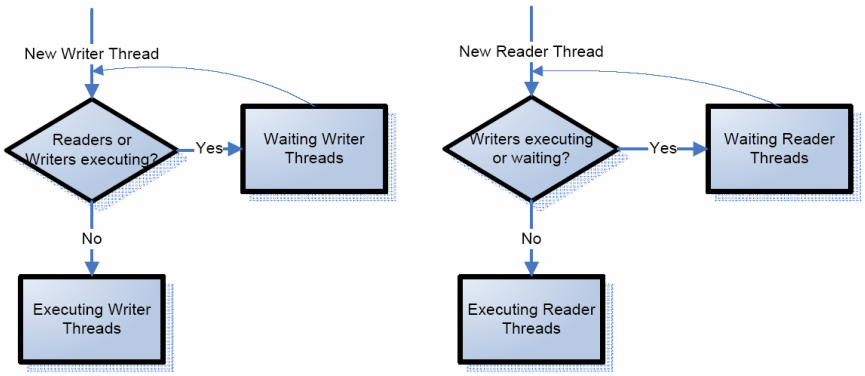

Figure 12 – Readers-Writer logic

RW-Lock implementation uses two locks to implement the mechanism. An example of the use of RW-Lock is given in Figure 13. Class *List* implements two methods: *put* (writer) and *get* (reader). Each time a reader or writer method is executed, the operation *acquire* attempts to get the respective lock before accessing the values. The *release* operation releases the lock.

```
class List{

  protected final RWLock rw = new RWLock();

  public void put(Object ob) throws InterruptedException{
    rw.writeLock().acquire();

    try{
      ...// insert new element
    }finally{
      rw.writelock().release();
    }
  }

  public Object get() throws InterruptedException{
    rw.readLock().acquire();

    try{
      ...// get element
    }finally{
      rw.readlock().release();
    }
  }
}
```

Figure 13 – Example of RW-Lock usage

The implementation of RW-Lock allows the participation of methods defined in distinct classes in the same RW-Lock instance, since the reference of that instance can be shared between the objects.

Lock management functionality is defined in class *RWLock*, shown in Figure 14. The *beforeRead* method blocks each reader thread when there are writer threads running or waiting to run. The *beforeWrite* method blocks writer threads when there are reader or writer threads running.

```
public class RWLock{

   ...// variables

   protected boolean allowReader(){
      return waitingWriters == 0 && activeWriters == 0;
   }

   protected boolean allowWriter(){
      return activeReaders == 0 && activeWriters == 0;
   }

   protected synchronized void beforeRead() throws
        InterruptedException{
      ++waitingReaders;
      while(!allowReader()){
         ...// exception handling
         wait();
         ...// exception handling
      }
      --waitingReaders;
      ++activeReaders;
   }

   protected synchronized void beforeWrite() throws
        InterruptedException{

      ++waitingWriters;
      while(!allowWriter()){
         ...// exception handling
         wait();
         ...// exception handling
      }
      --waitingWriters;
      ++activeWriters;
   }

   public synchronized void afterWrite(){
      --activeWriters;
      notifyAll();
   }

   public synchronized void afterRead(){
      --activeReaders;
      notifyAll();
   }

   public RLock readLock() {
      return rlock;
   }

   public WLock writeLock() {
      return wlock;
   }

   ...// implementation of private classes RLock and WLock
}
```

Figure 14 - RWLock class definition

3.2.3 Barrier

Many iterative problems waste potential parallelism by using coarse granularities. Segmentation of iterative problems is required to improve performance, by creating one parallel activity per each segment. When segments are dependent between each other, each activity executing one segment should periodically wait for the others to complete – e.g., when data reduction[3] is necessary to merge the results of a computation.

Barrier [29] is a *condition synchronisation* mechanism – more precisely an *activity synchronisation* – used to set blocking points for a set of threads. Threads reaching such points block until a specified threshold number of blocking threads is reached (Figure 15). Usually, Barrier is applied to a set of concurrent activities that cross one particular execution point, forcing those activities to synchronise at that point.

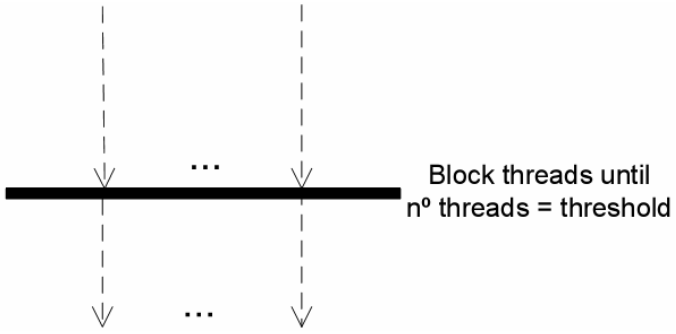

Figure 15 – Barrier synchronisation

Figure 16 shows the code of Barrier while Figure 17 shows the application of barrier to a set of tasks, each one associated to a distinct thread. After each iteration, threads invoke the *barrier* method on *Barrier* type object – all parts share the same *Barrier* instance passed by a constructor parameter of each *Part* object – where it waits for the other threads to finish the iteration. For instance, if barrier has a threshold of ten threads, each of the parts block when the method *barrier* is invoked and unblocks when the tenth thread calls the method.

[3] Data reduction refers to the process of merge the results of some computation executed by several threads/processes.

```
Class Barrier{
   ...// fields

   Barrier(int n){ /*...*/ }

   //...

   protected synchronized void barrier(){
      int index = --count;
      if(index > 0){
         int r = resets;

         try{
            do{ wait(); } while (resets == r);
         }catch(InterruptedException ie){
            ...// exception handling
         }

      } else {
         count = nThreads;
         ++resets;
         notifyAll();
      }
   }
}
```

Figure 16 – Barrier implementation

```
class Part implements Runnable {
   final Barrier b;
   ...// other fields and methods

   Part(Barrier barr, /* ... */){ b = barr; }

   public void execute(){ /* ... */ }

   public void run() {
      //...
      for(int i = 0 ; i< nIter ; i++) {
         execute();
         b.barrier();
      }
      //...
   }
}

class Whole {
   Barrier b = new Barrier(n);
   Threads[] thr = new Thread[n];

   for(int i = 0 ; i < n ; i++){
      threads[i] = new Thread(new Part(b, /* ... */);
      threads[i].start();
   }
   //...
}
```

Figure 17 – Barrier usage

3.2.4 Scheduler

The synchronisation mechanism implemented in many OO languages is inflexible, as object monitors cannot be configured to use a specific scheduling policy. The Scheduler pattern is independent of any scheduling policy and is specified in a per case basis. Specification of a scheduling order can be determined dynamically according to the state of the object and/or the method parameters passed on the call. Readers-Writer (section 3.2.2) is a particular implementation of Scheduler, where a different policy is applied to two different kinds of methods: *readers* and *writers*.

Each thread attempting to execute the synchronised code invokes the method *enter* on *Scheduler* object (Figure 18). In case a thread is currently executing the code, the attempter thread passes to wait state and is inserted in a waiting queue, until the Scheduler wakes it. Each time the running thread finishes execution – by executing method *done* –, a new thread is selected from the queue, according to the specified scheduling policy and carries out the execution. Implementation of method *enter* and *done* is shown in Figure 19. Method *findNextRequest*, invoked in method *done*, enables the selection of the new running thread, according to a specific policy. By redefining the method in subclasses, it is possible to configure the pattern instance with a customised scheduling policy.

Scheduler can be used to implement policies that can optimise synchronisation. As an example, it is possible to specify a priority scheme that allows threads with a higher priority to execute before the others, or use historical data to take decisions about the order of executing threads.

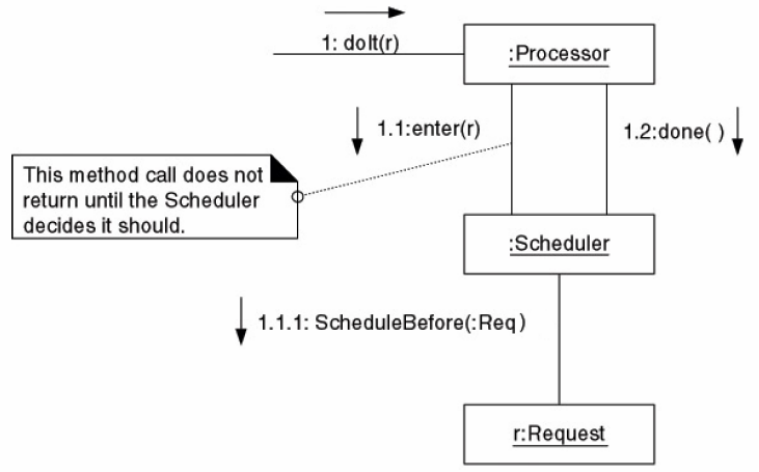

Figure 18 –Scheduling interaction between objects

```
class Scheduler{
   // before execute the method
   public void enter() throws InterruptedException{
      Thread thisThread = Thread.currentThread();

      synchronized (thisThread){
         while(thisThread != runningThread)
            thisThread.wait();
      }
   }

   // after the current thread finishes its execution
   public synchronized void done(){
      if(runningThread != Thread.currentThread())
         throw new IllegalStateException("Wrong Thread");

      int waitCount = waitingThreads.size();

      if(waitCount <= 0)  // if there are no waiting threads
         runningThread = null;
      else { //else determines next request
         runningThread =
            waitingThreads.get(findNextRequest(waitingThreads));
         synchronized(runningThread){
            runningThread.notifyAll();
         }
      }  // if waitcount
   }
}
```

Figure 19 – Scheduler implementation

3.2.5 Waiting guards

Waiting guards is a data synchronisation mechanism that restricts the execution of methods that depends on the state of the object. When some precondition is not satisfied, two situations can occur: (1) raise an exception – also known as balking –, (2) wait until the precondition is satisfied – also known as blocking wait. In balking form implementation, when the invoked method precondition is not met, an exception is raised which should be caught by the client. Afterwards, the client can give up or wait for a predetermined amount of time before invoking the method again. In blocking wait approach, the threads are blocked when attempting to execute methods and the precondition is invalid. Execution is only allowed when the precondition validity changes.

Waiting guards implement the blocking wait policy. When a precondition is not satisfied, the client thread blocks until some action that changes the precondition validity is performed, and then performs a reevaluation of the precondition. This process is repeated until the precondition is valid (Figure 20).

The implementation of Waiting Guards is shown in Figure 21. The precondition is evaluated before the execution of the *someMethod* method body and, when invalidated, the thread goes to waiting state. Methods changing object state that affect the precondition of other methods should perform *notifyAll*, in order to force waiting threads to reevaluate their preconditions.

Waiting guards should not be applied to single-thread applications or the program will stall permanently, because there is no other thread that changes the precondition validity and consequently unblocks the blocked thread.

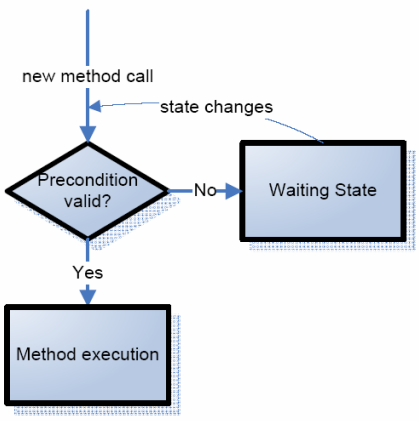

Figure 20 – Waiting Guards logic

```
public synchronized void someMethod() {
   try{
      while(! condition() ) {
         wait();
      }
   }catch(InterruptedException c) {
      ...// Exception handling
   }
   ...// method implementation;
}

public synchronized void methodThatChangesPrecondition() {
   notifyAll();
}
```

Figure 21 – Waiting guards implementation

3.3 Analysis of concurrency patterns

Patterns are proven solutions proposed to solve specific program design issues, which can introduce a great amount of changeability into applications. Notwithstanding, the use of patterns tend to increase application complexity [37]. A relation between changeability and analysability exists: increasing design changeability implies the decreasing of designs analysability, and vice versa. A deep study about this characteristic can be found in [37].

Several problems were identified on the use of object oriented pattern implementations: code scattering [9]; poor modularity [43]; lack of unpluggability, as adding or removing a pattern to/from a system often requires an invasive, difficult to reverse change [4]; and non-reusability of pattern implementations [43]. Pattern implementations lead to crosscutting code that is tangled and spreaded over domain specific code. Patterns loose its modularisation characteristic when they are mapped from design level to code level.

Whilst part of patterns logic is encapsulated and reused in specific classes – e.g., Future, Barrier, Scheduler – parts of the patterns code is spreaded and tangled with participant classes[4]. For instance, a simple pattern such as One-way requires, for each method executed asynchronously, the creation of a *Runnable* class, instantiation of class *Thread* and starting the thread. Such code introduces complexity into class definition and limits reusability of both pattern and participant classes. Patterns code is intrusively added to participant classes, thus preventing its reuse in other, non-related contexts and harm code maintainability. In addition, pattern code is spreaded throughout many participant classes, which prevents reuse of the domain specific code in different contexts.

3.4 Summary

Patterns are design level solutions that can be used to help the development of concurrent applications. Mechanisms help programmers to solve computational problems related with concurrency. Object-oriented implementations of the constructs presented in this dissertation have been used for several years in the development of concurrent applications.

[4] participant classes are classes that defines the base functionality and also defines patterns crosscutting code

A set of traditional concurrency patterns and mechanisms is presented in this chapter. PM classification is two-fold: Synchronisation and Parallel PM. Synchronisation PM is concerned with coordination of concurrent activities in order to avoid access anomalies – e.g., race conditions –, while Parallel PM concerns the creation and management of concurrent activities with the purpose of augmenting the applications performance and responsiveness.

Though pattern implementations improves application flexibility, patterns code is intrusively added to participant classes, preventing the reuse and maintainability of both pattern and participant classes code.

Aspect oriented programming aims to modularise crosscutting concerns, which is a necessary condition to achieve reusability and unpluggability of pattern implementations. The next chapter introduces AOP, presenting its basic concepts, which are mentioned in subsequent chapters.

Chapter 4. Aspect oriented programming (AOP)

This chapter presents the fundamentals of Aspect Oriented Programming and the core constructs of the AspectJ language, which includes the use of idioms to promote reuse of code. An overview of the use of annotations as a quantification model to compose concurrency code with the main functionality code is also presented.

Design of object oriented applications entails system decomposition into smaller units, namely classes. Each abstraction of an object in the real world is associated to one class. Class definition encloses the base functionality and a set of other concerns – e.g., concurrency, persistence, and logging. Each concern should belong to a single module, but is fragmented and tangled with base functionality. Figure 22 illustrates the decomposition of a program into concerns. The domain specific code – represented as black – is tangled with security, persistence and concurrency code, represented as green, read and yellow, respectively. Such code is known as *crosscutting code* or *tangled code* [24] and the subjacent concern is called *crosscutting concern*. One of the crosscutting concerns identified earlier arose in concurrent object oriented applications, where concurrency synchronisation constraints interfere with inheritance, originating the inheritance anomaly problem [34][35].

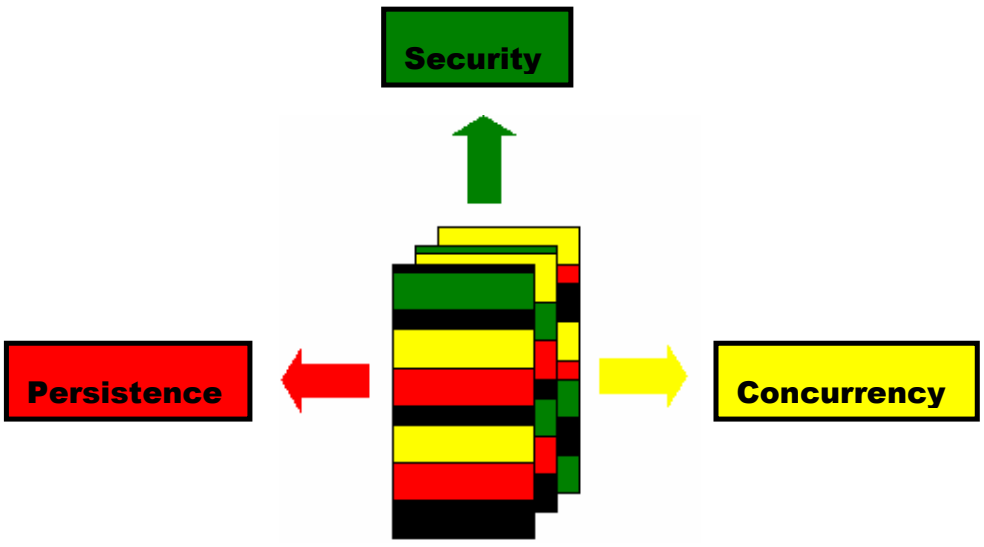

Figure 22 – Coexistence of base functionality and crosscutting concerns

A good design helps to improve modularity in object oriented applications. However, several crosscutting concerns cannot be modularised using the object oriented decomposition model – e.g., some pattern implementations.

AOP makes it possible to localise within a single module code that would be otherwise scattered throughout multiple modules. Concern-specific code is scattered and tangled with functional code, reducing modularity, understandability, code reuse and augmenting applications complexity [37]. Figure 23 shows the ParticleApplet class taken from [29], differentiating the application basic functionality from the concurrency code. Concurrency code – colored as grey – implements thread spawning and management logic and, at the same time, protects classes against data inconsistencies due to concurrent access.

```
public class ParticleApplet extends Applet {
  protected Thread[] threads = null;    // null when not running

  //...
  protected Thread makeThread(//…)
  {
    //utility
    Runnable runloop = new Runnable() {
      public void run() {
        //...
        for(;;) {
          p.move();
          canvas.repaint();
          //...
        }
        ...// exception handling
      }
      return new Thread(runloop);
  }
  public synchronized void start() {
    int n = 10;

    if (threads == null) {
      Particle[ ] particles = new Particle[n];
      for (int i = 0; i < n; ++i) particles[i] = new Particle(50, 50);
      canvas.setParticles(particles);
      threads = new Thread[n];
      for (int i = 0; i < n; ++i) {
        threads[i] = makeThread(particles[i]);
        threads[i].start();
      }
    }
  }
  public synchronized void stop() {
    if (threads != null) {
      for (int i= 0; i < threads.length; ++i) threads[i].interrupt();
      threads = null;
    }
  }
}
```

Figure 23 – Particle Applet class

Aspect oriented programming enables the modularisation of crosscutting code by localising related concerns within aspects. Thus, each class defines only base functionality whilst non-core functionality can be modularised within aspects. This results in simpler code that is easier to develop and maintain, and has greater potential for reuse.

To create the final representation of the system, aspects must be woven with base functionality. Aspect code composes with application code at specific points in the program execution, namely, *join points*. Examples of join points are the invocation or execution of method calls, object instantiation, or the access to a member variable.

AspectJ is the selected language for the development of AOP implementations which support this work. It was chosen for the similarities and compatibility with Java applications and their general acceptance and popularity.

The following sections present some concepts inherent to AOP. Such concepts are illustrated using AspectJ code implementations.

4.1 AspectJ

AspectJ is a general purpose language which provides modularity support for the implementation of crosscutting concerns. AspectJ is an extension to Java, comprising the following characteristics [23]:

- *Upward compatibility* - all legal Java programs are legal AspectJ programs.
- *Platform compatibility* - all legal AspectJ programs must run on standard Java virtual machines.
- *Tool compatibility* - it must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs, documentation tools, and design tools.
- *Programmer compatibility* - Programming with AspectJ must feel like a natural extension of programming with Java.

AspectJ supports two types of crosscutting mechanisms: static and dynamic. Static crosscutting changes the program structure, by allowing the introduction of variables or methods into an existing class. It also allows a class to implement an interface or to derivate from another class. Dynamic crosscutting changes the applications behaviour by using a set of novel constructs added to the Java language. They can be summarised to: *pointcuts*, *advices* and *aspects*.

4.2 Static crosscutting

AOP allows the change to the static structure of classes in a crosscutting way, which includes member introduction, type-hierarchy modification, compile-time error and warning declaration and the softening of checked exceptions.

The intertype declaration mechanism of AspectJ enables member introduction – i.e., fields, methods and constructors – and type-hierarchy modification by adding super-types and interfaces to specific classes – subject to Java's existing type rules. In addition, declaration of errors and warnings is also possible, when pointcuts used in the declaration are statically determinable – i.e., pointcuts that use the pointcut designators *this()*, *target()*, *args()*, *if()*, *cflow()*, and *cflowbelow()* presented in Table 3 and Table 4 cannot be used.

Table 1 shows the general form of each construct of intertype declaration mechanism.

Table 1 – Inter-type declaration constructs

General form	
`[Modifiers] FieldType TargetType.Id;`	`Inter-type field declaration`
`[Modifiers] ReturnType TargetType.Id(Formals) [throws TypeList] { Body };`	`Inter-type method declaration`
`[Modifiers] TargetType.new(Formals) [throws TypeList] { Body };`	`Inter-type constructor declaration`
`declare parents : [ChildTypePattern] implements [InterfaceList];`	`Declaration of new implemented interfaces`
`declare parents : [ChildTypePattern] extends [Class or InterfaceList];`	`Super-type declaration`
`declare error : <pointcut> : <message>;`	`Compile-time error declaration`
`declare warning : <pointcut> : <message>;`	`Compile-time warning declaration`

As an example, AspectJ static introduction presented in Figure 24 adds the interface *Serializable* to the list of interfaces implemented by class *Point*, presented in Figure 25. Additionally, method *migrate* is introduced by the aspect into class *Point*.

```
public aspect StaticIntroduction {
   declare parents: Point implements Serializable;

   public void Point.migrate(String node) {
      System.out.println("Migrate to node" + node);
   }
}
```

Figure 24 - Example of a static crosscutting aspect

```
public class Point {
   private int x=0;
   private int y=0;

   public void moveX(int delta) {  x+=delta; }
   public void moveY(int delta) {  y+=delta; }

   public static void main(String[] args) {
      Point p = new Point();
      p.moveX(10);
      p.moveY(5);
   }
}
```

Figure 25 - Point class definition

4.3 Dynamic crosscutting

Aspects are the units of crosscutting implementations composed by Java language abstractions and AspectJ abstractions, as *pointcuts* and *advices*. Join points are specific points in the program execution; *pointcuts* are used to refer to a collection of join points and to capture their execution-context information and, finally, *advices* define the behaviour added at specific join points.

4.3.1 Pointcuts

Pointcut construct specifies a set of join points and collects context information from those join points.

The general form of a named pointcut is:

<visibility-modifier> pointcut <name>(ParameterList) : <pointcut_expression>;

The *visibility-modifier* can assume the visibility modifiers of Java – i.e., *private*, *default (package)*, *protected*, or *public*. If the pointcut is declared as *abstract*, the *pointcut expression* is absent and the definition of abstract pointcut expressions is done in sub-aspects.

pointcut_expression is built by composing pointcut designators, using the operators *&&*, ||, and *!*. AspectJ pointcut designators (PDs) identify a set of join points, obtained by filtering the program join point space. PDs matching are three-fold: based on join point *kind*, *scope* and join point *context* [6].

Table 2 presents a list of AspectJ pointcut designators to quantify join points according to their kind and using their specification syntax. Such quantification can be restricted to a specific scope (Table 3) and execution context (Table 4). Pointcut designators based on execution context can be used to restrict intercepted objects to certain types and retrieve information from the context to be used by advices – e.g., capturing the reference of the intercepted object.

Table 2- Matching Based on Join Point Kind

Designator	
call(*Method-Signature*)	call to a method or constructor matching *Method-Signature*
execution(*Method-Signature*)	Execution of a method or constructor matching *Method-Signature*
get(*Signature*)	A reference to a field matching *Signature*.
set(*Signature*)	An assignment to a field matching *Signature*.
handler(*TypePattern*)	The handling of an exception of a type matched by the specified *TypePattern*.
staticinitialization(*TypePattern*)	Execution of a static initializer of any type matching *TypePattern*.
initialization(*Constructor-Signature*)	The initialization of an object where the first constructor called in the construction of the instance matches *Constructor-Signature*.
preinitialization(*Constructor-Signature*)	The pre-initialization of an object where the first constructor called in the construction of the instance matches *Constructor-Signature*.
adviceexecution()	The execution of an advice body.

Table 3- Matching Based on Scope

Designator	
withincode (*Method-Signature*)	A join point arising from the execution of logic defined in a method or constructor matching.
within(*TypePattern*)	Matches a join point arising from the execution of logic defined in a type matching *TypePattern*.
cflow(*pcut-expression*)	Matches a join point in the control flow of a join point P, including P itself, where P is a join point matched by *pcut-expression*.
cflowbelow(*pcut-expression*)	Matches a join point below the control flow of a join point P, where P is a join point matched by *pcut-expression*; does not include P itself.

Table 4- Matching Based on Join Point Context

Designator	
Target(*Type*)	Matches a join point where the target object is an instance of *Type*.
target(*id*)	Matches a join point where the target object is an instance of the type of *id* and *id* is bound in the pointcut definition or, for anonymous pointcuts, in the associated advice declaration.
this(*Type*)	Matches a join point where the currently executing object is an instance of *Type*.
this(*id*)	Matches a join point where the executing object is an instance of the type of *id* and *id* is bound in the pointcut definition.
args(*Type*,..)	Matches a join point where the arguments are instances of the specified types.
args(*id*,..)	Matches a join point where the arguments are instances of the type of *id* and *id* is bound in the pointcut definition or, for anonymous pointcuts, in the associated advice declaration.
if(*expression*)	Matches a join point where the Boolean *expression* evaluates to true.

4.3.2 Advices

Advices add behaviour to join points quantified by pointcut predicates. Advices follow the syntax:

```
[before | <Type> around] (<Parameter_list>):
  <pointcut_expression>
{ ...// added behaviour }
  or

after (<Parameter_list>) [returning | throwing] :
  <pointcut_expression>{ ...// added behaviour }
```

The *before* advice adds the specified behaviour before the execution point associated to the join points captured by *pointcut_expression*. *around* advices replace the original behaviour at join points with the advice code. However, the original code can be executed using the *proceed* construct inside the *around* advice definition. *after* advices add the new code to the end of each execution point. *pointcut_expression* embodies an expression comprising the composition of multiple pointcuts.

Figure 26 shows a typical example of a logging aspect, applied to Point class. In this example, a message is printed for every call to methods *moveX* and *moveY*. Pointcut *loggingPoints* specifies the pointcut expression to intercept such methods and captures references of the invoked object and method parameter.

```
public aspect Logging {
  protected pointcut loggingPoints(Point obj, int disp) :
    call(void Point.move*(int)) && target(obj) && args(disp);

  void around(Point obj, int disp) : loggingPoints(obj, disp) {
    System.out.println("Move called: target object = " + obj + "
      Displacement " + disp);
    proceed(obj, disp);       // proceed the original call
  }
}
```

Figure 26 – Logging aspect

4.4 AspectJ idioms for code reuse

Modularisation of crosscutting concerns is an achievement that would lead to code reusability. Though modularisation is a necessary condition to achieve reusability, it is not a sufficient condition, as not all code is reusable. Essential parts of the aspects behaviour are common to many contexts, whereas some behaviour is specific to particular join points.

Reuse of crosscutting concerns requires the capture of reusable code – i.e., behaviour common to multiple implementation instances – to abstract aspects, which can be extended by concrete aspects – i.e., by the aspects which subclasses the abstract ones. Concrete aspects contain the variable parts tailored to a case-specific code base that defines the case-specific join points to be captured in the logic declared by the abstract aspect (Figure 27) and the additional behaviour particular to the instance of use. AspectJ enables the development of reusable implementations of patterns and mechanisms, by moving each reusable part to a separate module – i.e., abstract aspect – independent of any case-specific code.

AspectJ structure of reusable implementation is aligned with the template advice idiom [14]. In template advice idiom, the abstract aspect specifies *hook pointcuts* and/or *hook methods*. Whenever an aspect is reused in a concrete application, the abstract aspect is extended and the *hook pointcuts/methods* should be defined – if it was specified as abstract – or overridden – if it already has a default implementation that it should be changed. An abstract pointcut does not have implementation. It is formed by the pointcut name and the list of parameters captured from the context in the form:

```
protected abstracted pointcut <pointcut_name>(<list of parameters>);
```

Optional pointcuts cannot be declared as abstract or their definition will be required in concrete aspects. However, optional pointcuts can be declared without the pointcut expression, which can be defined in concrete aspects if it is necessary.

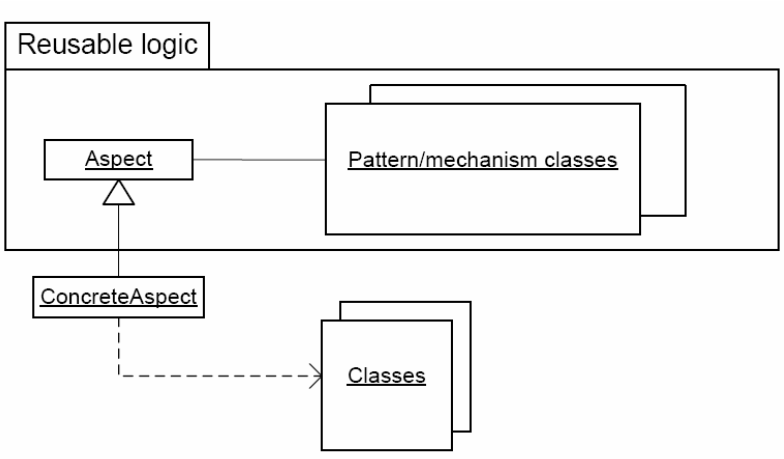

Figure 27 – Reusable implementations structure

Composition of crosscutting concerns with base functionality, at class level, can be facilitated by adding an interface to the list of interfaces implemented by the intercepted class (Figure 28), using the intertype declaration mechanism of AspectJ. Added interfaces are known as marker interfaces [13] and their purpose is to mark classes which are to be intercepted by the aspect. By doing that, definition/redefinition of inherited pointcuts is avoided in view of the fact that aspects intercept all classes implementing such interface. However, marker interfaces can only be used when the quantification granularity performed by the aspect is always the *class*.

Summing-up, abstract pointcuts are used to define the hooks in case-specific code taken by the aspect; abstract methods are used to configure pattern-specific parameters and marker interfaces to annotate the classes used by the reusable code.

```
public aspect <aspect_name> extends <reusable_aspect> {
   declare parents :
      <case-specific class> implements <interface_name>;
}
```

Figure 28 – Using intertype declaration to add new interfaces

4.5 Annotations

Annotations is a metadata facility introduced in Java 1.5 Tiger [21] implemented as modifiers that can be added to the code, more precisely to package declarations, type declarations, constructors, methods, fields, parameters, and variables.

Java 1.5 Tiger includes built-in annotations and supports the creation of new custom annotations. AspectJ 5 enables pointcuts to quantify over annotations, which simplifies the process of quantification, as pointcuts are able to only refer the annotation name instead of an element syntax (Figure 29). Annotations have two basic roles: describe the elements behaviour and provide a hook for aspects to compose. In the former, by describing the elements behaviour, annotations contribute to a better comprehension about the effect of aspects on the domain specific code, without having to analyse the code of each aspect in the application. In the latter, annotations simplify aspect composability, as pointcuts defined on abstract aspects quantify over the annotations inserted directly on base functionality.

Pointcuts suffer from the fragile pointcut problem [44][18] where pointcut declarations result in a tight coupling between aspect and base system. In consequence, non-local

changes may break pointcut semantics. The fragile pointcut problem can be ameliorated by the use of annotations. By using annotations as attributes describing some property or some role of element – i.e., class, method or field – aspects become more independent from specific elements of the syntax. Consequently, addition of new elements or changes on class names, field names or method signatures does not require changes on aspects that intercept such elements.

Whereas annotations reduce aspect decoupling from intercepted classes, they suffer from non-locality – i.e., information about which elements implement a specific concern is explicit but scattered across multiple modules –, which is one of the problems that aspects are supposed to solve. This problem is analysed in [25].

```
public class <class_name> {
   //...
   @Oneway
   public void execute(){}
}

// Aspect declaration
public aspect <aspect_name>{
   protected pointcut onewayPoints() : execution(@Oneway * *.*(..));

   //...
}
```

Figure 29 – Annotated methods

Chapter 5. AOP concurrency implementations

This chapter presents AOP implementations of the concurrency patterns and mechanisms presented in Chapter 3, with the purpose of bringing to concurrent applications the usual benefits of AOP, which can avoid some common problems identified in traditional object oriented implementations. Firstly, the motivation for the use of AOP to create an equivalent implementation of OO concurrency patterns will be presented, followed by the implementation design, the description of granularity allowed to model concurrency and the tools used to support these implementations. The AOP implementations are presented along with the pointcut interface required to use the implementation and, in most cases, an equivalent annotation-based solution is also shown. Composition of aspects is also an issue discussed in this chapter.

5.1 Motivation for the use of AOP pattern implementations

According to the ISO/IEC 9126[5] standard, two important parameters for quality of object oriented design maintenance should be considered: (1) *changeability* allows a design to be easily changed; (2) *analysability* enables understandability of design. Nordberg [37] studied the relation between these two parameters.

Concurrency pattern code is intertwined with base functionality: pattern code is scattered across many classes, tangled with code that is not related to the pattern and thus makes it hard to reuse both pattern and application base functionality. Modularisation of pattern code can be partially achieved by capturing the reusable code to specific classes. Though, even when a significant part of the pattern code is modularised, the participant classes should implement the code necessary to perform method invocations on pattern specific classes and lead the responses and exceptions.

AOP promises to improve modularisation of crosscutting code, which can be used to extract pattern-related code to aspects, potentiating modularisation, (un)pluggability of concurrency code and reuse of both concurrency and base functionality code.

---

[5] ISO/IEC 9126 – 1999 "Software Product Evaluation – Quality characteristics and Guidelines for their use"

Hannemann et al [15] performed a comparison between several object oriented pattern implementations – i.e. GoF patterns [12] – and equivalent aspect oriented implementations. In that comparison, he detects benefits in a significant percentage of the AOP patterns analysed. Such benefits are described in terms of the ability of AO pattern implementations to overcome the problems identified in the OO counterparts.

5.2 Pattern implementations design

The structure of the AspectJ implementations presented in this chapter comprises an abstract aspect encapsulating the reusable abstractions and a concrete aspect tailored to a case-specific code base that defines the case-specific join points to be captured in the logic declared by the abstract aspect and sometimes additional behaviour (see section 4.4).

Pointcuts declared in abstract aspects comprise an interface which is exposed to concrete aspects, in order to enable the composition of reusable code with the case-specific code at specific points in the application. In addition, AspectJ reusable implementations use *template methods* to configure parameters in the aspect [12] – e.g., to define the number of blocking threads on barrier mechanism. *Template methods* define the skeleton of an algorithm in terms of abstract operations which subclasses override to provide concrete behaviour. Such methods are specified in abstract aspects and implemented in concrete aspects.

Quantification over annotations is an alternative to quantification over traditional syntax-based join points. The latter requires definition of pointcuts inherited from the abstract aspect, whilst to implement the annotation-based solution, each method, class and attribute used by the pattern should be annotated in order to be captured by the aspect. In that case, definition of pointcuts in concrete aspects is not required, as pointcuts are defined in abstract aspects to capture annotation-based join points in the application. An equivalent implementation with annotations is present for each pattern, except for those where, due to mechanism specificity, an equivalent full implementation with annotations is not possible – e.g., whenever some case specific pattern behaviour should be specified by abstract methods.

In most of the mechanisms, one aspect instance should manage multiple pattern instances. A global hash map maintains, for each intercepted object – or in some cases for each join point localisation –, the respective pattern instance. An alternative implementation would

be to use *perthis* and *pertarget* to create an aspect instance per each pattern instance, which holds the state required to manage a single pattern, avoiding the use of a global map. Although it works in single-processor and single-core machines, several tests were done in the context of this dissertation using multi-core and multi-processor machines – using the configuration presented in section 5.4 – and in all of these tests, *perthis* and *pertarget* constructs revealed an unexpected behaviour, as a non deterministic number of aspect instances are created per each intercepted object. For that reason, such constructs are not used in the implementations presented in this dissertation.

5.3 Concurrency granularity

Concurrency patterns and mechanisms implemented in Java enable a higher level of semantic expressivess when compared with the equivalent AOP constructs. In Java we can apply a mechanism at instruction level – i.e., to a single instruction or a group of instructions. AspectJ equivalent implementations restrict quantification expressiveness to join points – e.g., method call or field access. AspectJ quantification model only can intercept a limited set of events preset by the language, which mitigates the possibility of aspects to have a fine-grain control of applications. For instance, pointcuts cannot capture the execution of a particular instruction or a group of instructions in the program. Such characteristic can be overcome by means of refactoring [11]. One example is the use of *extract method* technique to extract code to a new method, enabling the quantification by the aspect. Another possible solution would be the extension of AspectJ join point types, as done by Harbulot and Gurd, which created a join point type for loops [16].

Figure 30 shows an example where refactoring can be used with the purpose of letting the code amenable to be quantified by pointcuts. The *compute* method has a *for* loop with an instruction that cannot be directly quantified by a pointcut. The loop instruction can be extracted to method *loopBody* (Figure 31) to enable aspects to quantify over that instruction, as it is wrapped inside a method.

Despite the aforementioned limitations, this collection covers the most important constructs used to model concurrency, presented essentially in [29][40]. For several applications the granularity level provided by AspectJ is enough and, in those circumstances, this collection should be able to substitute the equivalent Java ones with the benefits identified before, without resorting to refactoring.

```
public void compute(){
   ...// local variables

   for(int i = 0 ; i < 150 ; i++){
     a[i] = (a[i-1]*a[i]*a[i+1])/3;
   }
}
```

Figure 30 – *compute* method

```
public void compute(){
   ...// local variables

   for(int i = 0 ; i < 150 ; i++){
     loopBody(i);
   }
}

private void loopBody(int i){
   a[i] = (a[i-1]*a[i]*a[i+1])/3;
}
```

Figure 31 – *Extract method* refactoring example

5.4 Tools used in the development of AOP implementations

Several tools were used in the development of this collection. Eclipse 3.1 IDE supports the development of aspect oriented and object oriented components. Sun JDK[6] 1.5.0_3 and AJDT[7] 1.3.0 were used to support development of Java and AspectJ applications, respectively.

Eclipse includes features to assist the refactoring of code. Code refactoring can be used to support the preparation of object oriented code to accept the aspects. As an example, the *extract method* refactoring technique presented in Figure 31 is one of the refactoring techniques supported by Eclipse. Furthermore, aspects can be easily included and excluded to and from the build path, which makes easier the task of plug and unplug the aspects on main functionality. This feature can help the debugging of concurrent applications as the code can be tested with and without concurrency, whenever the concurrency code is unpluggable.

[6] Java Development Kit – http://java.sun.com/javase/

[7] AspectJ Development Tools – http://www.eclipse.org/ajdt/

5.5 Implementations

This section presents the AOP collection of concurrency patterns and mechanisms[8]. Such collection embodies the equivalent AOP implementations of the traditional object oriented concurrency implementations presented in Chapter 3, with the purpose of assessing the feasibility of AOP to develop reusable implementations of those concurrency patterns and mechanisms.

For several patterns presented in Chapter 3, the Java constructs were extended to include new aspect-oriented features. For instance, one-way was extended to perform *sleep*, *join* and interruption of threads, which are features used to control threads. The equivalent Java constructs cannot be used, as the classes do not hold the reference to the object that reifies the thread – i.e., references to thread objects are contained within the aspect.

This section exposes each AOP implementation in the following manner: starts by characterising and contextualising the pattern, describes their usage, presents corresponding implementation and when it is feasible shows the equivalent implementation with annotations.

5.5.1 One-way calls

One one-way reference implementation in Java (see section 3.1.2) requires the creation of a *Runnable* class with the code that will be executed by the new thread. To perform an asynchronous call, we need to add several extra lines of code associated with thread creation and spawning that cannot be reused in other circumstances.

The *OnewayProtocol* abstract aspect defines the one-way reusable logic that can be used through concretisation of pointcut *onewayMethodExecution* and optional definition of *join*, *interrupt* and *interruptAll* pointcuts. Interface specification of *OnewayProtocol* is shown in Figure 32 in the context of a concrete aspect skeleton, followed by the respective implementation that is presented in Figure 33. Pointcut *onewayMethodExecution* specifies the events that run concurrently. The aspect creates a new thread per each captured join point, which will run the method code. In the *around* advice defined in Figure 33,

[8] All implementations, code samples and benchmarks are available from http://gec.di.uminho.pt/ppc-vm/conccollection/

proceed()[9] runs inside a *Runnable* anonymous object. Thus, that object contains the method invocation logic executed by the new spawned thread. It is possible to spawn threads into a specific thread group by means of *getThreadGroupName* method.

Several times it is necessary to block the main thread until all threads that were started by it terminate. The concrete aspect may optionally define a blocking point where the calling thread waits for the termination of all spawned threads. Pointcut *join* specifies the join points where the main thread should block waiting for the spawned threads to terminate, by calling *Thread.join* per each thread started before.

All started threads are registered along with the thread that started them. Method *registerThread* inserts thread references in the hash map *threads*, enabling the aspect to relate the current thread to the threads spawned by it. This is required to enable the feature *join*. Pointcuts *interrupt* and *interruptAll* also use that registration data structure to interrupt threads spawned by the aspect: *interrupt* specifies the join points where all threads created by the current thread should be interrupted and *interruptAll* does the same for all threads created by the aspect.

```
public aspect <aspect_name> extends OnewayProtocol {

  protected pointcut onewayMethodExecution(Object servant) :
    <pointcut definition>;

  protected pointcut join() : <pointcut definition>;

  protected pointcut interrupt() : <pointcut definition>;

  protected pointcut interruptAll() : <pointcut definition>;

  protected String getThreadGroupName() {
    return /* thread group name */;
  }
}
```

Figure 32 – Concrete aspect skeleton of the *OnewayProtocol*

[9] The *proceed* statement executes the method defined before be captured by the aspect.

```
public abstract aspect OnewayProtocol {
  // interface with the subaspects
  private WeakHashMap <Thread,ArrayList<Thread>> threads = new
    WeakHashMap<Thread,ArrayList<Thread>>();

  // ... state variables

  void around(final Object servant): onewayMethodExecution(servant) ||
        onewayAnnotation(servant){

    Thread t = new Thread(new ThreadGroup(getThreadGroupName()),
      new Runnable(){
        public void run(){
          // ...
          proceed(servant);  // the method is executed inside a
                             // Runnable object
          // ... exception handling logic
      }
    });
    registerThread(t);
    t.start();
  }

  after() : join(){
    joinThreads();
  }

  ...// advices to interrupt and interruptAll pointcuts
  ...// advices to pointcuts which capture annotations

  private synchronized void joinThreads(){
    ArrayList<Thread> arr;
    arr = threads.get(Thread.currentThread());

    for(int i = 0 ; i < arr.size() ; i++){
      // ...
      arr.get(i).join();
      // ... exception handling logic
    }
    threads.remove(Thread.currentThread());
  }

  protected String getThreadGroupName(){
    return Thread.currentThread().getThreadGroup().getName();
  }

  private synchronized void registerThread(Thread t){
    ArrayList<Thread> arr;
    if((arr = threads.get(Thread.currentThread())) == null){
      arr = new ArrayList<Thread>();
      arr.add(t);
      threads.put(Thread.currentThread(),arr);
    }else{
      arr.add(t);
    }
  }
  ...// other auxiliary methods
}
```

Figure 33 - One-way reusable implementation

Annotations

The annotations mechanism of Java 1.5 Tiger [25] provides an alternative to definition of pointcuts to quantify asynchronous methods. All methods annotated with annotation *@Oneway* are intercepted by the aspect *OnewayProtocol*. Thus, definition of pointcut *onewayMethodExecution* in concrete aspects is no longer needed. Likewise, definitions of *join*, *interrupt* and *interruptAll* pointcuts can similarly be replaced by the use of annotations. In addition, threads can be forced to sleep for a specified amount of time through the use of annotations. Annotations *@SleepBefore* and *@SleepAfter* annotate methods in which the current thread sleeps for a specified amount of time before or after their execution – sleep time is specified by an annotation parameter. Figure 34 shows pointcuts defined in *OnewayProtocol* to quantify over annotations.

```
// annotations captured from classes

protected pointcut onewayAnnotation(Oneway parameters):
  execution(@Oneway * *.*(..)) &&  @annotation(parameters);

protected pointcut sleepBeforeAnnotation(SleepBeforeExecution time):
  execution(@SleepBeforeExecution * *.*(..)) && @annotation(time);

protected pointcut sleepAfterAnnotation(SleepAfterExecution time):
  execution(@SleepAfterExecution * *.*(..)) && @annotation(time);

protected pointcut joinBeforeAnnotation() :
  execution(@JoinBeforeExecution * *.*(..));

protected pointcut joinAfterAnnotation() :
  execution(@JoinAfterExecution * *.*(..));

protected pointcut interruptAnnotation() :
  execution(@InterruptThreads * *.*(..));

protected pointcut interruptAllAnnotation() :
  execution(@InterruptAllThreads * *.*(..));
```

Figure 34 – One-way pointcuts for annotations

Annotations can be useful and sometimes essential in situations where concurrency is intrinsic to the algorithm. Modularisation of concurrency code when it cannot be unplugged from the core functionality – i.e., when the algorithm modelled by that code is concurrent by nature –, compromises the understandability of domain-specific code, as it cannot be analysed isolated from the concurrency code. As an example, in one-way, some method invocations only make sense for anyone that reads the code if he knows that method will be executed in parallel. This issue is further discussed in section 6.4.

Annotations listed in Table 5 are used by the aspect to control creation, management and interruption of threads. Attributes formatted as italic are optional. In *@Oneway*, attribute *saveState* is interpreted by the aspect to save the spawned thread reference to be used latter to perform thread joining or interruption. By default, *saveState* is true and *threadGroup* assumes the same *threadgroup* of the thread that creates it. Thread groups have a particular significance when barrier is used, as the selection of threads is based on thread groups.

Table 5 - Annotations used by *OnewayProtocol*

Annotations	
`@Oneway(`*`threadGroup`*`, `*`saveState`*`)`	Spawns each annotated method into a new thread associated to a thread group specified by *threadGroup* parameter. If *saveState* is defined as *true* the aspect saves the reference to the object which represents such thread.
`@JoinBeforeExecution`	Wait for threads spawned by current thread before the method execution.
`@JoinAfterExecution`	Wait for threads spawned by current thread after the method execution.
`@SleepBeforeExecution(time)`	Sleep for a specified amount of time before the execution of method.
`@SleepAfterExecution(time)`	Sleep for a specified amount of time after the execution of method.
`@InterruptThreads`	Interrupt threads spawned before by the current thread.
`@InterruptAllThreads`	Interrupt all threads spawned before by the aspect instance.

5.5.2 Future

Future extends one-way calls by adding the logic that allows the method invoked asynchronously to return a value to the client. Upon one-way implementation, the future adds synchronisation and communication logic between the main thread and the spawned thread. From the point where the method is invoked asynchronously to the point where the returned value is used, the real value is replaced by a fake object. That object should be instantiated with the same type of the real one. Two different AOP implementations of

futures were created: the first uses introspection[10] to automatically identify the returned object type in order to instantiate an object of same type; in the second the programmer creates the fake object explicitly.

Two different abstract aspects were created to consider each of the aforementioned future implementations: in the *FutureReflectProtocol* (Figure 35) the fake object – whose purpose is to replace the object that holds the result of a computation – is created by reflection, through identification of its *Class* type which is used to instantiate the fake object. The *FutureProtocol* aspect (Figure 36) declares the *getFakeObject* abstract method that is defined in concrete aspects, which intends to create and return the fake object explicitly by the programmer.

Pointcut *futureMethodExecution* defines the events related to the invocation of asynchronous methods and pointcut *useOfFuture* defines the join points where the values returned by such methods are consumed. Each thread blocks on the join points captured by *useOfFuture*, in case the methods associated with join points captured by *futureMethodExecution* have not returned.

```
public aspect <aspect_name> extends FutureReflectProtocol{
  protected pointcut futureMethodExecution(Object servant) :
  <pointcut definition>;
  protected pointcut useOfFuture(Object servant) :
    <pointcut definition>;
}
```

Figure 35 – Concrete aspect skeleton of the *FutureReflectProtocol*

```
public aspect <aspect_name> extends FutureProtocol{
  protected pointcut futureMethodExecution(Object servant) :
  <pointcut definition>;
  protected pointcut useOfFuture(Object servant) :
    <pointcut definition>;
  protected Object getFakeObject();
}
```

Figure 36 – Concrete aspect skeleton of the *FutureProtocol*

FutureReflectProtocol uses reflection to identify the object type returned by the method invoked asynchronously. Such type is used to create the fake object. Though, each fake

[10] Introspection represents the capability of object-oriented programming languages to access to objects meta-level information.

object should be instantiated by means of a constructor without parameters. Class instantiation using constructors with parameters is complex, as it requires the identification of parameter types by reflection again and the instantiation of those corresponding classes, which may require the use of a constructor with parameters again, potentially generating a cyclic identification and instantiation of classes. Thus, fake objects which do not implement the constructor without parameters must use *FutureProtocol* to provide an explicit means to instantiate fake objects.

Figure 37 shows the relevant parts of *FutureReflectProtocol* aspect. The first *around* advice intercepts invocations of methods and the second *around* advice intercepts accesses to the returned object. Both advices start by creating a fake object and a *Future* type object where the returned value will be stored. Fake objects are instantiated by reflection and registered in a hash map that associates fake objects to futures in the method *mapFake2Futures*. After method execution, the returned object is stored in the associated *Future* object. In the *around* advice acting on pointcut *useOfFuture*, the fake value is replaced by the genuine one. If it is not yet available, the client thread blocks until the new spawned thread stores the value in the future object. Method *unRegisterGet* removes the fake from the hash map and returns the real object.

Objects used to represent numbers – i.e. objects which type is descendant of class Number – are the exception of the rule which disallows objects that do not implement the constructor without parameters to be instantiated. Fake objects descendent of *Number* are instantiated with a specific value. Thus, creation of fake objects is implemented by methods *createFakeNumberParameters* to create the parameters of objects descendent of *Number* class*, createFakeNumberObject* to create objects descendent of *Number* class and *createFakeObject* to create other type instances.

The aspect *FutureProtocol* (Figure 36) has a different implementation of the same functionality of *FutureReflectProtocol* that does not resort to reflection. Method *getFakeObject* is defined explicitly on concrete aspects with the code associated to the creation of fake objects. Consequently, a new concrete aspect should be created per each different object type returned by the future. This solution replaces the solution based on reflection when the returned object does not implement a constructor without parameters – for the reason described before. Furthermore, as long as this solution avoids reflection, the performance costs of reflection can be avoided when this implementation is used, at cost of less reusability.

```
public abstract aspect FutureReflectProtocol {
   ...// state variables and pointcuts described before

   Object around(final Object servant) :
      futureMethodExecution (servant) && (! call (Number+ *.*(..))) {
      Class returnType = ((MethodSignature)
         thisJoinPoint.getStaticPart().getSignature()).getReturnType();
      Object fk = createFakeObject(returnType, fk);
      final Future fut = new Future();

      Thread t = new Thread(new Runnable(){
         public void run(){
            //...
               fut.setValue(proceed(servant));
            ...// exception handling
         }
      });
      mapFakeObjects2Futures(fk,fut); t.start();
      return fk;
   }

   Object around(final Object servant) :
      futureMethodExecution(servant) && (call (Number+ *.*(..))){
      ...// Similar to previous advice but specific to Number subclasses
   }

   Object around(Object servant) : useOfFuture(servant) {
      Object s = unRegisterGet(servant);
      if(s != null) servant = s;
      return proceed(servant);
   }

   private Object createFakeNumberParameters(/* parameters */) {
      //... create and returns the fake parameter descendent of Number
   }

   private Object createFakeNumberObject(/* parameters */) {
      //...
      fake = returnType.getConstructor(String.class).newInstance(
         parameter.toString());
      //... exception handling
      return fake;
   }

   protected synchronized void mapFakeObjects2Futures(/* parameters */){
      //... map object to future
   }

   private Object createFakeObject(Class returnType, Object fk) {
      //...
      fk = returnType.newInstance();
      //... exception handling
      return fk;
   }

   protected synchronized Object unRegisterGet(Object o){
      ...// removes the entry on the map and returns the Future value
   }
}
```

Figure 37 – Futures implementation with reflection

Annotations

FutureProtocol allows the quantification by means of annotations. Each method invoked asynchronously must be annotated with annotation *@Future* (Table 6). The method where the value is consumed – i.e., where the fake object is substituted by the real one – must be annotated with *@FutureClient*. In addition, the pointcut *targetTypeCall* and method *getFakeObject* must be defined with the returned object type and the code for the object instantiation, respectively.

Implementation of futures with annotations can be inefficient, as long as every method invocation on the objects, whose types are specified in *targetTypeCall*, are intercepted to verify if the object reference was registered by the aspect in the hash map. That situation is caused by the inability to annotate the instruction where the value returned by the future is needed. To reduce the overhead generated by this implementation, the number of objects with the same type of the returned object, used inside the method annotated with *@FutureClient* should be limited.

Table 6 - Annotations used by *FutureProtocol*

Annotations	
`@Future`	`Methods annotated with @Future are invoked asynchronously.`
`@FutureClient`	`Used to annotate methods that use the future returned by methods annotated with @Future.`

5.5.3 Barrier

The *BarrierProtocol* abstract aspect implements the AOP version of Barrier, which specifies an interface (see Figure 38) formed by pointcuts – which captures the join points where threads must synchronise – and methods – to specify the threshold number of blocking threads and the target thread group. Two optional pointcuts may be defined: (1) *barrierBeforeExecution* defines events where threads must block, immediately before the execution of methods associated with those events; (2) *barrierAfterExecution* has a similar purpose, but in this case, the threads block after method execution. Barriers apply to a specific set of threads. Selection of such threads can only be carried out at thread group level, as the selection of individual threads is not feasible – i.e., it is not possible to quantify specific thread instances in AspectJ. Accordingly, the method

getThreadGroupName should be redefined in order to specify the thread group name of the target barrier threads.

In order to enable the creation of barriers specific to a thread group it is essential to specify the correct thread group at the time when the threads are created. The one-way mechanism presented in section 5.5.1 allows the specification of thread groups either by method definition or by an annotation parameter.

The implementation of Barrier presented in Figure 39 is based on cyclic barrier [29]. It intercepts the join points defined in concrete aspect pointcuts and blocks all the threads that reach one of the specified join points, until the threshold number of blocking threads is reached.

Method *getNumberThreads* should be implemented on concrete aspects to set the number of blocking threads. Thus, a new concrete aspect is created for each threshold value. In the *applyBarrier* method, each barrier related join point holds a *State* object which maintains the number of threads that have reached the join point. In addition, the *State* object lock is used by *wait* – invoked in *barrier* method – to block threads at the barrier point.

```
public aspect aspectname extends BarrierProtocol{

   // barrier number of threads can be specified by:
   protected int getNumberThreads() {
      return <number of threads>;
   }

   // target Threadgroup can be specified by:
   protected String getThreadGroupName() {
      return <threadgroup name>;
   }

   // and define one of the following:
   protected pointcut barrierAfterExecution() : <pointcut definition>;

   protected pointcut barrierBeforeExecution() : <pointcut definition>;
}
```

Figure 38 - Concrete aspect skeleton of the *BarrierProtocol*

```
public abstract aspect BarrierProtocol {
  //... State variables

  //... pointcuts used to quantify annotations
  //... pointcuts declared to be specified on concrete aspects

  protected  void barrier(State s){  // Barrier logic
    int index = --s.count;
    if(index > 0){
      //...
      do{ s.wait(); } while (s.resets == r);
      ...// exception handling
    } else {
      s.count = s.nThreads;
      ++s.resets;
      s.notifyAll();
    }
  }

  after() : barrierAfterExecution(){
    applyBarrier(thisJoinPointStaticPart.toShortString(),
      getThreadGroupName(), getNumberThreads());
  }

  ...// similar advices (e.g. associated to annotations)

  //  can be redefined on subaspects
  protected int getNumberThreads(){  return 0; }

  //  can be redefined on subaspects
  protected String getThreadGroupName(){
    return Thread.currentThread().getThreadGroup().getName();
  }

  protected void applyBarrier(/* parameters */){
    //...
    State s =mapJoinPoint2State(textJoinPoint, threadGroup, nThreads);
    synchronized(s){  barrier(s); }
  }

  private synchronized State mapJoinPoint2State(/* parameters */){
    ...// return barrier associated to a joinpoint and register
    ...// on Hashmap if such association does not exist
  }

  ...// State class definition
}
```

Figure 39 - Barrier reusable implementation

Annotations

Methods where threads synchronise can be annotated either with *@BarrierBeforeExecution* or *@BarrierAfterExecution*, depending if it blocks before or after the method execution (Table 7). Element-pair values in annotations allow the definition of parameter values that can be used to specify the number of blocking threads

and the target thread group name. Element-pair values, namely *nThreads* and *threadGroup*, gives the number of threads and the target thread group where the barrier should apply, respectively. For instance, for five threads blocking after method execution associated to thread group *calculus*, the annotation is

@BarrierAfterExecution (nThreads = 5, threadGroup = "calculus")

Barrier usage without annotations requires the definition of method *getNumberThreads* to specify the number of threads. That solution requires a new concrete aspect per each value representing the number of threads. Annotations avoid this problem, given that number of threads is specified through an annotation argument and consequently that information can be obtained from the context of each captured join point, which allows the use of a single aspect instance to create and manage all barrier instances.

Table 7 - Annotations supported by *BarrierProtocol*

Annotations	
`@BarrierBeforeExecution( nThreads, threadGroup )`	Block threads belonging to thread group *threadGroup* before the method execution, until the number of blocking threads reaches nThreads.
`@BarrierAfterExecution( nThreads, threadGroup )`	Block threads belonging to thread group *threadGroup* after the method execution, until the number of blocking threads reaches nThreads.

5.5.4 Active Object pattern

Active objects decouple method invocation from its execution. Each active object runs into its own thread of control. The AOP version of active objects places the pattern related code within an aspect and allows participant classes to be oblivious of their roles in the pattern.

To specify which objects are active objects, the *ActiveObject* interface should be added to the list of interfaces implemented by the class (Figure 40). Pattern logic composes with the target application through the introduction of a marker interface [13] into active object classes, using the intertype declaration mechanism of AspectJ.

```
public aspect <aspect name> extends ActiveObjectProtocol {
   declare parents : <case-specific class> implements ActiveObject;
}
```

Figure 40 - Concrete aspect skeleton of the *ActiveObjectProtocol*

Figure 41 presents an overview of the AspectJ active object architecture and Figure 42 the corresponding implementation. Whenever an active object instance is created, the *before* advice (defined in Figure 42) intercepts the pointcut *create* and associates the object to a scheduler object that assumes the communication channel role between client threads and active object threads. Each method invocation performed on the active object is intercepted by *callMeth*, wrapped within a *concurlib.activeobject.Callable* object and afterwards is stored in the activation list. The activation list is the active object scheduler queue – implemented by the *ActivationList* class – where the method invocations are stored to be executed by the active object dedicated thread. The active object thread is continuously picking requests from the queue and carrying out their execution.

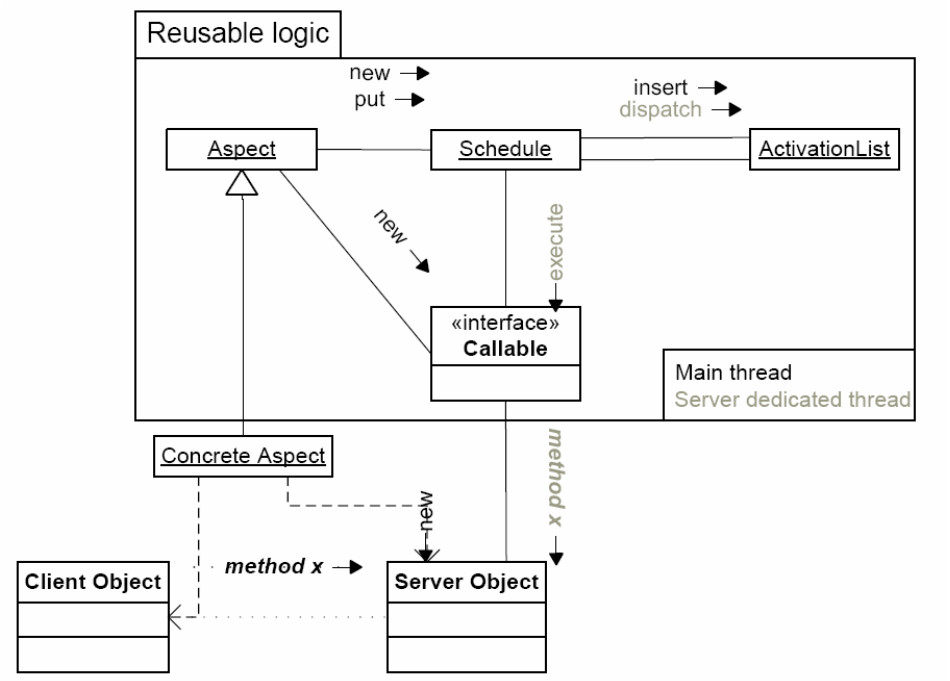

Figure 41 – AOP Active Object architecture

Both invoker and active objects are oblivious of their role in the pattern. Methods are invoked by the client object on active objects directly, without pass invocations through a proxy object. *ActiveObjectProtocol* aspect intercepts method invocations and stores them

in the activation list. The active object dedicated thread – which is created and managed by the aspect – executes sequentially the method calls queued in the activation list. Thus, client objects are coded to perform regular calls on server objects and the server objects to receive regular invocations, as the pattern code is modularised in the aspect. The active object aspect can be plugged and unplugged without having to change the code of any participant class.

```
public abstract aspect ActiveObjectProtocol {
  protected interface ActiveObject{};  // marker interface

  ...// state variables
  ...// pointcuts exclusive for annotations

  protected pointcut create(ActiveObject s) :
    execution(ActiveObject+.new(..)) && this(s);
  protected pointcut callMeth(ActiveObject s) : call(public *
    ActiveObject+.*(..)) && target(s);

  before(ObjectActive s) : create(s) {
    MQScheduler mqs = new MQScheduler(50);
    synchronized(this){  hash.put(s, mqs); }
    (new Thread(mqs)).start();
  }

  Object around(final ActiveObject s): callMeth(s){
    Message ms = new Message();
    MethodRequest mr = new MethodRequest(new Callable(){
      private Object msg = null;
      public void call(){ msg = proceed(s); }
      public Object getValue(){ return msg; }
    },ms);

    sendToQueue(mr,s);

    return waitForValue(mr, s);
  }
  ...// other advices / auxiliar methods
}
```

Figure 42 - Method call interception of Active Object

Annotations

The *@ActiveObject* annotation replaces *declare parents* clauses on concrete aspects to introduce the *ActiveObject* interface into the class. The *ActiveObjectProtocol* aspect (Figure 43) make the classes implement that interface, which simplifies the pattern plugging process, since the programmer does not have to use the intertype declaration mechanism to add the interface to the class.

```
public abstract aspect ActiveObjectProtocol {
   //...
   declare parents :
      @concurlib.annotations.ActiveObject * implements ActiveObject;
   //...
}
```

Figure 43 – Static introduction of *ActiveObject* interface

5.5.5 *Synchronized* mechanism

Synchronized is implemented in the Java language using the *synchronized* modifier on method declarations or using the construct

synchronized(<object>){ ...// code }

Each object holds its own lock, which is used to ensure the exclusive access to methods or regions declared as *synchronized*. Although *synchronized* methods share the same object lock, the *synchronized* block construct enables the use of an arbitrarily chosen lock.

AspectJ implementation of *synchronized* allows synchronisation using either the intercepted object lock or an independent object lock. Synchronisation granularity is the method or field – field level granularity enables the synchronised access to object fields. In order to synchronise blocks of code, the programmer would resort to refactoring, by extracting synchronised code to a new method, in order to be quantified by either pointcut declared in *SynchronizeProtocol* (Figure 44). Pointcuts *synchronizedUsingCapturedLock* and *synchronizedUsingSharedLock* would be defined to add synchronised behaviour to any method or field access, using the captured instance lock captured from the context or the aspect instance lock. In the latter, a unique lock can be used to synchronise access to methods of multiple objects. As a result, only one lock can be used per aspect instance.

In Figure 45, both around advices implemented in *SynchronizeProtocol* wraps the intercepted method execution or variable access within a Java *synchronized* construct.

```
public aspect <aspect_name> extends SynchronizeProtocol {
   protected pointcut synchronizedUsingCapturedLock(
        Object targetObject):
      <pointcut definition>;
   protected pointcut synchronizedUsingSharedLock():
      <pointcut definition>;
}
```

Figure 44 – Concrete aspect skeleton of the *SynchronizeProtocol*

```
public abstract aspect SynchronizeProtocol {

   ...// aspect variables...

   ...// pointcuts...

   Object around(Object targetObject) :
        synchronisedUsingCapturedLock(targetObject){

      synchronized(targetObject) {
        return proceed(targetObject);
      }
   }

   Object around() : synchronisedUsingSharedLock() {
      synchronized(this){
        return proceed();
      }
   }

   ...//definition of methods and other advices
}
```

Figure 45 - AO implementation of *synchronized* mechanism

Annotations

Synchronised methods can, optionally, be annotated with *@Synchronized* in substitution of pointcut definition. Abstract aspects capture elements annotated with *@Synchronized* and retrieve the object reference to use the object lock in the synchronisation process. Therefore, each intercepted method or variable access uses the respective object lock to synchronise access to it. Optionally, the synchronised mechanism can use a specific lock to synchronise access to methods defined in different class instances.

In order to identify a particular lock in the application, the annotation attribute *id* should be settled. Each aspect instance implements a map from *ids* to object locks. Thus, an *id* is associated uniquely to a specific lock, which can be used in the following manner:

@Synchronized(id = "lockName")

Methods *moveX* defined on class *Point* (Figure 46) and *translateAndZoom* defined on class *Circle* are annotated with *@Synchronized* using the same lock *id*. If *id* was not provided, each object lock would be used to control the access to annotated methods.

```
class Point{
   ...// variables

   @Synchronized(id="move")
   public void moveX(int delta) {  x+=delta; }
}

class Circle{
   int radius;
   Point p;

   @Synchronized(id="move")
   public void translateAndZoom(int deltaX, int deltaY, int zoom){
      p.moveX(deltaX);
      p.moveY(deltaY);
      p *= zoom;
   }
}
```

Figure 46 - Shared locks with annotations

5.5.6 Waiting Guards

Waiting guards is used in case the execution of methods requires precondition validation. If the precondition validation fails, the thread blocks until a change of state triggers a precondition revaluation.

As in previous AOP implementations presented in this dissertation, Waiting guards crosscutting code is modularised into an abstract aspect. Aspect *WaitingGuardsProtocol* implements the reusable part of Waiting Guards. *WaitingGuardsProtocol* implements an interface comprised by: (1) pointcuts that quantify over calls to methods where the mechanism ought to apply and methods that may change the precondition validity and, consequently, force each blocked thread to reevaluate its precondition; (2) methods that define the precondition and establish the timeout value that can, optionally, be defined in order to set the waiting time before the precondition reevaluation. Figure 47 presents the interface of Waiting Guards.

Pointcut *blockingOperation* specifies join points where the precondition is checked, whereas the pointcut *deblockingOperation* specifies methods that may change precondition validity and then forcing a reevaluation. Method *precondition* returns the logical value representing the precondition validity. This method receives two parameters that can be used to retrieve information from the join points context: *targetObj* that captures the intercepted object reference and *args* that captures method arguments. Such information can be used to define the condition. Optionally, time over value can be redefined by

overriding the method *getWaitingTime*. By default it returns zero, which is ignored by the *wait* construct.

```
public aspect <aspect_name> extends WaitingGuardsProtocol {
  protected pointcut deblockingOperation(Object targetObject) :
    <pointcut definition>;
  protected pointcut blockingOperation(Object targetObject):
    <pointcut definition>;

  protected boolean preCondition(Object targetObj, Object[] args) {
    return <precondition validity>;
  }

  //and optionally override method getWaitingTime
  protected long getWaitingTime(){
    return <time in milliseconds>;
  }
}
```

Figure 47 - Concrete aspect skeleton of the *WaitingGuardsProtocol*

Each thread reaching a join point captured by *blockingOperations* forces precondition validation and when it is not valid, the thread passes to *wait* state (Figure 48). By contrast, an advice acts on join points captured by *deblockingOperations* and notifies all blocked threads, forcing them to reevaluate their conditions.

```
public abstract aspect WaitingGuardsProtocol {
  //...
  protected long getWaitingTime(){ return 0; }

  before(Object ob, Object[] parameters) :
    blockingOperation(ob) && args(parameters){

    //...
    synchronized(ob) {
      while(! preCondition(ob, parameters)) {
        ob.wait(getWaitingTime());
      }
    }
    ...//exception handling logic
  }

  after(Object ob) : deblockingOperation(ob) {
    synchronized(ob) {
        ob.notifyAll();
    }
  }
}
```

Figure 48 - AO reusable implementation of Waiting Guards

As the precondition definition is essential to instantiate waiting guards, this pattern cannot be fully implemented with annotations. This problem was addressed before with futures

(section 5.5.2). Basically, the precondition code should be defined in the aspect and only delegates to annotations the specification of methods where the precondition should apply. Configuration of waiting guards using annotations requires the use of multiple semantics, divided between the classes and the aspect and can contribute to make the use of the mechanism harder. For that reason, the implementation with annotations is not addressed here.

5.5.7 Readers-Writer Lock

Readers-Writer lock differentiates accesses that change object state from the ones that just read it, with the purpose of increasing object level concurrent access. Accordingly, methods that do not change object state – i.e., readers – can be executed by multiple threads, but at the most one thread – i.e., a writer thread – is allowed to be executing exclusively methods that change the object state.

In order to specify which methods change object state and which ones read it, four pointcuts may be specified in concrete aspects (Figure 49): *readMethodObjectLock* and *readMethodSharedLock* capture executions of reader methods; *writeMethodObjectLock* and *writeMethodSharedLock* capture the execution of writer methods.

**SharedLock* pointcuts quantify over methods synchronised by a single shared lock. As in the *Synchronized* mechanism presented in section 5.5.5, a single lock would be used to synchronise access to methods implemented in different classes. **ObjectLock* pointcuts perform synchronisation using one lock per target. A map of objects to locks maintains the lock reference of each intercepted object. Shared locks do not require a structure to save lock references, as only one lock is maintained per each concrete aspect.

```
public aspect <aspect_name> extends RWLockProtocol {
  protected pointcut readMethodObjectLock(Object targetObject) :
    <pointcut_definition>;
  protected pointcut writeMethodObjectLock(Object targetObject) :
    <pointcut_definition>;
  protected pointcut readMethodSharedLock() :
    <pointcut_definition>;
  protected pointcut writeMethodSharedLock() :
    <pointcut_definition>;
}
```

Figure 49 - Concrete aspect skeleton of the *RWLockProtocol*

RW Lock implementation is shown in Figure 50. Appropriate lock types – i.e., reader locks or writer locks – shall be acquired before the execution of join points and released after execution. Lock management functionality is defined in class *RWLock* (Figure 14).

```
public abstract aspect RWLockProtocol {
   ...// state variables
   ...// other variables
   ...// pointcuts referred before

   before(Object targetObject) : readMethodObjectLock(targetObject) {
      RWLock lock = mapObjectCaptured2Lock(targetObject);
      readLockAcquire(lock);  // acquire lock for reading
   }

   after(Object targetObject) : readMethodObjectLock(targetObject) {
      RWLock lock = mapObjectCaptured2Lock(targetObject);
      lock.readLock().release();
   }

   before(Object targetObject) : writeMethodObjectLock(targetObject) {
      RWLock lock = mapObjectCaptured2Lock(targetObject);
      writeLockAcquire(lock);   // acquire lock for writing
   }

   after(Object targetObject) : writeMethodObjectLock(targetObject) {
      RWLock lock = mapObjectCaptured2Lock(targetObject);
      lock.writeLock().release();
   }

   ...//other methods and advices
}
```

Figure 50 - AO reusable implementation of RW lock

Annotations

@Reader and *@Writer* annotations (Table 8) can be used to annotate reader and writer methods, respectively. Both annotations receive an optional *id* value, uniquely representing the lock identity. A single lock is created per each *id*, enabling the association of methods from several classes to a specific lock, referenced by *id* in a similar way of synchronised mechanism. If *id* is not specified, the aspect uses a different lock per each object intercepted by the aspect. This option fits well when every intercepted object should hold its own lock.

Table 8 – RW Lock Annotations

Annotations	
`@Reader(id)`	Specifies a *Reader* method. Parameter *id* specifies the lock id. If it is not defined, a different lock is used per each object intercepted.
`@Writer(id)`	Specifies a *Writer* method. Parameter *id* specifies the lock id. If it is not defined, a different lock is used per each object intercepted.

5.5.8 Scheduler

Scheduler enables exclusion according to a specified scheduling order. Each thread attempting to execute a scheduled method blocks and is inserted into a queue until the mechanism wakes it. The AOP reusable implementation of scheduler is shown in Figure 51. By default, scheduling order is FIFO. However, a different order can be specified or a different scheduling policy can be implemented – e.g., by using information captured from the context. The method *selectRunningThread* should be defined in order to specify the scheduling order, which returns the position of the next running thread. It receives by parameter the thread queue and the intercepted object reference. Such parameters can be considered in the rule specification to determine the next running thread.

Each aspect instance manages many schedulers at different join point locations. A hash map of join point to array of threads is held by the aspect to maintain the list of waiting threads at each join point location.

Pointcut *scheduledMethodExecution* specifies the events intercepted by the scheduler. The *around* advice presented in Figure 52 intercepts the *scheduledMethodExecution* pointcut and, per each thread reaching one of the join points, the method *enter* stores the thread object reference in the array associated with that join point localisation and blocks the thread until it will be selected to be the next running thread. Each time the current thread finishes its execution, the method *done* selects the next running thread and removes the reference from the queue.

```
public aspect <aspect_name> extends SchedulerProtocol{
  protected pointcut scheduledMethodExecution(Object targetObject) :
    <pointcut_definition>;

  protected int selectRunningThread (ArrayList th, Object
       targetObject){
    return <next_thread_position>;
  }
}
```

Figure 51 - Concrete aspect skeleton of the *SchedulerProtocol*

```
public abstract aspect SchedulerProtocol {
  ...// variables and pointcuts referred before
  Object around(Object targetObject) :
    scheduledMethodExecution(targetObject) {
    //...
    try{
      enter(thisJoinPoint);
      return (proceed(targetObject));
    } catch(InterruptedException e) {    //exception handler logic
    } finally{ done(thisJoinPoint, targetObject); }
  }

  public void enter(String textJoinPoint) /*...*/ {
    Thread thisThread = Thread.currentThread();
    mapJoinPoint2ThreadSet(/*...*/);
    synchronized(thisThread){
      while(thisThread != runningThread) thisThread.wait();
    }
    removeRequest(textJoinPoint);
  }

  public synchronized void done(/* parameters */) {
    //...
    ArrayList<Thread> arr = waitingThreads.get(textJoinPoint);
    if(arr.size()==0) runningThread = null;
    else { //else determines next request
      runningThread= arr.get(selectRunningThread(arr, targetObject));
      synchronized(runningThread) { runningThread.notifyAll(); }
    }
  }

  // Returns the position of next running thread
  protected int selectRunningThread(parameters)
    //thread execution order logic
  }
  ...//definition of other methods and advices
}
```

Figure 52 - Scheduler implementation

Annotations

Scheduler can be engaged into applications by using annotations, although only FIFO[11] and LIFO[12] scheduling orders are allowed. Table 9 summarizes the use of *@Scheduled*. The annotation would be used with one of the enumerated parameter values, in the form: *@Scheduled(Scheduled.Order.FIFO)* for FIFO; *@Scheduled(Scheduled.Order.LIFO)* for LIFO. Other scheduler orders require redefinition of the *selectRunningThread* method, which should be implemented as described before.

Table 9 – Scheduler Annotations

Annotations	
`@Scheduled (Scheduled.Order.FIFO)`	`Used to annotate the elements with Scheduler access. The Scheduler order is FIFO.`
`@Scheduled (Scheduled.Order.LIFO)`	`Used to annotate the elements with Scheduler access. The Scheduler order is LIFO.`

5.6 Aspect composability

Composition of patterns and mechanisms between each other is possible whenever such compositions are valid in equivalent Java implementations.

Each aspect capturing a pattern implementation should be applied to participant classes according to a specific order. Consequently, the programmer should be aware of the impact of each aspect in the base code. For instance a barrier should only be applied when threads were created before. Therefore, one-way or futures should precede the barrier. In AspectJ, an aspect would be defined to specify the aspects precedence, e.g.,

declare precedence : Oneway, Barrier;

A profound study about composition of mechanisms by combining them and analysing the impact of one on the others requires the implementation and analysis of a large set of case studies with different natures. That analysis is out of the scope of this work and can be tackled in future work.

[11] First In First Out

[12] Last In First Out

5.7 Summary

Several aspect-oriented implementations were presented in this chapter. Such constructs present a better modularity and reuse when compared with object oriented equivalent implementations. Notwithstanding, the use of AOP constructs is restricted to the language join point model, as long as the constructs only should be applied at limited points in the program – i.e., at join points implemented by the language. As described before, such problem can be ameliorated by means of refactoring.

Some of concurrency mechanisms presented here were previously implemented with aspects. JBoss AOP [22] and AspectJ 5 Developer's Notebook [46] use the *@Oneway* annotation in void methods to specify execution in separate threads. However, additional thread functionality such as *join*, *interrupt*, *sleep* and definition of some parameters – e.g., thread group – was not implemented. An aspect-based non-reusable implementation of futures is presented in [7] that modularises thread spawning code within an aspect. However, management of futures must be written in the classes that use them. AO non-reusable implementations of RW lock are presented in [7][26]. These implementations only support a lock per object and do not support annotations. The collection presented in this dissertation included a reusable implementation of most well known patterns and mechanisms for concurrency. Both annotation and traditional pointcut model is supported whenever possible and is provided several parameters to configure each mechanism – e.g., specify the thread group in barrier and one-way.

One factor that would limit scalability of presented solutions is the number of pattern instances managed by the aspect. This happens because abstract aspects implement a map structure to keep track of pattern instances associated with specific join points. Even that overhead can be negligible for typical applications due to hash maps efficiency, it can be problematic for large applications, mainly because accesses to hash maps are synchronised. Such problem can be overwhelmed by augmenting the number of aspect instances and, consequently, restricting the number of join points intercepted by pointcuts of each aspect. As a result, aspects keep a smaller number of entries in the hash maps and enable the parallel access to hash maps.

Use of annotations is best suited for situations where each join point has a particular configuration. For instance, the *sleep* construct is only implemented with annotations, as each captured join point has a different sleep time. An equivalent implementation with

abstract pointcuts requires the definition of a setter method and consequently the creation of a new aspect per each sleep value, which reduces considerably the reusability of such solution.

Futures and waiting guards cannot be fully implemented with annotations. Such constructs require other artifacts not compatible with annotations in order to be instantiated – e.g., methods. Therefore, programmers should be aware of many semantics, which are divided between the annotations written on base functionality and the pointcuts/methods coded in aspects. This solution can be confusing and error prone and, for that reason, it is preferable the version without annotations.

Chapter 6. Illustration examples

This chapter illustrates the use of patterns and mechanisms presented in the previous section to model concurrency in the following applications: Water Tank [29], Fibonnaci and Particle Applet [29].

Concurrency logic is added to core functionality by creating concrete aspects which specify the case-specific events. Concurrency code was removed from the original examples and in some cases the method names were changed for clarity reasons. Both annotation and pointcut based solutions are used to compose concurrency with main functionality, though the pointcut-based solution is preferred whenever the modularisation of concurrency code does not difficult the comprehension of the application code.

6.1 Water Tank

The Water tank application simulates a set of water tanks that are continuously filling and emptying with random amounts of water. Each tank is limited by its capacity and avoids operations that overflow or underflow the tanks capacity.

The Water tank example illustrates the use of active objects, readers-writer lock, one-way and waiting guards. It shows how to apply multiple reuses of the same aspect to a particular tank instance, as well as applying reuses of different aspects. The application structure is described as the follows: water tanks are represented by WaterTank objects and the interaction with each tank is performed by AddWaterController and RemoveWaterController objects, respectively. The former adds water to tanks whilst the latter removes water from tanks.

Methods implemented in class *WaterTank* (Figure 53) can be classified as readers or writers. Instances of *WaterTank* have a preset capacity and an actual volume, as well as public writer operations *addWater* and *removeWater* and the reader operation *getCurrentVolume*. RW-Lock can be applied to *WaterTank* as a mechanism that avoids mutual exclusion synchronisation on concurrent access of reader methods to state variables. *RWLockWaterTank* subclasses the *RWLockProtocol*, which intercepts methods annotated with *@Reader* and *@Writer* annotations and adds the respective behaviour. An alternative pointcut-like implementation is presented in Figure 54.

Classes *AddWaterController* (Figure 55) and *RemoveWaterController* (Figure 56) interact with tanks, adding and removing water to and from them. Methods *addWaterToTank* and *removeWaterFromTank* are annotated with *@Oneway* to be captured by *OnewayProtocol*, in order to be executed in a separated thread. Such methods do not make sense without concurrency: they implement an infinite cycle which should be executed into a separated thread or else the thread starves in one of these cycles.

```
public class WaterTank {
  private float capacity;
  private float currentVolume;

  ...//constructors

  @Writer
  public void addWater(float amount) {
    ...//add the water to tank
  }

  @Writer
  public void removeWater(float amount) {
    ...//remove the water from the tank
  }

  @Reader
  public Float getCurrentVolume(){
    return currentVolume;
  }

  //...
}
```

Figure 53 - WaterTank class

```
public aspect RWLockWaterTank extends RWLockProtocol {

  // This pointcuts should be declared when annotations are not used
  protected pointcut readMethodObjectLock(/*...*/):
    execution(* WaterTank.getCurrentVolume(..)) &&
    this(targetObject);

  protected pointcut writeMethodObjectLock(/*..*/):
    (execution(* WaterTank.addWater(..)) ||
    execution(* WaterTank.removeWater(..))) &&
    this(targetObject);
}
```

Figure 54 - RW Lock concrete aspect

```
public class AddWaterController {
   ArrayList<WaterTank> tanks = null;

   AddWaterController(ArrayList<WaterTank> tankList){
      tanks = tankList;
   }

   public void work(){
      for(int i = 0 ; i < 50 ; i++)  addWaterToTank(i);
   }

   @Oneway
   private void addWaterToTank(int tankNumber) {
      while(true)
         tanks.get(tankNumber).addWater((float) Math.random()*10);
   }
}
```

Figure 55 – AddWaterController class

```
public class RemoveWaterController {
   ArrayList<WaterTank> tanks = null;

   RemoveWaterController(ArrayList<WaterTank> tankList){
      tanks = tankList;
   }

   public void work(){
        for(int i = 0 ; i < 50 ; i++)  removeWaterFromTank(i);
   }

   @Oneway
   private void removeWaterFromTank(int tankNumber) {
      while(true)
         tanks.get(tankNumber).removeWater((float) Math.random()*10);
   }

   ...// other methods
}
```

Figure 56 - RemoveWaterController class

To avoid the overflow and underflow of tanks, filling and emptying operations can only be realised when the amount of water in the tank after the operation is positive and does not surpass the tank capacity. Until those conditions are not met, threads executing those operations should stay blocked. Concrete aspects *OverFlowWaitingGuard* and *UnderFlowWaitingGuard* avoid water tank overflowing and underflowing, respectively. They ensure that threads executing methods that would lead to an invalid state stay blocked until a change of state triggers a reevaluation of the condition. Figure 57 presents the implementation of aspect *OverFlowWaitingGuard* in which the pointcut *blockingOperation* defines the operations that risk leading to an illegal state, the operations

that can unblock threads blocked before – whenever the precondition is not met – using the *deblockingOperation* pointcut and the precondition itself that is defined by the *preCondition* method.

```
public aspect OverFlowWaitingGuard extends WaitingGuardsProtocol {
  protected pointcut blockingOperation(/*...*/):
    call(* WaterTank.addWater(..)) && target(targetObject);

  protected pointcut deblockingOperation(/*...*/):
    call(* WaterTank.removeWater(..)) && target(targetObject);

  protected boolean preCondition(/* arguments */) {
    WaterTank wt = (WaterTank) targetObject;
    Float amount = (Float) args[0];
    return (wt.getCurrentVolume() + amount) <= wt.getCapacity();
  }
}
```

Figure 57 - Water tank overflow waiting guard

An aspect weaving order should be specified, by defining the precedence of each aspect over the other. A new aspect was created (Figure 58) to establish such an order. Firstly, the one-way mechanism is applied to create threads, followed by the guards to ensure that objects cannot transit to an invalid state and, finally, the readers-writer lock. Aspect precedence should obey such a sequence to ensure that precondition validation occurs after creation of threads and before the readers-writer synchronisation, whenever the pointcuts of those aspects quantify over the same join points.

```
public aspect Precedence {
  declare precedence : OnewayConcrete, OverFlowWaitingGuard,
    UnderFlowWaitingGuard, RWLockWaterTank;
}
```

Figure 58 – Aspects precedence in Water Tank

6.2 Water Tank using Active Objects

An adaptation of the water tank implementation illustrates the use of active objects. In that implementation, each instance of *WaterTank* class is an active object. Hence, a concrete subaspect of *ActiveObjectProtocol* (Figure 59) is defined to add the *ActiveObject* interface to the list of interfaces implemented by the classes, in order to be captured by the abstract aspect.

Each *WaterTank* method invocation is executed by the thread dedicated to the active object. As a consequence, waiting guards cannot be used with active objects since the

object dedicated thread must not be blocked, given that it is the only thread executing the object code. If waiting guards conditions are not met, the active object thread became blocked forever. Those invariants can be respected through balking strategies – i.e., raising an exception when the operation cannot be performed.

```
public aspect ActiveObject extends ActiveObjectProtocol {
  declare parents : WaterTank implements ActiveObject;
}
```

Figure 59 - Water tank implemented with Active Objects

6.3 Fibonacci

The widely known recursive fibonacci function is used to illustrate the use of future calls to introduce concurrency into fork/join applications. Figure 60 presents class *Fibonacci*, a sequential Java implementation of the fibonnaci function. This class defines method *compute*, which recursively computes the fibonnaci value.

```
public class Fibonacci {
  protected long value;

  Fibonacci(long val) { value = val; }

  public Long compute() {
    if (value <=1) return(value);
    else {
      Fibonacci f1 = new Fibonacci(value-1);
      Fibonacci f2 = new Fibonacci(value-2);
      Long r1 = f1.compute();
      Long r2 = f2.compute();
      return (r1+ r2); // performs r1.longValue()+r2.longValue()
    }
  }

  public static void main(String args[]) {
    Fibonacci fibo = new Fibonacci(12);
    Long result = fibo.compute();
    System.out.println("Fibonacci result :" + result.longValue());
  }
}
```

Figure 60 - Java implementation of Fibonacci

Figure 61 presents the aspect that adds futures behaviour to a sequential fibonacci implementation. Invocation of the *compute* method is asynchronous and since it returns a value, a future should be used to return the value to the client. Asynchronous invocations using futures requires the definition of pointcut *futureMethodExecution*, which captures the join points where the reusable implementation applies.

Whenever a Fibonacci object is instantiated to compute a high Fibonacci value, the number of concurrent activities – i.e., the number of newly created threads – grows proportionally, which can severely reduce performance as the number of threads greatly exceeds the number of available processors. To limit the number of parallel calls, the pointcut designator *if* is included in the pointcut expression.

```
public aspect FutureFibonacci extends FutureReflectProtocol {
  protected pointcut futureMethodExecution (Object servant) :
    call(Long Fibonacci.compute()) &&
    if(((Fibonacci) servant).value>8) &&
    target(servant);

  protected pointcut useOfFuture(Object servant) :
    call(* Long.longValue()) && target(servant);
}
```

Figure 61 - Fibonacci implementation using futures

6.4 Particle Applet

Particle applet is a toy application based on movable bodies controlled by threads that randomly change their locations. Class *ParticleCanvas* contains references to all particles. Whenever method *paint* is invoked, the *ParticleCanvas* object invokes method *draw* on every particle (Figure 62). Class *Particle* (Figure 63) maintains the state of each particle – using fields *x* and *y*. Class *ParticleApplet* (Figure 64) shows the movement of particles inside an applet viewer: a set of squares, each one representing one particle in the set.

```
public class ParticleCanvas extends Canvas {
  ...//  constructor and accessor methods

  public void paint(Graphics g) { // override Canvas.paint
    Particle[ ] ps = getParticles();
    for (int i = 0; i < ps.length; ++i)  ps[i].draw(g);
  }
}
```

Figure 62 – *ParticleCanvas* implementation

```
public class Particle {
  //fields x and y, constructors

  public synchronized void move() {
    x += rng.nextInt(10) - 5;
    y += rng.nextInt(20) - 10;
  }
  public void draw(Graphics g){ ...// draw rectangle }
}
```

Figure 63 - Particle class definition

In its original form [29], *Particle* contains concurrency code unrelated to the core logic – colored as grey. We can remove the various occurrences of *synchronized* keyword by replacing it with the aspect *Synchronisation* (Figure 65) and localise thread management code – including spawning – in *Oneway* (Figure 66). *onewayMethodExecution* pointcut specifies the methods that run asynchronously. Pointcut *interruptAll* specifies the event associated with the execution of method *stop* to interrupt all threads spawned in the context of the aspect. To implement a step-wise movement among particles one barrier should be introduced after each movement (Figure 67). Threads are unblocked when the last thread executes the movement.

```
public class ParticleApplet extends Applet {
   ...// null when not running
   protected Thread[] threads = null;

   protected Thread makeThread(final Particle p) {
      ...//utility
      Runnable runloop = new Runnable() {
         public void run() {
            //...
            for(;;) {
               p.move();
               canvas.repaint();
               //...
            }
            ...// exception handling
         }
      }
      return new Thread(runloop);
   }

   public synchronized void start() {
      int n = 10; // just for demo
      if (threads == null) {
         Particle[ ] particles = new Particle[n];
         for (int i = 0; i < n; ++i)
            particles[i] = new Particle(50, 50);
         canvas.setParticles(particles);
         threads = new Thread[n];
         for (int i = 0; i < n; ++i) {
            threads[i] = makeThread(particles[i]);
            threads[i].start();
         }
      }
   }

   public synchronized void stop() {
      if (threads != null) {
         for (int i= 0; i < threads.length; ++i) threads[i].interrupt();
         threads = null;
      }
   }
}
```

Figure 64 - Particle Applet

This application was selected for its intrinsically concurrent behaviour. Thus, concurrency code can be modularised but is not unpluggable, as the application is concurrent by nature and then sequential code by itself does not make sense. For instance, if we move elsewhere the concurrency code shown in Figure 64, the isolated implementation of *makeThread* may not be understandable, unless the impact of the aspect in the sequential code is taken into account. Thus, to understand the logic of method, it is necessary to analyse the method code and the impact of aspects on the method. If we use annotations to tag the method, we can modularise concurrency code and give information about the concurrent behaviour added by the aspect to programmers. Figure 68 shows method *makeThread* devoid of concurrency – the method name was changed to *moveParticle* due to semantics.

By tagging the method *moveParticle* with *@Oneway* (Figure 68) – in replacement of the use of aspect *Oneway* presented in Figure 65 –, we create a hook which leaves the method amenable to be intercepted by aspects. Aspects subclassing *OnewayProtocol* enables the capture of annotation related join points and creates a new thread per each method annotated, avoiding the definition of pointcut *onewayMethodExecution*, as the reusable implementation quantifies directly on annotations. *@Oneway* annotation describes asynchronous intentionality of method. A similar situation occurs in method *stop*, since it becomes empty when we remove concurrency-related code. Figure 69 shows the use of *@InterruptAllThreads* to tag methods where all threads should be terminated.

```
public aspect Synchronisation extends SynchronizeProtocol {
  protected pointcut
    synchronisedUsingCapturedLock(ObjectcapturedLock):
      (execution(* ParticleApplet.start(..)) ||
      execution(* Particle.move(..))) &&
      this(capturedLock);
}
```

Figure 65 - Synchronisation aspect

```
public aspect Oneway extends OnewayProtocol{
  //...
  protected pointcut onewayMethodExecution() :
    execution(* ParticleApplet.moveParticle(..));

  protected pointcut interruptAll() :
    execution(* ParticleApplet.stop(..));
}
```

Figure 66 – Oneway aspect

```
public aspect Barrier extends BarrierProtocol{
   //...
   protected pointcut barrierAfterExecution() :
      execution(* Particle.move());

   protected int getNumberThreads(){ return 10; }
   protected String getThreadGroupName() { return "ParticleMover"; }
}
```

Figure 67 - Example of the use of Barrier

```
@Oneway
protected void moveParticle(final Particle p) {
   for(;;) {
      p.move();
      canvas.repaint();
   }
}
```

Figure 68 - Use of annotations to represent the intention of a given method

```
@InterruptAllThreads
public void stop() {}
```

Figure 69 – Interrupt all threads created by the aspect using *@InterruptAllThreads*

6.5 Summary

This chapter presents several examples to illustrate the use of the AOP concurrent reusable patterns and mechanisms. Whenever possible, aspect composition via pointcut definition is preferred. Both versions – using pointcuts and annotations as the interface to specify events – are presented whenever it is considered important for the sake of comparison.

Harbulot and Grud [16] use AspectJ in an attempt to separate the core functionality from parallelisation issues. Several experiments are made, on the basis of various parallel benchmarks, to move thread-related code and message-passing code into aspects. Harbulot and Grud conclude that most of parallel applications require refactorings to take advantage of AO approaches, as parallel code is not generally developed in an OO style. All code related to concurrency issues is placed into a single aspect, without further structuring. Our work achieves a similar goal through a collection of aspects dealing with similar issues, which lead to a higher level of modularity and reuse.

When applications are concurrent by nature, modularisation of concurrency code can leave base code in a form that may not be understood when it is analysed by itself. In those

circumstances, it is preferable to use annotations to describe the element concurrent behaviour and simultaneously provide a hook for aspects to facilitate aspect composition.

Programmers should be aware of the impact of each aspect on applications. Use of multiple aspects intercepting the same join points may require specification of aspects precedence.

Several lines of code used to model concurrency were modularised in the applications presented in this chapter. Several benefits were observed: (1) creation of concurrent applications can be less error-prone as the code of patterns and mechanisms is reusable and only requires programmers to specify the application events captured by the aspects; (2) code readability and analysability is improved, as a consequence of concurrency modularisation, (3) reusability of base functionality is increased as it can be used without concurrency and (4) debugging of code is simplified by unplugging concurrency. The defect can be traced to concurrent or core behaviour, as the sequential version of the application can be executed without concurrency code.

AOP reusable implementations can be composed with domain specific code at class, method or field level. Method-level composition would require in a few cases restructuring of applications to enable quantification by the aspects.

Chapter 7. Performance

This chapter present the result of tests performed using an adaptation of the Java Grande Forum (JGF) [42] multithreaded benchmarks to measure the overhead of the AOP constructs presented in this dissertation when compared with the equivalent OO constructs.

Performance overheads of AOP implementations are classified into three categories:

1. retrieval of join point context;
2. management of global join point history;
3. object and aspect overheads.

Some of the abstract reusable aspects need to capture specific join point context information to be used by the concrete aspect instance. For instance, the barrier reusable aspect can be used to simultaneously manage multiple barriers, each one associated to a join point localisation in the code. The target barrier instance in the aspect is selected by resorting to information captured from the join point context – e.g. join point localisation or the reference of the intercepted object. Join point context can be retrieved by using *thisJoinPoint** values from the AspectJ API and *Thread.currentThread*, or can be passed explicitly in pointcut designators *this* or *target*.

Aspects on a number of occasions manage history of each join point context, maintaining data structures for each join point. This can be performed through a collection – i.e., a hash map – that associates each join point context to particular data structures. Depending on pattern, each structure can be associated to a particular join point location or a specific object. As an alternative, a new aspect could be created per each intercepted object – by using *perthis* and *pertarget* –, to avoid maintaining a map of objects to pattern instances. Such a solution was rejected due to an observed non-safety characteristic of *perthis* and *pertarget* on multi-core machines in current versions of AspectJ compiler – i.e., AJDT 1.3.0. This issue is addressed in 5.2.

Moving code to an aspect also introduces overheads related to aspect instantiation and management, additional objects and method calls. This is a cost due to higher modularisation. The Java Grande Forum (JGF) [42] multithreaded benchmarks were updated in the context of this dissertation to measure the impact of the aforementioned

overheads. These provide low-level benchmarks to measure the overhead of barriers, synchronised methods and fork/join of threads using concurrent programming Java mechanisms (CPJ). Similar benchmarks were developed for each pattern/mechanism not included in the original JGF benchmark, and for the AO versions using the implemented collection.

Thread spawning benchmarks – e.g., One-way, Futures and Active Objects – perform a short non-trivial calculation on a separate thread. Synchronisation benchmarks spawn several threads accessing a single shared counter, which implements the benchmarked synchronisation policy. JGF benchmark style was followed as close as possible, even if sometimes it was not in the most natural AO way.

Table 10 presents the overhead percentage, calculated through the formula

$$Overhead = (CPJ–AOP)/CPJ$$

CPJ is concurrent Java style implementations and AOP our AO implementations. The results were collected for 1, 4, 16 and 64 threads. Presented values are median of 5 executions collected on two unloaded machines, both with Sun JDK 1.5.0_3 and AJDT 1.3.0: a AMD Athlon XP 1800+, 512 MB RAM DDR 266, running Windows XP, and a Intel Dual Xeon 3,2 GHz (with Hyper-thread enabled), 1MB L2, 1 GB RAM DDR2 400, running CentOS 4.1. For reference, it is provided the CPJ number of operations per second using 4 threads on both machines.

Table 10 - Overhead of AO implementations relative to CPJ implementations.

	Athlon XP (N°threads)				Dual Xeon (N°threads)				Operations/s	
	1	4	16	64	1	4	16	64	XP	Xeon
One-way	7	20	17	12	15	15	12	10	6K	12K
Futures	20	31	28	29	13	12	6	4	6K	12K
Barrier	93	45	39	41	88	24	24	27	200K	100K
Active Object	64	44	46	43	9	6	2	6	100K	100K
Synchronisation	2	17	6	13	9	9	8	1	600K	2M
Waiting Guards	19	26	16	18	31	47	44	47	400K	700K
Readers-Writers	0	36	33	31	4	33	29	29	1M	300K
Scheduler	78	63	61	55	8	13	13	15	100K	70K

Table 11 presents the relative cost, in percentage of the total cost, for each implementation. The results were obtained by developing specialised aspects for each benchmark, removing each cost component, and represent the average on these two machines – measurements among machines are close enough to be significant to present only the average values.

The implementation of Barrier, Waitings guards and Scheduler uses *thisJoinPoint** variables to retrieve jointpoint context information. In these implementations, context retrieval introduces a significant part of overheads (see Table 11). Barrier execution with a single thread betrays an unusually high overhead because the thread never blocks in the barrier and still backs the barrier load. Waiting guards also have significant object/aspect overheads, since the aspect oriented version introduces additional locks and notifications. We can say that this cost is mainly due to a higher modularity and not the cost of moving to reusable implementations. One-way uses *Thread.currentThread* to retrieve the running thread and a hash map of hash maps to maintain relationships among creator and created threads. These are relatively low-cost functions when compared to thread management cost. Futures have higher join point context overheads: they rely on reflection to identify and instantiate fake values returned by methods and use a hash map to manage futures. Futures implementation without reflection does not have such a cost, however, for comparison purposes it was considered the less efficient implementation – i.e., the one that uses reflection.

Table 11 - Relative cost, in percentage of total cost in AspectJ implementations.

	One-way	Futures	Barrier	Active Object	Synchronise	Waiting Guards	Readers-Writer	Scheduler
Join point Context	8	48	83	---	---	40	---	88
Global Execution History	27	52*	14	47	---	---	92	9
Aspect/Object Overheads	65		3	53	100	60	8	3

* It was not possible to separately measure global execution history and aspect/object overheads

Active object requires a global map to associate each intercepted object to the corresponding scheduler – described as part of the active object pattern. As the pattern is

applied to the object – captured with pointcut designator target – it does not need join point-specific information. Readers-Writer uses hash map to associate each object to a RW lock, which accounts for most of the overheads on these machines. Synchronisation does not require context information and history as the aspect uses the target object *mutex* and *monitor* captured from the context.

It should be noticed that the above are worst-case scenarios – most real cases are likely to yield better performance – and that no attempts were yet made to optimise the presented implementations.

Access to join point history must be synchronised – this overhead was also included in global history execution overhead –, which seems to be the main constraint to scalability. Join point context and history overheads can be reduced by creating case-specific aspects for each context. Thus, one aspect instance would manage a smaller number of pattern instances, which increases parallelism when the history data structures are accessed.

In all benchmarks, AO implementations enable unpluggability of concurrency mechanisms. This helps to validate the benchmark code itself, by comparing execution times with and without concurrency, ensuring that they are measuring the concurrency mechanisms overhead.

Chapter 8. Discussion

This chapter discusses benefits, limitations and performance trade-offs of using AOP-based implementations to develop concurrent applications. Some insights are presented, taken from the analysis of AOP implementations of concurrency, addressing issues such as the pointcut quantification model, the use of annotations as hooks for composition and the limitations of the use of this collection.

8.1 High-level OO concurrent approaches

This section provides an overview of high-level OO concurrent approaches to develop concurrent applications.

AOP can replace reflective systems for some problems. However, instead of using run-time reification, AO languages perform compile-time weaving, which can lead to higher application performances. Reflective systems – or meta-level architectures – allow the programmer to change the system behaviour by providing access to the meta-architecture. Examples of such concurrent systems are ABCL/R3 [33][32] and MPC++ [19]. ABCL [48] provides active objects, one-way calls and futures. These high-level abstractions for concurrency can be replaced by our equivalent AO implementations. Even priority-based scheduling can be implemented using the scheduler pattern to provide higher priority to specific message types.

Use of specialised languages has the advantage of minimising the gap between the intentions of the programmer and the representation in source code of those intentions. However, it also requires a significant upfront investment in designing and implementing the specific language, as well as developing a specialised weaver for the language. This approach is not scalable [31]. By contrast, this limitation can nowadays be avoided with the use of a relatively mature general-purpose AOP language.

The collection presented in this dissertation provides an AO approach to develop concurrent applications. Use of this collection combined with plain Java – without Java concurrency constructs – can be considered an alternative to a high-level OO concurrent programming language.

In our approach Java is used as a core language and concurrency issues as additional language features. This approach has two main benefits. First, traditional support to concurrency features degrades performance even when the features are not used – e.g., as occurs in Java –, as long as each object implements thread lock management even when concurrency features are not used in the program. A similar benefit was observed in middleware systems, where a performance improvement was attained by extracting non-core features to aspects [49]. Second, treating concurrency issues as additional features – modelled by aspects – helps to develop concurrent applications where concurrency issues are more modular and can be unplugged. This approach eases application development, since concurrency can be added at a later development stage and can be unplugged to allow the use of core functionality without concurrency or for debugging purposes.

Implementing equivalent high-level concurrency constructs using traditional OO approaches requires a significant amount of effort. One example is the transparent implementation of futures – e.g., without following the library approach of Java 5 –, in a way similar to several high-level OO concurrent languages. Traditional approaches require the same kind of source code modification and a complex parsing of method calls, returned values and subsequent use of those values. An AO implementation may reduce significantly the implementation effort, as the pattern code can be reused by other pattern instances.

8.2 Quantification analysis

Concurrency behaviour is plugged into applications at specific events quantified by pointcuts. Consequently, use of constructs is limited to events that can be quantified in aspects according to the pointcut quantification model. Thus, it is not possible to introduce a pattern at an arbitrary point in the application, but only at locations reached by the language join point model – e.g., method calls or variable accesses.

Code refactoring would be necessary in some cases to make the code amenable to be composed with new pattern instances. One example is when a pair of instructions needs synchronised access: the instructions should be extracted to a new method, enabling its quantification by aspects.

Table 12 presents a summary of presented implementations, referring the granularity of quantification used – method, field and class levels –, support of annotations (QA),

composition transparency (CT) [15] and interception of multiple participant classes (MPC) using a single aspect instance.

All patterns except Active Object support method-level quantification. Active Object is restricted to class-level quantification as it represents a class role with an associated behaviour. Field-level quantification can be used whenever we want to bind some pattern to a field access, for reading or writing. Field quantification is performed on scheduler, synchronized and barrier reusable implementations.

Table 12 - Analysis of mechanisms/patterns.

		One-way	Futures	Barrier	Active Object	Synchronized	Waiting guards	RW lock	Scheduler
Granularity	Class	--	--	--	X	--	--	--	--
	Method	X	X	X	--	X	X	X	X
	Field	-	--	X	--	X	--	--	X
Analisys Criteria	QA[13]	X	--	X	X	X	--	X	X*
	CT[14]	--	--	X	--	X	X	X	X
	MPC[15]	--	--	X	--	X	X	X	X

* Possible but with restrictions

Some problems would require multiple mechanism instances capturing the same join points. That property is known as composition transparency [15]. Such situations do not origin problems when aspect instances do not interfere with each other. Composition transparency is feasible for the implementations presented here whenever compositions are valid in equivalent Java implementations. For instance, an active object class can either be

[13] QA- Quantification of Annotations

[14] CT- Composition Transparency

[15] MPC- Multiple Participant Classes

an active object class or not. It does not make sense to have more than one active object aspect composing to the same class.

The WaterTank class presented in Figure 70 uses annotations to demonstrate the composition transparency of multiple aspect instances – an equivalent implementation can be done with pointcut specification instead of annotations. Method *addWater* is annotated with *@BarrierAfterExecution* in place of be intercepted by the Barrier aspect. That annotation represents two barriers applied at the same execution join point: (1) barrier intercepts threads of *LockW* thread group, with a threshold of five threads and (2) threads of *LockR* thread group with a threshold of three threads.

```
public class WaterTank {
  private float capacity;
  private float currentVolume;

  //...
  @BarrierAfterExecution(nThreads={5,3},threadGroup={"LockW","LockR"})
  public void addWater(Float amount) //throws OverflowException
  {
     currentVolume += amount;
  }
}
```

Figure 70 – Multiple Barrier instances applied to the same point

Barrier, synchronized, reader-writer lock, waiting guards and scheduler are able to include instances of multiple, type-unrelated classes in the context of a single mechanism. This can be useful in many situations – e.g., creating locks involving many classes or implementing a barrier with multiple, optional synchronisation points. Figure 71 shows the use of SynchronisationProtocol to apply a shared lock to two methods defined in two distinct classes.

Although it is possible to handle multiple participant classes, waiting guards depend on values captured from the context in order to perform precondition validation. In such situations, it might not be straightforward to define a precondition that would be applied to various type unrelated classes.

```
public aspect Synchronisation extends SynchronizeProtocol{
  protected pointcut synchroniseUsingSharedLock() :
    execution(* particleapponeway.ParticleApplet.start(..)) ||
    execution(* particleapponeway.Particle.move(..));
}
```

Figure 71 – Using the same lock to synchronise access to methods of distinct objects

8.3 Annotations

Annotations can be useful and sometimes essential in situations where concurrency is intrinsic to the situation at hand. In that case, if we modularise concurrency, the functional code may possibly become hard to understand by itself. This problem was mentioned in section 6.4. Annotations describe facets of behaviour by giving information about the behaviour added by the aspects, to the person looking at the code. Hence, annotations have two basic roles: describe the concurrent behaviour of a method and provide hooks for aspects to compose. By using annotations as attributes describing some property or some role of an element – i.e., class, method or class fields –, aspects become more independent from element syntax. Consequently, changes on class names, field names or method signatures do not cause changes on aspects that intercept such elements. This problem is known as the *fragile pointcut problem* [44]. Other limitations of composition through the use of pointcuts are presented in [18].

Not all AOP PM can be fully implemented with annotations. Some AOP reusable logic requires the definition of methods. Configuration values are typically returned by anchor methods implemented in the concrete aspect, as the *getNumberThreads* method, that configures the aspect with the number of synchronisation threads at Barrier instance. Element-value pairs can be used with annotations to specify configuration values in replace of methods definition. Such information can be specified at annotation level, as a form of parameter defined with the annotation. This solution can be used whenever the values are known at compile time. Though, whilst information specified on annotations are static, method-like implementations enable the capturing of configuration values from the application context. Furthermore, some aspects require the definition of case-specific code – e.g., precondition definition on waiting guards –, which only can be defined using methods.

Though annotations loose coupling between aspects and the classes to which they compose, they suffer from non-locality, which is one of the problems that aspects are supposed to solve. Thus, the capture of which methods are participants in the concern is explicit but scattered across multiple modules. This problem is analysed in [25].

The OpenMP [39] model uses annotations to express concurrency issues, in a way similar to our annotations. Both approaches support the development of concurrent applications where concurrency is specified through code annotations that can be ignored when

concurrency is unplugged. As an example, OpenMP's *parallel for* annotations resemble *@Oneway* annotations and OpenMP's *critical* annotations are similar to *@Synchronized* annotations. Our annotations are inserted at class level – i.e., it is not possible to insert an annotation within specific object instances –, providing less flexibility and we do not provide many features of OpenMP, such as loop scheduling. On the other hand, we provide a more high-level way to structure applications, by leveraging OO annotations. In addition, we include high level mechanisms not present in OpenMP, namely futures and RW locks.

8.4 Limitations

Most of the limitations of the AOP collection are related with AspectJ limitations in obtaining local join point context information. This is partly due to the fact that abstract pointcuts typically preset the context information that is captured, which limits the information that can be captured from the context. For instance, the waiting guards mechanism requires the definition of a precondition to control the execution of methods, which is formed by an expression that would use information from the context – e.g., method parameters – that cannot be captured if it is not considered upfront in the pointcut specification. Though, one method can have one parameter of a specific type whilst the other method can have two parameters with different types. Thus, a reusable pointcut cannot be specified, as the number of parameters is unknown. Waiting guards and futures overcome the problem by resorting to reflection – AspectJ and/or Java reflection – to obtain the needed information. This solution yields maximum reusability but pays a price in performance.

Futures only should be applied to methods with non-primitive return types. However, Java 5 introduces *Autoboxing* to wrap primitive types into objects and vice-versa, which avoids such limitation. Replacement of fake objects with the real ones requires interception of all accesses to objects of that type and not just accesses to instances returned by future calls, with the purpose of verifying if such object is a fake object or not. That verification can degrade performance, though the problem can be minimised by limiting the scope of the pointcut. In case performance degradation is not acceptable, the futures mechanism has an alternative implementation which requires definition of a separate concrete aspect per each invoked object type, letting the programmer define the fake object instantiation explicitly. The latter solution yields better performance at the expense of reusability.

Reusable aspects are global entities that cannot distinguish among specific class instances. As a consequence, it is not possible to apply a barrier to specific threads – or to the objects that represent the threads. This limitation can be partially overcomed by enabling a barrier to apply to a specific thread group. To do that, each thread spawned by one-way mechanism can be associated to a specific thread group.

Chapter 9. Conclusion

Concurrent programming is gaining importance with the advent of multi-core and multi-threading processors. Programming with concurrency is a hard task, usually left to experts, namely due to the execution unpredictability introduced by concurrency. Reusability of components is difficult when they implement concurrency, because otherwise many components may need to be rewritten to be used in non concurrent contexts.

Patterns were introduced as good practices to design good solutions for common problems [12]. Several concurrency patterns were identified to cover numerous concurrency problems, namely One-way calls, Futures, Active Objects, Barriers, among others. Although patterns can improve the design of concurrent applications, they lack for maintainability as patterns-related code is tangled with participant classes code and scattered over multiple classes, which limits reusability of both the main functionality and patterns code. In addition, debugging is made more complex as the defect cannot be traced to concurrent or core behaviour.

Traditional programming languages provide a weak support for development of concurrent applications. Separation of concurrent code from main code is one desired characteristic to support the debugging process, reduce complexity and enhance reusability, though such separation is not a built-in feature in conventional object oriented languages. Aspect oriented programming aims to modularise crosscutting concerns in applications. Such concerns do no align well with class decomposition and therefore should be implemented separately from the classes. Concurrency is an example of crosscutting concerns and therefore it is a good candidate to be modularised using AOP techniques.

This dissertation presents a collection of reusable patterns and mechanisms implemented in AspectJ. Each aspect wraps the pattern logic that would be plugged into applications by means of pointcuts or annotations which are defined by programmers to be used as hooks for aspects to compose. Applications can be developed without concurrency and concurrency patterns are plugged afterwards into applications, through the creation of concrete aspects that inherit from base aspects.

By aspectising concurrency patterns, modularisation of concurrent code enables code reuse, ameliorates code readability and analysability and aims to achieve a more

independent application development, test and configurability. Reuse of concurrency code substantially reduces concurrency related errors as long as patterns code can be well tested and reused several times. Furthermore, due to the unpluggability of concurrency code, sequential code can be debugged and reused without concurrency.

Abstract pointcuts typically preset the context information, which may limit reusability of AOP concurrency constructs: acquisition of context specific information required by the reusable code is preset, which limits the flexibility to capture specific data to be used by the reusable code. Concurrency code can only be added at specific points in the applications, restricted to the locations quantifiable by pointcuts, which may limit expressiveness when programmers use concurrency constructs within applications.

Modularisation may introduce decomposition costs, as long as new objects are created and new relationships are established which can degrade performance. Composition of aspects should be managed carefully, as the introduction of new patterns impacts on other patterns previously introduced. Thus, precedence control may be necessary to avoid errors related with dependencies between pattern instances.

Annotations can be used to simplify aspect composition by describing the points in applications captured by the aspects. In addition, annotations describe concurrent behaviour of program elements – e.g., classes and methods. This characteristic of annotations is useful, as long as concurrency modularisation enables applications main code to be oblivious of concurrency. As a consequence, anyone who reads the code will be unaware of the impact of concurrency on the application, unless reading the aspects code, which can limit comprehension of applications.

This dissertation does not address the impact of each pattern over the others by thoroughly analysing pattern compositions. It also does not analyse applications in terms of separation of concurrency code by describing which concurrency code can be modularised in concurrent applications and neither presents a methodology to describe how to transform sequential applications into concurrent applications.

This collection implements a set of common patterns and mechanisms for concurrency. In future work, other concurrency patterns can be added to this collection. As many patterns were implemented upon java mechanisms for concurrency, the use of *java.util.concurrent* package of Java 5 promises to make the code more efficient.

Bibliography

[1] Andrews, G., Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000.

[2] Andrews, G., Schneider F., Concepts and Notations for Concurrent Programming, ACM Computing Surveys, 1983.

[3] Bergmans, L. M. J., Composing Concurrent Objects: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs, Ph.D. thesis, University of Twente, Netherlands, June 1994.

[4] Budinsky, F., Finnie, M., Yu, P., Vlissides, J., Automatic code generation from Design Patterns. IBM Systems Journal 35(2): 151-171.

[5] Buhr, P., Harji, A., Concurrent Urban Legends, Concurrency and Computation: Practice and Experience, 17:1133-1172, Wiley InterScience 2005.

[6] Colyer A., Clement A., Harley G., Webster M., Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools, Addison Wesley 2004.

[7] Colyer A., Making concurrency a little bit easier, blog entry, January 2005, www.aspectprogrammer.org/blogs/adrian/ 2005/01/making_concurre.html

[8] Dustard, David W., Concepts of Concurrent Programming. Curriculum Module CM-24. Pittsburgh, PA: Software Engineering Institute, April 1990.

[9] Florijn, G., Meijers, M., Winsen, P. van., Tool support for object-oriented patterns, Proceedings of ECOOP 1997.

[10] Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972.

[11] Fowler, Marting, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman, 1999.

[12] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[13] Grand, M., Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, 2nd Edition, Wiley, 1998.

[14] Hanenberg, S., Schmidmeier, A., Unland, R., AspectJ Idioms for Aspect-Oriented Software Construction, 8th EuroPLoP, Irsee, Germany, June 2003.

[15] Hannemann, J., Kiczales, G., Design Pattern implementation in Java and in AspectJ, OOPSLA 2002, Seattle, USA, November 2002.

[16] Harbulot, B., Gurd, J., Using AspectJ to Separate Concerns in Parallel Scientific Java Code, AOSD 2004, Lancaster, UK, March 2004.

[17] Haustein, M., Löhr, K., JAC: Declarative Java Concurrency, Concurrency and Computation: Practice and Experience, vol. 18, no. 5, April 2006.

[18] Herrejon, R., Batory, D., Taming AspectJ Composition: A Functional Approach, Technical Report TR-05-27, May 2005.

[19] Holmes, David, Synchronisation Rings Composable Synchronisation for Object-Oriented Systems, Ph.D. thesis, Macquarie University, Sydney, October 1999.

[20] Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte J., Tezuka, H., Konaka, H., Maeda, M., Kubota, K., Design and Implementation of Metalevel Architecture in C++ - MPC++ Approach, Workshop on Reflection and Metalevel Architecture (Reflection'96), San Francisco, CA, April 1996.

[21] Java 1.5 Specification, http://java.sun.com/j2se/1.5.0/docs/api/index.html

[22] JBoss AOP web site. http://www.jboss.org/products/aop.

[23] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., An Overview of AspectJ. ECOOP 2001, Budapest, Hungary, Springer Verlag LNCS vol. 2072, pp. 327-353, June 2001.

[24] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J. Aspect Oriented Programming, ECOOP'97, Jyväskylä, Finnland, June 1997.

[25] Kiczales, G., Mezini, M., Separation of Concerns with Procedures, Annotations, Advice and Pointcuts, ECOOP'05, Glasgow, UK, July 2005.

[26] Laddad, R., AspectJ in Action – Practical Aspect-Oriented Programming, Manning 2003.

[27] Lamport, L., A Simple Approach to Specifying Concurrent Systems, Comm. ACM 32, 1 (Jan. 1989), 32-45.

[28] Lavender, R. G., Schmidt, D. C.., Active Object: An Object Behavioral Pattern for Concurrent Programming, 2nd Annual Pattern Languages of Programming (PLoP'95) Conference.

[29] Lea, D., Concurrent Programming in Java, Second edition, Addison-Wesley, 1999.

[30] Lopes C. V., D: A Language Framework for Distributed Computing, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, USA, November 1997.

[31] Lopes, C. V., AOP: A Historical Perspective (What's in a Name?). In Aspect-Oriented Software Development, pages 97–122. Addison-Wesley, 2005.

[32] Masuhara, H., Matsuoka, S., Yonezawa, A., Implementing Parallel Language Constructors Using a Reflexive Object Oriented Language, Workshop on Reflection and Metalevel Architecture (Reflection'96), San Francisco, USA, April 1996.

[33] Masuhara, H., Yonezawa, A., Design and Partial Evaluation of Meta-Objects for a Concurrent Reflective Language, ECOOP'98, Brussels, July 1998.

[34] Matsuoka S., Yonezawa A., Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In Research Directions in Concurrent Object-Oriented Programming (Agha G., Wegner P., et al., editors), pp. 107-150, MIT press, 1993.

[35] Matsuoka, S., Kenjiro, T., Akinori, Y., Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, Proceedings of the OOPSLA'93 Conference on Object-oriented Programming Systems, Languages and Applications, Pages 109-126, 1993.

[36] McHale C., Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance. Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, Ireland, October 1994.

[37] Nordberg III, M. E., Aspect-Oriented Dependency Management, In Aspect-Oriented Software Development, pages 557–584. Addison-Wesley, 2005.

[38] Oaks, S., Wong, H., Java Threads, 3rd edition, O'Reilly 2004.

[39] OpenMP architecture review board, OpenMP Application Program Interface, Version 2.5, May 2005, www.openmp.org

[40] Schmidt, D. C., Stal, M., Rohnert, H., Buschmann, F. (eds), Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley & Sons 2000.

[41] Silva A. R., Concurrent Object-Oriented Programming: Separation and Composition of Concerns using Design Patterns, Pattern Languages, and Object-Oriented Frameworks, Ph.D. thesis, Technical University of Lisbon, March 1999.

[42] Smith, A., Bull, J., Obdrzálek, J. A Parallel Java Grande Benchmark Suite, Supercomputing 2001, Denver, November 2001.

[43] Soukup, J., Implementing Patterns, In: Coplien J. O., Schmidt, D. C. (eds.) Pattern Languages of Program Design, Addison Wesley 1995, pp. 395-412

[44] Störzer, M., Koppen, C., PCDiff: Attacking the Fragile Pointcut Problem, Interactive Workshop on Aspects in Software (EIWAS) 2004, Berlin, Germany, September 2004.

[45] Sutter, Herb, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb's Journal, 30(3), March 2005.

[46] The AspectJ 5 Development Kit Developer's Notebook, http://eclipse.org/aspectj/doc/next/adk15notebook/

[47] Yonezawa, A. Tokoro, M. (ed). Object-Oriented Concurrent Programming, MIT Press, 1987.

[48] Yonezawa. A., ABCL: an Object-Oriented Concurrent System, MIT Press, 1990.

[49] Zhang, C., Jacobsen, H., Resolving Feature Convolution in Middleware Systems, OOPSLA'04, Vancouver, Canada, October 2004.