An Evaluation of AOP for Java-based Real-time Systems Development

Shiu Lun Tsang

A dissertation submitted to the University of Dublin in partial fulfilment of the requirements for the degree of Master of Science in Computer Science

2003

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Shiu Lun Tsang

September 15, 2003

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Shiu Lun Tsang

September 15, 2003

Acknowledgements

I would first like to thank my supervisor Dr. Siobhán Clarke for her help, not just during this dissertation, but throughout the whole year. Her guidance and input have been invaluable and greatly appreciated.

Special thanks goes to Elisa Baniassad for her helpful suggestions and comments and to Vinny Reynolds for all his help with Simulator.

To the NDS class, for making this year an experience to remember. Thank you.

Finally I would like to thank my family and friends for their continued support and simply for being there when needed.

Abstract

Advances in programming language have improved the software developer's ability to achieve *separation of concerns* (SOC). Although object-oriented (OO) technology offers greater ability than previous language paradigms to achieve this goal, it still has difficulty localising concerns that do not fit naturally into a single program module. Aspect-oriented programming (AOP) introduces a new style of decomposition for capturing such crosscutting behaviour that further enhances a software developer's ability to achieve SOC.

Although studies have shown that AOP is beneficial to many OO systems, it has yet to be proven for real-time systems development, where systems are bound by strict temporal constraints. Languages such as Real-time Java have encouraged the use of OO languages in real-time system development, but may result in codebases that significantly differ from conventional software systems due to the necessity of additional real-time constructs.

The goal of the dissertation was to evaluate the use of AOP in the development of Java-based real-time systems. The main focus was on the seven enhanced areas introduced by the Real-time Specification of Java. The Sentient Traffic Simulator application developed by the Distributed Systems Group in Trinity College, was extended to satisfy specific temporal constraints, with a subsequent re-implementation using AOP enhancements. The overall development process involved an initial analysis of areas in the original system that required real-time behaviour and the later identification of crosscutting real-time functionality that exists in the codebase.

Evaluation of the effectiveness of AOP for Real-time Java was performed to illustrate the impact of this paradigm on understandability, maintainability, reusability, and testability attributes.

Based on the results gathered, it has been concluded that AOP does provide useful enhancements but the improvements shown are significantly less impressive than previous studies carried out for AOP and OOP. The overall effectiveness of AOP is highly application specific and dependant on the amount of aspectual behaviour exhibited in the codebase of the for Java-based real-time system.

Table of Contents

List of Figures

List of Tables

1 Introduction

Advances in programming languages have aimed to improve the software developer's ability to achieve separation of concerns (SOC). SOC refers to the identifying, encapsulating, and manipulating of those parts of a system that are relevant to a particular concern [1]. If successfully achieved, this can result in great benefits to system understandability, maintainability, testability, and reusability.

Currently OO programming is the dominant programming paradigm where a problem is decomposed into objects that abstract behaviour and data in a single entity. Yet OO still has limitations in achieving total SOC [2]. Irrespective of how well designed and implemented, OO software may eventually lead to the problem scenarios of code scattering (concerns spread over many modules) and code tangling (concerns which are tightly intermixed) [2, 3], which oppose the SOC principle.

Aspect-oriented Software Development (AOSD) [4], which encompasses Aspect-Oriented Programming (AOP) and other AOSD approaches, has built on the existing OO paradigm as the latest language advancement in the pursuit of achieving SOC. AOSD has introduced concepts that are designed to enable developers' to cleanly capture and modularise crosscutting functionality in a system which were previously unavailable. A large volume of research has been carried out in the area of AOSD and in particular AOP with studies highlighting the SOC benefits that this new paradigm offers in the development of (non real-time) software systems [2, 5, 6].

The development of real-time systems differs significantly to that of conventional software applications [7]. The principles of predictability and performance, which are essential in a real-time environment, can only be achieved with stricter and more carefully designed programming constructs and practices than those used applied in non real-time applications. Traditionally real-time systems have been developed using procedural languages such as C and ADA or in some cases even assembler. Yet these languages do not provide the programming ease or simplicity evident in the OO paradigm. Moreover the growing popularity and programmer familiarity with OO languages in the development of distributed systems has thus encourages them to be adopted in the real-time domain.

At present Java is one of the most commonly used programming languages in software development. However, traditional Java constructs incur unpredictability in performance which proves unsuitable for a real-time environment. The limitations of Java for real-time systems development have therefore resulted in an extension to its API which introduces a number of concepts enabling the language to satisfy real-time constraints. Real-time Java (as this extended version is known) enhances its traditional counterpart in seven areas that were previously unsupportive of real-time principles.

By introducing additional constructs, the development of Java-based real-time systems may lead to a codebase that may differ significantly to that of traditional (non real-time) applications. Therefore the effectiveness of AOP in such a domain is unclear.

1.1 Research Goal

The main aim of the dissertation is to determine if AOP techniques provide better separation of concerns for the development of Java-based real-time systems. The focus of the project is on how AOP affects code implementing the seven enhanced areas of Real-time Java as set out by the Real-time Specification Group (RTSJ) [8] and an evaluation of the usefulness of AOP will be based on these outlined areas.

1.2 Research Approach

To achieve the goals of the dissertation, a Java-based real-time test application will be implemented in two versions, one using Real-time Java in pure OO programming, and a subsequent re-implementation using Real-time Java with AOP enhancements. The Sentient Traffic Simulator developed by the Distributed Systems Group in Trinity College will be the application adopted for the dual development. On completion of the two cases an evaluation of each will be performed highlighting their implementation differences.

The main focus will be on the seven enhanced areas introduced in the Real-time Specification for Java. These are the areas where Java's API has been extended to satisfy the real-time principle of predictability. They are thread scheduling and dispatching, memory management, synchronization and resource sharing, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, and physical memory access.

Evaluation of the effectiveness of AOP for the above seven areas was performed to illustrate the impact of this paradigm on understandability, maintainability, reusability, and testability attributes. An OO metrics suite was selected as the basis for this evaluation as they are primarily designed to highlight important SOC issues regarding encapsulation and inheritance. In addition adopting such a metric will enable a further assessment into the suitability of OO metrics for evaluating AO software.

Finally the data gathered from the evaluation will be analysed to conclude benefits and drawbacks of each approach in relation to each of the seven Real-time Java areas. The results will in turn be evaluated against the research question, and will assess the effectiveness of AOP in the domain-specific, Java-based real-time arena.

1.3 Work plan

Accomplishing the research goal involves the following tasks:

- Research the main considerations of system development in the real-time arena
- Examine the seven enhanced areas associated with Java-based real-time systems that distinguish them from conventional Java systems

- Understand the advantages/disadvantages of AOP for separation of concerns
- Determine suitable metrics for assessing the impact of AOP in Java-based real-time systems
- Implement a Java-based real-time system
- Re-implement the system using AOP techniques
- Evaluate the effectiveness of AOP techniques for the implemented Java real-time system based on the metrics found

1.4 Research Contributions

Research in the RTJava area has mainly been concentrated on the seven enhanced areas as set out by the Real-time Specification of Java. Substantial work has been carried out to provide an acceptable implementation for the specification, while other work has focused on the evaluation of the suitability of implemented specification for the (hard and soft) real-time arena.

Being a relatively new programming paradigm, AOP and the larger Aspect-oriented Software Development (AOSD) domain, is a wide research area in the growing AOSD community. Previous work has mainly focused on certain non-functional areas that are common to most information systems such as tracing, logging, synchronization and exception handling. Large amount of statistics and figures have been made available with the majority heralding the arrival of AOP as a solution to modularising crosscutting functionality. Very few have highlighted potential drawbacks that this relatively new paradigm brings with most research choosing to emphasis its benefits. Other work in the AOP area relates to suitable metrics that are applicable for assessing the effectiveness of AOP development. No suites of metrics have yet to be developed and work in this area is growing.

Very few studies have been carried out on the areas of RTJava and AOP in conjunction. Those of which known are documented in the State of the Art chapter (chapter 2) of the dissertation.

The work that was carried out was aimed to bring certain contributions to both the areas of real-time system development and AOP. They consist of the following:

- AOP evaluation for real-time systems characteristics. The seven enhanced areas of RTJava highlight all the characteristics that distinguish real-time systems from non real-time ones. By applying AOP techniques to all these areas provides a solid basis for the study of how effective this paradigm is for real-time systems development. This contrasts previous studies which have mainly focused on applying AOP to individual areas but not all seven in conjunction. By applying AOP to all seven gives a high level analysis of Java-based real-time programming with the added dimension of interactions that may occur between the areas.

- An evaluation of metrics that could be applied for assessing the benefits and drawbacks of AOP to OOP systems.
- Highlights the affect of AOP enhancements of numerous system attributes, namely understandability, maintainability, reusability, and testability.
- Identifies some myths that are present in previous AOP research where the consideration of certain issues may have AOP favourable results less compelling and conclusive.

1.5 Sentient Traffic Simulator

The Sentient Traffic Simulator (Simulator) was developed by a member of the Distributed Systems Group, Trinity College as part of an unrelated research project. This application was designed to simulate a sentient traffic management system that allows autonomous sentient vehicles to co-exist safely and efficiently. It is important to note that the application ignored any real-time considerations and did not include any code that provided real-time related attributes or functionality. Thus this along with the hard-real time behaviour that the Simulator exhibits was the main reasons why this application was chosen for this dissertation. More information about the Sentient Traffic Simulator is available at [9].

1.6 Development Environment

All implementation was carried out with Eclipse version 2.0 Integrated Development Environment with Sun's JDK 1.4. The Real-time Java implementation used was the TimeSys RTSJ Reference Implementation [10]. AspectJ version 1.1 was used for the aspect approach. All the above are running on Linux RedHat 7.1 with the TimeSys Linux/RT 3.0 GPL kernel [11].

1.7 Dissertation Roadmap

The remaining chapters of the dissertation are organised as follows:

Chapter 2: State of the Art

This chapter describes the current state of the art in the areas of real-time systems, in particular Real-time Java and AOSD. Several approaches to AOSD are documented. The chapter also discusses evaluation metrics that may be adopted for evaluating aspect-oriented software. The related work of the above areas is also described.

Chapter 3: Object-oriented Design and Implementation

This chapter describes the object-oriented (OO) version of the design and implementation of the Sentient Traffic Simulator. The details of system prior to any real-time alterations (i.e. original design and

implementation) are given. The subsequent OO design and implementation of Simulator inclusive of Real-time Java constructs is then provided, with each of the seven enhanced areas addressed.

Chapter 4: Aspect-oriented Design and Implementation

This chapter provides the design and implementation of the AO version of the Simulator.

Chapter 5: Evaluation Metrics

This chapter gives an in depth description of the Chidamber and Kemerer metrics suite that is used in the evaluation of the Simulator. Each of the metrics in the suite are described along with the affects that aspect-orientation may have on each.

Chapter 6: Evaluation

This chapter provides an evaluation of the use of aspect-oriented techniques for each of the seven Real-time Java areas. The gathered metric results are analysed and a discussion on the implications of these results on certain system attributes (understandability, maintainability, reusability, and testability) is given.

Chapter 7: Conclusions

This chapter provides the concluding remarks based on the evaluated results and some future work related to the dissertation.

2 State of the Art

This chapter describes the current state of the art in the areas of real-time systems and Aspect-oriented Software Development (AOSD). In particular, the areas of Real-time Java (RTJava) and aspect-oriented programming will be addressed. The chapter also documents potential metrics that can be used in the evaluation of aspect-oriented software.

2.1 Real-time Systems Defined

This section provides an overview of real-time systems.

2.1.1 Definition of a real-time system

A real system is one which "must produce correct responses within a definite time limit" [12]. The effect if not responding within this limit is either performance degradation and/or system malfunction. It is important to note that the word "time" needs to be taken in the context of the total system. Depending upon that system this time limit could be microseconds, seconds, minutes or even hours. Yet regardless of the time limit, the computer system must be guaranteed to respond with the correct response within its specified time limit. In general, real-time systems are either event-triggered based on interrupts, or are time-triggered based on deadlines [13].

2.1.2 Characteristics of a real-time system

All real-time system posses a number of special characteristics [7, 14]. According to [15] Real-time systems have a number of distinguishing characteristics. They are:

- **Timely** in the sense that it's possible to predict how long a particular task will take to execute and it's possible to initiate task execution at a specific time.
- Typically **concurrent**, with multiple sequences of execution (threads) if varying priority simultaneously (or pseudo-simultaneously) active within a task or process.
- **Responsive** so as to interact with physical interfaces (hardware) to the surrounding environment in a predictable and usually rapid manner.
- Often **embedded** into a system such that the behaviour of the real-time software system is indistinguishable from that of the host.
- Highly **predictable**, in that they do the right thing all of the time.
- **Robust**, in that they will do the right thing under unexpected or even erroneous conditions.

Real-time systems vary as to the degree to which they meet the above criteria. [16] divides software into four distinct groups based on time. Time plays a greater deal of importance in the latter two described. Soft real-time systems are those where response times are important but the system will still function correctly if deadlines are occasionally missed, while hard real-time systems are those where it's absolutely imperative that responses occur within a specified deadline. Missing a hard deadline can result in catastrophic consequences or loss of system performance.

2.1.3 Real-time dimensions

The space of real-time problems has at least three useful dimensions: the precision with which the time is measured, the importance of consistency, and the shape of the utility curve around the deadline [17].

2.1.3.1 Precision of Measurement

The precision of the units the real-time system uses to measure a period of time helps to characterise the type of system. Some systems use a submicrosecond as a time unit while others use conventional seconds. The time-scale pyramid shown below indicates the differing time constraints for different devices and applications [17].

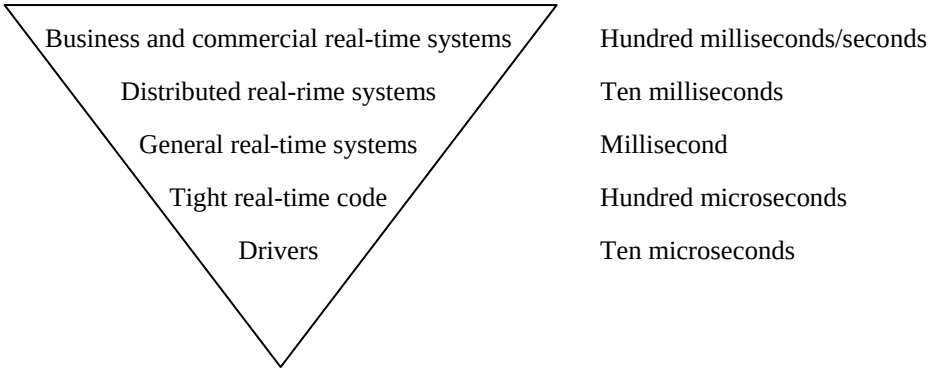

FIGURE 1: TIME-SCALE PYRAMID

2.1.3.2 Consistency

A vital property in the real-time domain is the notion of determinism, meaning that timing is predictable at the precision that is specified by the system. Consistency is more that mere determinism. The central idea of consistency in real-time is to reduce to a minimum the difference between expected performance and the worst possible performance. For example, it is useful to know that an urgent event will occur in your system at sometime between 10 and 200 microseconds. However it is better still to know that this

time interval will be between 50 and 70 microseconds. Reducing the gap between the *best-case* and *worst-case* can greatly improve the overall performance of the system.

2.1.3.3 Utility Function Curve

Utility occurs in the context of real-time when the system is late or misses a deadline. Dependant on the type of system (non real-time, soft or hard-real time), missing a deadline will result in different results or responses. The idea of the utility function curve is to aid in the design of real-time systems. They can depict when a time constraint is missed and help programmers identify the consequences and seriousness of early, late or non completion of an operation.

2.1.4 Myths of real-time systems

There are some myths associated with real-time systems that need to be expelled. One is that “real-time systems need to be fast” [12]. This may not necessarily be the case. Real-time systems need only respond within a specified time limit, whether it is microseconds, seconds, minutes or hours, depending on the specific system. Another myth is that real-time systems must be compact and require the use of minimum amount of memory. Again this is the requirement of a particular system, but is not a defining feature of a real-time system. To summarise, a real-time system is a system which must provide an appropriate response within a certain time period.

2.2 Java technology and real-time

This section describes the background related to transforming Java into a language suitable for real-time systems development.

2.2.1 The language requirements for real-time systems

There are two main additions to the requirements of a programming language to be used in the development of real-time systems [12, 15]. Firstly the language must provide predictable performance when handling different events as they occur. Thus the designer of the system must be able to predict the delivered performance of the system under all possible situations. Secondly, the language must support the servicing of multiple events, where theses events appear at random and asynchronously. This implies that it is possible to prioritise the processing of these events by separate, potentially asynchronous, processes. Other requirements relate to the execution speed and memory requirements. This is because some real-time systems need to be very fast or have to be able to run within limited resources such as memory. Yet the primary concern remains the predictability of the performance of the system, ensuring that the response time of the system is within prescribed limits. The performance characteristics of any language used for real-time systems must be deterministic.

2.2.2 Java for real-time systems

The Java programming language was originally designed by Sun to facilitate the development of embedded systems software [18, 19], but has grown into the prominent language for web programming because of its ability to simplify the development of flexible, portable, distributed applications with high-level graphical user interfaces [18].

Java, as it stands, does not guarantee the performance predictability required for real-time systems, thus it is often claimed, “Java is not an appropriate language for real-time programming” [20]. Here Johnson et al. state some of the reasons for why Java cannot ensure a real-time acceptable level of determinism:

- The nature of real-time systems requires a priority-based scheduler. In Java, tasks are modelled using threads, but working with thread priorities is difficult.
- Monitors in Java present the priority inversion problem [21], and there is no guarantee about the policy of waiting queues
- The resource management abilities need to be extended to cover four aspects: the ability to track usage (resource accounting), the ability to reclaim the resources of an activity when it terminates (resource reclamation), the ability to reassign resources to an activity (resource reassignment), and the ability to change the resources assigned to activities (resource negotiation).
- From a real-time perspective, asynchronous even handling present’s problems in Java, as a thread can be asynchronously interrupted by another.
- Java’s garbage collector runs as a separate thread that, once started, cannot be stopped, making the entire system non-deterministic.

Many real-time systems are developed in C, and increasingly in C++. Yet developers have increasingly started to consider Java as a potential platform for implementation of real-time systems. Java is certainly the modern computing language that possesses all the elements required for well engineered reliable code. Indeed [12] has shown that there is a one to one matching between features of Java and those of ADA 95 [7], a language which is often considered to be particularly suited to real-time systems.

2.2.3 Java’s Real-time Obstacles

2.2.3.1 Garbage Collection

The biggest obstacle to predictable performance in Java is the garbage collector. This is a process which runs within its own low priority thread under three situations:

- Whenever the processor is free

- Memory needs to be reclaimed
- The user indicates that garbage collection is required

The problem is that the garbage collector effectively makes the program non-deterministic. In addition it is no possible to specify exactly when it will occur. The JVM will decide when and if the garbage collection process will execute, using a programmer's request for garbage collection as an indicator rather than an absolute request. The result of this is that it could execute some time later during a time critical operation which may result in failure to meet the specified time constraint.

2.2.3.2 Dynamic Class Loading and Initialisation

Dynamic class loading is a significant impediment to real-time behaviour. Java incorporates the concept of late binding. In all Java programs, the JVM may postpone the actual loading of a class file until it is actually referenced. This is an effort to speed up the execution of a Java program. However this introduces a degree of non-determinism that is problematic for the real-time domain. Even if it were possible to introduce determinism to class loading, class initialisation is always performed at he time of first active use. This requirement is imposed by the Java language Specification [22]. Thus this too will be non-deterministic, as the initialisation of a class triggers a sequence of class initialisation methods that is inherently unpredictable.

2.2.3.3 Object Construction and Initialisation

Similar to the issues of class initialisation, where a class instance (i.e. object) is created, it triggers a recursive sequence of constructor invocations. The time required for this will depend on the depth of the class hierarchy and the nature of the constructors themselves, introducing a non deterministic property that may not be suitable for real-time development.

2.2.3.4 Dynamic Method Binding

Dynamic binding is a method that uses inheritance to achieve polymorphism can impose performance penalties if poorly implemented. In such a case, the JVM must determine which actual method to invoke when given an abstract function, and this incurs some performance or time penalty. Again an unpredictable aspect is introduced, making real-time development unsuitable.

2.2.3.5 Exception Handling

Exception handling too imposes a significant performance penalty. It is not possible to predict the amount of time that elapses between when an exception was first raised and when it reaches the handler.

2.2.3.6 Distributed Systems

The ability to easily implement a distributed system is one of the promises of Java. Yet maintaining a meaningful temporal relationship amongst distributed nodes is a challenge.

2.2.3.7 JVM Execution Speed

Java as a language which compiles byte codes which are then interpreted is inherently slower than its alternatives [17]. JVM's are said to be 10 to 20 times slower than C code [15]. There have however been many attempts to increase the speed of Java. Examples include the development of both native and Just In Time compilers [12, 15].

2.2.4 Benefits of Java for Real-time Systems Development

Java has proven itself as an excellent vehicle for the development of portable, robust, object-oriented software. Recent efforts have tried to make Java more palatable to real-time system developers. Corsaro and Schmidt state in [19] that Java has become an attractive choice for developing real-time systems for the following reasons:

- It has a large and rapidly growing programmer base and is taught in many universities
- It has a virtual machine architecture - the Java Virtual Machine (JVM) - that allows Java applications to run on any platform that supports a JVM
- It has a powerful, portable standard library that can reduce programming time and costs
- It offloads many tedious and error-prone programming details, particularly memory management, from developers into the language runtime system
- It has desirable language features, such as strong typing, dynamic class loading, and reflection/introspection
- It defines portable support fro concurrency and synchronization

In [23], the Requirements Group for Real-time Extensions to the Java Platform mentions the following additional reasons as motivation for the use of Java by the real-time community:

- It has a higher level of abstraction that allows for increased programmer productivity
- It is relatively simpler than C++
- It is relatively secure, keeping software components (including the JVM itself) protected from one another
- It is designed to support component integration and reuse

2.2.5 Real-time Java development forces

Encouraged by the benefits Java could bring to real-time systems development, there are several groups of industrial and academic organisations conducting ongoing work for transforming Java into a suitable real-time programming language. These are described below:

2.2.5.1 Real-time Java Experts Group

The Real-time Java Experts Group (RTJEG) is divided into a core team of consisting of expert developers, of which are actively involved in the process of creating the Java technology from the start and an advisory team. Chartered by the Java Community Process (JCP) [24], RTJEG released the Real-time Specification for Java (RTSJ) in 2000. RTJEG are undoubtedly the most influential group involved.

2.2.5.2 Requirements Group for Real-time Extensions to the Java Platform

The National Institute of Standards and Technology (NIST) [25] sponsor the Requirements group which consists of representatives from the computing industry and academia. During a series of workshops, the group produced a basic requirements document for RTJava. Much of the further development work is based on this document.

2.2.5.3 J-Consortium

The J-Consortium [24] has tried to offer the community an alternative to the dominance of the RTJEG. J-Consortium have been unsatisfied with RTSJ, has provided criticism and proposed to create a specification of their own. Currently the J-Consortium has come up with several ideas, but few of these have been realized.

2.2.6 Real-time specifications for Java

As stated above, conventional Java implementations are unsuitable for developing real-time systems. To address these problems the RTJEG has defined the Real-time specifications for Java (RTSJ) [8]. [8, 19, 26] all document some of the capabilities introduced by the RTSJ which enable real-time system development:

- New memory management models that can be used in lieu of garbage collection.
- Access to raw physical memory
- A higher resolution time granularity suitable for real-time systems
- Stronger guarantees on thread semantics when compared to conventional Java

In general, the RTSJ is a specification that aims to improve Java language's capabilities towards meeting the requirements set by the real-time domain. RTSJ extends the Java API and refines semantics of certain constructs to support the development of real-time systems.

In order to make Java into an acceptable real-time system programming language, the Java constructs that may result in any inherent determinism needed to be addressed. Consequently, to address the non-deterministic nature of conventional Java, the RTSJ extended the current Java Application Programming Interface (API), in seven main areas [8, 21, 27]. It is the enhancements and introduction of new constructs in these areas that introduce level of determinism that makes it acceptable for real-time systems development. Java, with the addition of these real-time constructs as specified by the RTSJ is known as Real-time Java (RTJava). These seven areas specified by the RTSJ are described below:

1. Thread scheduling and dispatching – adds real-time threads that have scheduling attributes that are more carefully defined than the scheduling for ordinary Java threads.
2. Memory management – adds tools and mechanisms that help programmers write Java code that does not need garbage collection.
3. Synchronization and resource sharing – adds synchronization properties suitable for real-time threads.
4. Asynchronous event handling – adds asynchronous event handler class and mechanisms that associates asynchronous events with happenings outside the JVM.
5. Asynchronous transfer of control – adds a mechanism that lets a thread change control flow in another thread. It is essentially a carefully controlled way for one thread to throw an exception into another thread.
6. Asynchronous thread termination – adds mechanisms that lets threads terminate in a safe and controlled manner.
7. Physical memory access – adds mechanisms that let the programmer control where objects will be allocated in memory and to access memory at particular addresses.

2.2.7 Seven Enhanced Real-time Java Areas

The seven areas are discussed in approximate order of their relevance to real-time programming. However, the semantics and mechanisms of each of the areas are all crucial to the acceptance of the RTSJ as a viable real-time development platform.

2.2.7.1 Thread Scheduling and Dispatching

The RTSJ adds real-time threads that have scheduling attributes that are more carefully defined than that for ordinary Java threads. Regardless of the scheduling type used (priority scheduling, periodic scheduling, or deadline scheduling), the way tasks are scheduled on the processor is central to real-time system development. Non real-time environments are casual about the details of scheduling, but real-time environments must be much more precise. With temporal constraints required in real-time systems, stronger semantics for scheduling are necessary. One of the concerns of real-time programming is to ensure the timely or predictable execution of sequences of machine instructions.

Real-time threads introduced by the RTSJ can be used where a traditional thread is required. The two are similar but many features of the RTSJ are only available from real-time threads including: extended priorities, scoped memory, asynchronously interrupted exceptions, and periodic scheduling.

To support these additional features, RTSJ introduces the concept of a *schedulable object* via the `Scheduler` interface. The RTSJ includes three additional classes that are schedulable objects; `RealtimeThread`, `NoHeapRealtimeThread` and `AsyncEventHandler` [8]. The `RealtimeThread` class can be used to create threads that *can* be interrupted by garbage collection activities at any time and for unpredictable lengths of time. `NoHeapRealtimeThread` can be used for time critical tasks. This form of thread cannot access the heap, and therefore cannot be interrupted by the garbage collector. They execute asynchronously with the garbage collector. In fact they may execute with or suspend the garbage collector at any time.

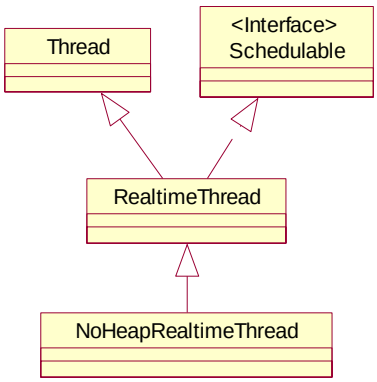

FIGURE 2: RTSJ REAL-TIME THREAD CLASS HIERARCHY

It is important to note that in the real-time domain, there must be a *timely execution of threads* [8, 17]. This means that the programmer can determine, by analysis and/or testing of the program, whether (particular) threads will always complete execution before a given timeliness constraint. This in effect is

the essence of real-time programming; the addition of temporal constraints to the correctness conditions for computation [14]. The additional threads introduced to the Java API under this area enable this *timely execution*.

2.2.7.2 Memory Management

One of the most valuable benefits of Java is its automatic memory management. However automatic garbage collection can pose a serious problem in real-time programming, because this process could cause unpredictable latencies. Thus the RTSJ have added tools and mechanisms that help programmers write Java code that does not need the garbage collection.

To provide Java with the determinism that is required for real-time programming, the RTSJ introduces the concept of memory areas. A memory area represents an area of memory that may be used for the allocation of objects. When a thread creates a new object, it is allocated in the default memory area unless the thread explicitly specifies that the object should be allocated in another memory area. There are four types of memory areas; *physical memory*, *immortal memory*, *scoped memory*, and *heap memory* [8].

Physical memory allows objects to be created within a specific range of physical addresses that have particular important characteristics, such as memory that has substantially faster access. Objects created in *immortal memory* persist until the JVM terminates. Immortal memory is commonly used on hard real-time systems, as they are never subject to garbage collection or movement. *Scoped memory* areas associate memory with an execution scope. A memory scope is used to give bounds to the lifetime of any objects allocated to it. Objects in scoped memory are not garbage collected, but are destroyed immediately when a particular code region comes to an end. The primary issue with scoped memory is to ensure that their use does not create dangling references (references to objects allocated in scoped memory that have been de-allocated) [28]. Multiple threads can share a scoped memory area, and the area will remain active until the `run` method of all threads return. After the thread exits the scope, it can no longer access objects allocated there and the JVM is free to recover memory used there. Scopes can also be nested. Both immortal and scoped memory areas can be assigned to physical memory address ranges. *Heap memory* represents an area of memory that is the heap. The RTSJ does not affect objects that are in heap memory.

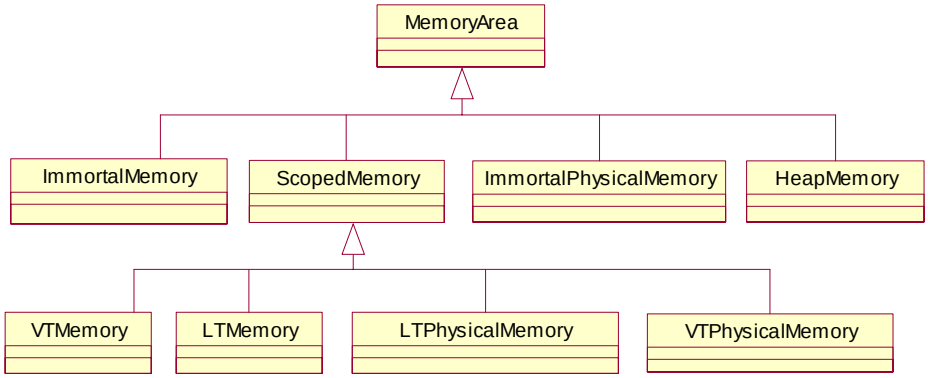

FIGURE 3: RTSJ MEMORY MODEL CLASS HIERARCHY

With the extension of the Java API to include these memory classes, the programmer can allocate objects to non garbage collected areas. This allows program logic to allocate objects in a Java like style, ignore the reclamation of those objects, and not incur the latency of any implemented garbage collection algorithm which may be critical to real-time constraints.

2.2.7.3 Synchronization and Resource Sharing

In real-time systems, the timely execution of threads is an essential concept. With precise temporal constraints associated with real-time systems, blocking of thread execution caused by synchronization will pose problems, as the blocking period may produce unpredictable timing measures. Thus the RTSJ provides mechanisms in synchronization in order to retain the timeliness property (Obviously if real-time systems do not use sharing of resource, then the synchronization is not of relevance).

In RTSJ, the concept of *highest priority thread* mostly indicates what the next thread to be chosen by the dispatcher for execution should be [17]. In the case where threads waiting to access a resource have equal priority, they are queued using the first-in, first-out (FIFO) principle.

In addition to the synchronization construct, the RTSJ introduces synchronization without the need for locking. This is done through the concept of wait-free queues. The idea with such queues is to allow wait-free communication between real-time threads and normal Java threads. The problem is that synchronization access objects shared between real-time threads and threads might cause the real-time threads to incur delays due to the execution of the garbage collector. Wait-free queues enables that at least one end of the queue will never cause a thread using it to wait for garbage collection. These queues can be either classified as wait-free write queues or wait-free read queues.

2.2.7.4 Asynchronous Event Handling

Many real-time systems are event drive. They typically interact closely with the (asynchronous) real world where things happen and the system responds to. Therefore a meaningful real-time system must be able to interact with asynchronous events in a timely and predictable manner. The time between an event and the service of the event is overhead on real-time responsiveness. Thus RTJava includes efficient mechanisms for accommodating this inherent asynchrony. Asynchronous event handlers are the solution to utilise the benefits of creating threads to service events without taking a performance penalty.

To handle such *happenings* (external events), the RTSJ provides the `AsyncEvent` class to represent each unexpected asynchronous event. Each `AsyncEvent` object is associated with an arbitrary number of `AsyncEventHandler` objects. Theses are *schedulable objects* (similar to threads) that include parameters for controlling scheduling and memory. When an asynchronous event occurs, the associated `AsyncEvent` is fired (with `AsyncEvent.fire()` command), which in turn starts the associated `AsyncEventHandler` object(s).

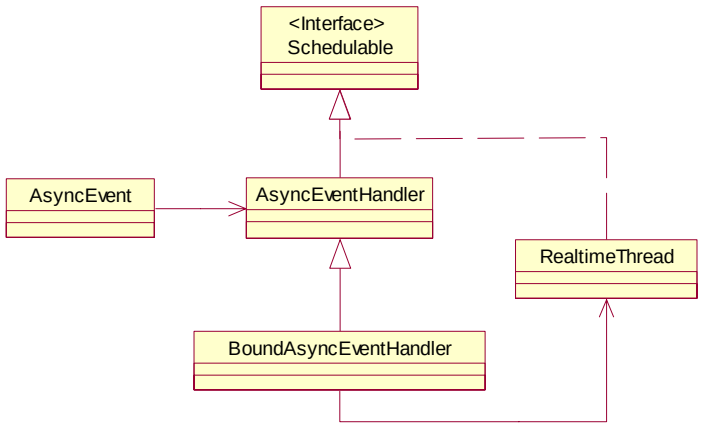

FIGURE 4: RTSJ ASYNCHRONOUS EVENT CLASS HIERARCHY

2.2.7.5 Asynchronous Transfer of Control

In the real-time domain there may be instances where it is necessary or logical to immediately and efficiently transfer the current point of execution to another location. Consequently the RTSJ includes a mechanism, which allows programs to programmatically change the focus of control to another Java thread. It enables a thread throw an exception into another thread. Standard Java includes a similar mechanism, `thread.interrupt()`, but this is weak for real-time requirements. Thus RTJava defines an `AsynchronouslyInterruptedException` class, for asynchronous exceptions. This class has special rules that make the exceptions safe for use, allowing threads to be interrupted in a controlled manner.

2.2.7.6 Asynchronous Thread Termination

RTSJ also includes functionality to handle asynchronous termination of threads that may occur due to unexpected conditions or events. Earlier versions of Java provided mechanisms such as `stop()` (that has been deprecated) and `destroy()` in class `Thread`. However these are unsuitable for the real-time domain due to their inconsistent results and susceptibility to deadlock.

The RTSJ avoids the dangers of `stop()` and `destroy()` through a combination of the `interrupt` method on real-time thread objects and the asynchronous transfer of control techniques. These can be used to notify a thread to terminate itself safely and within real-time constraints.

2.2.7.7 Physical Memory Access

Special types of memory are closely related to performance such as slow memory, cached memory, etc. Whereas performance is not a real-time issue, predictable performance is and some memory attributes have a large impact on the predictability of code that uses the memory. An application can get improved performance by putting the most used data in fast memory. For these reasons, the RTSJ includes the physical memory classes. Theses let it store objects in the right kind of memory.

RTSJ defines classes that allow programmers byte-level access to physical memory and construction of objects in physical memory from code. Physical memory objects are `ScopedPhysicalMemory` (which consists of `VTPhysicalMemory` and `LTPhysicalMemory`), `ImmortalPhysicalMemory`, `RawMemoryAccess`, and `RawMemoryFloatAccess`. Some of these memory classes are depicted in the RTSJ Memory Model Class Hierarchy in figure 3. The introduction of these classes provides pointer-like objects and direct access to physical memory that greatly increases the usefulness of Java for the real-time community.

2.2.8 Current RTSJ Implementations

Until recently there were no implementations of the RTSJ, which hampered the adoption of Java in real-time systems. It also hampered the analysis and evaluation of the benefits and drawbacks of the RTSJ programming model. Several implementations of the RTSJ are now available.

2.2.8.1 TimeSys RTSJ Reference Implementation

TimeSys has developed the *official* RTSJ Reference Implementation (RI) [10], which is a fully compliant implementation of Java that implements all mandatory features of RTSJ. The RI provides a provides a proof of implementation of the RTSJ along with a virtual machine required to test a Java-technology based application against the real-time specification. The RI runs on all Linux platforms, but the synchronization control mechanisms are available to the RI only when running under TimeSys Linux/RT. The figure below shows the architecture of the TimeSys platform. The real-time Java

application contains the bytecode that is interpreted by the JVM, which in turn was written for a given host system.

Note that the implementation carried out in this dissertation uses the TimeSys Linux/RT platform and the TimeSys RTSJ RI available at [11].

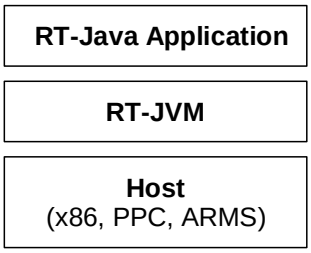

FIGURE 5: TIMESYS REAL-TIME JVM ARCHITECTURE

2.2.8.2 jRate

jRate is an open-source RTSJ-based extension of the GNU Java Compiler (GCJ) runtime systems that were developed at the University of California. jRate adds support for most of the features required by the RTSJ. The jRate project is discussed in greater detail in the Related Work section (section 2.5) of this chapter.

2.2.8.3 Real-time Java for Flex

The Program Analysis and Compilation group at MIT created the Flex Java Compiler infrastructure, along with the real-time Java for Flex [29]. This implementation differs from the TimeSys RI in a number of aspects; the primary one being the use of an Earliest Deadline First scheduling algorithm instead of the priority based pre-emptive scheduler.

2.3 Aspect-oriented Software Development

Aspect-oriented Software Development (AOSD) has recently been proposed as a new paradigm for the separation of concerns in software development. Separation of concerns is considered important to manage complexity and increase adaptability in a program and has become a vital software engineering principle [1]. Researchers in AOSD are largely driven by this fundamental goal and the techniques offered by AOSD make it possible to modularise crosscutting concerns within a software system.

Currently object-oriented programming (OOP) serves as the methodology of choice for most new software development projects. However, while many of the OOP techniques are useful, they are inherently unable to modularise all concerns in a complex system [2]. A typical system consists of several concerns. The limitation of the OOP approach is its inability to localise concerns that do not

naturally fit into a single program module, or even several closely related modules. Such concerns are defined *crosscutting concerns* as they cross-cut or span multiple (often unrelated) implementation modules [30].

OOP code suffers from two phenomena's (resulting from *misalignment* between requirements and code); t*angling* and s*cattering* [2, 3]. Tangling occurs when multiple concerns are addressed in a single module (or class), making the module harder to understand and maintain. Scattering results when the implementation of a concern is spread over multiple modules, leading to the risk of inconsistencies at each point of use. [3, 30] both describe in detail how code tangling and scattering can adversely effect software development.

AOSD puts a greater focus on crosscutting concerns that that of OOP and other language paradigms. It provides language mechanisms that explicitly capture crosscutting concerns in a modular way and thus achieving the benefits that results from improved modularity; code that is easier to design, implement, maintain, reuse and evolve. AOSD introduces the notion of an *aspect*, a well-modularised crosscutting concern, and shows how we can take crosscutting concerns out of modules and place them in a centralised location [31]. In most cases, aspects tend to be implementations of non-functional properties that are absent from the initial requirements document [6]. Common examples are security, synchronisation, and tracing.

There are currently several approaches to AOSD that have been adopted for dealing with crosscutting concerns. To date, the most widely used approach is the *Aspect* approach or aspect-oriented programming (AOP) [2, 30].

2.3.1 The Aspect Approach

The Aspect approach known more commonly as aspect-oriented programming (AOP) was first introduced in [2]. The problem that AOP aims to provides mechanisms that all programmers to handle concerns which cannot be constrained easily into modular form. Such mechanisms allows for each concern to be addresses separately with minimal coupling, and thus facilitating the programmer in achieving the separation of concerns goal with less duplicated code that is easier to understand and maintain and more reusable. AOP involves three different development stages:

- Aspectual decomposition decomposes the requirements to identify crosscutting concerns
- Concern implementation implements each concern separately
- Aspectual recomposition specifies the recomposition rules and uses this information to compose the final system

AOP introduces two new elements to the set of standards tools used in the development of a software system; *aspect language* and *aspect weaver*. The aspect language is the language used solely used to program the aspects and differs from the *component language* which programs the components. The function of the aspect weaver is to weave the aspects and the component code together. The output is a combination of the two languages which is the executable program. At present AspectJ is the most predominant language supporting the AOP approach to AOSD.

2.3.1.1 AspectJ

AspectJ is an aspect-oriented extension to Java. Its language specification defines various constructs and their semantics that support aspect-oriented concepts [32]. The constructs introduced by AspectJ enable the support for modular implementations of crosscutting concerns. Because AspectJ is an extension of Java, every valid Java program is also a valid Aspect program. The AspectJ compiler produces class files that comply with Java byte code specification, enabling any compliant JVM to interpret the produces class files. Figure 6 taken from [30] shows the process outlined above.

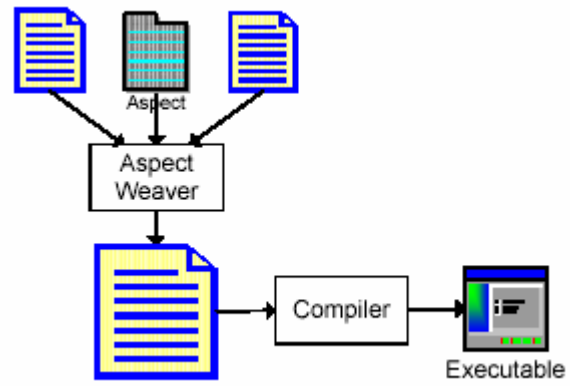

FIGURE 6: ASPECT WEAVER

[31] documents two kinds of crosscutting implementations that AspectJ supports; *dynamic crosscutting* and *static crosscutting*. The first allows the definition of additional implementation to run at certain well-defined points in the execution of the program, while the second defines new operations on existing types.

AspectJ Language Constructs

AspectJ adds a number of additional concepts to the Java language; *joinpoints, pointcuts, advices and aspects*.

A critical element of AspectJ is the **joinpoint model**. The joinpoint model provides a common frame of reference that makes it possible to define the dynamic structure of crosscutting concerns. Joinpoints are

well-defined points in the execution of the program such as method calls, a conditional check or and assignment. They are predefined and also have a context associated with them, meaning that it is necessary to specify what joinpoint is to take action when defining pointcuts.

Pointcuts (or pointcut designators) define arbitrarily large sets of points in the execution of a program. Essentially they are a predicate defined on a set of joinpoints. They let you refer a joinpoint (collection) and certain values at those joinpoints. [3] identifies a number of different types of pointcuts. They are field access pointcuts, exception handler pointcuts, class initialization pointcuts, control flow based pointcuts , self argument type pointcuts, conditional test pointcuts, and named and anonymous pointcuts.

Advice is code, that implements additional behaviour, which executes when a joinpoint is reached. AspectJ provides three different advices; *before advice, after advice,* and *around advice.* Before and after advices execute before and after method(s) specified in a pointcut, while around advice executes in place of the method(s) specified in a pointcut (i.e. the implementation in the around advice replaces that of the specified method).

The final additional construct is the **aspect**. Aspects are AspectJ's units of modularisation with crosscutting implementation provided in terms of joinpoints, pointcuts, and advices. The definition of an aspect is very similar to the definition for a class. They act as AspectJ's unit of modularisation. In addition to all the features that a class can possess (data, operations, be instantiated and inherited from etc.), aspects also have the ability to enhance the behaviour of other classes through weaving. So while classes contain variable and methods, aspects contain variables, methods and weaves.

Currently AspectJ offers good integration with popular IDEs (Integrated Development Environment) such as Eclipse, Forte, JBuilder, Emacs, and NetBeans [33].

AspectJ Critique

[6] outlines some critique of AspectJ. The areas are *dynamic weaving, protection mechanisms, abstract methods, exception handling,* and *orthogonality*. The documented reason that for these issues is that AspectJ is a relatively new language, and that these criticisms have only arisen because developers have been given more power and to let the user community experiment and discover what is right.

Dynamic weaving is a powerful mechanism that some feel resemble self-modifying code. This follows from the property that method calls in AspectJ includes advice that is associated with a method call. If a program can change its advice patterns and executing a method with this tailored advice, then it would seem that the program has succeeded in modifying and subsequently executing this altered code. However, in reality, this is not the case. AspectJ's dynamic weaving does not actually involve modifying code. Instead it results from dynamic decisions regarding where to place code for execution.

Protection mechanisms are an issue that is raised because aspects cut across classes. And because of this, the part of the class that cross cuts a class can be considered as part of the class itself. This gives rise to

the problem that if it is part of the class, then the aspect should have full access to all the class data members. This in affect violates information hiding and encapsulation.

At present in AspectJ either a subclass, or an aspect that advices a subclass may handle an abstract method defined in a superclass. Therefore, if an abstract method is handled by an aspect, then the class will become dependant on that aspect. In practice, an aspect should by its nature have no dependant classes, meaning that abstract methods should always be implemented by a subclass.

Exception handling is a vital part of any programming paradigm, and often creates many issues in programming language design due to their high coupling with other language features. Yet, there is very little documented about AspectJ in relation to exception handling.

The functions provided by aspects and classes is said to violate the principle of orthogonality [6]. Using AspectJ, a programmer can create an aspect that has identical behaviour to that of a class. What results is overlapping between constructs and reduced performance.

2.3.1.2 Other Aspect Languages and Tools

This section outlines some of the other languages and tools that are available for the AOP approach to AOSD.

Aspect Languages

The table below details the aspect languages that are currently available in addition to Java and AspectJ as detailed in [4].

Aspect Language	Definition
AspectC++/C	AO extension of C++ and C respectively
AspectC#	AO extension of C# (of Microsoft.Net Framework)
Aspect.pm	Set of Perl libraries for AOP
AspectR	AO concepts for Ruby
AspectS	AO extension of Squeak/Smalltalk
Pythius	AO construct for Python
Caesar	AOP language focused on multi-view decomposition and aspect reusability
Apostle	AspectJ like extension to Smalltalk
ArchJava	AO extension of Java with explicit software architecture constructs
EAOP	AOP for Java with expressive crosscuts, explicit aspect composition, and dynamic aspect management

Table 1: Aspect Languages

Aspect Tools

The table below details the aspects tools that are currently available as described in [4].

Tool	Functions
JAC (Java Aspect Components)	Java framework for distributed for developing AOP focusing mainly on developing and AO middleware layer
JMangler	A framework that allows for load-time transformation of Java programs
JMunger	A tool that makes it possible to insert Java code into existing classes at load time
PROSE (Programmable Service Extensions)	AOP platform based on the Java Virtual Machine that allows dynamic weaving and un-weaving
UMLAUT	A framework for building application specific weavers to weave multi-dimensional high-level UML design models into detailed design models suitable for implementation, simulation or validation
Aspect Browser	A graphical software tool that uses the map metaphor to assist finding, exploring, and manipulating crosscutting concerns in software
Aspect Mining Tool (AMT)	A tool that facilitates the identification and exploration of concerns in a software system
Noah	A tool for AOP based on software interaction based on ISL (Interaction Specification Language)
FEAT (Feature Analysis and Exploration Tool)	A tool similar to AMT. It is developed as an Eclipse IDE plug-in and is used for locating, describing, and analysing implementations of concerns in Java.
Concern Graphs	A tool that provides a representation of concerns in system code that can be extracted from an existing codebase
Weave.NET	This tool aims to provide a language independent mechanism for supporting AOP that is also consistent with the .NET Framework component model

Table 2: Aspect Tools

2.3.2 Other AOSD Approaches

This section details some alternative approaches that currently exist for AOSD.

2.3.2.1 Hyperspaces

Hyperspaces were designed to enable multi dimensional separation of concerns (MDSOC) [34]. MDSOC is defined as refer to flexible and incremental separation, modularization, and integration of software artefacts based on any number of concerns [1]. With this AOSD approach, concerns are grouped into

dimensions which represent a set of disjoints (i.e. concerns that have no units in common). Such dimensions enable the explicit identification of any concerns and management of relationships among concerns. The main advantage of hyperspaces is that they are designed to be utilised in all phases of the software lifecycle, thus promoting traceability between phases, and limiting the impact of change throughout. HyperJ is the tool that implements hyperspaces [34].

2.3.2.2 Compositional Filters

The Compositional Filters (CF) approach to AOSD first detailed in [35], extends the conventional OO model in a modular way. The CF model provides mechanisms for adding an open-ended range of concerns to the object model with violating their basic mechanism. This is achieved through the manipulation of incoming and outgoing messages. The most significant components are the input filters (for received messages) and output filters (for sent messages). A single filter specifies a particular manipulation of messages. A combination of filters will therefore compose the behaviour of an object. Each filter consists of a pattern definition. Messages are matched against this pattern, and the filter subsequently determines the action to be performed depending on the acceptance or rejection of this message in the filter. ComposeJ [36] and ConcernJ [37] are two existing tools that support the CF approach to AOSD.

2.3.3 Concern Manipulation Environment

The Concern Manipulation Environment (CME) is currently being developed by IBM is a flexible, extensible, interoperable environment for AOSD [38]. CME consists of a suite of tools that enable the creating, manipulating, and evolving AO software, across the full software lifecycle. It has been designed to replace the hyperspace approach and provide added support to the MDSOC.

2.4 Evaluation Metrics

Currently no metrics have been proposed or devised for the evaluation of AO system [39]. This section describes some of the OO metrics that may be applied for evaluating AO software.

2.4.1 The Chidamber and Kemerer (C&K) Metrics Suite

The C&K metrics suite is an OO metrics suite that is designed to measure the key elements of OO software: encapsulation, abstraction, and inheritance [40]. This metrics suite was the first to be based on sound measurement theory, and has been empirically validated. For this reason, the C&K suite has been chosen for the evaluation in this dissertation. The suite is described in more detail in the Evaluation Metrics chapter (chapter 5) of the dissertation.

2.4.2 The MOOD (Metrics for Object Oriented Design) Metrics

The MOOD metrics set [41] includes the following metrics; (1) Method Hiding Factor (MHF); (2) Attribute Hiding Factor (AHF); (3) Method Inheritance Factor (MIF); (4) Attribute Inheritance Factor (AIF); (5) Polymorphism Factor (PF); (6) Coupling Factor (CF). Each of those metrics refer to a basic structural mechanism of the object-oriented paradigm such as encapsulation (MHF and AHF), inheritance (MIF and AIF), polymorphism (PF) , message passing (CF) and are expressed as quotients. The exact definition of each is documented in [41].

The mood metrics have not been chosen for the evaluation carried. [41] documents some problems regarding the mood set. The main drawbacks being its imprecise definitions and unlike C&K metrics suite, the mood set have yet to be validated empirically.

2.4.3 Morris

In his Master's thesis [42], Morris proposed a set of metrics for productivity measurement. The metrics included; (1) Methods per class; (2) Inheritance Dependencies; (3) Degree of Coupling between Objects; (4) Degree of Cohesion of Objects; (5) Object Library Effectiveness; (6) Factoring Effectiveness; (7) Degree of Reuse of Inheritance Methods; (8) Average Method Complexity; (9) Application Granularity. The exact definitions of each are detailed in [42].

Although some of the metrics are similarly named to those of the C&K suite, their definitions differ slightly. The focus of the Morris metrics is on productivity only and ignores other attributes such as maintainability and testability which are important elements required in my evaluation. As with the MOOD metrics, those proposed by Morris have not been empirically validated.

2.5 Related Research/Work

This section documents some of the ongoing work that is related to the adoption AOSD in the real-time domain using RTJava. Also described are several real-time applications that under development using AOSD.

2.5.1 Translation of Java to Real-time Java Using Aspects

Memory management and scoped memory was one of the areas introduced by the RTSJ which allows programmers to allocate memory to objects that will not be garbage collected. [43] documents the use of aspects and AspectJ to transform a Java program to a RTJava program that satisfies real-time constraints. The research focuses on the area of memory management and the use of aspects for dealing with scoped memory.

The main goal of the research is to define an automatic way for translating Java code into scope aware RTJava code (i.e. automating the creation of RTSJ memory scopes). To achieve this, the concept of probing aspects is introduced. A *reference-probing* aspect is used to determine thee legal RTSJ `ScopedMemory` assignments. It builds a *doesReference graph* to track which objects refer to other objects, with object instantiations and assignments used as join points. In turn a *liveness-probing aspect* is defined to determine the points at which objects are dead or unreachable and thus flushed from memory.

This study showed that by using AOP techniques for dealing with the scoped memory area addressed by RTSJ, many benefits associated with AOP are gained, including more modular code, lower development costs, and code that is easier to understand and maintain. From a real-time perspective, two additional benefits are attained; having aspect code in the target program will introduce real-time predictability; and user source files are unchanged as separate aspect source code will describe any real-time modifications. Thus this study provided a good basis for the work when implementing the memory management area in the Simulator. It also indicated that aspects were likely to have some impact (whether good or bad) when used in the development of the RTJava areas.

2.5.2 On Aspect-Orientation in Distributed Real-time Dependable Systems

The design and implementation of real-time systems usually involves the consideration for many non-functional properties. [44] documents how AOP can be used to address these non-functional issues and what the benefits are. The main issues addressed are distribution, real-time, and fault tolerance, with aspects for each developed using AspectC++.

The research introduces a real-time aspect to handle temporal constraints posed by the real-time domain. The idea of a *watchdog* (per-thread watchdog timer) is proposed to monitor and regulate the execution of component code, ensuring that it is processed within the required time boundaries.

An interesting finding of this study indicated that aspects could be useful in improving real-time systems performance predictability. Without AOP, the watchdog code would have to be inserted manually into the component code wherever necessary. In addition to code scattering, this may introduce an unnecessary overhead if the component is reused outside the real-time domain. This problem is exasperated if the component is only subject to time constraints when called from specific locations in the application.

2.5.3 jRate

jRate is currently under development at the University of California, Irvine (UCI). jRate is an open source a-head-of-time compiled implementation of the RTSJ that is developed using AOSD techniques.

[19] describes its design and performance evaluation as compared to other RTJava implementations that were developed using traditional programming approaches, mainly OO.

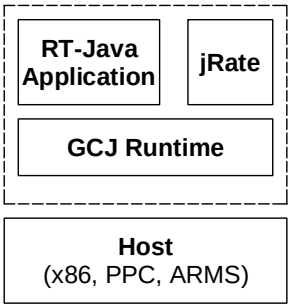

FIGURE 7: JRATE ARCHITECTURE

The goal of the project was to make jRate a generative RTSJ implementations, thus adopting AspectJ and AspectC++ during development. The advantage of this according to Corsaro and Schmidt was that these aspect languages will be better capable of dealing with static and dynamic crosscutting properties of the implementation. Within the implementation, aspect-oriented constructs are used for the implementation of several RTSJ areas such as memory areas, real-time threads and scheduling, and asynchrony.

The performance evaluation presented in the paper is based on the RTJPerf benchmarking suite developed by Corsaro and Schmidt for RTSJ [19]. The results show that jRate is both more efficient and predictable than the RTSJ reference implementation. The main problem with jRate is that at present it does not implement many core RTSJ features. Future work associated with jRate is to complete the RTSJ implementation, with the long term goal to create a set of components that allow users to generate custom RTJava implementations tailored for particular requirements and environments. And according to the authors this will most likely be done using the AOSD approach. Due to these limitations, this dissertation did not adopt the jRate implementation of the RTSJ, but instead used the TimeSys RI (described above).

2.5.4 AIRES

The AIRES (Aspects in Real-time Embedded Systems) project being carried out by BBN Technologies [45]. A major finding of this project is the identification of some of the areas where AOSD were limited in its support for developing distributed real-time embedded (DRE) systems. One reason for this is the immaturity of AOSD, while others are due to the specific characteristics of DRE applications.

The aim of the AIRES project is to advance the state of the art of AOSD by addressing the above limitations. To achieve this, certain parts of the AOSD technology will be investigated and developed. At

present, the project is at the stage of developing Aspect Suites for Embedded Systems with several innovative features. Such features include; a unified approach to AOSD that will address both functional and systemic aspects; investigation of the composition, and conflicts between aspects in regard to real-time facets, in particular timing and synchronisation; and research development of DRE application using AOSD, including static analysis and design of composed aspects and more advance weavers for dealing with temporal constraints.

2.5.5 FACET

FACET or Framework for Aspect Composition for an Event Channel [46] is a project aimed at investigating the development of customisable middleware using AOSD. The main focus of the project has been the development of a real-time event channel in Java using AspectJ. The purpose and benefits derived from using aspects is to weave in features, namely real-time specific features, required for the target application. The advantages given are those that are resulting from AOSD, including better modularisation and separation of concerns, code that is easier to understand and maintain, and simpler and more flexible feature addition.

Currently FACET has developed a based implementation that allows features to be added to target applications by weaving aspects. Although not developed using RTJava, the real-time event features have been developed using AspectJ, highlighting the benefits of using AOSD for real-time system development.

2.5.6 Real-time Filters

In real-time environments, at least some of the classes in the system impose temporal constraints upon the execution of methods. When such classes are reused in other applications, change to either the application requirements or to the real-time constraint specifications in subclasses may result in excessive redefinitions of superclasses whereas this would be intuitively unnecessary. [47] refers to this as a *real-time specification anomaly*, which arises when real-time specifications are mixed with application code or when independently define but related specifications are composed together. As a solution to these anomalies, the Compositional Filters approach to AOSD is used. With it, the concept of a real-time composition filter is introduced. Such filters can be used to affect real-time characteristics of messages that are received or sent by an object. Proper configuration of these filters allows real-time constraints to be specified and reused, resulting in decoupling that ensure that the aforementioned anomalies will not occur. Unlike this dissertation, this study does not adopt the aspect approach, but it further highlights a growing interest in real-time systems development using AOSD.

2.5.7 An Initial Assessment of Aspect-Oriented Programming

[48] documents a number of experiments that were carried out to assess the capabilities of AOP. In the experiments, participants were divided into two groups, one working with AspectJ, the other working with an object-oriented control language. The performance and experiences of the participants were then compared while carrying out two common programming tasks; debugging and change.

The first experiment, debugging, highlights that the AspectJ participants were able to finish the tasks faster that the participants using Java. The second experiment, change, contrasts the first with the AspectJ participants requiring more time to complete tasks than the participants using the non-aspect oriented control language Emerald.

The research raises two interesting insights into AOP. Firstly, programmers may be better able to understand an aspect-oriented program when the effect of the aspect code has a well defined scope. Secondly the strategies programmers use to address tasks perceived to be related to aspect code may be altered due the presence of that aspect code. The first insight supports the findings that were established during the evaluation of the Simulator. Although there were cases in which aspects eased system understandability, there were also instances where the understanding of the application was more difficult due to a wider scope that was caused as a result of aspect code. These findings are discussed in more detail in the Evaluation chapter of the dissertation (chapter 6).

2.5.8 Challenges of Aspect-Oriented Technology

Alexander et al. in [49] describe the research they conducted into the costs and benefits associated with AOSD in terms of its impact on software engineering. The research aims to understand both the strengths and limitations of AOSD and to raise awareness of the potential negative side effects of its use. Some of the criteria used for this analysis are; understandability, emergent properties and fault resolution, implicit changes in syntactic structure and semantics, and effects on cognitive burden. Thus this research highlights a number of important points that need to be considered when developing application using AOSD.

3 Object-oriented Design and Implementation

This chapter describes the design and implementation of the Sentient Traffic Simulator both prior to any alterations to include RTJava or AOP related constructs and also inclusive of RTJava modifications. In the case of the latter, each of the seven enhanced areas are detailed along with the affects that they have on the original implementation.

3.1 Background

The original design and implementation of the Simulator, prior to any alterations to include Real-time Java or AOP related constructs, had three distinct goals related to sentience and intelligence [9].These goals are:

- To design an accurate as accurate a model for a vehicle as possible
- To accurately reflect the sentient properties of a vehicle and the data obtained from sensors
- To accurately reflect the communication between vehicles through the use of sensors

However, despite the hard real-time properties of the Simulator, real-time compliance was not a goal of the original implementation.

The Simulator itself consisted of a number of vehicles, either normal cars or emergency vehicles, driving on a four lane road. The objective for each vehicle is to reach its predefined desired speed (which differs for each). In order for each vehicle to reach its desired speed, sensor information is sent and read by all vehicles in the network. Through the interpretation of this data, vehicles can paint an accurate picture of the entire road network and all its components. Thus a particular vehicle can “sense” the speed and locations of other vehicles and hence react to its surroundings. For example a vehicle must reduce its speed if it senses a slower vehicle ahead in order to avoid collisions. Another example is if a car senses an emergency vehicle approaching, it must change lanes to let it easily pass by.

The original design of the Simulator is shown in figure 8. The implementation (without any real-time functionality) consisted of eight classes:

- Vehicle.java – implements the properties and mechanics of real-world vehicles (such as the car length and width, its maximum speed etc.), sentience (enabling it to react intelligently to its surroundings), and communication (consisting of reading and sending of sensor information between vehicles)
- Car.java – subclass of Vehicle with specific protocols. These protocols specify the rules relating to how a car is to be operated within the sentient environment. Sample rules

include changing lanes if a slower car is in front of them, to maintain the safety margin between cars, and changing lane if an emergency vehicle is behind.

- EmergencyVehicle,java – subclass of Vehicle with specific protocols (as compared to Car)
- ViewableCar.java – contains the specific properties visible to all vehicles through sensors
- ListOfCars.java – representation of all the vehicles in the network
- Road.java – representation of the road network
- RoadPanel.java – graphical output of the program
- Simulation.java – main class

Of the eight classes, RoadPanel.java is the only class responsible for graphical functions. The remaining seven classes implement the core functionalities of the simulation such as object sentience and context based behaviour. It is these remaining seven classes that exhibit the need for real-time behaviour. Such constraints are introduced to ensure that certain functions (in these classes) meet critical temporal constraints of the real-world as ignored from the original implementation.

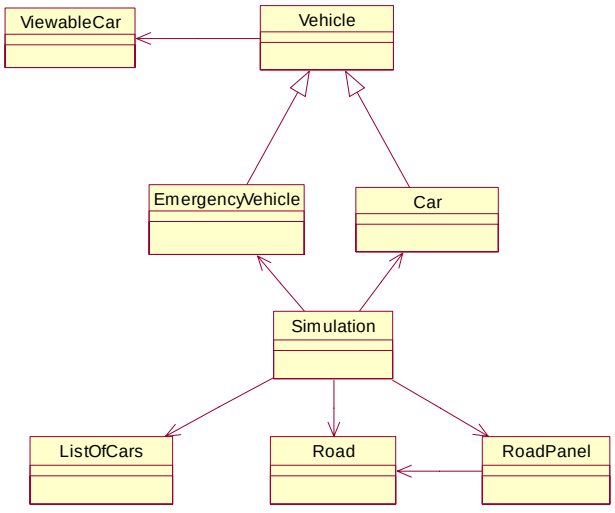

FIGURE 8: SENTIENT TRAFFIC SIMULATOR CLASS STRUCTURE

[9] gives a detailed description of the original design and implementation of the Simulator.

In contrast, figure 9 illustrates the class structure of the Simulator inclusive of RTJava constructs.

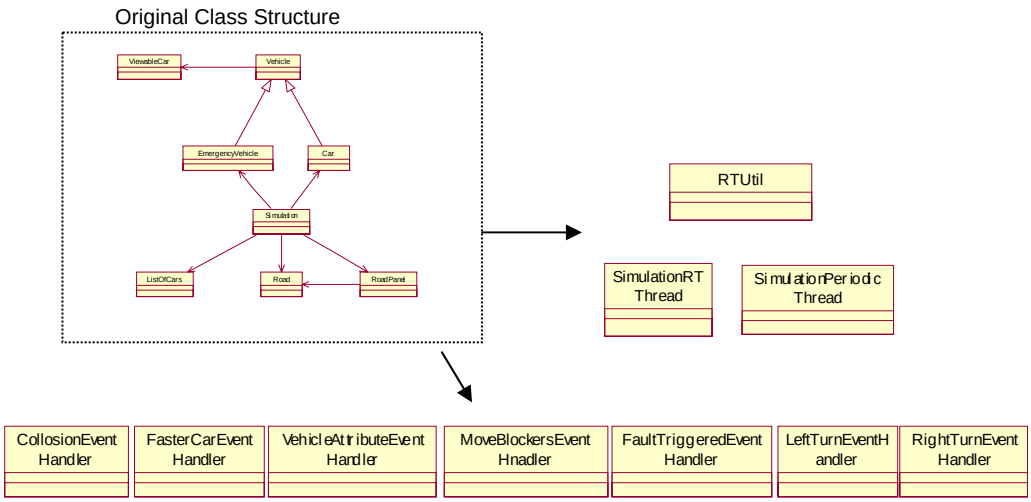

FIGURE 9: SENTIENT TRAFFIC SIMULATOR CLASS STRUCTURE WITH RTJAVA CONSTRUCTS

There are only minor changes made to the original design to enable the inclusion of the seven enhanced RTJava areas. Firstly there are the additions of a number of handler classes that are required for the handling of the different types of asynchronous events that are thrown by the Simulator in the real-time implementation). Secondly there is the use of an `RTUtil` class, which provide utility functions related RTJava constructs. And thirdly two `RealtimeThread` subclasses were implemented which provide specific real-time threading behaviour required for the Simulator.

The identification of real-time concerns was performed in two manners. Firstly existing constructs in the Simulator which require alteration to satisfy real-time constraints were determined. Functionality implemented with using threading, object creation, and synchronization is the main examples of this. Thus existing threads in the system which were originally programmed using conventional Java threads were, in the real-time version, altered to real-time threads. Similarly for objects that are created in the Simulator. The original implementation assigned all objects to the heap meaning it was possible for these objects to incur unpredictable time delays due to garbage collection activities. This of course is unacceptable in a real-time environment. Therefore objects, in the real-time version of the implementation need to be allocated to memory that is not affected by any unpredictable behaviour.

The second method for identifying real-time concerns was less trivial and required a far deeper understanding of the original implementation and its execution flow. An examination of the codebase was performed to determine which points in the program's execution required RTJava constructs. For example asynchronous event handling was used to replace previous implementation dealing with certain events. For example, when a vehicle senses that there is a slower vehicle ahead, then an event is fired to notify it to slow down and avoid collision. This previously consisted of another method call to handle

this operation. Thus RTJava constructs are used instead as they provide a more real-time correct way of dealing with such functionality. Also certain functionality were grouped together in the original implementation and thus executed in a single thread manner. This may be problematic in a real-time context with each operation incurring time delays. In the original implementation of the Simulator, the thread responsible for providing the driving operations of a vehicle was also responsible for the interpreting of sensor information of other vehicles, updating its own sensor information, and also the driving manoeuvres of the vehicle itself (i.e. change lanes, accelerate, decelerate). Thus it was necessary to introduce a single interruptible thread to handle each task above. These were the two main tasks carried out to identify where real-time behaviour and RTJava constructs were necessary to enable the Simulator to satisfy real-time constraints, with examples of each case described.

The table below shows the three main requirements as set out for the original implementation, and the RTJava alterations that are required for each to transform the Simulator into a system that is compliant with the real-time property of predictability performance.

Requirement	**RTJava Constructs Required**
Accurate Vehicle Model	Thread scheduling & dispatching, Memory management, Asynchronous event handling, Asynchronous transfer of control, Asynchronous thread termination, Physical memory access
Sentience	Memory management, Synchronization and resource sharing, Asynchronous event handling, Physical memory access.
Communication	Thread scheduling & dispatching, Memory management, Synchronization and resource sharing, Asynchronous event handling, Asynchronous transfer of control, Asynchronous thread termination, Physical memory access

Table 3: Simulator Requirements and RTJava Constructs Relationship

3.2 Thread Scheduling and Dispatching

Real-time threads differ from conventional Java threads as they possess scheduling attributes that are more carefully defined. Real-time threads differ in that they are created with a set of parameters specifying real-time attributes. The parameters are:

- Scheduling parameters – specifies the scheduler to be used and the priority given to the thread
- Release parameters – waiting period for a thread when `waitForNextPeriod` method is used
- Memory parameters – sets limits on the amount of memory a thread can use

- Memory area parameter – the memory pool used by the thread (either heap, immortal, or scoped)
- Processing group parameter – specifies the particular processing group that a thread is in. The consequence of this is the resource consumption activities of all the threads in a group cannot exceed the allocated amount for that group
- Logic parameter – refers to an instance of a class that implements the `Runnable` interface

The constructor of the `RealtimeThread`, with the six parameters, is as follows:

```
public RealtimeThread (SchedulingParameters scheduling,
      ReleaseParameters release,    MemoryParameters memory,
      MemoryArea area, ProcessingGroupParameters group,
      java.lang.Runnable logic)
```

On creation of a real-time thread, all the above parameters need to be specified reflecting the particular characteristics that the new thread is required to have. If a null value is given for any of the parameters, then the default value (as specified in the RSTJ) is used.

The Simulator requires the use of a number of threads throughout its execution flow. Threading was used in the following instances:

- To start the simulator – this is done in addition to the main thread (i.e. the main thread starts another thread)
- Each Vehicle object is run in a separate thread – this thread was responsible for a number of operations such as driving, detecting collisions, updating vehicle attributes, reading sensor information, and performing manoeuvres (turn right, turn left, slow down).

Thus in the OO real-time version of the Simulator, the above threads were transformed to real-time threads. In addition, the second thread specified above was separated into numerous threads, one for each operation.

Figure 10 highlights the classes that implemented thread scheduling and dispatching functionality. (The prominent coloured classes are those that contain the functionality)

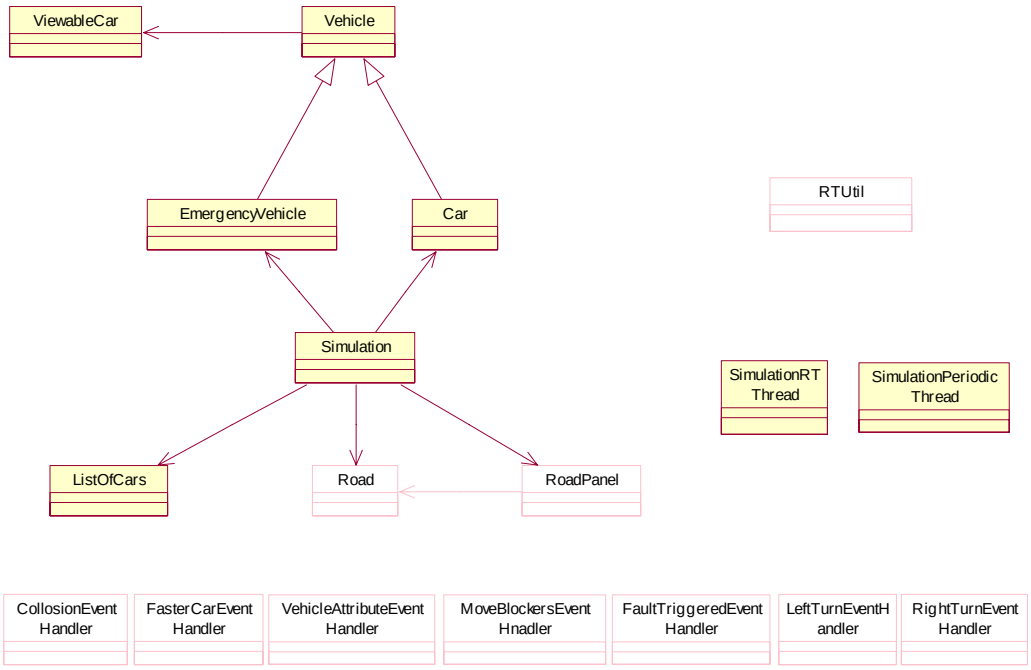

FIGURE 10: CLASSES CONTAINING THREAD SCHEDULING AND DISPATCHING FUNCTIONALITY

Two types of real-time threads were implemented in the Simulator (by specifying different constructor parameters). The first type reflects the behaviour of a normal thread, executing operations specified in its run method on the calling of the start method. The second type, known as a periodic thread, differs slightly. The distinguishing factor is that these type of thread repeats its operations (implemented in run) in iterations of a specified time. Thus calling start the thread will call its run method, wait for the specified time interval and then perform its run again.

Implementation of the above threads is done in two ways in the Simulator. The first techniques utilises subclassing of the `RealtimeThread` class. The approach is beneficial when a several real-time threads in the system require the same real-time behaviour (i.e. have the same `RealtimeThread` instantiation arguments). This would therefore enable each of threads to be more easily created. This is because the programmer will not need to create and initialise all `RealtimeThread` arguments each time a new thread is created, but instead simply call the constructor of the `RealtimeThread` subclass. The second technique differs and involves the creation and initialisation of `RealtimeThread` arguments at each point of thread instantiation. This is the problem that the subclassing technique aimed to solve. However this technique is beneficial when each real-time thread in a system requires a variation in their real-time behaviour (i.e. different values for any of the `RealtimeThread` constructor arguments). Subclassing in this instance would be of little benefit.

The first technique used subclassing for implementing real-time threads. This was possible as numerous real-time threads in the Simulator had identical behaviour (i.e. the same constructor parameters). As a result it was possible to specify the behaviour of these two different types of real-time threads used in the Simulator as subclasses of `RealtimeThread`. Creation of these threads could be done with a no argument constructor eliminating the need for all the thread parameters to be constructed and specified at the point of thread creation.

Two subclasses of `RealtimeThread` are implemented in the Simulator to reflect the two types of real-time threads used. The code for the two `RealtimeThread` subclasses, `SimulationRTThread` and `SimulationPeriodicThread`, in the Simulator are shown below:

```
public class SimulationRTThread extends RealtimeThread {
    public SimulationRTThread() {
       super((SchedulingParameters)
              (new PriorityParameters(
                     PriorityScheduler.MIN_PRIORITY + 30)),
              (ReleaseParameters) (new AperiodicParameters(
                     null, null, null, null)),
             new MemoryParameters(MemoryParameters.NO_MAX, 0),
             (MemoryArea) (ImmortalMemory.instance()), null, null);
    }
}
public class SimulationPeriodicThread extends RealtimeThread {
    public SimulationRTThread() {
       super((SchedulingParameters) (new PriorityParameters(
                     PriorityScheduler.MIN_PRIORITY + 20)),
              (ReleaseParameters) (new PeriodicParameters(
              new RelativeTime(), new RelativeTime(50, 0),
                     null, new RelativeTime(5000, 0),
                     null,null)),
             new MemoryParameters(MemoryParameters.NO_MAX, 0),
             (MemoryArea) (ImmortalMemory.instance()), null, null);
    }
}
```

As shown, the two classes extend the `RealtimeThread` superclass and differ only in the parameters passed to a call to `super`. These parameter differences are significant as they indicate the real-time behaviour of thread execution.

It is important to note that if additional variations of real-time threads (different to those above) are used in the system, more subclasses need to be added to reflect the differences in behaviour.

The second implementation version of real-time threads in the Simulator does not utilise subclassing. Instead all parameters are specified at the point of thread creation. Thus the six parameters need to be constructed and passes wherever a new real-time thread is instantiated. The code for this is specified below:

```
SchedulingParameters scheduling = new PriorityParameters(
       PriorityScheduler.MIN_PRIORITY + 20);
ReleaseParameters release = new AperiodicParameters(null, null, null, null);
MemoryParameters memory = new MemoryParameters(
       MemoryParameters.NO_MAX, 0);
MemoryArea area = ImmortalMemory.instance();
ProcessingGroupParameters group = null;


drivingRT = new RealtimeThread(scheduling, release, memory, area,  group,
null)
...
```

This method gives programmers the freedom to specify specific parameters each time to reflect the different behaviour that each real-time thread may require. Thus if each real-time thread created is slightly different to others, this is the programming practice that would be followed. It must be noted that although this method provides the flexibility of creating contrasting thread behaviour for each real-time thread, this was not done in the Simulator. Each real-time thread used the thread properties identical to those described for the subclasses above. That is threads are either periodic or not.

Moreover the creation of real-time threads, threads in RTJava also differ to conventional threads in relation to a feasibility check that needs to be performed before they can be started. This feasibility check (using the `isFeasible` method) is carried out on all real-time threads to ensure that the scheduler has enough resources to satisfy the resource requirements in the thread's scheduling parameters, release parameters, memory parameters, and processing group parameters. If the scheduler cannot allocate the necessary resources, the `isFeasible` method returns false, and does not execute the thread. Thus the code for checking this feasibility and starting a real-time thread in the Simulator is shown below:

```
RealtimeThread rt;

...

if (!rt.getScheduler().isFeasible()) {
       throw new Exception("Problem starting real-time thread");
} else {
       rt.start();
}
```

The code illustrates that the particular `RealtimeThread` object that is to be started needs to retrieve its assigned scheduler object. Once this is complete, it is determined if this scheduler has the necessary resources required for running the thread. An exception is thrown if insufficient resources are available; otherwise a call to start is made. In the Simulator, this function is implemented as a separate `startRTThread` static method which can be shared among all real-time threads.

3.3 Memory Management

As stated before, the problem with Java for real-time systems is that its garbage collector introduces unpredictability. Memory management in RTJava enables the creation of objects that are not subject to garbage collection and will therefore not experience delays caused by its unpredictable execution. RTJava provides a number of different memory types that are not subject to garbage collection: physical memory, immortal memory, and scoped memory.

The memory types of physical memory and scoped memory are not implemented in the Simulator. Physical memory could not be implemented due to hardware constraints while non garbage collectable objects in the Simulator were not associated with any execution scope. In contrast, objects (that were not garbage collectable) were required to persist until the JVM terminates. Consequently the immortal memory type was implemented for objects (not subject to garbage collection) in the Simulator.

Figure 11 shows the classes that implemented memory management functionality.

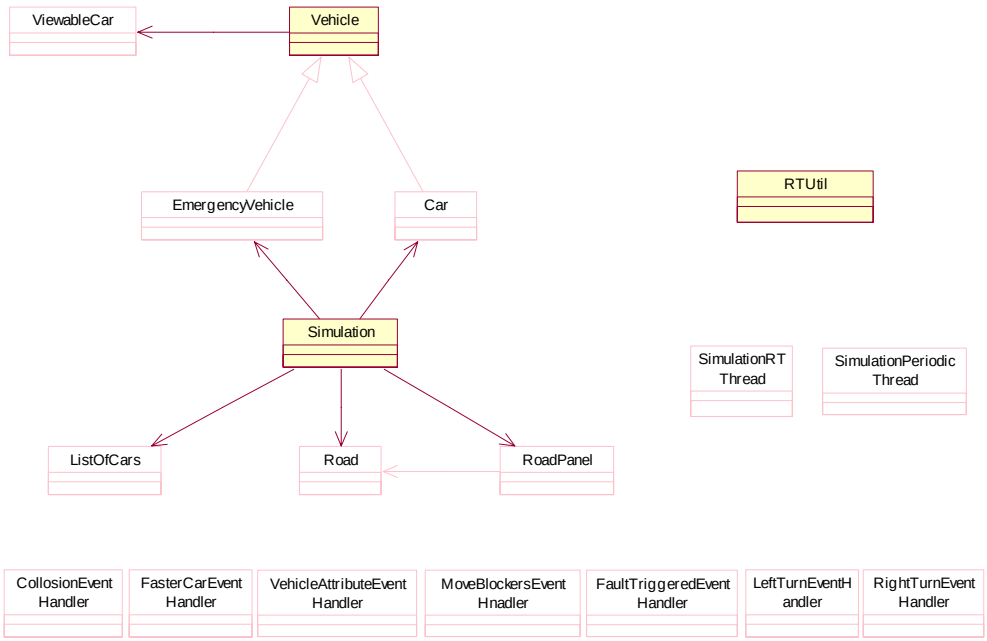

FIGURE 11: CLASSES CONTAINING MEMORY MANAGEMENT FUNCTIONALITY

RTJava provides a number of mechanisms for creating immortal objects. Some of these are associated with threading (where threads are assigned to different memory types as specified by the memory area constructor parameter). The remaining mechanisms are used for the creation of immortal objects. Creation of immortal objects can be classified into two categories – objects *with no* argument constructors and objects *with* argument constructors.

As the original implementation ignored any real-time necessities, all objects were allocated to the heap, meaning that they are all subject to garbage collection (and will therefore incur delays when it is executed). This unpredictability is not satisfactory in a real-time environment, thus objects (in the real-time version) of the Simulator should be assigned to memory that is not subject to garbage collection activities.

Of the objects in the Simulator the following were to be created in immortal memory:

- Car.java – argument constructor
- EmergencyVehicle – argument constructor
- ViewableCar – argument constructor
- ListOfCars – no argument constructor
- Road – no argument constructor

These objects are central to the operation of the Simulator and are subject to real-time constraints. Therefore the operation of these objects should not be unpredictably delayed by the garbage collector. Consequently they are chosen to be assigned to immortal memory and will persist until the JVM terminates.

The creations of immortal objects (for both arguments and no argument constructors) cannot be done using the traditional `new` operator on the constructor. Instead immortal object allocation involves a more verbose codebase.

Examples of the code used in the Simulator for the creation of objects with non argument constructors (`ListOfCars` and `Road`) are illustrated below:

```
roadNetwork = (Road) ImmortalMemory.instance().
        newInstance(Road.class);
listOfCars = (ListOfCars) ImmortalMemory.instance().
        newInstance(ListOfCars.class);
```

The combination of the `newInstance` method and Java's reflection mechanism enables objects with non argument constructors to be allocated to immortal memory.

The code for objects with argument constructors differs quite significantly and involves a number of additional steps. Also argument constructors require additional methods which provide the specific operations required for immortal memory. Thus in order to create immortal objects, a call one of these methods is required. The codebase for the `ViewableCar` object with argument constructor to be allocated to immortal memory are shown below:

```
public static ViewableCar createViewableCar(String name, boolean isVeh) {
        ViewableCar vcar = null;
        try {
                Class[] paramTypes = new Class[2];
                Object[] params = new Object[2];
                paramTypes[0] = Class.forName("java.lang.String");
                paramTypes[1] = Class.forName("java.lang.Boolean");
                params[0] = name;
                params[1] = new Boolean(isVeh);
                Class classType = Class.forName("ViewableCar");
                        Constructor constructor =
                                                classType.getConstructor(paramTypes);
                vcar = (ViewableCar) ImmortalMemory.instance().
```

```
                newInstance(constructor, params);
        } catch (IllegalAccessException iae) {
             System.out.println("Illegal Access Exception in creating
                    immortal viewable car object");
        } catch (InstantiationException ie) {
            System.out.println("Instantiation Exception in creating
                    immortal viewable car object");
        } catch (ClassNotFoundException cnfe) {
            System.out.println("ClassNotFound Exception in creating
                    immortal viewable car object");
        } catch (NoSuchMethodException nsme) {
            System.out.println("NoSuchMethod Exception in creating
                    immortal viewable car object");
        }
        return vcar;
    }
```

Creating the above object and others with argument constructors is clearly more verbose than both normal Java objects and immortal objects with non argument constructors. Additional steps relating to building parameter using `Class` and `Object` arrays as well as a `Constructor` object are necessary. Also creation of these objects is done through calling the above `createViewableCar` method as opposed to either the `new` operator (for normal Java objects) or `newInstance` (on `ImmortalMemory` as described for non argument constructor objects).

3.4 Synchronization and Resource Sharing

The concept of synchronization in RTJava is the same as that for normal Java systems. From a design and coding perspective, what programmers view is essentially identical. Only the implementation details of the specification (Java vs. RTJava) differ. RTJava strengthens the semantics of Java synchronization for real-time systems.

Consequently the design and implementation of the synchronization properties of the Simulator remain unchanged. On examination of the codebase, real-time compliant synchronization was required wherever synchronization was previously implemented. In addition it was also determined (through examining the implementation) that there were no other points in the programs execution that required synchronization properties (i.e. making the Simulator real-time compliant did not necessitate any additional synchronization of objects).

Figure 12 shows the classes that implemented synchronization and resource sharing functionality.

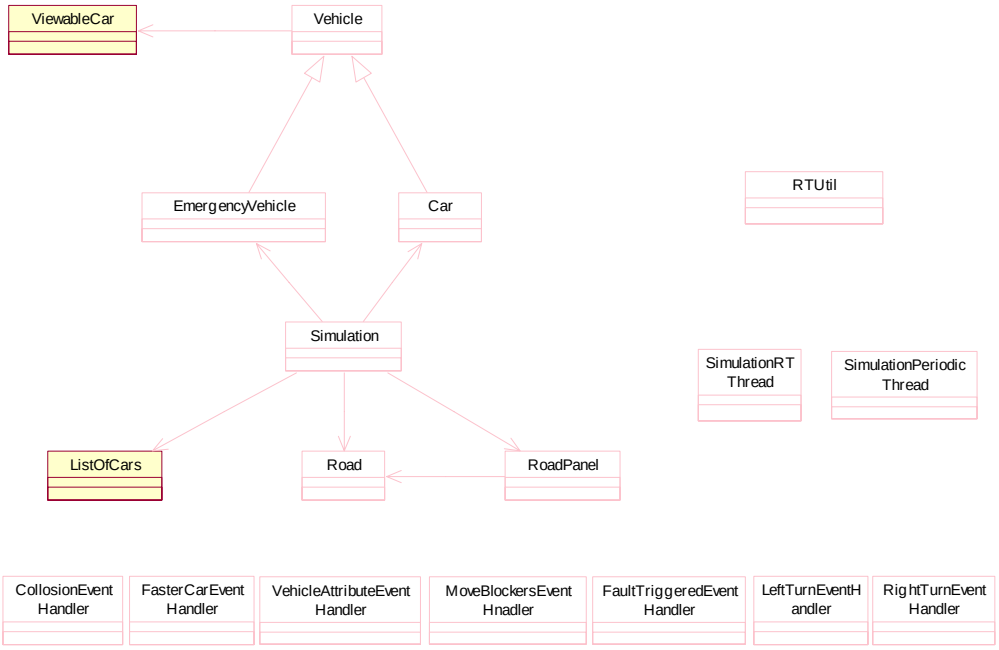

FIGURE 12: CLASSES CONTAINING SYNCHRONIZATION AND RESOURCE SHARING FUNCTIONALITY

In the Simulator, locking of the `ListOfCars` object was required. This object is accessed by all vehicles in the network for gathering data about other vehicles. Real-time access is essential as vehicles require this information in order to paint a global picture of the road network (i.e. where other cars are located in relation to itself, the velocity they are travelling at etc.). Therefore this concurrent access needs to be controlled in a real-time manner. (Obviously a single object containing all details about the network that all vehicles read and write to is insufficient in the real-world. However for illustration purposes, the Simulator uses a single shared object for such purposes).

The RTJava synchronization code is shown below:

```
public synchronized void clearPoll(String name) {
	while (updated == false) {
		try {
			wait();
		} catch (InterruptedException e) {
			throw new Exception("Synchronization Problem");
		}
		updated = false;
		pollValue = -1;
```

```
                linkedCar = new String("");
                parameter = 0;
                updated = true;
                notify();
        }
}
```

As evident, this code does not differ significantly from that of traditional Java synchronization. The `clearPoll` method is synchronized, and the accessing thread is forced to `wait` if the `updated` variable equals false. Also a call to `notify` is made when the executing thread is finished the operations described within.

The concept of wait-free queues is also implemented in the Simulator. However these queues were not implemented as part of the core functionality, but simply for illustration purposes. The reason for this is that core classes which contain the functional behaviour did not require any thread and real-time thread interaction (All threads in the Simulator are real-time threads).

The code shown below is based on test implementations in the Simulator (and is not part of the final implementation). It represents the periodic threads in the Simulator that accept new period values from a wait-free write queue. This was done in the Simulator to decrease the time between consecutive sensor polls that are executed by each vehicle.

```
public static void readPeriod() {
        RelativeTime newPeriod;
        WaitFreeReadQueue queue;
        RealtimeThread rt = currentRealtimeThread();
        while(true) {
                newPeriod = (RelativeTime)queue.read();
                // update period in thread if possible
                if(newPeriod != null) {
                        PeriodicParameters oldParam;
                        oldParam = (PeriodicParameters)
                rt.getReleaseParameters();
                        oldParam.setPeriod(newPeriod);
                }
                rt.waitForNextPeriod();
        }
}
```

The accessing thread (i.e. the current thread above) is allowed to read the queue without waiting. This is one of the principles of `WaitFreeReadQueue` objects (but `WaitFreeWriteQueue` require synchronized access). The time period is read by the thread and the old waiting period is set with the new period read previously. The real-time thread then waits for the specified period before repeating the above steps.

3.5 Asynchronous Event Handling

Asynchronous event handling enables a system to handle events that may occur asynchronously outside of the JVM (These external events are known as *happenings* in RTJava). Asynchronous event handling is based on the same concept as traditional event handling in Java – binding events to triggers and handlers and firing the event.

In the case asynchronous event handling, `AsyncEventHandler` classes are used for happenings. An asynchronous event causes the `run` method in the `AsyncEventHandler` class to execute. The `run` method calls the `handleAsyncEvent` method once for each time the event is fired.

In the Simulator, asynchronous events are thrown when a vehicle needs to carry out a particular action depending on sensor data received from other vehicles. For example if a car sensed an emergency vehicle was behind, then it should change lanes. In such a case, an asynchronous event is fired to notify the car that it should change lanes. Thus asynchronous events are adopted in the following contexts:

- Emergency vehicle behind – change lane (left or right)
- Slower car ahead – change lane (left or right)
- Slower car ahead and not possible to change lanes – slow down
- Received new sensor data – update vehicle attributes

Asynchronous events are also thrown in the system when faults (exceptions) occur in the Simulator. Thus whenever an exception is raised, it is handled by an `AsyncEventHandler` subclass (`FaultTriggeredEventHandler`) which performs the necessary operations to handle such faults.

Figure 13 shows the classes that implemented asynchronous event handling functionality.

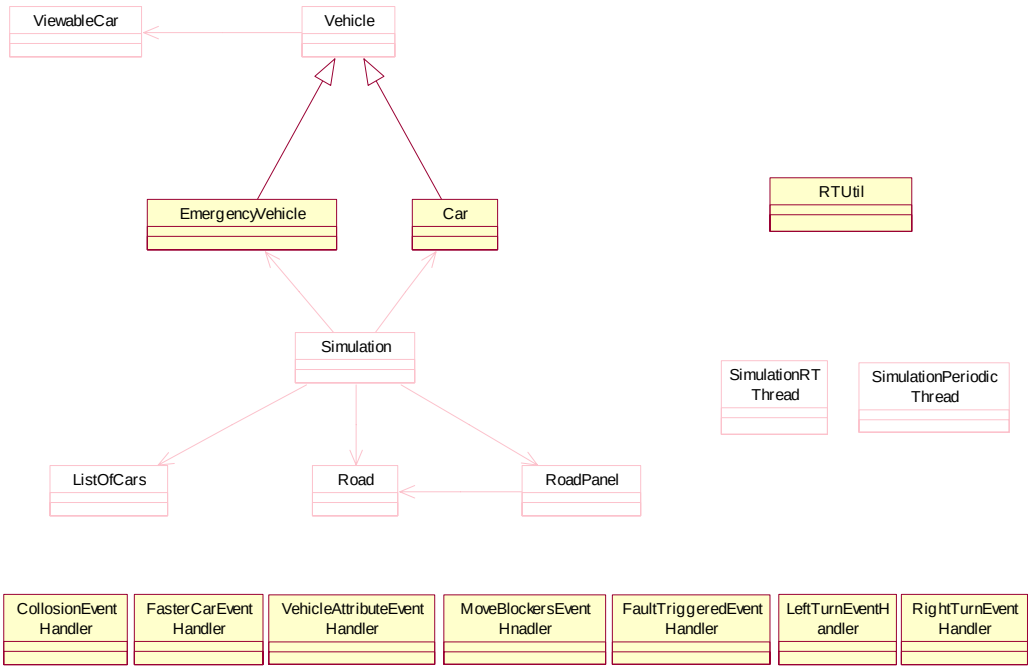

FIGURE 13: CLASSES CONTAINING ASYNCHRONOUS EVENT HANDLING FUNCTIONALITY

The introduction of asynchronous event handling into the Simulator not only required the alteration of core classes (for firing asynchronous events), but also a set of asynchronous event handler classes used for performing the necessary operations when a particular asynchronous event is fired. Seven asynchronous event handler classes have been added to the Simulator in total. These are: `CollisionEventHandler`, `FasterCarEventHandler`, `FaultTriggeredEventHandler`, `LeftTureEventHandler`, `RightTurnEventHandler`, `MoveBlockersEventHandler`, and `VehicleAttributeEventHandler`. All these classes extend the `AsyncEventHandler` class.

The code below illustrate the binding of events to a particular event name and the adding of handlers to these events.

```
public static void bindAsyncEvent(Class eventClass, String bindName) {

        AsyncEvent event = null;
        AsyncEventHandler handler = null;
        MemoryArea immortal = ImmortalMemory.instance();
        try {
                handler = (AsyncEventHandler)
                immortal.newInstance(eventClass);
```

```
            event = (AsyncEvent)
                    immortal.newInstance(AsyncEvent.class);
        } catch (InstantiationException e) {
                System.out.println("Instantiation Exception");
        } catch (IllegalAccessException ie) {
                System.out.println("Illegal Access Exception");
        }
        event.addHandler(handler);
        event.bindTo(bindName);
...
```

The method accepts a Class and String object which represent the class of asynchronous event and the name the event is to be bound respectively. An asynchronous event and an asynchronous event handler objects are created in immortal memory (to avoid garbage collection activities). The asynchronous event handler is reference to the particular event (represented by the argument passed). Finally the asynchronous event is bound to both the handler and the name as shown.

The call `event.fire()` illustrates the actual firing of asynchronous events when required. These calls are scattered throughout the application and called when necessary.

Also shown is the code for the `LeftTurnEventHandler` asynchronous event handler classes. This class like all other asynchronous event handler classes provide concrete implementation for the abstract `handleAsyncEvent` method declared in the `AsyncEventHandler` parent class.

```
public class LeftTurnEventHandler extends AsyncEventHandler{
    public void handleAsyncEvent() {
       // code that is executed when this
       // particular async event is fired
        Vehicle.shiftLaneLeft();
    }
}
```

Each of the asynchronous event handler classes inherit from the `AsyncEventHandler` class and thus are required to implement the abstract `handleAsyncEvent()` method. This method is automatically called when an event of the asynchronous event handler class type is fired. Thus operations implemented in the `handleAsyncEvent()` method are those that are executed for handling a specific asynchronous event.

3.6 Asynchronous Transfer of Control

Asynchronous transfer of control (ATC) is a mechanism that lets one thread throw an exception into another thread. RTJava provide two mechanisms for the implementation of ATC – the *fire* method and Java's exception handling mechanism.

The fire method for ATC is a combination of the fire method on `AsynchronouslyInterruptedException` (`AIE`), together with the `AIE` object's `doInterruptible` method. This mechanism allows asynchronous interrupts to be directed at particular methods. The advantage of this is that `AIE` can be associated with a `doInterruptible` method and thus execute the required operations when such an exception is thrown.

The exception handling mechanism for ATC differs slightly in that it is adopted when `AIE's` are not related with any specific `doInterruptible` methods. Such exceptions are known as generic AIE. Thus handling of these generic AIE's are done using conventional exception handling techniques with the `catch` block catching the thrown AIE. This case differs from the fire method as it allows AIE's to be propagated through multiple `catch` clauses until it reaches the threads `runnable` method.

In the Simulator, there are two instances that require ATC. The first instance relates to the updating of Vehicle attributes which are visible by other vehicles in the network (i.e. its `ViewableCar` object). If more recent sensor data arrives before the previous update is complete, then the real-time thread still performing the previous update needs to be interrupted. A newer thread which is executed to update the attributes with the more recent data causes the interruption. The second case is similar and relates to vehicles reading data about other vehicles. If this more up to date sensor information is available about other vehicles before this operation is complete, then it is interrupted. A newer thread which is started to read the more recent data causes the interruption of the previous thread.

The above ATC cases are implemented in the Simulator in two versions, one using the fire method and the other using the exception handling mechanism. This is done in order to illustrate both mechanisms which programmers may choose when implementing ATC.

Figure 14 highlights the classes that implemented asynchronous transfer of control functionality.

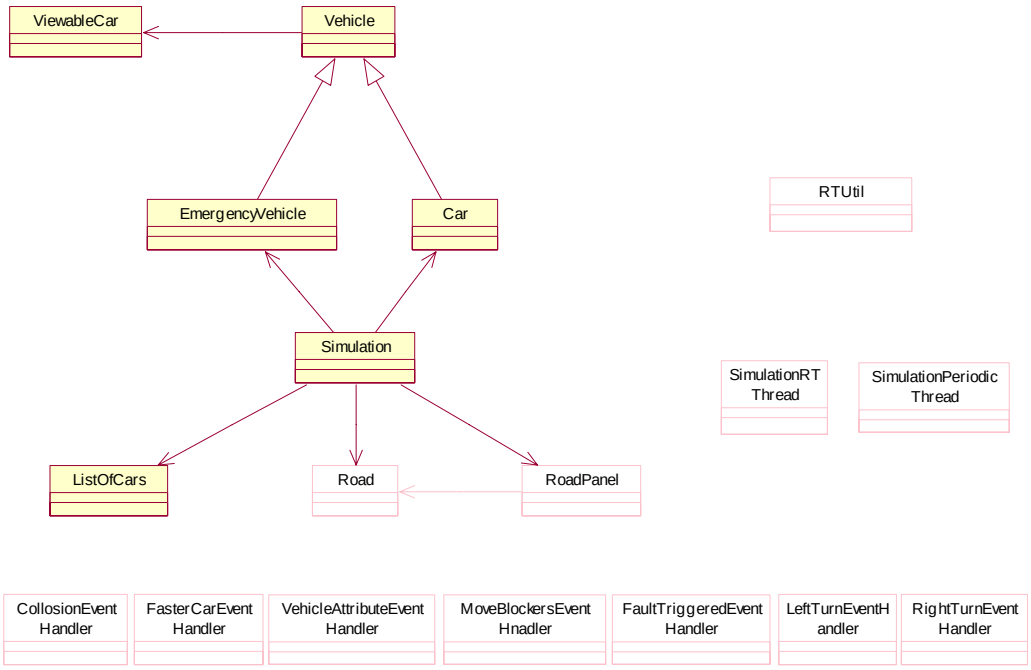

FIGURE 14: CLASSES CONTAINING ASYNCHRONOUS TRANSFER OF CONTROL FUNCTIONALITY

The code shown below shows the Simulator's implementation of ATC with the fire method.

```
AsynchronouslyInterruptedException locAie;
...
locAie = new AsynchronouslyInterruptedException();
locAie.doInterruptible(new Interruptible() {
       // called by JVM when a real-time thread is interrupted
       public void interruptAction(AsynchronouslyInterruptedException ie) {
              System.out.println("Thread interrupted");
              // operations to execute if real-time thread is interrupted
       }
       public void run(AsynchronouslyInterruptedException aie) throws
              AsynchronouslyInterruptedException {
              // interruptible code goes here
       }
});
```

The interruptible code is called with the `doInterruptible` method. Code with this method is designed to expect to be interrupted by a thrown AIE. As shown the interruptible code is placed in the

run method which throws an `AsynchronouslyInterruptedException.` The `interruptAction` method is provided in cases where the code needs to know it has been interrupted and take some action, such as clean up or logging of errors.

The AIE is fired using the `fire` method of `AsynchronouslyInterruptedException`.

```
ListOfCars.locAie.fire();
```

The exception handling mechanisms differs from this quite significantly. Interruptible code is simply placed in the try block with the subsequent catch statement, catching the `AsynchronouslyInterruptedException` that is fired when the code is interrupted. The code for this method, as implemented in the Simulator is shown below:

```
try {
	// interruptible code goes here
} catch (AsynchronouslyInterruptedException aie) {
	// operations that are called when an AIE is fired
}
```

It is clear that the above has the same structure as normal exception handling code. The operations that are to be executed when an AIE is fired are placed in the `catch` block (equivalent to that of the `interruptAction` method for the fire method of ATC). The interruptible code is placed in the `try` block.

The AIE is fired in the same way as the fire method.

```
ListOfCars.locAie.fire();
```

3.7 Asynchronous Thread Termination

Calling the `interrupt` method on a real-time thread provides a common way for terminating threads. However it is not a reliable general purpose way to kill threads as other parts of the system affected by the terminated thread may be left in an inconsistent state. Thus the common way to handle thread termination is to use ATC techniques (in conjunction with the `interrupt` methods) as it allows for clean up (restoring inconsistent data). This is made possible as the call to `interrupt` throws an AIE. When this occurs, the thrown exception which will propagate through all previously visited `catch` clauses and `interruptAction` methods until it reaches the thread's `runnable` method. Therefore it is possible for operations to be implemented in the clauses and methods that will perform any clean up that is necessary.

The Simulator implements thread termination in two instances. They are identical described above for ATC – the fire method and the exception handling mechanisms of Java.

Figure 15 shows the classes that implemented asynchronous thread termination functionality.

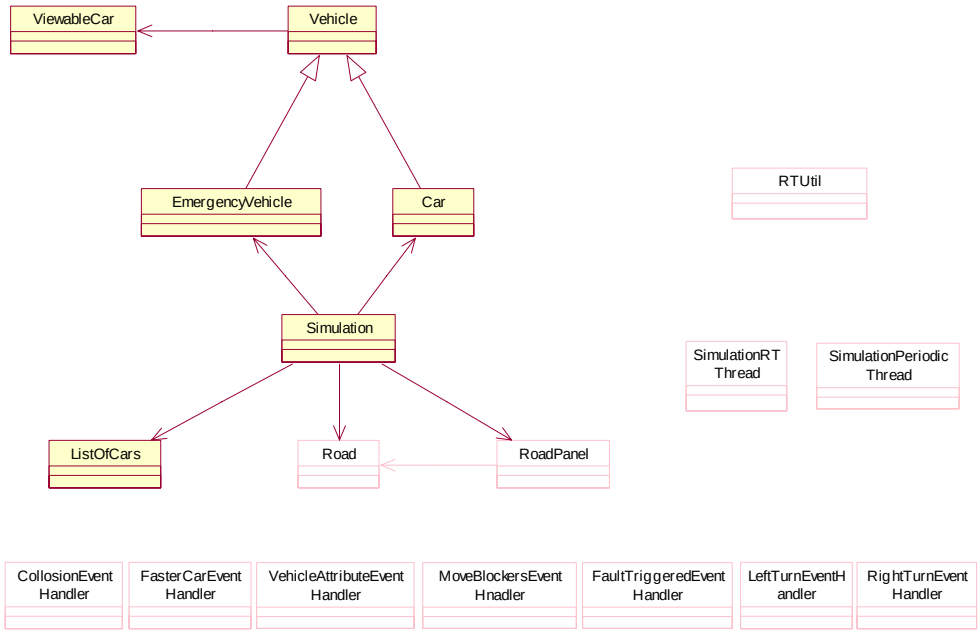

FIGURE 15: CLASSES CONTAINING ASYNCHRONOUS THREAD TERMINATION FUNCTIONALITY

The code below illustrates the implementation of the asynchronous thread termination area as done using the fire method (for ATC):

```
AsynchronouslyInterruptedException locAie;
RealtimeThread rt;
...
locAie = new AsynchronouslyInterruptedException();
locAie.doInterruptible(new Interruptible() {
	public void interruptAction(AsynchronouslyInterruptedException ie) {
		System.out.println("Thread interrupted");
		rt.interrupt();
		// perform clean up operations
	}
	public void run(AsynchronouslyInterruptedException aie) throws
		AsynchronouslyInterruptedException {
		// interruptible code goes here
	}
});
```

It is clearly visible that this code is almost identical to that as for ATC. The only difference is the call to the `interrupt` method at the start of the `interruptAction` method which is used to terminate the thread. The operations that are performed after this call enable clean up meaning that the thread is terminated safely.

The code for the exception handling method is shown below:

```
RealtimeThread rt;
try {
	// interruptible code goes here
} catch (AsynchronouslyInterruptedException aie) {
	// operations that are called when an AIE is fired
	rt.interrupt();
	// perform clean up operations
}
```

A similar concept applies here. The call to `interrupt` to terminate the thread again can be safely performed with the clean up operations carried out after.

3.8 Physical Memory Access

RTJava defines classes for programmers to directly access physical memory from code. This can be done in two ways described below.

Raw Memory Access (`RawMemoryAccess` class) defines methods that allow the programmer to construct an object that represents a range of physical addresses. A set of set and get methods allow the contents of the physical memory to be accessed through offsets which are interpreted as byte, short, int, long, float, or double values.

Physical memory areas differ from raw memory access in that a raw memory access object cannot contain objects or references to objects. Physical memory areas, in contrast, do provide this capability. The `ScopedPhysicalMemory` and `ImmortalPhysicalMemory` classes allow for the flexibility of allocating objects to particular memory addresses that represent a certain type of memory. For example Java objects that are used frequently may be placed in the physical memory addresses of fast RAM in order to optimise performance.

The physical memory access was not implemented in the Simulator due to constraints in the hardware available (i.e. laptop) and the limitations that they had in accessing physical memory locations. The remaining descriptions on design and evaluation in this area are based on test and sample code used during the course of the dissertation.

On examination of the Simulator, there were a number of areas that required the creation and access of physical memory addresses. Vehicles (`Vehicle`, `Car` and `EmergencyVehicle` objects) all require physical memory access. Attributes in these objects such as velocity and position coordinates are regularly accessed and modified and would therefore benefit from being allocated to fast physical memory.

Figure 16 shows the classes that the physical memory access functionality was designed for.

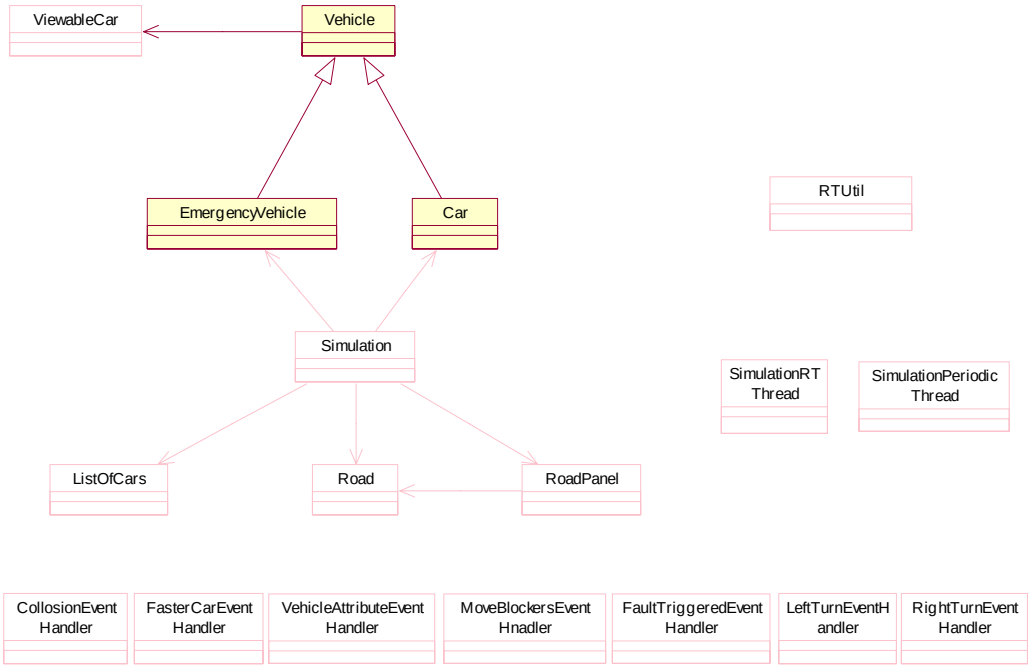

FIGURE 16: CLASSES CONTAINING PHYSICAL MEMORY ACCESS FUNCTIONALITY

The code below shows both the raw memory accesses to physical memory areas as designed for the Simulator.

```
public static void setValuesPhysicalMemoryAccess() {
	try {
		String[] type = new String[3];
		type[0] = new String("velocity");
		type[1] = new String("X pos");
		type[2] = new String("no cache");
		RawMemoryAccess rma = new RawMemoryAccess(type, 512);
		int pos1 = rma.getInt(100);
		int pos2 = rma.getInt(50);
```

```
            rma.setInt(100, 0);

            rma.setInt(50, 32);

            //getShort, getByte, setShort, setByte etc...


      } catch (SecurityException sec) {

            System.out.println("SecurityException in Physical Memory
                  Access");

      } catch (OffsetOutOfBoundsException offset) {

            System.out.println("OffsetOutOfBoundsException in Physical
                  Memory Access");

      } catch (SizeOutOfBoundsException size) {

            System.out.println("SizeOutOfBoundsException in Physical Memory
                  Access");

      } catch (UnsupportedPhysicalMemoryException unsup) {
            System.out.println("UnsupportedPhysicalMemoryException in
                  Physical Memory Access");

      } catch (MemoryTypeConflictException mem) {
            System.out.println("MemoryTypeConflictException in Physical
                  Memory Access");

      }
}
```

The String array is used to specify the specifics of the physical memory being accessed. The constructor for `RawMemoryAccess` has asked for 512 bytes of memory that maps the above velocity and X pos for non cached access. The series of `set` and `get` methods enable the accessing of raw memory through the `RawMemoryAccess` object.

4 Aspect-oriented Design and Implementation

This chapter describes the design and implementation of the aspect-oriented version of the Sentient Traffic Simulator. The seven areas of RTJava are individually addressed and details of the impact aspects have in each are given.

4.1 Thread Scheduling and Dispatching

The AOP version of the Simulator in this area aims to identify and eliminate any scattered or tangled implementation related to thread scheduling and dispatching. AOP improvements in this area have been directed at thread creation and the starting of real-time threads.

The first case relates to the design of the Simulator which includes the `RealtimeThread` subclasses. As stated previously, these classes were introduced to specify specific threading behaviour that is shared between numerous threads in the system. In the OO implementation, each subclass is required to instantiate each `RealtimeThread` parameter separately. Thus the introduction of an aspect classes can remove this crosscutting behaviour from each `RealtimeThread` subclass, placing this functionality in a single modular unit. Figure 17 shows the capturing of this crosscutting functionality and the subsequent classes that are affected.

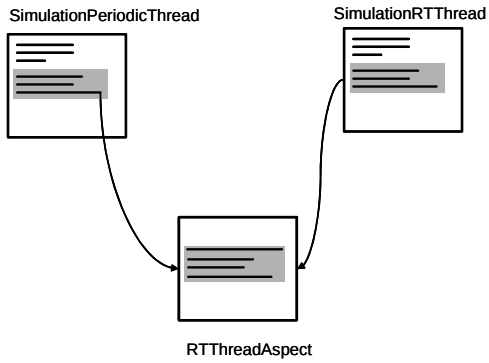

FIGURE 17: ASPECT FOR REAL-TIME THREAD SUBCLASSING

The AO version of the Simulator creates an `RTThreadAspect` aspect class for crosscutting thread functionality. The code below shows the how the AOP is utilised for the creation of one of the thread types (`SimulationRTThread`) in the system:

```
pointcut createRTThread():execution(* SimulationRTThread.new());

RealtimeThread around():creatRTThread()  {
       SchedulingParameters scheduling = new PriorityParameters(
```

```
            PriorityScheduler.MIN_PRIORITY + 20);
        ReleaseParameters release = new AperiodicParameters(null, null,
        null,  null);
        MemoryParameters memory = new MemoryParameters(
            MemoryParameters.NO_MAX, 0);
        MemoryArea area area = ImmortalMemory.instance();
        ProcessingGroupParameters group = null;
        Logic logic = null;
        return new RealtimeThread(scheduling, release, memory, area,
        group, logic);
}
```

The pointcut captures all calls to the `SimulationRTThread` constructor. When this constructor is called, the implementation detailed in the around advice is executed in place of the constructor implementation. It is here that all the parameter creation and initialisation takes place in addition to the actual instantiation of the real-time thread.

Chapter 3 describes an alternative method to subclassing when creating real-time threads. This method removes the need for subclassing completely and instead initialises all `RealtimeThread` parameters prior to thread creation in aspects. As stated previously this is a common practice adopted by programmers when each thread requires slightly different properties, such as scheduling, memory or processing group. Thus if aspects were to be adopted here, then each point where a thread is created would be need to be identified and a separate pointcut declaration and advice implementation (specifying the `RealtimeThread` parameters) would need to be provided for each thread. And although an different aspect would be required for each variation of the `RealtimeThread` object, the overall modularity would be improved, with crosscutting thread initialisation code now placed in a single modular unit and not scattered and tangled throughout the codebase.

Yet the difficulty previously described is reduced somewhat if particular classes used a single type pf thread. For example, all threads in the Vehicle class used the parameters originally used in the `SimulationRTThread` subclass. If this were the case, a single aspect could be adopted to capture this initialisation behaviour that may occur in several occasions in the Vehicle class. The code below illustrates this point.

```
pointcut createVehicleRTThread()
:execution(* Vehicle.RealtimeThread.new());

RealtimeThread around():createVehicleRTThread()  {
        SchedulingParameters scheduling = new PriorityParameters(
            PriorityScheduler.MIN_PRIORITY + 20);
```

```
        ReleaseParameters release = new AperiodicParameters(null, null,
        null, null);
        MemoryParameters memory = new MemoryParameters(
                MemoryParameters.NO_MAX, 0);
        MemoryArea area area = ImmortalMemory.instance();
        ProcessingGroupParameters group = null;
        Logic logic = null;
        return new RealtimeThread(scheduling, release, memory, area,
        group, logic);
}
```

The pointcut captures all `RealtimeThread` constructor calls occurring in the Vehicle class. Thus the aspect eliminates the need for the thread parameters to be created in the Vehicle class. Instead the aspect provides this functionality. The around advice again provides the initialisation arguments required for and the actual thread creation. Consequently all that is required in the vehicle class is:

```
RealtimeThread rt = new RealtimeThread();
```

A similar case applied at a higher level of granularity. If the package structure of the system reflected the types of threads in the each package. Aspects have the ability to specify points in the program related to different packages. This is done using the `within` AOP construct. Therefore if all threads in a particular package had identical real-time behaviour, then the initialisation parameters for these could be encompassed in a single modular unit. The code below illustrates this ability:

```
pointcut createNonperiodicPackRTThread()
:execution(* .RealtimeThread.new()) && within(nonperiodic);


RealtimeThread around():createNonperiodicPackageRTThread()  {
        SchedulingParameters scheduling = new PriorityParameters(
                PriorityScheduler.MIN_PRIORITY + 20);
        ReleaseParameters release = new AperiodicParameters(null, null,
        null, null);
        MemoryParameters memory = new MemoryParameters(
                MemoryParameters.NO_MAX, 0);
        MemoryArea area area = ImmortalMemory.instance();
        ProcessingGroupParameters group = null;
        Logic logic = null;
        return new RealtimeThread(scheduling, release, memory, area,
        group, logic);
}
```

In this case the pointcut is implemented to capture any `RealtimeThread` constructor calls that occur in the `nonperiodic` package. The remainder of the implementation is the same as described above with the `around` advice providing the real-time thread instantiation behaviour. The implementation for the periodic package of the Simulator is similar differing only on the argument of within and the parameters given for instantiating the `RealtimeThread`.

The two cases described above for using aspects to specify threading behaviour in either specific classes or packages has a significant consequence. It eliminates the need for subclassing `RealtimeThread`, altogether as is the case for the OO implementation. Aspects could be used in place of subclasses for specifying the threading properties of each real-time thread. However, it is noted that this practice (of organising particular types of threads in this class or package way) is not commonly practiced and is only done here to illustrate the ability of AOP in this area.

Finally, in this area, aspects can also be applied for the starting of real-time threads. As stated previously, a feasibility check is performed before the starting each real-time thread. This provides a clear case of code scattering and tangling that is evident throughout the Simulator. Figure 18 shows this capturing of crosscutting code and its affected classes.

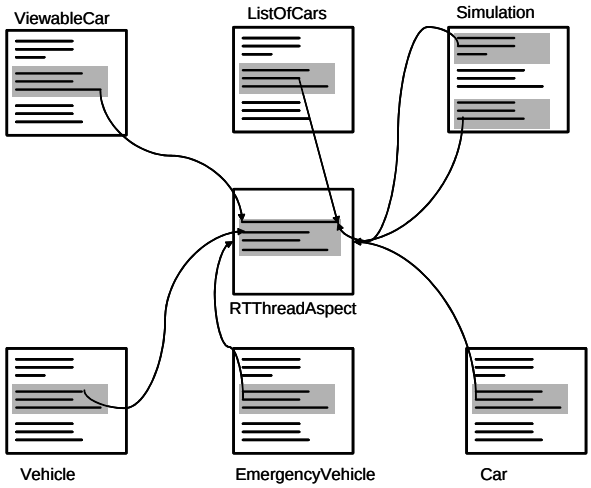

FIGURE 18: ASPECT FOR STARTING REAL-TIME THREADS

The code illustrated below shows how the previously scattered and tangling code can be eliminated.

```
pointcut executeRTThread():call(* *.RealtimeThread.new(..));

after() : executeRTThread() {
       RealtimeThread rt = (RealtimeThread) thisJoinPoint;
       if (!rt.getScheduler().isFeasible()) {
```

```
            RTUtil.fireFaultEvent();
        } else {
            rt.start();
        }
}
```

The pointcut declaration captures all the points in the programs execution where real-time threads are to be started. In this instance they are started immediately after creation. The code in the `after` advice captures the real-time thread executing at this join point. A feasibility check is then performed for this thread and if sufficient resources are available a call to `start` is made. Otherwise an asynchronous is fired indicating a problem has occurred.

4.2 Memory Management

As described in the previous chapter, the creation of objects in immortal memory is not as trivial as that for objects allocated to the heap. Using the `new` operator is no longer possible, with additional calls required for immortal memory allocation. These additional steps that are needed for immortal object creation are a good example of code scattering that could be better modularised through AOP.

The first case refers to objects with no argument constructors. As described earlier, this involves a call to `ImmortalMemory`'s `newInstance` method.

```
listOfCars = (ListOfCars) ImmortalMemory.instance().
        newInstance(ListOfCars.class);
```

The use of aspects can be beneficial here with this reference to `ImmortalMemory` eliminated from wherever such object instantiation occurs. Aspects can weave this immortal memory related code to wherever necessary and provide the setup calls before the call to create the new object is made. Figure 19 illustrates the classes affected by the use of aspects in this instance.

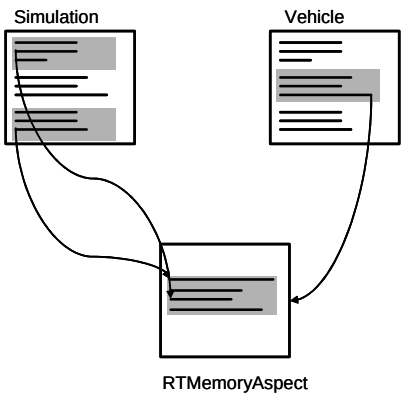

FIGURE 19: ASPECT FOR ALLOCATING IMMORTAL MEMORY TO A NO ARGUMENT CONSTRUCTOR OBJECT

In effect aspects allow objects to be allocated into immortal memory using the conventional new operator. The code below shows the code of the aspect.

```
pointcut createListOfCarsMemory():call(simulation.ListOfCars.new());


ListOfCars around():createListOfCarsMemory() {
	ListOfCars loc;
	try {
		loc = (ListOfCars) ImmortalMemory.instance().
			newInstance(ListOfCars.class);
	} catch (IllegalAccessException iae) {
		System.out.println("Illegal Access Exception in creating
		ListOfCars memory");
	} catch (InstantiationException ie) {
		System.out.println("Instantiation Exception in creating
		ListOfCars memory");
	}
	return loc;
}
```

The pointcut captures all instances where the `ListOfCars` constructor is called. The related around advice specifies the implementation that will replace that of the constructor. Thus the aspect now provides the functionality for creating the `ListOfCars` object, hiding the real-time related constructs from the core classes.

Similarly is the case for immortal objects with argument constructors. Aspects can be adopted to perform all the steps required for building and initialising parameters. The overall effect is the same as that for no argument constructors. Objects (with argument constructors) can also be instantiated with the new operator when aspects are used. The aspect class provides the immortal memory and constructor related functionality. Figure 20 shows the classes affected by the above.

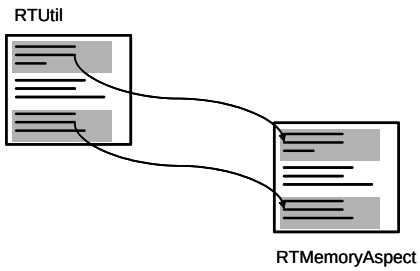

FIGURE 20: ASPECT FOR ALLOCATING IMMORTAL MEMORY TO A ARGUMENT CONSTRUCTOR OBJECT

Shown below are the code of the aspect class for argument constructor objects and the resultant code from the core classes where these objects were created.

```
pointcut createViewableCarMemory(String thread, boolean isVeh) :

call(simulation.ViewableCar.new(String, boolean)) && args (thread,
       isVeh);


ViewableCar around(String thread, boolean isVeh) :
createViewableCarMemory(thread, isVeh) {

       ViewableCar vcar = null;

       try {

              Class[] paramTypes = new Class[2];

              Object[] params = new Object[2];

              paramTypes[0] = Class.forName("java.lang.String");

              paramTypes[1] = Class.forName("java.lang.Boolean");

              params[0] = thread;

              params[1] = new Boolean(isVeh);

              Class classType = Class.forName("ViewableCar");

              Constructor constructor =
                     classType.getConstructor(paramTypes);

              vcar = (ViewableCar) immortalMemory.instance().newInstance(

                     constructor, params)

        } catch (IllegalAccessException iae) {

            System.out.println("Illegal Access Exception in creating
                     viewableCar memory");

        } catch (InstantiationException ie) {

            System.out.println("Instantiation Exception in creating
                     viewableCar memory");

        } catch (ClassNotFoundException cnfe) {

            System.out.println("ClassNotFound Exception in creating
                     viewableCar memory");

        } catch (NoSuchMethodException nsme) {

            System.out.println("NoSuchMethod Exception in creating
                     viewableCar memory");

        }

        return vcar;

       }
```

The above is an example of an aspect used for argument constructor objects. The pointcut captures all of the `ViewableCar` constructor calls in the Simulator, as in the case of argument constructor objects, replaces the constructors implementation with that specified in the `around` advice block.

The main issue with aspects in this case is that a separate pointcut and advice are required for each object that is to be allocated to immortal memory. However, the opposing argument is that this functionality, which was originally scattered throughout the codebase, is now in a single modular unit (which is the main aim of AOP). Also object instantiation has been made easier as the `new` operator can now be used as opposed to the more verbose method originally required.

There is the argument that the use of aspects in this case somewhat resembles that of the Factory Method design pattern [50]. This will be discussed in more detail in the Evaluation chapter of the dissertation.

Although aspects allow for objects, even in immortal memory, to be created using the `new` operator, there remains the argument that this is not beneficial. This is because a more verbose and long-winded mechanism for allocating immortal objects makes it perfectly clear that a particular allocation is from immortal memory. Therefore the possibility of confusion related to the memory area of objects would be non existent which is not the case in the AOP approach. This will be discussed further in the Evaluation chapter of the dissertation.

4.3 Synchronization and Resource Sharing

Previous studies have shown how AOP can provide better modularisation of synchronization code [3, 51]. As the implementation of the synchronization area of RTJava is the same as that for traditional Java, RTJava can also gain such modularisation benefits through the use of AOP enhancements.

Aspect enable improved modularisation as it eliminates the need `wait()` and `notify()` constructs when accessing shared resources. Such calls, which maybe scattered and tangled throughout the codebase, are removed from core code and placed in a single modular unit. Figure 21 shows the classes affected by the introduction of the `SynchronizationAspect`.

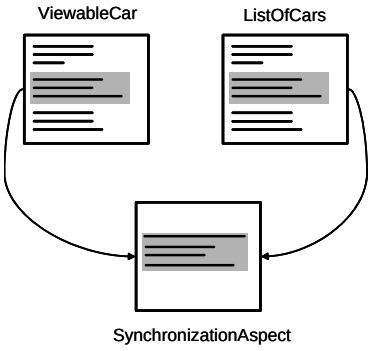

FIGURE 21: ASPECT FOR SYNCHRONIZATION AND RESOURCE SHARING

The code below shows the code contained in the SynchronizationAspect of the Simulator and the changes to the original code that contained the synchronization constructs.

```
pointcut updateAction():execution(* ViewableCar.*Poll(..));
before():updateAction(){
	synchronized(ViewableCar.linkedCar) {
		while(ViewableCar.updated == false) {
			try {
				ViewableCar.linkedCar.wait();
			} catch (InterruptedException ie){
			...
after():updateAction() {
	ViewableCar.linkedCar.notify();
}
```

The pointcut above captures the point in program execution when methods ending in Poll are executed. The reason being all methods ending in Poll are to be synchronized. The before advice specifies the operations to be carried out before the capture method (e.g. clearPoll) is executed. This is where access to the resource is requested. If not, then the accessing thread is forced to wait. The after advice performs the notification after the operations in the capture method have been executed.

```
public void clearPoll(String name) {
	updated = false;
	pollValue = -1;
	linkedCar = new String("");
	parameter = 0;
	updated = true;
}
```

The above shows the resulting code in the core class with the inclusion of the SynchronizationAspect. The synchronized constructs are removed as the wait and notify calls are no longer required.

RTJava also introduces the notion of wait-free queues, which provide synchronization without the need for locking. However it was determined that these queues cannot be constructed as aspects due to the specifics of their implementation.

4.4 Asynchronous Event Handling

Asynchronous event handling may be classified into a number of categories in relation to the examining of aspectual behaviour – the binding of events to event names and handlers, firing of asynchronous events and the actual handling of the fired event.

The code for the binding of a particular asynchronous event to names and handlers is described above. The use of aspects in this case allows this code to be removed from core classes and placed in a single aspect class. The resultant code for this is shown below:

```
pointcut bindEvent(Class eventClass, String bindName)
:call(* *.bindAsyncEvent(Class, String)) && args (eventClass, bindName);

void around(Class eventClass, String bindName)
: bindEvent(eventClass, bindName) {
	AsyncEvent event = null;
	AsyncEventHandler handler = null;
	MemoryArea immortal = ImmortalMemory.instance();
	try {
		handler = (AsyncEventHandler) immortal.newInstance(eventClass);
		event = (AsyncEvent)immortal.newInstance(AsyncEvent.class);
	} catch (InstantiationException e) {
		System.out.println("Instantiation Exception in binding async
			event");
	} catch (IllegalAccessException ie) {
		System.out.println("Illegal Access Exception in binding async
			event");
	}
	event.addHandler(handler);
	event.bindTo(bindName);
}
```

The pointcut captures all calls to the `bindAsyncEvent` method which take arguments of `Class` and `String` types. Again the around advice includes the implementations that is executed in place of those in the `bindAsyncEvent` method. Thus all constructs related to asynchronous event handling are contained in the aspect and not the classes containing the core functionality. Figure 22 illustrates the affect classes on the above aspect on the Simulator.

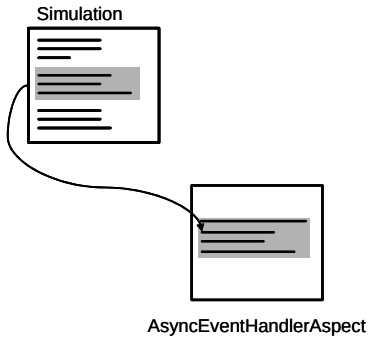

FIGURE 22: ASPECT FOR BINDING ASYNCHRONOUS EVENTS

The functionality specified in the `around` advice block replaces the implementation of the `bindAsyncEvent` method. Consequently any implementation in the replaced method becomes unused. Thus when a call to this method is made, the operations in the around advice are executed in place of the `bindAsyncEvent` method. The only consequence of placing this asynchronous event binding code into an aspect is that this code is extracted into a separate modular unit, thus improving the overall modularity of the Simulator.

Asynchronous event handling also involves firing of the asynchronous event depending on environment conditions. Although the firing of events is scattered throughout the codebase, capturing this crosscutting behaviour as an aspect is not possible. The reason for this is that AspectJ does not support the identifying of execution points inside method boundaries (where events are fired). Consequently, specifying exact points in the execution of the program where asynchronous events are fired cannot be encapsulated in aspects pointcut declaration. Therefore the Simulator does not implement any aspects related to the firing of asynchronous events.

The final case for asynchronous event handling is the actual handler classes which provide the operations that occur when a particular event is fired. Because the actual functionality specified in the `handleAsyncEvent` method is specific for different events, they cannot be encapsulated into an aspect. However all asynchronous event handler classes must extend the `AsyncEventHandler` class (and provide concrete implementation for the `handleAsyncEvent` method). This declaration is a prime example of crosscutting code which can be modularised into an aspect. Figure 23 illustrates the classes affect by the use of such an aspect.

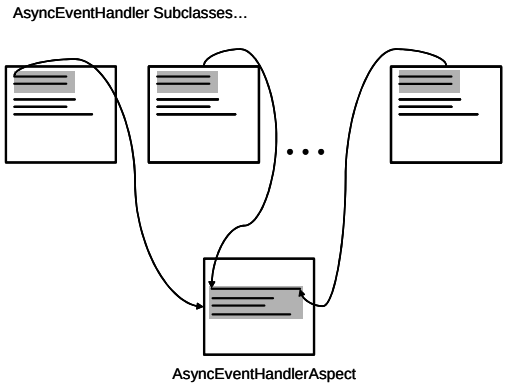

FIGURE 23: ASPECT FOR EXTENDING ASYNCEVENTHANDLER

The code for this as implemented in the aspect is shown below:

```
declare parents: eventHandler.*EventHandler extends AsyncEventHandler;
```

The `declare parents` declaration is known as an inter-type declaration. This case specifies that all classes ending in `EventHandler` and in the `eventHandler` package are to extend the `AsyncEventHandler` class. The consequence of this is that each asynchronous event handler class no longer needs to provide this inheritance declaration.

```
public class LeftTurnEventHandler {
    public void handleAsyncEvent() {
       // code that is executed when this
       // particular async event is fired
       Vehicle.shiftLaneLeft();
    }
}
```

The above `AsyncEventHandler` subclass, as a result of using aspects, longer have to provide the code indicating their inheritance to the `AsyncEventHandler` class. This is the case for all `AsyncEventHandler` subclass.

4.5 Asynchronous Transfer of Control

As described previous chapter, ATC area of RTJava can be implemented using either the *fire* method or using traditional exception handling mechanisms. AOP enhancements can be adopted in both instances in an attempt of improving the modularisation of ATC code.

In relation to the first methods, the *fire* method, this does exhibit some aspectual behaviour. Firstly the interruptible code block is required to throw an `AsynchronouslyInterruptedException`. And

this is the case wherever the fire method is used for ATC. Consequently, a single aspect can be used to capture this crosscutting behaviour. The code for the aspect which carries out this function is shown below:

```
declare soft : AsynchronouslyInterruptedException :
        execution(* *.run(AsynchronouslyInterruptedException))
        && within(simulation);
```

The `declare soft` indicates that all run methods which takes an `AsynchronouslyInterruptedException` argument throws an `AsynchronouslyInterruptedException.` This aspect applies to all classes in the `simulation` package as indicated by the `within` construct. Figure 24 illustrates the above.

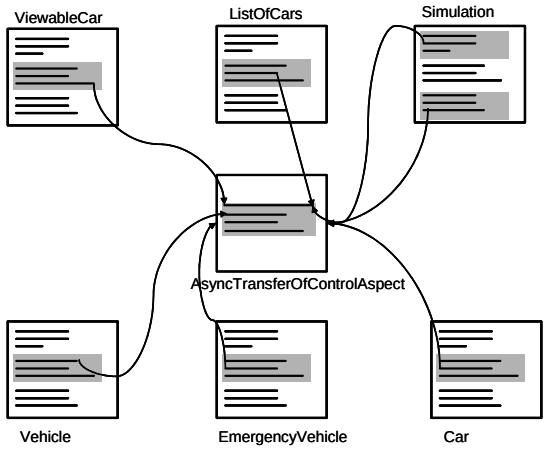

FIGURE 24: ASPECT FOR ASYNCHRONOUS TRANSFER OF CONTROL

In addition, it may be possible to provide the implementation of the `interruptAction` method in an aspect. Note that however, this is only possible if the operations performed by this method are identical in every instance. This is the case in the Simulator, thus enabling the implementation for this method to be specified in an aspect class. The code below illustrates the above:

```
pointcut weaveInterruptAction() : call(* *.interruptAction());

void around() : weaveInterruptAction() {
        System.out.println("Thread interrupted");
        // operations to execute if real-time thread is interrupted
}
```

However, it is important to note that this is only possible when the implementation for every `interruptAction` method is identical. This is of course unacceptable for most systems and is only done as illustration purposes for this dissertation.

Aspects can also capture crosscutting behaviour when using exception handling for ATC. There are two important points to note with using aspects for exception handling. Firstly AspectJ does not support for capturing the catching of exceptions inside method boundaries. The work-around for this limitation consisted of wrapping exception catching in methods. This is again, not common or acceptable for system development, but is done for the illustration purposes of this dissertation. Secondly, the degree in which aspects are effective in exception handling is based on the number of reaction types of certain exceptions. Thus if the number of reaction types to exceptions is much smaller than the number of places where the particular exception is caught, then the more meaningful aspects become.

In the Simulator's case of ATC, the operations that are performed when an AIE is thrown are identical in each case. Therefore there is sufficient redundancy in the system to demonstrate the potential of aspects. This is, as above, not common practice and is unacceptable but for the purpose of the dissertation. The code below reflects the points described above:

```
pointcut catchAIE() : call (* *.interrupted*(..))
	& within(simulation);


after() : catchAIE() {
	catch (AsynchronousInterruptedException AIE) {
		System.out.println("AIE Thrown, RTThread interrupted");
	}
}
```

If it were the case that different operations were required for each specific case for the catching of AIE, then a separate aspect would be needed for each. The code below shows the resultant impact on the aspect class:

```
pointcut catchAIE() : call (* *.interruptableUpdateViewableCar(..))
	&& within(simulation);


after() : catchAIE() {
	catch (AsynchronousInterruptedException AIE) {
		// perform specific operations when an AIE is thrown for
		//in this case
	}
}
```

```
pointcut catchAIE() : call (* *.interruptibleUpdateLOC(..))
	&& within(simulation);
after() : catchAIE() {
	catch (AsynchronousInterruptedException AIE) {
		// perform specific operations when an AIE is thrown for
		// in this case
	}
}
```

The aspect now includes the addition of a new pointcut and related advice blocks. The classes affected by this ATC technique are the same as that for the fire technique shown in figure 24.

It is argued that aspects may instil bad programming practices for the handling of exceptions whereby programmers become lazy in their implementation in handling exceptions. Aspects may give such programmers an easy alternative by allowing them to delegate exception handling responsibilities to the aspect and encouraging the same reaction for exceptions. This argument is further discussed in the Evaluation chapter of the dissertation.

4.6 Asynchronous Thread Termination

As the OO version of asynchronous thread termination is somewhat identical to that of ATC, (differing only in a call to interrupt), the same applied for the AO versions of the two areas. Thus the same principles apply here as they did above for ATC.

Thus aspects can be adopted in this area as with ATC. These are the modularisation of the throwing of an `AsynchronouslyInterruptedException`, the implementation of the `interruptAction` method (where possible) or the capturing of the exception handling code when adopting this method for ATC.

Yet it must be also noted that the limitations of where the above modularisation of aspectual behaviour is possible still persist for this area, as with ATC.

4.7 Physical Memory Access

Due to the specifics of physical memory access, the exhibition of aspectual functionality in this area is quite limited. Because each physical memory access may be read or written to different memory addresses indicated by different position variable, this makes capturing of this behaviour in a meaningful aspect very difficult, if not impossible. Yet although the accessing of physical memory is unique in each case, each access shares a common characteristic whereby a number of physical memory access exceptions are thrown and thus need to be handled. This is illustrated in the code described in the OO implementation for this area.

Aspects can therefore be used to modularise this exception catching code which occurs at every point of physical memory access. The code below shows the aspect for this case and the impact on the original physical memory access implementation.

```
pointcut catchPhysicalMemoryException()
: call (* *.PhysicalMemoryAccess(..)) && within(simulation);

after() : catchPhysicalMemoryException() {
       catch (SecurityException sec) {
       } catch (OffsetOutOfBoundsException offset) {
       } catch (SizeOutOfBoundsException size) {
       } catch (UnsupportedPhysicalMemoryException unsup) {
       } catch (MemoryTypeConflictException mem) {}
}
```

Similar to the exception catching aspect implemented for ATC, the pointcut captures all methods in the `simulation` package ending with `PhysicalMemoryAccess`. The after advice provides the implementation for the catching of the different exceptions that are thrown during access of physical memory addresses. The classes affected by the above are shown in figure 25.

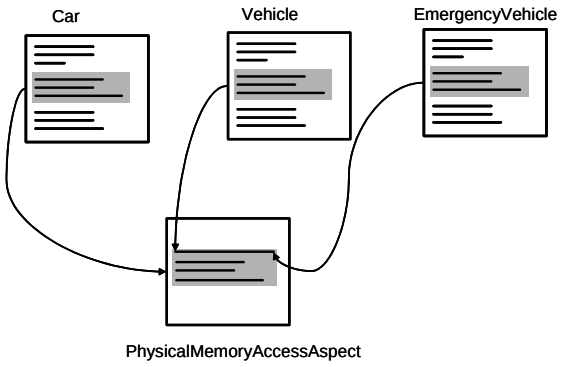

FIGURE 25: ASPECT FOR PHYSICAL MEMORY ACCESS

The same principles apply here as for the earlier cases for exception handling described for ATC and asynchronous thread termination areas. The wrapping of exception catching in methods is required for AspectJ. Also aspects are only significantly useful if the number of reaction types is much smaller than the number of places where exceptions are caught. If a number of reaction types exist, then a different aspect is required for each reaction type. Moreover, the same argument related to aspects instilling a bad exception handling practice in programmers applies here as it did for the exception handling cases above. This will be discussed further in the Evaluation chapter (chapter 6) of the dissertation.

5 Evaluation Metrics

This chapter provides a more detailed description of the Chidamber and Kemerer (C&K) metrics suite that is used in the evaluation of the Simulator. Each of the C&K metrics are addressed, describing their implications for system attributes and the affects of aspect-orientation on these metrics.

5.1 Background

Software metrics are a way of qualifying software design. Several metrics suites have been proposed for evaluating the design of object-oriented systems. Such metrics differ from traditional measures such as lines of code and comment percentage and instead are designed to focus on the combination of function and data as an integrated object [52].

As of yet, no metrics have been proposed for aspect-oriented systems [39]. Yet because AOP builds on existing object-oriented concepts, the metrics used for evaluating object-oriented systems are can be applied for aspect-oriented systems. However further research is required into this are as the effects and accuracy of applying OO metrics to evaluate AO software still remains an open issue.

Some of the important attributes evaluated by OO metrics are:

- Understandability
- Maintainability
- Reusability
- Testability

Therefore the metrics applied for evaluation of a system must be effective in measuring one or more of the above attributes. The C&K metric suite described below consist of a series of metric which highlight the affects of OO design and implementation on the attributes above.

5.2 The Chidamber and Kemerer Metrics (C&K) Suite

The Chidamber and Kemerer Metrics (C&K) Suite is a suite of metrics designed for evaluating object-oriented designs and is detailed in [40]. The metric suite itself is based on sound measurement theory and has been empirically as well as theoretically validated. Its purpose is to measure the key elements of object-oriented software such as encapsulation, abstraction, and inheritance. The suite has been adopted by many for evaluating object-oriented systems including the Software Assurance Technology Centre (SATC) at NASA Goddard Space Flight Centre [52].

The suite itself contains six metrics and are identified as; (1) Weighted Methods per Class; (2) Depth of Inheritance Tree; (3) Number of Children; (4) Coupling Between Objects; (5) Response For a Class; (6) Lack of Cohesion in Methods.

The remaining sections of the chapter describe the metrics of the C&K metric suite. The details of how aspect-orientation can impact on each metric are taken from [39] and are also described below.

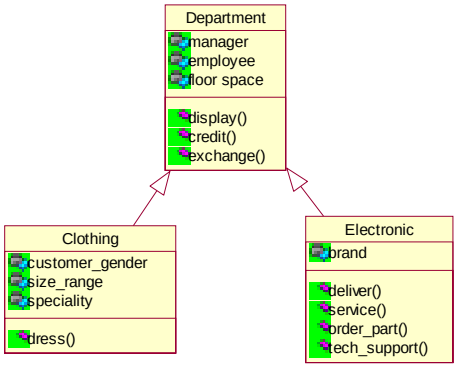

FIGURE 26: OBJECT-ORIENTED APPLICATION

5.2.1 Weighted Methods per Class (WMC)

5.2.1.1 Background

WMC is a measure of the number of methods implemented within a class. It is calculated by counting the number of methods in each class. Referring to figure 26 above WMC for *Clothing* = 1 and WMC for *Electronic* = 4. WMC is designed to measure understandability, maintainability, and reusability [39, 40] as follows:

- The number of methods in a class reflects the time and effort required to develop and maintain the class.
- The potential impact on children will be greater if there are a larger number of methods in a class since children inherit all the methods of its parent class.
- Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

5.2.1.2 Impact of AOP

AOP may impact the WMC metric as aspects may help reduce the number of methods implemented within a class. Aspects combine cross-cutting functionality into modular units, thus eliminating code that

may be tangled in a core class. Applying aspects will extract aspectual functions. This will therefore reduce the number of tangled methods in a class and in turn reduce the WMC. Aspects may also reduce the WMC in the case where a sub-class overrides its parent's functionality with its own aspectual behaviour. Exception handling may be such an example, where a sub-class may have to replace a super class function with its own method for handling certain exceptions. Thus if the handling of the exception was implemented as an aspect, this would eliminate the need for the subclass to add a new function its own exception handling and therefore reduce the WMC measure.

5.2.2 Depth of Inheritance Tree (DIT)

5.2.2.1 Background

DIT is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes. Thus the root will have a DIT of 0 while a direct child of the root will have a DIT of 1. In figure 26, class *Department* is the root and has a DIT of 0 while class *Clothing* has DIT of 1. From the viewpoint described in [40], DIT measures understandability, reusability, and testability as follows:

- The deeper a class is within the hierarchy, the number of methods it is likely to inherit will be greater making it more complex to predict its behaviour.
- Since more methods and classes are involved, deeper trees also constitute greater design complexity.
- Deeper inheritance trees give a greater potential for reuse of inherited methods.

5.2.2.2 Impact of AOP

AOP may have an impact on DIT measure by reducing the depth of the inheritance tree. This will occur in the case where subclasses are defined only for the purpose of applying their own implementation of aspectual behaviour. Aspects can be used to replace the aspectual behaviour of such subclasses and thus eliminating their need and thus reducing the depth of the inheritance tree.

5.2.3 Number of Children (NOC)

5.2.3.1 Background

NOC is a count of the number of immediate subclasses of a class in the hierarchy. This measure is an indicator of the potential influence a class can have on the design and on the system. From figure 26 class *Department* has a NOC of 2, while a leaf class such as *Electronic* has a NOC of 0. NOC measures efficiency, reusability, and testability as follows:

- The greater number of children, the greater the likelihood of improper abstraction of the parent class and may be a case of misuse of subclassing.
- Since inheritance is a form of reuse, the greater the number of children, the greater the reuse.
- If a class has a large number of children, testing time will increase as the class may require more testing of the methods in the class.

5.2.3.2 Impact of AOP

The impact AOP on the NOC measure is the same as that of the DIT measure described in Section 5.2.2.2.

5.2.4 Coupling Between Objects (CBO)

5.2.4.1 Background

CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related classes on which a class depends. Thus two classes are coupled are when methods declared in one class use methods or instance variable of the other class [40]. CBO aims to measure reusability, maintainability, testability, and understandability. Excessive coupling hinders modular design and prevents reuse. The more independent a class is, the easier it is to reuse. Higher coupling also results in maintenance difficulties as the system is more sensitive to changes. The measure of coupling is useful to determine how complex the testing of various parts of the system is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be. Moreover, strong coupling complicates a system since a class is harder to understand if it is interrelated with other classes. The complexity of a system can be reduced by designing systems with the weakest possible coupling between classes.

5.2.4.2 Impact of AOP

The presence of aspects is likely to decrease the coupling between core classes, but increase the coupling between core classes and aspect classes. This is because aspects are new entities on which core classes depend. Core classes are more likely to be reused than aspects therefore increasing the coupling between aspects and core classes and in effect decreasing the coupling between core classes themselves will be beneficial. As stated before, a lower level of coupling between core classes is likely to foster more modular design and increase reuse.

5.2.5 Response For a Class (RFC)

5.2.5.1 Background

RFC is defined as the number of methods in the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. This includes all methods accessible within the class hierarchy. The RFC for *Department* in figure 26 above is the number of methods that can be invoked in response to a message by itself, by *Clothing*, and by *Electronic*. Thus the RFC for *Department* = 3 [(self) + 1 (*Clothing*) + 4 (*Electronic*) = 8]. The aim of RFC is to measure the amount of communication with other classes. RFC is designed for measuring understandability, maintainability, and testability as follows:

- The larger the RFC, the greater the complexity of the class.
- Testing and debugging also becomes more complicated if a large number of methods can be invoked in response to a message, thus requiring a greater level of understanding on the part of the developer.

5.2.5.2 Impact of AOP

AOP has an impact on RFC as the presence of aspects is likely to increase the RFC of a system. This is because in addition to communicating with other classes, classes also have to communicate with aspects. However the advantage of aspects in this instance is that they can be designed in a way that encapsulates logic and the objects with which a class communicates in a modular way.

5.2.6 Lack of Cohesion in Methods (LCOM)

5.2.6.1 Background

LCOM is the degree to which methods within a class relate to one another. LCOM counts the number of different methods within a class that reference a given instance variable. This metric evaluates efficiency and reusability. High cohesion indicates good class subdivision. Low cohesion increases complexity, thereby increasing the likelihood of errors during development. Also lack of cohesion implies that classes should be split into two or more subclasses with increased cohesion.

5.2.6.2 Impact of AOP

Functionality that crosscuts modules in a system reduces the cohesion of a class. Thus aspects have a noticeable affect on LCOM as aspects increase cohesion by extracting crosscutting behaviour.

5.2.7 Interpretation of C&K Suite

[52] have proposed a set of guidelines as to how to interpret the metrics in the C&K suite. Table 4 below summarises the objective for the values of the metrics.

METRIC	OBJECTIVE
Weighted Metric per Class	Low
Coupling Between Objects	Low
Response For a Class	Low
Lack of Cohesion of Methods	Low
Depth of Inheritance Tree	Low (trade-off)
Number of Children	Low (trade-off)

Table 4: C&K Metric Suite Interpretation Guidelines

However, as indicated in the last two metrics, there is a trade-off with some of the metrics. A high DIT will increase maintainability complexity but also shows increased reuse. Similarly a high NOC will increase the testing effort but will also increase the extent of reuse efficiency. Thus developers must be aware of the relationship between the metrics as altering the size of one can impact areas such as testing, understandability, maintainability, development effort and reuse as shown in Table 5 below. X indicates that a metric has an impact on one of the systems attributes.

	WMC	DIT	NOC	CBO	RFC	LCOM
Understandability	X	X		X	X	
Maintainability	X			X	X	
Reusability	X	X	X	X		X
Testability		X	X	X	X	

Table 5: Effect of Metrics on Object/Aspect Oriented System

6 Evaluation

This chapter provides evaluation of the use of AOP techniques for the development of Java-based real-time systems. The two version of the implementation, AO and OO, are analysed with data relating to the seven enhanced areas of RTJava, as developed in each paradigm, gathered. The C&K metric has been applied to each of the areas in both the OO and the AO versions. Conclusions are then drawn from the analysed will either support or dispute the research question posed. The remaining sections in the chapter provide both a quantitative and qualitative analysis of the effectiveness of AOP when applied to the seven enhanced areas of RTJava with regards to better separation of concerns.

6.1 Metric Results

The following section details the values gathered for each of the C&K metrics when applied to the seven enhanced areas of RTJava. Note that a more detailed description of each of the metrics is available in Evaluation Metrics chapter (chapter 5) of the dissertation. The values of each of the metrics are given as a reduction or increase representing the change that is incurred to the particular metric as a direct consequence of using aspects (i.e. the difference in the metric values between the OO and AO versions of the Simulator's implementation).

6.1.1 Weighted Methods per Class

Weighted Methods per Class (WMC) is a measure of the number of methods implemented within a class. Table 6 shows the results for this metric as gathered for the implementation of the seven RTJava areas in the Simulator.

7 Enhanced RTJava Areas	Difference
Thread Scheduling & Dispatching	0
Memory Management	+2
Synchronization & Resource Sharing	+2
Asynchronous Event Handling	+1
Asynchronous Transfer of Control	+6
Asynchronous Thread Termination	+6
Physical Memory Access	+4

Table 6: Weighted Methods per Class Results

As documented in chapter 5, aspects may help reduce the number of methods in a class. However the opposite has occurred with the results depicting an increase in WMC for the majority of the RTJava

areas. It is interesting to note that the WMC was not reduced by the use of aspects in any of the seven areas. There are several reasons for this increase. Firstly, aspects themselves have methods (or advice such as `around`, `before`, and `after`) that provide the implementation that is to be weaved to the specified joinpoints declared in the aspect. Consequently, any reduction in methods that aspects do provide (through modularising crosscutting concerns) are somewhat countered by the methods present in the aspect itself. For example, this applies for the memory management area where a separate method is required when allocating argument constructor objects to immortal memory. The function of this method is described in chapter 3 of the dissertation. Aspects in this case enable this method to be removed (as discussed in chapter 4), thus reducing the WMC. Yet the aspect itself subsequently introduces a method to provide equivalent functionality of the removed method, hence cancelling out the original reduction.

Secondly, the increase in WMC may be caused when aspects capture crosscutting behaviour but does not eliminate the methods in which this behaviour was contained. This is the case for the synchronization and resource sharing area. Aspects capture the `wait` and `notify` constructs but do not remove the method. Yet additional methods are present in the system due to the methods in the aspects that have to be accounted for. Thirdly, an increase in WMC may be a result of a limitation of AspectJ. As discussed in chapter 4, it does not support the catching of exceptions inside method boundaries. The work-around for this consisted of wrapping exception catching in methods. Thus in addition to the increase in methods caused by aspect methods, there was a further increase due to the wrapping of code that throws exceptions. This is the case in the RTJava areas of asynchronous transfer of control, asynchronous event handling, and physical memory access. Without this limitation, the increase in these areas may not have been as high, but further investigation is required to support this.

6.1.2 Depth of Inheritance Tree

Depth of Inheritance Tree (DIT) is the maximum length from a class node to the root of the tree. Table 7 shows the results for this metric as gathered for the Simulator's implementation of the RTJava areas.

7 Enhanced RTJava Areas	Difference
Thread Scheduling & Dispatching	-1
Memory Management	-
Synchronization & Resource Sharing	-
Asynchronous Event Handling	-
Asynchronous Transfer of Control	-
Asynchronous Thread Termination	-
Physical Memory Access	-

Table 7: Depth of Inheritance Tree Results

AOP, as stated in chapter 5, may reduce the DIT in cases where subclasses are defined only for the purpose of applying their own aspectual behaviour. Thread scheduling and dispatching, as implemented in the Simulator, is the only area that is applicable for this metric. Other RTJava areas do no utilise subclassing for this reason and therefore do not alter the value of the DIT metric. In relation to the threading area, aspects remove the need to subclass when providing specific real-time thread behaviour. Aspects enabled the capturing of initialisation arguments that are the contrasting factor in the two threading subclasses in the Simulator – `SimulationRTThread` and `SimulationPeriodicThread`. Thus by encapsulating this behaviour in an aspect, the need for these subclasses is removed, reducing the DIT by one.

6.1.3 Number of Children

Number of Children (NOC) is the number of immediate subclasses of a class in the hierarchy. The table below (table 8) shows the results for this metric as gathered for the implementation of the seven RTJava areas in the Simulator.

7 Enhanced RTJava Areas	Difference
Thread Scheduling & Dispatching	-2
Memory Management	-
Synchronization & Resource Sharing	-
Asynchronous Event Handling	-
Asynchronous Transfer of Control	-
Asynchronous Thread Termination	-
Physical Memory Access	-

Table 8: Number of Children Results

The NOC metric provides a similar result to that of the DIT metric. In this instance the NOC count for the `RealtimeThread` class (of the `javax.realtime` package) is decreased by two (the number of `RealtimeThread` subclasses that exist in the Simulator). This is a consequence of aspects removing the need for the `RealtimeThread` subclasses as described for the DIT metric in this chapter.

6.1.4 Coupling Between Objects

Coupling between Objects (CBO) is a count of the number of other classes to which a class is coupled. The table below (table 9) shows the results for this metric as gathered for the implementation of the seven RTJava areas in the Simulator.

7 Enhanced RTJava Areas	Difference
Thread Scheduling & Dispatching	0
Memory Management	-2
Synchronization & Resource Sharing	+2
Asynchronous Event Handling	-2
Asynchronous Transfer of Control	+3
Asynchronous Thread Termination	+3
Physical Memory Access	-12

Table 9: Coupling between Objects Results

Chapter 5 describes how AOP is likely to decrease the coupling between core classes, yet increase the coupling between core classes and aspects. This is deemed beneficial as core classes are more likely to be reused. The CBO metric shows the most varying results of all the metrics of the C&K suite with both reductions and increases caused by the use of aspects. The variation of these values is related to the number of different objects each RTJava area requires to reference in its implementation and the number of different classes each area is implemented in. For example, classes which provide physical memory access code are coupled to five separate exception objects that may be thrown during physical memory access. Aspects remove the need to reference these exceptions in the core classes (as described in chapter 4), therefore eliminating the coupling between these core classes and each of the five exceptions thrown. Each of these classes is now coupled with an aspect, which in turn is coupled to these five exceptions (as the aspect provides the exception handling code). In the Simulator, physical memory access occurs in three separate classes. Although these classes are coupled to the aspect, the net result is a reduction in CBO. This reduction would further increase if physical memory access code was present in more classes. The synchronization and resource sharing area in contrast shows an increase in CBO. As the wait and notify constructs are part of an Objects behaviour, modularising this behaviour does not reduce coupling. Yet classes containing synchronization code (two classes in the Simulator) are, in the AO version, also coupled to the aspect (`SynchronizationAspect`). This therefore increases the CBO metric in proportion to the number of different core classes that implement the synchronization functionality. Thus these two cases indicate that CBO does increase coupling between aspects and core classes, while the degree to which they reduce coupling between core classes is dependant on the number of different objects required for the implementation of each RTJava area.

6.1.5 Response For a Class

Response For a Class (RFC) is the number of all methods that can be invoked in response to a message to an object of the class or by some method in the class. Table 10 shows the results for this metric as gathered for the implementation of the seven RTJava areas in the Simulator.

7 Enhanced RTJava Areas	Difference
Thread Scheduling & Dispatching	+2
Memory Management	+14
Synchronization & Resource Sharing	+4
Asynchronous Event Handling	+7
Asynchronous Transfer of Control	+6/+3
Asynchronous Thread Termination	+6/+3
Physical Memory Access	+6

Table 10: Response For a Class Results

Chapter 5 documents how the RFC metric is likely to increase in the presence of aspects. Although this by RFC's definition leads to greater complexity, the positive point is that aspects can encapsulate the logic and the objects with which a class communicates in a modular way. As is evident from table 10 above, the RFC of all seven areas are increased in the AO version of the implementation. This is a direct consequence of the reason stated in chapter 5 (RFC increases because also have to communicate with aspects which did not occur in the OO version of the implementation). The variation of the RFC increases in each area is related to the number of additional methods that aspects introduce in that area. This is because as the number of methods in a class increases (WMC), then the number of responses for a class (RFC) is likely to increase also.

6.1.6 Lack of Cohesion in Methods

Lack of Cohesion of Methods (LCOM) is the degree to which methods within a class are related to one another. The table below (table 11) shows the results for this metric as gathered for the implementation of the seven RTJava areas in the Simulator.

7 Enhanced RTJava Areas	Difference
Thread Scheduling & Dispatching	-
Memory Management	-
Synchronization & Resource Sharing	-2
Asynchronous Event Handling	-
Asynchronous Transfer of Control	-
Asynchronous Thread Termination	-
Physical Memory Access	-

Table 11: Lack of Cohesion of Methods Results

As evident from table 11 above, the LCOM metric is only applicable in the synchronization and resource sharing area. This is the only area that requires the sharing of an instance variable between different methods within a class (which is what the LCOM metric measures). The `updated` boolean variable used for synchronization is referenced by two methods in the Simulator. Aspects enable this `updated` variable to be extracted from the core class, and it is therefore no longer shared among methods within the synchronization code. This in turn reduces the LCOM value for the Simulator.

6.2 Summary of Metric Results

This section provides a summary of the figures gathered above giving a good indication of the comparative impact of each of the seven areas on the attributes of the system. Table 12 summarises the metric results of the evaluation. The affects of AOP on each RTJava area are shown as an increase or decrease in each of the C&K metric values. The changes in metric values, which are a direct cause of using AOP, provides a good indication of the comparative impact that AOP has on the attributes of each area as implemented in the Simulator. The next section describes the affect of AOP on the system attributes in greater detail.

	WMC	DIT	NOC	CBO	RFC	LCOM
Thread Scheduling & Dispatching	0	-1	-2	0	+2	-
Memory Management	+2	-	-	-2	+14	-
Synchronization & Resource Sharing	+2	-	-	+2	+4	-2
Asynchronous Event Handling	+1	-	-	-2	+7	-
Asynchronous Transfer of Control	+6	-	-	+3	+6/+3	-
Asynchronous Thread Termination	+6	-	-	+3	+6/+3	-
Physical Memory Access	+4	-	-	-12	+6	-

Table 12: Summary of Metric Results

6.3 Affect of AOP on System Attributes

This section gives a description the affects AOP techniques have on the specific system attributes in terms of the seven enhanced RTJava areas: understandability, maintainability, reusability, and testability.

6.3.1 Understandability

The Evaluation chapter of this dissertation (chapter 6) describes how the C&K metrics suite measure the understandability attribute of a software system. WMC, DIT, CBO, and RFC are given as the metrics which measure understandability. The figure below illustrates the effects of AOP on the seven RTJava areas in terms of this attribute as evaluated for the Simulator. The metric values again are shown as an increase or decrease to those gathered for the OO version implementation. These changes in metric values are a direct result of using of AOP constructs in the implementation.

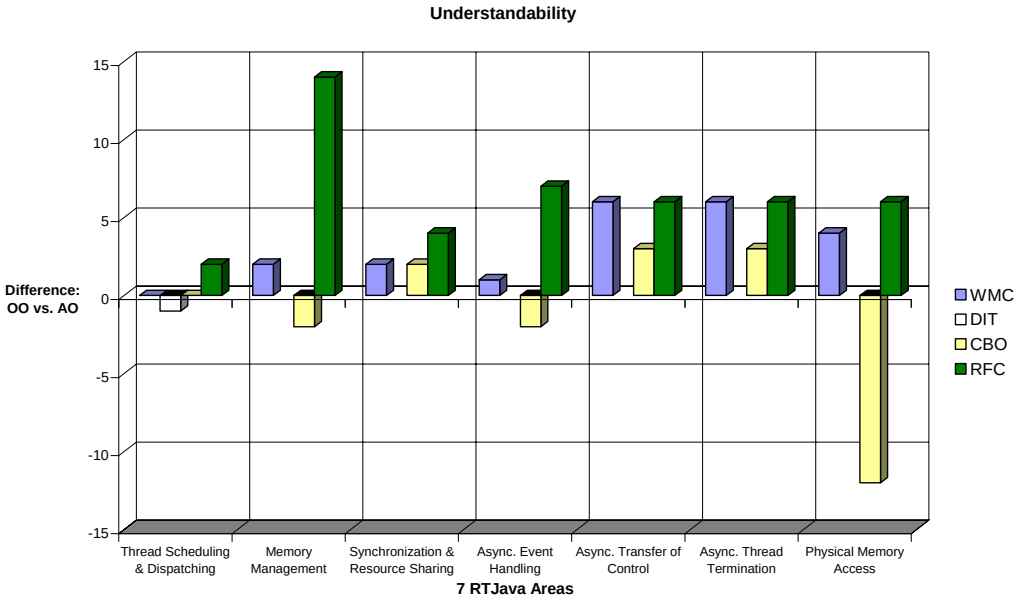

FIGURE 27: EFFECT OF AOP ON UNDERSTANDABILITY

It is evident from figure 27 above that, in most cases, the metrics for evaluating understandability have increased, thus indicating that AOP has an adverse affect in terms of the Simulator's understandability. The level of increase is slightly different for each area, showing that AOP has varying affects on the understandability of each area. It is interesting that in some cases, there are both increases and reductions to metrics. This illustrate that AOP may improve the understandability of an area in one aspect but reduce it in another. This is clearly evident in physical memory access area. The CBO is substantially decreased due to the elimination of coupling between core classes and exception classes, while the RFC

metric is increased due to additional methods present in the system that are contained in the aspect. A similar case applies for the memory management area. In addition, these two memory real-time memory related areas either contain a substantial increase (i.e. for memory management), or decrease (i.e. for physical memory access) in one of the metrics. This may indicate that the overriding affect on understandability in each area will be consistent with the metric value that has altered the most (i.e. in the memory management understandability will become more difficult as illustrated by the increase in RFC). The areas related to asynchrony show similar results for understandability which may indicate that asynchronous functionality is more difficult to understand when implemented using aspects. The remaining areas, thread scheduling and dispatching and synchronization and resource sharing, are less affected in terms of understandability than the other areas when implemented using AOP. However they still show an increase in complexity due to an increase of responses for a class and in the case of synchronization and resource sharing and increase in the number of methods within classes that are also more tightly coupled. Overall, the figures indicate the AOP increases the complexity of the Simulator, and therefore adversely affecting its understandability.

6.3.2 Maintainability

The Evaluation chapter (chapter 6) documents how the C&K metrics suite measure the maintainability attribute of a software system. WMC, CBO, and RFC are given as the metrics which measure maintainability. The figure (figure 28) below illustrates the effects of AOP on the seven RTJava areas in terms of this attribute as evaluated for the Simulator.

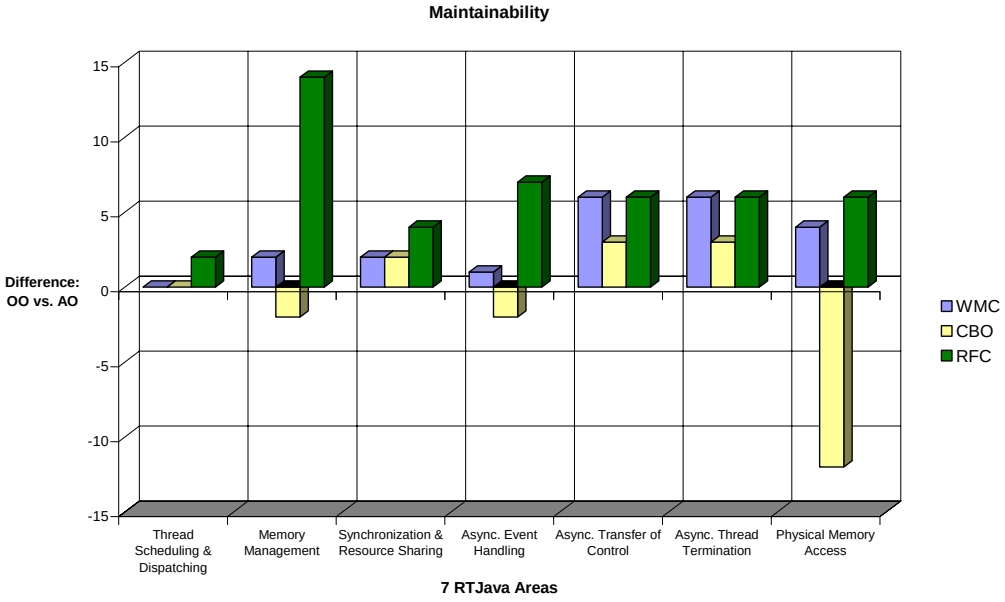

FIGURE 28: EFFECT OF AOP ON MAINTAINABILITY

The results for maintainability, like understandability, indicate that the Simulator is more difficult to maintain with the presence of aspects. A similar insight to the understandability attribute is evident here in that the maintainability of an area may be improved in one way but made more difficult in another. The areas of memory management and physical memory access illustrate this. In addition these two areas again show a substantial increase or decrease in one of the metrics which may be the overriding factor for the maintainability for that area. The areas of asynchrony again show similar results indicating a slight increase in the difficulty in maintaining the Simulator. The remaining areas, thread scheduling and dispatching and synchronization and resource sharing, like understandability, are less affected by AOP but still indicate a more complex system that is more difficult to maintain. Overall the metrics illustrate that AOP adversely affects the maintainability of the Simulator in some manner.

6.3.3 Reusability

The Evaluation chapter (chapter 6) also describes how the C&K metrics suite measure the reusability attribute of a software system. WMC, DIT, NOC, CBO, and LCOM are given as the metrics which measure reusability. The figure below (figure 29) illustrates the effects of AOP on the seven RTJava areas in terms of this attribute as evaluated for the Simulator.

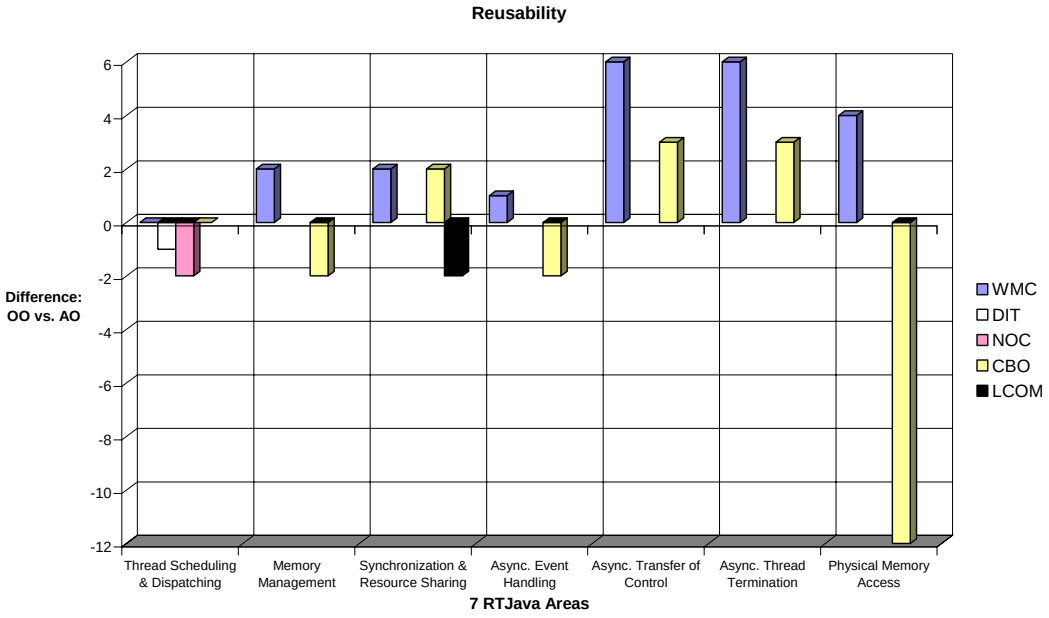

FIGURE 29: EFFECT OF AOP ON REUSABILITY

The results for reusability are more dispersed than those described for the previous two areas. The asynchrony areas in this instance have a greater contrast in results. The areas of asynchronous transfer of control and asynchronous thread termination are less likely to be reusable due to increases in the number

of methods and in the coupling between classes. Asynchronous event handling differs with a decrease in coupling, indicating greater reusability. The thread scheduling and dispatching area also improves this attribute through the use of aspects. Synchronization and resource sharing implemented with aspects results in improvements to reusability in one area while greater difficulty in the two others. The results of the remaining areas, memory management and physical memory access, show an increase in WMC and a decrease to CBO. This raises an interesting point regarding AOP and reusability. In these two areas the increase in WMC is a direct result of methods that are implemented the aspect (, which by C&K's definition make the aspects themselves harder to reuse). The opposite is the case for the core classes. The decrease in coupling is a direct result of core classes being less coupled to other core classes and more tightly coupled to aspects. Also methods in these classes have been removed (and replaced by those in the aspect). Consequently the core classes have become more reusable (due to this reduction in methods and coupling). And since core classes are more likely to be reused [39], the use of AOP in these areas seems to have a positive affect on the Simulator in terms to reusability.

6.3.4 Testability

The Evaluation chapter (chapter 6) also describes how the C&K metrics suite measure the testability attribute of a software system. DIT, NOC, CBO, and RFC are given as the metrics which measure testability. The figure below (figure 30) illustrates the effects of AOP on the seven RTJava areas in terms of this attribute as evaluated for the Simulator.

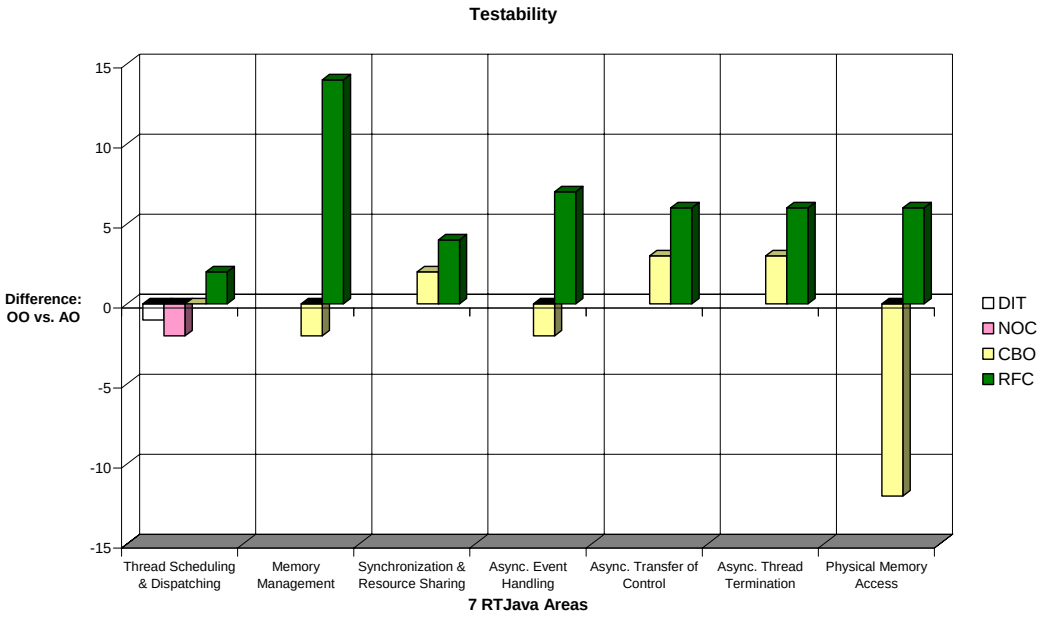

FIGURE 30: EFFECT OF AOP ON TESTABILITY

In terms of testability, the C&K metrics suite does not provide any reference to test suites adopted in their study to measure testability. Instead the suite measures testability based on the view that an increase in metric values described increases the overall complexity, which will therefore in turn increase the testability of a system. The figures depicted for the testability of the Simulator shows contrasting results in three of the areas. In all cases it has been determined that the increase in RFC is directly related to an increase in methods, brought about by using aspects. Where reduced, the coupling is a result of more loosely coupled core classes while where it is increased, coupling not only exists between core classes, but also between these classes an aspects. In these three areas (synchronization and resource sharing, asynchronous transfer of control and asynchronous thread termination), RFC and CBO are both increased. This may suggest that the overall testability of the Simulator is more difficult. This is because, in addition to the core classes, the aspects and their relationship with classes also have to be tested. In the thread scheduling and dispatching area, testing is eased due to a reduction in the number of subclasses. However testing classes containing threading behaviour may now be more difficult because they, in the AO version, need to communicate with aspects. The remaining areas (memory management, asynchronous transfer of control, asynchronous thread termination, and physical memory management) show contrasting values when measuring testability. The reduction in coupling in each case may indicate that the degree of coupling between core classes is reduced, therefore easing the testing of these classes. However, the increase in RFC may suggest that the testing of behaviour that relates aspects and core classes is more difficult.

6.3.5 Discussion

The metric results as evaluated for the attributes above (understandability, maintainability, reusability, and testability), raises some interesting issues regarding the actual benefits of AOP for these areas. Firstly, some of the AOP community have, in many studies, suggested that where aspects improve system modularity, they therefore also improve system attributes [6, 53, 54]. As shown in chapter 4, the modularity of the Simulator has improved through the capturing of crosscutting RTJava functionality. This, by some suggestions, should therefore improve the system in terms of the above attributes. However, the results of the metrics show otherwise, with the majority of the results showing that the use of AOP actually has an adverse affect in terms of system attributes. This in turn raises two interesting questions.

1. The first relates to the OO metrics used and their suitability for evaluating AO software. It may be the case that aspects, while improving modularity, do also improve system attributes. And this improvement is not highlighted in the metric results simply because OO metrics are unsuitable as they do not consider modularity measures when measuring system attributes. Thus further investigation is required into the metrics that are suitable for evaluating AO applications.

2. The second question raised relates to the causal relationship between modularity and system attributes. While some of the AOP community claim that improved modularity, leads to a subsequent improvement in system attributes, the resulted described above raises questions about this causal link between modularity and system attributes. That is while aspects improve modularity, they may not necessarily improve understandability, maintainability, reusability or testability. This supports the findings of [48]. In this study Walker et all raise the question about the relationship between aspects and an their benefits to system attributes. The notion of an *aspect core interface* is defined as the boundary between code expressed as an aspect and the functionally decomposed code. Their findings indicate that in a wider interface (i.e. it is necessary to look at both aspect code and large sections of the core code to understand the aspect code), the causal link between aspects and the benefits they give in terms of system attributes becomes uncertain. Further investigation into these two questions raised is therefore required.

Another interesting issue highlighted by the metric results, relates to the relationship between each of the system attributes. Because the same metrics are used in a more than one attribute evaluation, the results between them are relatively consistent. That is in the measure of understandability, maintainability, reusability, and testability, the results for asynchronous transfer of control and asynchronous thread termination, in each case and for all the metrics used in each measure, are always increased. The areas of memory management and physical access always have contrasting metric values in each attribute measure. The remaining cases are also relatively consistent. This suggests that if the evaluation results indicate that an RTJava area (or any functional concern in any system) is more easily understandable when using aspects, then it is likely to also be more maintainable, more reusable, and more easily tested. The main reason for this is that the same metrics are used in the evaluation of each system attribute.

The final interesting point highlighted by the evaluation results is the relationship between the RTJava areas. For example the two memory related RTJava areas both show an decrease in CBO and an increase in WMC and RFC. This may provide a good indication of the precise area that AOP benefits in these memory related areas. Similarly, the use of AOP in the asynchrony areas always results in an increase in WMC and RFC, thus indicating that AOP may not be of great benefit to the overall asynchrony area.

6.4 Scalability

It is important to remember that the values of the metrics described above reflect the design and implementation of the Sentient Traffic Simulator. The OO version of the Simulator was implemented in 16 classes, with the AO version adding to this number by 7 aspect classes, which by industry standards is a relatively small application. Consequently it is important to describe how the results gathered here may differ if the same process was carried out for a larger application.

It is estimated that in a larger system, the metric values will follow in the same direction (i.e. either increased or decreased) as they did for the Simulator. The values of metrics are estimated to be higher for increased values and lower for reduced values. The proportion of this increase will be dependant on the number of times a certain RTJava construct is used. For example in the memory management area, if one additional object were to be allocated to immortal memory then the metric values of WMC and RFC would increase by one while CBO would decrease by one. For two additional objects increase WMC and RFC by two and decrease CBO by two etc.

Although these figures grow in proportion to the number of usage instances, it is estimated that the modularity benefits provided by aspects will increase in a larger system. The main reason for this belief is that larger systems, in general, exhibit a greater amount of redundant code than evident in the Simulator. Aspects can capture this additional redundancy without incurring any additional costs (such as additional methods, advices, pointcuts etc.).

However a larger system will also increase the amount of aspects needed for capturing crosscutting behaviour. Take for example threading. A larger system may involve a greater variation of real-time threads used. Consequently additional aspects will be required one for each thread variation that exists.

To summarise, the results documented for the Simulator should scale in proportion to the number of instances that RTJava constructs are used throughout. And although certain metrics values will increase indicating a greater level of difficulty in terms of system attributes, the modularisation benefits are also greater.

6.5 Critique

This section critiques some of the findings as noted during the evaluation of the OO and AO version of the Simulator implementation.

6.5.1 Exception Handling

The exception handling functionality in the Simulator raises some interesting issues with regards to AOP for Java-based real-time systems development. Firstly, the development process exposed a weakness of AspectJ in terms of its ability to cleanly capture exception catching code. The problem is that exceptions can be caught and handled in arbitrary parts of the method's implementation and AspectJ does not currently provide support for capturing the catching of exceptions inside method boundaries. The work-around for this limitation, as utilised in the AO version of the Simulator, consisted of wrapping exception catching in methods. This of course is not generally acceptable or practiced in normal programming circumstances except for the purpose of this study. Yet it is due to this limitation that the WMC and RFC metric values calculated above has increased with the use of aspects, as additional methods are

necessary for every variation in exception catching code. Thus if AspectJ overcame this limitation, the aforementioned metrics would not have been so adversely affected.

Secondly, it was common practice in the Simulator implementation to have identical exception catching code wherever particular exceptions were thrown. This therefore reduced the number of aspect pointcuts and advices that were necessary to handle exception catching behaviour. If it were the case that there was a greater variation of exception handling code, then the metrics evaluated above would differ noticeably. Different variations in exception catching code would result in the need for either separate aspects altogether or additional pointcut and advice in an existing aspect. A single addition of either would be necessary for each variation in exception handling code that exists. This will, as a result, increase the values of WMC and RFC. CBO, DIT, and NOC metrics may also increase depending on whether new aspects are used to handle various exception handling cases.

The final issue relates to the notion that aspects may instil laziness in programmers when implementing exception handling code. It is argued that if aspects exist to handle certain exceptions, programmers may simply delegate exception handling to it. This is of course acceptable if the aspect provides the suitable exception catching code. The problem arises if the opposite is true with programmers delegating exception catching to aspects even if the handling implementation is not as precise as is needed or is completely unsatisfactory. Although this temptation for such laziness is greater when using AOP, due to this delegation ability, I believe that this would not be a common occurrence, especially in the real-time domain where critical operations dependant on such exception handling activities may be present.

6.5.2 Real-time Threading

The evaluation of the thread scheduling and dispatching area was based on the Simulator implementation consisting of two types of real-time threads. These two types were sufficient in satisfying the requirements of the system. In addition it was somewhat coincidental that various classes used the same type of thread. The consequence of this was that it enabled aspects to more easily capture crosscutting thread behaviour, thus leading to more favourable C&K metric results in this area. If it were the case that a greater variation of real-time threads were used throughout the Simulator and within classes themselves, developing aspects would have been more difficult. Firstly a different pointcut and advice block would need to the implemented for each thread variation that exists. Secondly, modularising crosscutting thread behaviour would be significantly more difficult. All real-time threads are created with the same constructor call (`new RealtimeThread(..)`), with variations for their behaviour specified by the values of `RealtimeThread` arguments. Consequently capturing and specifying different variations of thread behaviour in an aspect can no longer be done using the constructor call (as described in chapter 4). Thus it would now be necessary to identify the context in which each variation of a real-time thread is created and specifying each of these cases in a separate aspect pointcut. This is clearly

more difficult and error prone as each context in which a thread is created will have to be individually identified and specified and in turn implemented in the aspect. From a C&K metrics perspective the main changes will be in the WMC and RFC values due to the increase of advices needed to encapsulate additional thread variations in the aspect (`RTThreadAspect`). Yet metrics such as CBO will not change. This is because classes containing threads were all previously coupled with the `RTThreadAspect` aspect. Altering the real-time behaviour of a thread in one of these classes will not affect the coupling between the class and the aspect and hence the CBO.

6.5.3 Memory Management

Issues relating to the memory management area of RTJava have also been raised in the dissertation. The main discussion point here is in relation to the allocation of immortal memory. By capturing immortal object creation code, aspects enable all objects to be instantiated in the conventional way (using the `new` operator). This may be beneficial in terms of understandability and maintainability especially for developers unfamiliar with such real-time constructs. However, it is argued that the more long-winded approach, implemented in core classes, provides a better technique for all immortal allocation. This is because it makes it perfectly clear that a particular allocation is from immortal memory. Thus the two opposing views of the argument seem valid, which the choice of which to support dependant on the requirements of the system and developers involved. The second issue relating to this area is the argument of aspects versus the Factory Method design pattern [50] for object creation. The two approaches actually differ quite significantly. Creation of immortal objects using the Factory Method pattern would require the need to reference the class containing the factory method. Object creation would need to be performed by calling the factory method. This is in contrast to the aspect approach which does not introduce the need for interfaces and subclasses as required for the Factory Method. Instead immortal objects can simply be created using the `new` operator. Both approaches provide more modular ways of implementing objects creation, yet there are significant benefits to each approach. Design patterns provide a very efficient means of creating objects and are now widely used in OO applications. Developers are also becoming increasingly familiar in both how and when they should be adopted. Aspects can enable the encapsulating of object creation behaviour hiding implementation details where necessary. Objects can also be created in the conventional manner using `new`. Thus the choice really is dependant on which the developers favour and obviously any requirements of the system relating to this area.

7 Conclusions

This final chapter documents the conclusions that have been drawn from the evaluation results as documented in the Evaluation chapter (chapter 6) of the dissertation. It also presents some of the future work related to the dissertation that has been identified.

7.1 Conclusions from Findings

The original research question that the dissertation aimed to evaluate was the effectiveness of AOP techniques for separation of concerns in the development of the seven enhanced areas of Java-based real-time systems development. Following the evaluation of the two versions (OOP and AOP) of the Simulator, a number of conclusions were derived about AOP and its effectiveness for Java-based real-time systems development and also about the applicability of the C&K metric suite for assessing AO programs.

The conclusions drawn are the following:

- AOP enhancements can improve the modularity of real-time systems.
- The degree of effectiveness of AOP for developing Java-based real-time systems is highly application specific.
- OO metrics shows an apparent contradiction when applied to the evaluation of AO programs.
- The benefits provided by AOP are not as compelling as previous studies.

Firstly the use of aspects in the Simulator has greatly improved its modularity. As they are designed to do, aspects enable the modularisation of crosscutting concerns exhibited by the seven RTJava areas in the codebase, thus eliminating code scattering and code tangling. As described in the evaluation chapter, this can have significant consequences on the system attributes, understandability, maintainability, reusability, and testability. These areas benefit from this improved modularity, which may in turn improve the performance and productivity of development as opposed to the OO model.

Second, as described in the Implementation and Evaluation chapter of the dissertation, the degree of effectiveness of AOP in achieving SOC is dependant on application specifics. This is clearly illustrated in the Simulator's implementation of threading and exception handling (for ATC, asynchronous thread termination, and physical memory access). Only two types of real-time threads are supported in the Simulator, which enables the crosscutting behaviour of each to be easily captured as two aspects. However an increase in the number of different threads would make capturing this crosscutting behaviour less trivial. In such an instance a new aspect would be necessary for each different thread variation. A similar argument holds for that of exception handling. The Simulator's reaction to a thrown

`AsynchronouslyInterruptedException` object is the same in each case. This however may not be a common practice as varying implementations may be provided for exception catching depending on the context that the exception is thrown. If this is the case, the variations in exception handling reactions would, like threading, require a separate aspect for each reaction type. This addition causes C&K metric values to increase, thus reducing the benefits of aspects in terms of understandability, maintainability, reusability, and testability. In theory, aspects are more beneficial if the number of variations in real-time threads or exception handling code is significantly less than the number of places real-time threads are created or where exceptions are caught. When this is so, it indicates that there is a fair amount of redundancy in the code relating to these areas. Consequently the benefits of aspects in terms of SOC would be greater here than if a lower level of redundancy existed due to greater variations in thread or exception handling code.

Third, OO metrics may show an apparent contradiction when applied to the evaluation of AO programs. According to the C&K metrics suite, any increase in the metric values will adversely affect a system in terms of understandability, maintainability, reusability, and testability (i.e. all of these things become more difficult). In the majority of the RTJava areas, aspects result in an increase in these metrics, which as stated should adversely affect system attributes. However, by design, aspects are intended to capture crosscutting concerns and thus improve modularity. This intension is reflected in the Simulator with aspects providing better modularity in all RTJava areas. The resulting improvement in modularity should therefore benefit a system in terms of the aforementioned attributes as shown in previous studies [6, 53, 54]. Thus contrasting views in terms of the above system attributes are evident. While aspects indicate an improvement to systems attributes, the opposite is the case under OO metric evaluation. This raises two interesting questions.

1. Firstly the view of previous studies may be taken: that by improving modularity, understandability, maintainability, reusability, and testability benefits follow. This indicates that OO metrics may be unsuitable for evaluating AO applications. Thus further investigation into metrics suitable for evaluating AO systems is required. In the OO metrics suits there is no relationship between modularity and system attributes (i.e. values for modularity are not considered in the formulae for understandability, maintainability, reusability, and testability). It may be necessary to devise new metrics that take into account modularity measures when evaluating the affects of AOP on system attributes.

2. Secondly, as shown in the results, doubts have been raised about the causal link between modularity and the system attributes above. Thus although AOP improves modularity, according to these metrics, it's failed in its claim to help with understandability, maintainability, reusability, and testability. As stated in the previous chapter (chapter 6), this supports the findings of [48], which also raises questions about the causal link between aspects and the system benefits that they provide. Consequently further investigation is necessary to determine

both the suitability of OO metrics for AO software and also to examine the causal relationship between modularity (aspects) and system attributes which some of the AOP community claim are gained for free through such improved modularity.

Finally, it is evident from the dissertation that the results given here are far less compelling in favour of AOP than in previous studies. It has been highlighted here that AOP is beneficial in terms of modularity, however the degree of this effectiveness if highly dependant on application specifics. The evaluation of this dissertation is based on metrics that are used for evaluating OO designs. This is not the case for previous studies of AOP with evaluation typically based on traditional metrics such as lines of code (LOC). Thus it is difficult to compare previous studies with this dissertation due to the differences in evaluation criterion. However, if traditional metrics such as LOC were used in this evaluation, the AO version would seem better with a noticeable reduction in LOC (219 less LOC). Yet as described, it is evident that AOP does not turn out favourably in terms of the OO metrics. Consequently, this dissertation highlights that the benefits provided by AOP are less compelling than in other studies when evaluated using different means.

7.2 Future Work

This section details some future areas of work that are related to the dissertation.

7.2.1 Contradiction Investigation

During evaluation of the gather results, an apparent contradiction arose relating to the benefits that AOP brings to the system attributes understandability, maintainability, reusability, and testability. There were two main suggestions for the cause of this. One suggestion was that OO metrics may not be suitable for the evaluation of AO software and that new metrics should be devises which account for modularity measures when evaluating the affects of AOP on system attributes. This area thus requires further investigation. The second suggestion was that the results raise questions between the causal link between modularity and system attributes obtained through improved modularity. Therefore this relationship between the two should be more closely examined.

7.2.2 Physical Memory Access Area

The evaluation of the physical memory access area in the dissertation is based on design and sample code which was not fully implemented and tested. Therefore the next immediate step to be covered would be to accurately implement this area into the Simulator and carry out an evaluation on this area and the impact of subsequent results. However it has been noted that results for this should not stray by any significant amount from the evaluation depicted above.

7.2.3 Evaluation for Application Specific Real-time System

As previously illustrated certain areas of the Simulator provide identical implementation. This is the case are areas such as exception handling and threading. An interesting path for future work would be to apply AO techniques to existing Java-based real-time systems which may not share the same degree of code redundancy as the Simulator. Therefore an evaluation could be carried, that may better reflect the current state of the art of Java-based real-time systems.

7.2.4 Aspects for Other Real-time System Programming Languages

Presently most real-time systems are developed using procedural languages such as C, although the adoption of OO languages in this domain is increasing with the growth in popularity of C++ and Java. Therefore adopting the AO approach for other real-time system development languages may be interesting. Aspect languages such as AspectC and AspectC++ [55] could be used for re-implementation of current real-time systems with a subsequent evaluation carried out. The study for each individual analysis could then be contrasted highlighting the varying effects of aspects in each case.

7.2.5 Non Real-time Related Aspects

This dissertation focused solely on real-time related constructs of Java-based real-time systems. Other non functional areas which have been repeatedly studied in AOP research, such as tracing, logging, design by contract techniques, etc. have been ignored in this case. Inclusion of aspects for handling such crosscutting behaviour may further impact the system which current results do not accurately reflect. The significance of these additional aspects would provide an interesting avenue for future work.

7.2.6 Compositional Filters Approach for Real-time Systems Development

The current state of the art in the area shows the use of the compositional filters approach to AOSD for real-time system development. Thus as an alternative to the AOP, other AOSD approaches could be adopted and evaluated to determine their effectiveness for real-time systems development. Compositional filters, hyperspaces, and the growing Concern Manipulation Environment (CME).

7.2.7 Performance Evaluation of Aspects

The performance property is another important characteristic of real-time systems. Although aspects do improve the SOC in Java-based real-time systems, their impact on system performance has not been tested. It is estimated that they should not significantly affect performance as aspect weaving is carried out at compilation. However research should be conducted to test if AOP constructs result in any performance overheads.

8 Bibliography

1. Ossher, H. and P. Tarr. *Using Multidimensional Separation of Concerns to (Re)shape Evolving Software*. in *Communications of the ACM*. October 2001.
2. Kiczales, G., et al. *Aspect-Oriented Programming*. in *ECOOP*. June 1997.
3. Javaworld, *I Want My AOP*. March 2002, www.javaworld.com.
4. *The AOSD Website*, www.aosd.net.
5. Mendhekar, A., G. Kiczales, and J. Lamping, *RG: A Case Study For Aspect-Oriented Programming*. 1997, Xerox Palo Alto Research Centre.
6. Highley, T.J., M. Lack, and P. Myers, *Aspect Oriented Programming A Critical Analysis of a New Programming Paradigm*. 1999, University of Virginia.
7. Burns, A. and A. Wellings, *Real-time Systems and Programming Languages*. Third ed. 2001: Addison-Wesley.
8. Gosling, et al., *The Real-time Specification for Java*. 2000: Addison-Wesley.
9. Reynolds, V., *Sentient Traffic Simulations for an Intelligent Transport System (IST-2000-26031)*, University of Dublin, Trinity College.
10. *TimeSys. Real-time Specification for Java Reference Implmentation.* www.timesys.com/rtj, 2003.
11. *TimeSys. TimeSys Linux/RT 3.0.* www.timesys.com, 2003.
12. Hunt, J., *Real-time Java*. 1998, JayDee Technology Limited: www.PlanetJava.co.uk.
13. Tryggvesson, J., T. Mattsson, and H. Heeb, *JBED: Java for Real-time Systems.* Dr. Dobb's Journal, November 1999.
14. Nissanke, N., *Real-time Systems*. 1997: Prentice Hall.
15. Hayes, R.G., *Real-time Java*. December 2000, Department of Electrical Engineering and Computer Science College of Science and Engineering Loyola Marymount University: Los Angeles, USA.
16. Ive, A. *Implications of Real-time Garbage Collection in Java*. in *Java Virtual Machine Research and Technology Symposium*. November 2000.
17. Dibble, P.C., *Real-time Java Platform Programming*. 2002, Prentice Hall.
18. Nilsen, K., *Issues in the Design and Implementation of Real-time Java*. Java Developer's Journal, 1996. **1**.
19. Corsaro, A. and D. Schmidt. *Evaluating Real-time Java Features and Performance for Real-time Embedded Systems*. in *IEEE Real-Time Technology and Applications Symposium*. 2002. San Jose.
20. Johnson, M., *Real-time Java*. 2002, Christian Albrechts-Universitat zu Kiel.
21. Kolehmainen, M., *Synchronization in the Real-time Specification For Java*. 2001, University of Kuopio.

22. Gosling, J., B. Joy, and G. Steele, *Java Language Specification*. 1996: Addison-Wesley.
23. Nielsen, K. *Thoughts Regarding Sun's Real-Time Specification for Java*. in *The Real-time Embedded Systems Forum, The Open Group*. January 2002.
24. *The Java Community Process.* www.jcp.org.
25. *The National Institute of Standards and Technology.* www.nist.gov.
26. *The Real-time Specification for Java.* Computer Innovative Technology for Computer Professionals IEEE Computer Society, June 2000. **33**.
27. Piszcz, A. and K. Vidrine, *Real-time Java Commercial Product Assessment.* October 2000, MITRE: Washington, USA.
28. Beebee, W. and M. Rinard. *An Implmentation of Scoped Memory for Real-time Java*. in *EMSOFT*. 2001.
29. Rinard, M., *FLEX Compiler Infrastructure.* www.flex-compiler.lcs.mit.edu, 2002.
30. Pahlsson, N., *Aspect-Oriented Programming*. 2002, Department of Technology University of Kalmar: Kalmar Sweden.
31. Kiczales, G., et al. *An Overview of AspectJ*. in *ECOOP*. June 2001.
32. TheAspectTeam, *The AspectJ Programming Guide*. 2001.
33. Kersten, M. *AO Tools: State of the (AspectJ) Art and Open Problems*. 2002. OOPSLA.
34. *HyperJ website.* www.alphaworks.ibm.com/tech/hyperj.
35. Aksit, M., B. Tekinerdogen, and L. Bergmans. *Achieving Adaptability Through Separation and Composition of Concerns*. in *ECOOP*. 1997.
36. Wichman, J., *ComposeJ: The Development of a Preprocessor to Facilitate Compositional Filters in the Java Language*, in *Computer Science*. 1999, University of Twente.
37. Caro, P., *Adding Systematic Crosscutting and Super-Imposition to Compositional Filters*, in *Computer Science*. 2001, University of Twente.
38. *Concern Manipulation Environment website.* www.research.ibm.com/cme/.
39. Zakaria, A.A. and D.H. Hosny. *Metrics for Aspect-Oriented Software Design*. in *Workshop on Aspect-Oriented Modeling with UML*. March 2003.
40. Chidamber, S.R. and C.F. Kemerer. *A Metrics Suite for Object-Oriented Design*. in *IEEE Transaction on Software Engineering*. 1994.
41. Harrison, R., S.J. Counsell, and R.V. Nithi. *An Evaluation of the MOOD Set of Object-Oriented Software Metrics*. in *IEEE Transactions on Software Engineering*. June 1998.
42. Morris, K., *Metrics for Object-Oriented Software Development Environments*. 1989, M. I. T. Sloan School of Management.
43. Deters, M., N. Leidenfrost, and R. Cytron. *Translation of Java to Real-time Java using Aspect*. in *International Workshop on Aspect-Oriented Programming and Separation of Concerns*. August 2001. Lancaster, United Kingdom.

44. Gal, A., W. Schroder-Preikschat, and O. Spinczyk. *On Aspect-Orientation in Distributed Real-time Dependable Systems*. in *5th ECOOP Workshop on Object Orientation and Operating Systems*. June 2002. Malaga, Spain.
45. *The AIRES website.* www.dist-systems.bbn.com/projects/AIRES/.
46. *The FACET website.* www.cs.wustl.edu/~fhunleth/facet/.
47. Aksit, M., et al. *Real-time Specification Inheritance Anomalies and Real-time Filters*. in *ECOOP*. 1994.
48. Walker, R., E. Baniassad, and G. Murphy. *An Initial Assessment of Aspect-oriented Programming*. in *International Conference on Software Engineering*. 1999.
49. Alexander, R. and J. Bieman. *Challenges of Aspect-oriented Technology*. in *ICSE Workshop on Software Quality*. 2002.
50. Gamma, E., et al., *Design Patterns Elements of Reusable Object-Oriented Software*. 2000: Addison-Wesley.
51. Kiczales, G., *Getting Started with AspectJ*. 2001, University of British Columbia and Xerox Palo Alto Research Center.
52. Rosenberg, L.H., *Applying and Interpreting Object Oriented Metrics*. July 2003, Software Assurance Technology Center: www.satc.gsfc.nasa.gov.
53. Lippert, M. and C.V. Lopes. *A Study on Exception Detection and Handling Using Aspect-Oriented Programming*. in *ICSE*. 2000.
54. Hanenberg, S. and R. Unland. *Using and Reusing Aspects in AspectJ*. in *OOPSLA*. 2001.
55. *The AspectC and AspectC++ website.* www.aspectc.org/.