An Evaluation of Aspect-Oriented Programming for Java-based Real-time Systems Development

Shiu Lun Tsang, Siobhán Clarke, Elisa Baniassad
Distributed Systems Group,
Department of Computer Science,
Trinity College Dublin,
Dublin 2, Ireland
{ShiuLun.Tsang, Siobhan.Clarke, Elisa.Baniassad}@cs.tcd.ie

Abstract

Some concerns, such as debugging or logging functionality, cannot be captured cleanly, and are often tangled and scattered throughout the code base. These concerns are called crosscutting concerns. Aspect-Oriented Programming (AOP) is a paradigm that enables developers to capture crosscutting concerns in separate aspect modules. The use of aspects has been shown to improve understandability and maintainability of systems.

It has been shown that real-time concerns, such as memory management and thread scheduling, are crosscutting concerns [5, 6, 9, 11]. However it is unclear whether encapsulating these concerns provides benefits. We were interested in determining whether using AOP to encapsulate real-time crosscutting concerns afforded benefits in system properties such as understandability and maintainability. This paper presents research comparing the system properties of two systems: a real-time sentient traffic simulator and its Aspect-Oriented equivalent. An evaluation of AOP is presented indicating both benefits and drawbacks with this approach.

1. Introduction

When designing and building systems, it is often difficult to arrive at a design that modularises all required system features [12]. More often than not, there are certain features that will not fit well into any class structure we choose [4, 12, 14]. This is particularly true for certain kinds of functionality, such as logging, debugging or handling of real-time constraints. We build a well-formed object-model, and then try to align these features with it, but find we must compromise our design to make the dynamics work.

Design compromises often lead to code that is "tangled" and "scattered" [4, 12, 14]. If two features are tangled, their code is intertwined, and difficult to separate. If a feature is scattered, it means that code related to it appears around the system. Tangling and scattering are summed up in the term "crosscutting", and hurt certain system properties such as understandability, maintainability, testability, and reusability of our systems [15].

Aspect-Orientation (AO) [12] is a paradigm that is intended to let us separate tangled code, and encapsulate scattered code more effectively. AO allows us to express crosscutting behaviour in a separate "aspect" module, affording specification of behaviour that overlays our well-formed object-model without having to compromise it.

Several studies have shown that AO is effective at improving modularity [4, 12, 14]. However, real-time systems are designed significantly differently from non real-time systems due to timing and predictability constraints [2]: programming constructs used in the implementation of real-time systems are generally stricter and more carefully defined than those for non real-time systems. The results of studies on AO with non-real time systems do not necessarily reflect their appropriateness within the real-time domain. There have been studies on the use of AOP for real-time concerns (outlined in Section 5), but none has addressed whether the structural benefits translate to improved system properties.

In this paper, we evaluate how the use of an AO language compares to the use of an Object-Oriented (OO) real-time domain-specific language in terms of its effect on system properties (usability, maintainability, testability and reusability). We chose AspectJ [13] as the AO language, and Real-time Java

(RTJava) [7] as the domain specific language. RTJava provides an extension to conventional Java that addresses seven real-time areas of concern [1]. AspectJ is based on Java, but allows description of crosscutting behaviour in class-like aspect-modules.

We constructed two systems that both implemented a real-time sentient traffic simulation: OOSim, implemented in RTJava, and AOSim, implemented in AspectJ. We identified RTJava code that appeared in OOSim and enclosed it in aspects for the AspectJ version. We then assessed whether the use of aspects resulted in improvements in system properties.

In the next section we provide background information necessary to understand the real-time and Aspect-Oriented content in the paper. Section 3 describes the case study systems used in our work. Section 4 presents the results of our comparison between the aspect- and object-oriented implementations of the real-time simulator. Section 5 presents results from other studies related to our own. Section 6 summarises our experiment and findings.

2. Background

This section describes the two languages used. First, we discuss Real-time Java, describing its scope and the limitations of Java that it aims to address. Then we describe the Aspect-Oriented language, AspectJ, including a discussion of the AO paradigm itself, providing code examples.

2.1. Real-time Java

Java has a number of characteristics that make it difficult for real-time systems programming. Features such as automatic garbage collection and dynamic class loading introduce an unpredictability that breaks the temporal constraints associated with systems of a real-time nature. For example, the garbage collector may run at an unforeseen time leading to delays that may break the temporal constraints of a real-time system.

The Real-time Specification for Java (RTSJ) has directly identified seven distinct areas where Java is limited for developing of real-time systems, providing language enhancements for threading, overriding garbage collection, and handling of external events. The seven areas are:

- Thread scheduling and dispatching
- Memory management
- Synchronization and resource sharing
- Asynchronous event handling
- Asynchronous transfer of control
- Asynchronous thread termination
- Physical memory access

2.2. AspectJ

AspectJ is an Aspect-Oriented (AO) programming language. AO is intended to allow developers to modularise behaviour that does not fit cleanly into an object model.

2.2.1. Structure of AspectJ Program. AO applications consist of a "core" and "aspects". The core is a standard OO program and is implemented in a language such as Java. Aspects are separate modules that describe behaviour that is to be executed at precise locations in the core. These locations are also described in the aspect module. A "weaver" inserts calls to the aspect code into the core at compile time.

AspectJ is an aspect-oriented extension to Java. It provides the *advice* construct for describing code that is to be inserted into the core at locations called *joinpoints*. The code can be inserted *before* or *after* these locations. *Around advice* allows a programmer to specify code that should replace code at a joinpoint. It also provides the *pointcut* construct for specifying at which joinpoints code should be executed.

```
aspect TracingExample {
   //capture the execution of methods in
   //classes named * in methods named *
   // with any parameters in package
   // "example"

pointcut trace() : execution(* *(..)) &&
   within (example);

before():trace(){ //Execute this code
   //before the above point is reached
   System.out.println ("Entering"
      thisJoinPoint.getSignature().getName
      ());
}

after():trace() {//Execute this code
   //after the above point is reached
   System.out.println ("Existing"
      thisJoinPoint.getSignature().getName
      ());
}
}
```

Figure 1: AspectJ Code Snippet

2.2.2. Example. The following is a simple, classic example of encapsulating tracing behaviour using AspectJ. In standard OO programs, tracing behaviour such as `System.out.println` statements are generally intertwined with code requiring tracing

(tangling) and will also appear in multiple methods of the system (scattering). The following code illustrates the encapsulation of tracing code into a singular modular unit – the `TracingExample` aspect.

3. System

To conduct our experiment, we constructed aspect- and object-oriented real-time sentient traffic simulators. We called these AOSim and OOSim respectively. We initially constructed OOSim by evolving an existing sentient traffic simulator to include real-time considerations. We identified real-time code in OOSim and encapsulated it into aspects in AOSim. This section describes the original simulator, outlines the design of OOSim and AOSim, and provides implementation details of both simulators.

3.1. Original Design

The Sentient Traffic Simulator was developed independently by the Distributed Systems Group in Trinity College Dublin. The simulator models a traffic management system that allows vehicles to self-drive along a stretch of four-lane highway. Vehicles are equipped with sensors that allow them to determine speed and positioning information about vehicles surrounding them. Vehicles react to their sensor information: a vehicle must allow emergency vehicles to pass and must reduce speed if approaching a slower moving vehicle.

The design of this simulator is shown in Figure 2. It contains only the basic constructs needed to model the sentient-traffic system.

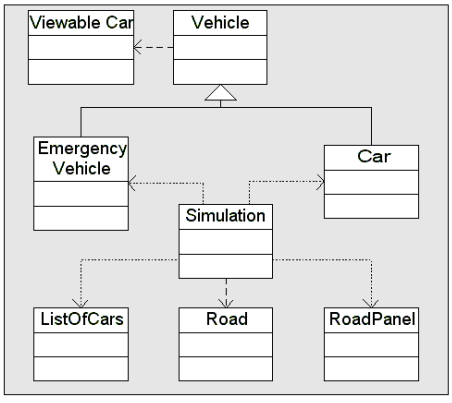


Figure 2: Sentient Traffic Simulator Class Structure

We chose the simulator as the base system for our experiment because it strongly requires real-time functionality to properly model a traffic system. In particular, scheduling and timing are of critical importance, because a vehicle needs the ability to react to any unexpected events in its environment in a timely and controlled manner.

3.2. OOSim Design

To construct OOSim, we identified where real-time behaviour was required in the original simulator, and implemented that behaviour using RTJava. In some cases we replaced functionality, while in others we added new functionality.

We examined functionality such as allowing emergency vehicles to pass, or slowing down, to assess where unpredictable delays occur, and where external events should be catered for. Unpredictable delays occurred in the way threads where handled, and also where objects were allocated to memory and became subject to garbage collection. We re-implemented this functionality using RTJava constructs.

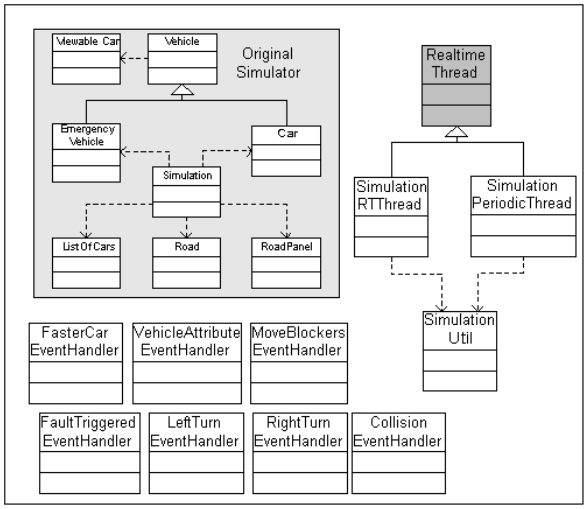


Figure 3: Sentient Traffic Simulator Class Structure with RTJava Constructs

Figure 3 illustrates the class structure of the Simulator that uses RTJava. A number of asynchronous event handler classes were added that are responsible for handling asynchronous events. `SimulationUtil` provides utility functions related to RTJava constructs. Two `RealtimeThread` subclasses, `SimulationRTThread` and `SimulationPeriodicThread` shown in Figure 3, were implemented which provide specific real-time threading behaviour required for the Simulator. Class `RealtimeThread` is shown in grey since it is part of the RTJava API.

3.3. AOSim Design

The aspect-oriented version of the Simulator (AOSim) was constructed by re-engineering OOSim.

For each real-time area, we identified RTJava code that appeared in OOSim. We encapsulated this code into aspects, with one aspect for each real-time area.

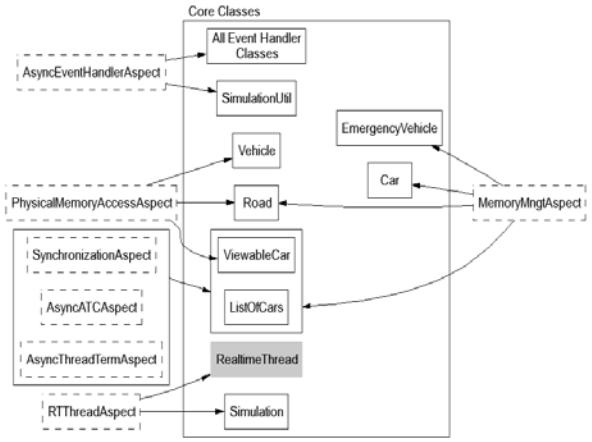


Figure 4: Simulator Aspects and Relationships

Dashed-boxes in Figure 4 depict the aspects used. There is one aspect for each real-time area. Arrows show the core classes into which aspect-code is woven. For example, the `MemoryManagement` aspect encapsulates crosscutting code for the `EmergencyVehicle`, `Car`, `Road`, `ViewableCar` and `ListofCars` classes. Once again, class `RealtimeThread` is shown in grey since it is part of the RTJava API.

- Real-time Threading aspect: encapsulates the creation of real-time threads through subclassing and the starting of real-time threads
- Memory Management aspect: encapsulates the creation of objects that are not subject to garbage collection
- Synchronization and Resource Sharing aspect: encapsulates the waiting, locking, and notifying of objects
- Asynchronous Event Handling aspect: encapsulates the binding of asynchronous event handlers to events, the firing of asynchronous events, and the declaring of `AsyncEventHandler` parent by asynchronous event handler subclasses
- Asynchronous Transfer of Control (ATC) aspect: encapsulates the throwing and handling of `AsynchronouslyInterruptedException`, the handling of interruptible code, and the firing of interrupts
- Asynchronous Thread Termination aspect: as with ATC
- Physical Memory Access aspect: encapsulates the handling of exceptions thrown when accessing physical memory

In AOSim, the locations of crosscutting code became joinpoints and we specified pointcuts to replace their functionality through weaving.

3.4. Implementation

This section describes the implementation of the each real-time area for the sentient traffic system. For each area, we first describe the use of RTJava in the OO system (OOSim), and then describe the AO version (AOSim).

3.4.1. Thread Scheduling and Dispatching.

OOSim: Programmers using RTJava have more control over how threads are scheduled than with conventional Java. The constructor for `RealtimeThread` is as follows:

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

OOSim uses `RealtimeThread` in two ways. The first approach involved subclassing `RealtimeThread`. The second approach uses anonymous inner classes.

Subclassing of the `RealtimeThread` was used for real-time threads with the same behaviour. These threads would have the same instantiation arguments for the `RealtimeThread` constructor. The subclasses are `SimulationRTThread` and `SimulationPeriodicThread`.

In the second approach, methods create real-time threads directly using the `RealtimeThread` as an anonymous inner class. This requires the programmer to specify all constructor arguments each time a thread is created. This was useful when different parameters are required for each individual real-time thread.

AOSim: Aspects were used for creation and starting of real-time threads. Encapsulating code needed for creating real-time threads requires wrapping

each segment of code in a method, and in turn wrapping those methods in advice to describe the RTJava inner class definitions and operations (A similar process is used for handling anonymous inner classes in the asynchronous transfer of control and asynchronous thread termination areas). Additionally, the creation of the `RealtimeThread` object requires a number of different parameters, with each parameter needing to be separately created. The createRTThread aspect captures all real-time thread creation calls and replaces them with parameter creation and initialisation code.

```
pointcut createRTThread():execution(*
   *.createNewSimulationRTThread());

RealtimeThread around():createRTThread()  {
   // thread creation code
}
```

Aspects were adopted for the starting of real-time threads. A feasibility check is performed before starting each one. The `executeRTThread` pointcut captures all thread construction calls. The after advice indicates that a feasibility check is executed after those calls.

```
pointcut executeRTThread():call(*
   *.createNewRealtimeThread(..));

after() : executeRTThread() {
   // code for checking feasibility and
   // starting of the real-time thread
}
```

3.4.2. Memory Management.

OOSim: RTJava introduces three new types of memory areas: physical memory, immortal memory (persistent until JVM termination), and scoped memory (immortal memory within an execution scope). We used immortal memory for all OOSim objects that we did not want garbage-collected: `Car`, `EmergencyVehicle`, `ViewableCar`, `ListOfCars` and `Road`.

We created immortal object memory with and without argument constructors. Creating objects with no argument constructors involved a call to the `newInstance` method of the `ImmortalMemory` class. Objects with argument constructors required a number of additional steps relating to the building of the parameters necessary to create that object. This involved additional operations involving `Class` and `Object` arrays as well as the `Constructor` object.

AOSim: Aspects remove the need for core classes to reference the `ImmortalMemory` object.

```
aspect MemoryManagement {
…
pointcut createViewableCarMemory(String
   thread, boolean isVeh)
   :call(simulation.ViewableCar.new(String,
   boolean)) && args (thread, isVeh);

ViewableCar around(String thread, boolean
   isVeh): createViewableCarMemory(thread,
   isVeh) {
      // building and initialising of
      // ViewableCar parameters and allocation
      // to immortal memory
}
```

This pointcut is an example of immortal memory creation for the `ViewableCar` object. The pointcut captures all of the `ViewableCar` constructor calls in the Simulator, including all the parameters. The implementation specified in the `around` advice replaces the constructors implementation in the `ViewableCar` object. In the `MemoryManagement` aspect there is one pointcut for each immortal object

3.4.3. Synchronization and Resource Sharing.

OOSim: The `ListOfCars` object in the Simulator requires locking because all vehicles access it to obtain information about nearby vehicles. The design and implementation of the synchronization of the `ListOfCars` object was translated into real-time from that of the original simulator.

AOSim: The code below shows the code contained in the `SynchronizationAspect` of AOSim.

```
pointcut updateAction():execution(*
   ViewableCar.*Poll(..));

before():updateAction(){
   // wait and lock object
}

after():updateAction() {
   // notify objects waiting for lock
}
```

The pointcut shown above specifies that all methods with names ending in “Poll” are to be synchronized. The before advice implements the waiting and subsequent locking of the shared resource before the specified method (e.g. `clearPoll`) is executed. The after advice performs the notification after the operations in the captured method have been executed.

3.4.4. Asynchronous Event Handling.

OOSim: In the original simulator, events are generated as a result of sensor information. For example, when an emergency vehicle is behind, a “change lane” event is generated, or when a slower car is ahead and the lanes on either side are busy, a “slow down” event is generated, and so on. OOSim handles

these events as asynchronous events. Asynchronous event handling enables a system to handle events or *happenings* that may occur asynchronously outside of the JVM. A number of asynchronous event handler classes were implemented in the real-time version of the Simulator, as seen in Figure 3. These classes provide the operations that are performed when a particular asynchronous event is triggered or fired. For example, when an exception is raised, it is handled by the `FaultTriggeredEventHandler` class, which performs the necessary operations to handle such faults.

AOSim: Aspects were used to bind events to handlers, fire events and to perform inheritance declarations in asynchronous event handler classes.

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

void around(Class eventClass,
   String bindName) {
   // perform binding and firing actions
}
```

The pointcut shown above captures all calls to the `fireAsyncEvent` method that takes arguments of `Class` and `String` types. The around advice includes the implementation that is executed in place of those in the `fireAsyncEvent` method. All constructs related to the binding and firing of asynchronous events is now contained in the aspect.

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

The `declare parents` declaration specifies that all classes ending in `EventHandler` and in the `eventHandler` package extends the `AsyncEventHandler` class. In OOSim, all asynchronous event handler classes must extend the `AsyncEventHandler` class (and provide concrete implementation for the `handleAsyncEvent` method). This aspect allows the inheritance declaration to appear only once.

3.4.5. Asynchronous Transfer of Control.

OOSim: Vehicle processing based on out-of-date sensor information needs to be interrupted. We used asynchronous transfer of control to throw an exception into the threads performing this processing. We used the *fire* approach, in which blocks of code can be interrupted (each object has a method called `interruptible*()`, for example `interruptibleUpdateViewableCar()`) and Java's exception handling mechanism.

Both mechanisms are implemented in the `ViewableCar` object. Arbitrary decisions were made as to which approach to apply.

AOSim: In the fire method approach, aspects are used to modularise the throwing of the `AsynchronouslyInterruptedException` exception:

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

The `declare soft` statement indicates that all run methods which takes the exception as an argument are required to throw an `AsynchronouslyInterruptedException` (`soft` refers to checked exceptions). This aspect applies to all classes in the `simulation` package as indicated by the `within` construct.

The code below specifies an aspect that handles interruptible code by catching all calls to the appropriate thread's `interrupted` method:

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

after() : catchAIE() {
   // catch AsynchronouslyInterruptedException
}
```

We were able to apply aspects in this way because the same code could be executed each time the exception was caught. If different operations had been required (based on who was throwing the exception, for instance), then a separate aspect would be needed for handling each one.

3.4.6. Asynchronous Thread Termination.

OOSim: The original simulator implemented thread termination by calling the `interrupt` method for normal thread. This is not a reliable way to kill threads because parts of the system affected by this terminated thread may be left in an inconsistent state. We changed this to use RTJava, which provides a means to handle thread termination that uses ATC techniques in conjunction with the `interrupt` method, defined by `RealtimeThread` in the RTJava API, and that allows for clean up (restoring inconsistent data).

AOSim: The implementation for thread termination nearly identical to the implementation for transfer of control, except that after throwing `AsynchronouslyInterruptedException`, a call to the `interrupt()` method is made.

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

after() : catchAIE() {
   // catch AsynchronouslyInterruptedException
   // call to interrupt()
}
```

3.4.7. Physical Memory Access. Hardware constraints kept us from being able to implement physical memory access for OOSim or AOSim. Instead, we provide a theoretical analysis of the design.

OOSim: 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. This would have involved the getting and setting of specific positions and sizes of physical memory locations as well as the throwing and handling of a number of exceptions.

AOSim: We would have used aspects to modularise exception-catching code at each point of physical memory access:

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

after() : catchPhysicalMemoryException() {
   // catch all thrown exceptions
}
```

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.

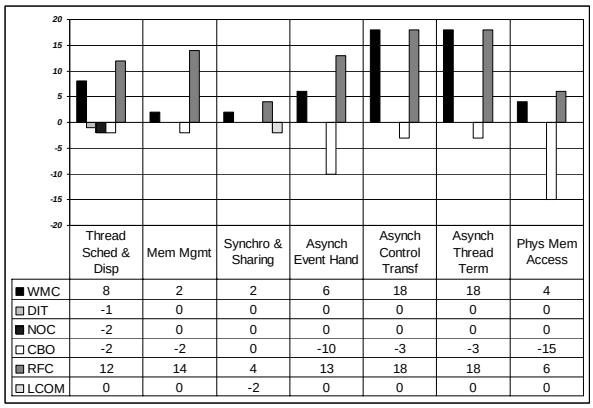
4. Results

This section provides the results for the comparison of OOSim and AOSim.

We used the C&K [3] metrics suite in our evaluation because it provides the most comprehensive and best validated set of measures [10]. We adapted the calculation of each metric for use with aspects.

The application of the C&K suite involves measurement against several metrics, including number of children, and number of methods per class. These *individual results* are then used in combination to assess system properties, such as testability and maintainability.

For each metric we describe how it is calculated, and its individual results, in terms of the change in value from OOSim to AOSim. These values are summarised in Figure 5. We then describe the metrics' combined effect on system properties. Finally, we discuss factors affecting the results.



	Thread Sched & Disp	Mem Mgmt	Synchro & Sharing	Asynch Event Hand	Asynch Control Transf	Asynch Thread Term	Phys Mem Access
WMC	8	2	2	6	18	18	4
DIT	-1	0	0	0	0	0	0
NOC	-2	0	0	0	0	0	0
CBO	-2	-2	0	-10	-3	-3	-15
RFC	12	14	4	13	18	18	6
LCOM	0	0	-2	0	0	0	0

Figure 5: Change in Individual Results due to Use of Aspects[1]

4.1. Individual Results: OOSim-AOSim

4.1.1. Weighted Methods per Class. Weighted Methods per Class (WMC) is a measure of the number of methods implemented within a class. To apply this to AOP, we counted aspects as classes, and advice blocks as methods belonging to aspects.

4.1.2. Depth of Inheritance Tree. Depth of Inheritance Tree (DIT) is the maximum distance from a class node to the root of the tree. DIT only changed in the area of thread scheduling. The OOSim `RealtimeThread` class needed two subclasses `SimulationRTThread` and `SimulationPeriodicThread` to specialise thread instantiation (Section 3.4.1). These classes were not required in AOSim because the specialised behaviour was moved into the `RTThreadAspect` aspect.

4.1.3. Number of Children. Number of Children (NOC) is the number of immediate subclasses of a class. There are two fewer child classes in AOSim, reducing the NOC by two. As was described for DIT, the `RealtimeThread` class in AOSim did not require subclassing.

[1] The values of each of the metrics are given as a reduction or increase representing the change that is incurred by the particular metric as a direct consequence of using aspects. Thus, a negative value indicates a reduction due to the use of aspects, whereas a positive value indicates an increase.

4.1.4. Coupling Between Objects. Coupling between Objects (CBO) is a count of the number of other classes from which elements are used i.e. calls or attribute accesses between classes. To apply this to aspects, we considered aspects coupled to classes only if the aspects explicitly name the classes. For instance, if we have the joinpoint `call(* *(..))`, then the aspect is not coupled to any classes. However, if we have the joinpoint `call(example.Test.methodName(..))`, then the aspect is coupled to `Test`. In five of seven real-time areas (all except synchronization and resource sharing), the CBO value has decreased. In the synchronization and resource sharing areas there was no change in coupling.

4.1.5. Response For Class. Response For Class (RFC) is the number of methods that can potentially be executed in response to a message received by an object of a class. When a call is made to a method that is affected by an aspect, that class will invoke code described in the aspect. These core-to-aspect invocations are counted when calculating RFC. RFC was increased in all real-time areas.

4.1.6. Lack of Cohesion of Methods. Lack of Cohesion of Methods (LCOM) is the degree to which methods within a class are related to one another in terms of shared variables. To adapt this for AO evaluation, we considered pointcuts and advice blocks to be methods. Only the synchronization and resource sharing real-time areas showed change to the LCOM, since in OOSim these classes require multiple methods to share an instance variable, while in AOSim they do not.

Table 1: Combination of Individual Results for System Properties

		Weighted Methods /Class	Depth of Inheritance Tree	Number of Children	Coupling Between Objects	Response for a Class	Lack of Cohesion of Methods
		WMC	DIT	NOC	CBO	RFC	LCOM
	Avg:	+8.28	-0.14	-0.28	-5	+12.14	-0.28
Understandability	+3.82	✓	✓		✓	✓	
Maintainability	+5.14	✓			✓	✓	
Reusability	+0.51	✓	✓	✓	✓		✓
Testability	+1.68		✓	✓	✓	✓	

4.2. Metrics Combined into System Properties

✓ in Table 1 shows which individual results are used when assessing system properties. Understandability relies on WMC, DIT, CBO, and RFC. These individual metrics all contribute to the effort required on the part of the developer to understand a code base. If the depth of the inheritance tree is high, for instance, it may take more effort to determine which attributes and behaviour are passed down to a given class. Maintainability relies on metrics that affect the changeability of the code base: if coupling is high, it will be more difficult to scope the ramifications of a change. Nearly all metrics are taken into account to forecast the effort required to reuse a portion of a code base; the more dynamic metric, RFC, however, is not taken into account because control flow is not as great a factor for reuse. Testability is indicated by all metrics contributing to the behaviour of a system including the RFC individual result: if more calls can result from calls to methods in a class, then the effort required to test that class will be increased.

Table 1 also shows the average change from OOSim to AOSim for each individual metric. Next to each system property is the average change for its relevant individual results. As before, a negative number indicates a benefit of AOSim over OOSim; a positive number indicates that AOSim was worse.

AOSim showed the greatest improvement in modularity (measured by a combination of LCOM and CBO [8]). Overall each of the system properties was hurt in AOSim. Understandability and maintainability suffered more than reusability and testability as they rely on both WMC and RFC, the two worst affected individual results.

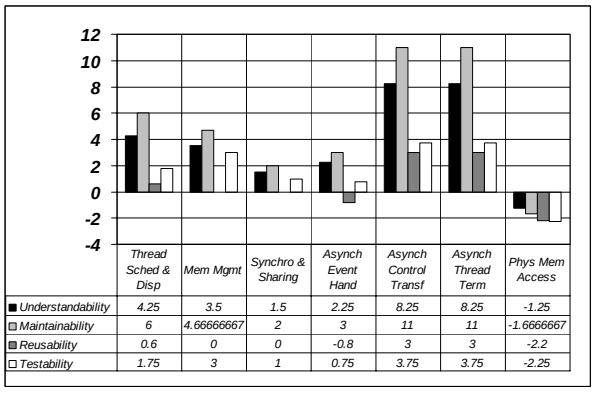


Figure 6: Average change over all system properties for real-time areas

4.3. Real-time Areas and System Properties

AspectJ showed greater impact in some real-time areas than in others. As is depicted in Figure 6, it was, on average, better for physical memory access where each of the system properties were improved and can be attributed to decreased coupling.

The remaining areas, on average, fared worse, due in large part for their poor performance in the WMC and RFC metrics.

4.4. Discussion

The results from Table 1 and Figure 6 show that some metrics fared better than others. For example, system properties declined in physical memory access but increased by varying amounts in all other areas.

Not surprisingly, modularity (LCOM and CBO) is never negatively affected by the use of aspects. By design, aspects result in code with greater modularity. We discovered that the use of wildcards (* construct) can maximise modularity improvements. They eliminate the need for explicit naming and hence further reduce coupling and cohesion. We found it beneficial to make design choices that will allow more generic method naming in the core, and allow wildcards to be used in aspects. Examples of such choices are: limiting the number of types of threads, or keeping uniform the way in which exceptions are handled for asynchronous transfer of control. The greater the code duplication, the greater the modularity benefits as provided by wildcards.

We found that encapsulating concerns into aspects if they are not crosscutting leads to more methods (WMC) and more complex control flows (RFC). This adversely affects system properties. Memory management and asynchronous transfer of control are examples of this. Our results suggest that delaying the encapsulation of RTJava concerns into aspects until crosscutting is evident will reduce the impact of this.

5. Related Work

We compare our work to two kinds of research: comparisons between AO and OO real-time systems, and the use of aspects in real-time research.

5.1. AO Comparisons to OO

A number of studies have shown AO to be significantly more effective than OO for system development [4, 12, 14]. The formats of these studies have all involved an initial development using OO and a subsequent redevelopment of the same system in AO. Evaluation has mostly been based on traditional metrics such as lines of code and sites of change, with the AO version of the applications being a clear winner in each study. For example in [12], Kiczales et al. illustrate how AO was adopted to significantly reduce the LOC in their image processing system, which in turn lead to benefits in reusability and maintainability due to reduced code tangling. The main difference between these studies and ours relates to the metrics used and the non real-time nature of the systems adopted for their evaluations.

jRate is an open source ahead-of-time compiled implementation of the RTSJ that is developed using AO techniques [5]. AspectJ and Aspect C++ were used to capture concerns such as memory, real-time threads, and asynchrony. Its design and performance is compared to other RTSJ implementations developed using OO. Their evaluation was based on the RTJPerf benchmarking suite [5], which contains a number of tests designed to measure the design and performance of RTSJ implementations. In terms of performance, jRate was found to be both more efficient and predictable than the RTSJ reference implementation. These gains were attributed to the use of aspect languages, since they are better able to deal with static and dynamic crosscutting concerns. This work contrasts ours due to its focus on RTJava performance over software engineering issues such as system properties.

5.2. Aspects Applied to Real-time Systems

[9] documents the methodology and benefits of applying AOP to address non-functional requirements such as distribution, real-time, and fault tolerance. Aspects for each requirement were developed using AspectC++. The research introduces a real-time aspect to handle temporal constraints posed by the real-time domain. 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. 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. This work is complementary to ours, since our metrics do not address predictability of systems, whereas this work does not address system design issues.

[6] describes work on automating the translation of Java code into scope aware RTJava code (i.e. automating the creation of RTSJ memory scopes). A *reference-probing* aspect is used to determine the legal RTSJ `ScopedMemory` assignments. Some of the benefits derived from AO included more modular code, lower development costs, and better real-time predictability. Though they found benefits of using aspects, they did not assess the impact that aspects have on code understandability and maintainability due to the indirection added by aspects.

6. Summary

The aim of our study was to evaluate the effectiveness of AOP techniques for separation of concerns in the development of the seven enhanced areas of Java-based real-time systems development. We applied the C&K Metrics suite to assess and compare an AO and OO system in terms of system properties.

We found that AOSim improved modularity over OOSim. This is indicated by the reduction in coupling (CBO) and cohesion (LCOM) in all of the seven RTJava areas. We also found that WMC, and RFC individual results were hurt. These individual results negatively affected all of the system properties, with the greatest negative impact on understandability and maintainability.

We found that for real-time systems, the greatest gain can come from making the aspect-core relationship generic and broad. We suggest minimizing the number of kinds of real-time threads, and maximising the amount of redundant code pulled into a single aspect.

7. Acknowledgements

We would like to thank Vinny Reynolds for supplying the original sentient traffic system. As well, we would like to thank Andrew Jackson for help with Aspect-encapsulation, and for comments on earlier drafts of this paper. Finally, we would like to thank the anonymous reviewers for their comments.

8. References

[1] Bollella. G., et al. *The Real-time Specification for Java*. Addison-Wesley, 2000.

[2] Burns, A., and Wellings, A. *Real-time Systems and Programming Languages: Ada 95, real-time Java and real-time POSIX*, Harlow: Pearson Education, 2001.

[3] Chidamber, S.R., and Kemerer, C.F. “A Metrics Suite for Object-Oriented Design” in Proceedings IEEE Transaction on Software Engineering, Vol.20, No.6, 1994, pp. 476-493.

[4] Coady, Y., and Kiczales, G. “Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code” in Proceedings International Conference on Aspect-Oriented Software Development (AOSD’03), Boston Massachusetts, USA, 2003, pp. 50-59.

[5] Corsaro, A., and Schmidt, D. “Evaluating Real-time Java Features and Performance for Real-time Embedded Systems” in Proceedings Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)**,** San Jose, California, USA. 2002, pp. 90-100.

[6] Deters, M., Leidenfrost, N., and Cytron R. “Translation of Java to Real-time Java using Aspect” in Proceedings International Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster, United Kingdom, 2001, pp. 25-30.

[7] Dibble, P. *Real-time Java: platform programming*, Prentice Hall, 2002.

[8] Fairley, R. *Software Engineering Concepts*, McGraw-Hill Series in Software Engineering and Technology, McGraw-Hill Book Company, 1985.

[9] Gal, A., Schroder-Preikschat W., and Spinczyk, O. “On Aspect-Orientation in Distributed Realtime Dependable Systems” in Proceedings Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS ‘02) , San Diego, California, USA, pp.216-270.

[10] Harrison, R., Counsell S.J., and Nithi R.V. “An Evaluation of the MOOD Set of Object-Oriented Software Metrics” in Proceedings IEEE Transactions on Software Engineering, Vol.24, No.6, 1998, pp. 491-496.

[11] Holmes, D., Noble, J., and Potter, J. “Aspects of Synchronisation” in Proceedings Technology of Object-Oriented Languages and Systems (TOOLS Pacific 25) Pacific, Melbourne, Australia, 1997, pp 7-18.

[12] Kiczales, G., et al. “Aspect-Oriented Programming” in Proceedings European Conference for Object-Oriented Programming (ECOOP’97), Jyväskylä, Finland, 1997, pp. 220-242.

[13] Kiczales, G., et al. “An Overview of AspectJ” in Proceedings European Conference for Object-Oriented Programming (ECOOP’01), Budapest, Hungary, 2001, pp. 327-353

[14] Lippert, M., and Lopes, C. “A Study on Exception Detection and Handling Using Aspect-Oriented Programming” in Proceedings International Conference on Software Engineering (ICSE’00), Limerick Ireland. 2000, pp. 418-427

[15] Tourwé, T., Birchau, J., and Gybels, K. “On Existence of the AOSD-Evolution Paradox” in Proceedings International Conference on Aspect-Oriented Software Development Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, USA, 2003.