Context Aware J2ME Applications With AspectJ, an investigation

Jerry Kiely

May 11, 2005

Abstract

This report is an investigation into the use of AOP, and specifically AspectJ, in the J2ME domain. It details the design and implementation of a J2ME *MIDlet* that uses AOP both to extend it's functionality and to adapt to it's changing context. It concludes by evaluating AOP in the context of J2ME, and suggests further research.

Acknowledgements

I would like to express my gratitude to Siobhan Clarke, my project supervisor, for her support and valued input. I would also like to give thanks to Darragh Curran for his editorial input and advice on matters grammatical. Also I would like to thank Hugh Gibbons for his help, encouragement and friendship over the years.

I would like to dedicate this work to Lauren, my beautiful daughter, and to Don Van Vliet.

Jerry Kiely
University Of Dublin, Trinity College
May 11, 2005

Contents

List of Figures

List of Tables

Preface

Over the last few years mobile devices have become increasingly ubiquitous. As a consequence of this, ever more demands are being placed on them. On top of this, software development release cycles have become increasingly tighter, to accommodate fluctuating demands. The industry, by necessity, is moving further and further away from older, monolithic waterfall style methodologies in order to keep up with the demands of an ever more technically literate consumer base.

New methodologies and technologies are emerging to accommodate this shift. From the agile processes (including *eXtreme Programming*) to *Aspect Oriented Software Development*, there is a palpable shift in the industry towards making projects more *adaptable* to changing requirements.

Coupled with this is a move towards a new type of software, one which is capable of *adapting* to its environment, and changing the way it behaves to accommodate it's changing context. Software that can sense that it's location is changing, or that it's memory is running low, or that it's network connection has dropped, and can change it's functionality accordingly.

This report represents an investigation into the problem of developing context aware / adaptive programs for mobile devices. It attempts to explain what it means for a program to be adaptive, and attempts to evaluate AOP (specifically *AspectJ*) in the context of Java's Micro Edition. It also attempts to demonstrate how the use of AOP can make a project more amenable to change.

Chapter 1

Introduction

1.1 Overview

The objective of this project was to develop a *context-aware* application for a small device, specifically a mobile phone / handset. To that end the project involved an investigation into two technologies, namely Java's Micro Edition (J2ME) and Aspect Oriented Programming, and to evaluate the usefulness of using both to develop applications that are both *context-aware*, and dynamically extendable.

There follows a brief description of the technologies and approaches used in this project. I set out by briefly describing J2ME in terms of it's component parts and it's limitations. I continue by describing what it means for an application to be adaptive, by referring to research on the subject of Adaptive Software, and discussing good style principles (specifically the Law of Demeter). I conclude with an introduction to Aspect Oriented Programming.

1.2 J2ME

J2ME is defined as a platform for limited devices, such as mobile phones. Unlike J2SE and J2EE it is intended for a wide variety of different devices, each with different specifications. As such J2ME is made up of a number of different platforms comprising *Configurations* and *Profiles*.

1.2.1 Configurations

J2ME is currently made up of 2 *configurations*, each representing different types of devices, with slightly different specifications.

Each configuration is made up of a virtual machine and a core set of classes. The virtual machines implement a subset of the core Java Virtual Machine specification. For example, because mobile phone hardware would not support floating point operations, the VM would not need to support `float` and `double` types, or any instructions that make use of those types. The core set of classes are based as much as possible on the corresponding J2SE classes.

CLDC

The *CLDC* (Connected Limited Device Configuration) is the configuration intended for mobile phones and the like. It is designed for devices with small amounts of memory (typically of the order of .5MB) and less powerful processors. It is the configuration most closely associated with Wireless Java.

CDC

The *CDC* (Connected Device Configuration) is the configuration intended for devices such as set top boxes that have slightly more memory (typically of the order of 2MB) than mobile phones, and more capable processors. It lies somewhere between the CLDC and J2SE.

1.2.2 Profiles

A *profile* adds to a configuration by providing extra classes that support specific devices. Example profiles include;

- Mobile Information Device Profile (MIDP)
- PDA Profile (PDAP)
- Foundation Profile

- RMI Profile
- Game Profile

MIDP is aimed at mobile phones and adds support for networking and UI development amongst other things. PDAP is aimed at PDAs with greater available memory and displays. The Foundation Profile adds core J2SE 1.3 libraries to the CDC. The RMI Profile adds J2SE Remote Method Invocation libraries to the Foundation Profile. The Game Profile will provide a platform for developing games on CDC devices.

As can be seen, some of these profiles are dependent on other profiles, or on the configurations they are based on directly.

MIDP

MIDP is an extension of the CLDC designed for devices with small footprints, small processors and limited displays, like mobile phones or pagers. MIDP applications are called MIDlets. MIDlets must extend the following abstract class;

```
javax.microedition.midlet.MIDlet
```

This becomes the main entry point for the MIDP application.

The implemented methods `startApp()`, `pauseApp()` and `destroyApp()` are provided so the environment can control the lifecycle of the MIDlet. Also the class `MIDlet` provides methods that the MIDlet can invoke to gain information about it's environment at runtime.

A MIDlet is packaged, together with it's dependent classes and run-time resources (for example, image and configuration files), into a *jar* (Java ARchive). The jar file's *manifest* must contain information about the contents of the jar. Another important file is the *jad* (Java Application Descriptor) file. This is stored separately from the jar file, and is also used to describe the contents of the jar. Jad files are very small (in comparison to the jar files they describe) and allow you to query the contents of the jar file before choosing to download it to the mobile device for installation and execution.

```
public class HelloMIDlet extends MIDlet implements CommandListener {

    private Form mMainForm;

    public HelloMIDlet() {
        mMainForm = new Form("HelloMIDlet");
        mMainForm.append(new StringItem(null, "Hello, Jeremiah!");
        mMainForm.addCommand(new Command("Exit", Command.EXIT, 0));
        mMainForm.setCommandListener(this);
    }

    public void startApp() {
        Display.getDisplay(this).setCurrent(mMainForm);
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

    public void commandAction(Command c, Displayable s) {
        notifyDestroyed();
    }
}
```

Figure 1.1: A simple MIDlet

1.3 Adaptive Programming

The object of this project was to design an application that could *adapt* to it's changing context. But what exactly does this mean? Are we referring to changing requirements at development time? When we talk about software that adapts, do we mean software that is designed to be amenable to architectural, or coding changes? Code that doesn't become overly fragile and dependant on a choice of pattern, middleware or framework? Or are we referring to changing behaviour at runtime? Are we referring to an application who's functionality changes according to it's physical context?

1.3.1 What is Adaptive Software?

In his book [1] on Adaptive Programming, Lieberherr answers the question "what is adaptive software?" with the following definition; "A program is called *adaptive* if it changes it's behavior according to it's *context*". I have listed a few of these contexts, which are relevant to this project, that a program can adapt to;

Run-time environment an adaptive program can improve performance by adapting to it's environment

Software architecture an adaptive program is not bound to a specific set of components or a specific software architecture

Computational resources an adaptive program is not bound to any specific resources

I will briefly describe these in more detail, and in a way that is relevant to the task at hand.

Environment

When you develop for hand held devices, or embedded devices, you find that resources are at a premium. The ability to monitor these resources and to adapt to the changes in usage over the course of program execution is of the utmost importance, in order to ensure the smooth running of the application.

On top of this, the application potentially shares it's environment with a number of other running processes, each of which may be vying for the same precious memory, etc. An application that demonstrates an awareness of this, and that has the ability to adapt to this is an important step forward.

Architecture

One of the biggest problematic issues experienced in software development projects is the effects of *early binding* to specific systems (Database Management Systems, Application Servers, etc.) or architectures. In most cases,

decisions made early on in a project have a habit of coming back and biting you further down the cycle. Usually this expresses itself as an unnecessary form of vendor or architecture tie in, locking you in to a specific platform, reducing the ability to support other platforms, and has an impact on the overall maintainability of the system.

The ability to leave these kinds of decisions to later on in the cycle would greatly improve the development process, and make for more portable (platform agnostic) and maintainable code.

Resources

To illustrate this point it is useful to consider a simple example; in GUI development there is a tendency to tightly couple presentation logic with business logic, resulting in code that doesn't port easily to other devices / environments. The ability to decouple the presentation logic (device specific) from the business logic (device independent) would be a tremendous boon.

In fact, the ability to treat the two entirely separately, and to be able to swap specific presentation layers in and out at either compile time or run time would be a tremendous advantage.

1.3.2 The Law Of Demeter

In order to successfully develop adaptive software, code modules need to be sufficiently loosely coupled. In order to ensure this loose coupling the question is are there any obvious style rules that should be observed? Or even when deciding to adapt an existing application is there a set of criteria to help evaluate the amenability to adaptation (the adaptiveness) of an application?

In answer to the question posed, we move on to the Law of Demeter, or more specifically, the *law of demeter for methods*, which can be summarised by considering that any method of any object should only call methods of the following objects;

1. itself
2. the parameters of the method

3. objects created within the method

4. attributes of the object

Objects should make little or no assumptions about the implementation of the objects they come into contact with, other than their exposed API. To put it plainly, and object shouldn't *ask* another object about it's internals in order to perform a particular task, but should *tell* it to perform the task, and not care about, or be exposed to, the other objects inner implementation.

```
public class MoneyProcessor {

    public Money getTotalsAndDoSomething(Customer customer) {

        Money amount = new Money();
        Iterator iterator = customer.accountIterator();
        while(iterator.hasNext())
            amount.add(((Account) iterator.next()).getTotal());
        // do something to the total...
        return amount
    }
    // ...
}
```

Figure 1.2: Example code that doesn't obey the LoD

In figure 1.2 we see a simple method that calculates the totals of all accounts belonging to a Customer and performs some calculation on that total. It makes a lot of assumptions about how a Customer object is implemented. In figure 1.3 we see an example that makes no assumptions about how the Customer object implements it's Accounts. Should the Customer's implementation change in any way, the method in figure 1.2 would need to be changed. So by applying the law of demeter for methods, as in figure 1.3, we have reduced the coupling between the above method and the Customer.

This approach was originally formulated [2] with respect to good design principals as applied to Object Oriented Programming languages, but applies equally to the context of adaptive programming, insofar as an application of

```
public class MoneyProcessor {

    public Money getTotalsAndDoSomething(Customer customer) {

        Money amount = customer.getAccountTotals();
        // do something to the total...
        return amount
    }
    // ...
}
```

Figure 1.3: Example code that obeys the LoD

the above rules both ensures that you reduce entanglement within your code at development time, and serves as a means of evaluating an application for adaptability post development.

1.3.3 What does Adaptive Programming give you?

The law of demeter has been extended to include *concerns*. Applied to the context of Adaptive Programming, it ensures a clean seperation of concerns, which results in code that is more adaptive. One of the approaches to achieve this is to use *traversals*.

Consider the following example of a method that *appends* the names of a collection of `Location` objects to a *List*;

```
public void init() {

    Enumeration enum = _model.getLocations().elements();
    while(enum.hasMoreElements()) {
        Location location = (Location) enum.nextElement()
        append(location.getName(), null);
    }
}
```

Figure 1.4: init() method: initial implementation

In figure 1.4 the `init()` method enumerates over the collection of `Location`

objects and appends each of their names in turn. The method knows way too much about the the way the model stores `Location` objects

```
public void init() {

    _model.iterateOverLocations(new Executable(){

        public void execute(final Object obj) {

            append(((Location) obj).getName(), null);
        }
    });
}
```

Figure 1.5: init() method: a simple traversal

We can see a simple example of a *traversal* in figure 1.5. It uses an *executable block* implemented as an *anonymous inner class* (used analogously to a closure in *smalltalk*). The calling object has no knowledge of the underlying implementation of how `Location` objects are stored. This enforces *concern-shyness* between the objects and ensures the underlying implementation (in this case, how the model keeps track of `Location` objects) is free to change, safe in the knowledge that it shouldn't affect objects that make use of it.

1.3.4 Summary

To sumarize, Adaptive Software is software that embraces change, both at development time and at runtime. It is software that is designed to be amenable to change. It is software not committed to a particular platform, framework or approach.

1.4 Aspect Oriented Programming

So how best to express these adaptations? We would like to be able to define adaptive behaviour in a modular manner, and in a way that allows us to

change our implementation as we see fit and as we need. Also we would like these adaptations to be re-useable, either across an application, or across projects.

Emerging from Xerox Parc in the mid nineties, Aspect Oriented Programming (AOP) was the result of the work of Gregor Kiczales and his group. It's aim was to provide a clean way to separate the *core concerns* of an application from it's *cross-cutting concerns*.

The core concerns of an application represent it's functional, or primary purposes. The cross-cutting concerns of an application represent the non-functional, or secondary purposes. For example, among the core business concerns of a CRM (Customer Relationship Management) application would be the ability to retrieve information relating to a specific customer, or the ability to update information relating to a specific customer, and among the cross-cutting concerns would be authorization, persistence, logging and synchronization modules.

We will now speak a little about AOP under the following headings (it's core concepts), and illustrate them with examples from AspectJ;

1. Join Points
2. Pointcuts
3. Advice
4. Aspects

1.4.1 Join Points

Join points are identifiable points in a programs execution. They are made up of, but not limited to, the following categories;

- Constructor Join Points
- Method Join Points
- Field Access Join Points

- Exception Handling Join Points
- Class Initialization Join Points
- Object Initialization Join Points

Join points are where the cross-cutting concerns can be added / woven and the code augmented. Each join point has an associated context. For example, a method invocation includes the *caller object*, the *target object* and any *arguments* passed to the method. Some example join points are listed in figure 1.6.

```
public HelloMIDlet() {}

public String getMessage() {
    return _message;
}

public void setMessage(String message) {
    _message = message;
}
```

Figure 1.6: Example Join Points

It is important to qualify the above with the notion of *exposed* join points. The exposed join points are the subset of all the possible join points that are made available to be captured by an AOP implementation.

In order to understand why an AOP implementation would choose to support this subset we shall consider a loop. A standard loop can be implemented as a `for` loop or a `while` loop. You could decide to capture the `for` loop's join points in an aspect. But if you decide to change the implementation from a `for` loop to a `while` loop, any `for` based join point becomes invalid.

1.4.2 Pointcuts

Pointcuts identify join points in the programs execution, and allow weaving to take place at those points. A pointcut can refer to join points through

a (regular) expression or by referring to other pointcuts. Pointcuts can be *named* or *anonymous*. Pointcuts can use familiar *unary* and *binary* operators to form complex expressions to identify join points. Some simple pointcuts are listed in figure 1.7.

```
pointcut callGetMessage():
    call(* HelloMIDlet.getMessage());

pointcut callSetMessage(String message):
    call(void HelloMIDlet.setMessage(String))
    && args(message);

pointcut newHelloMIDlet():
    execution(HelloMIDlet.new(..));
```

Figure 1.7: Example Pointcuts (AspectJ)

Pointcuts *capture* join points, and their associated *context*, and allow you to do something to those join points. For example, you could, at a certain method, perform logging before, and after the method's invocation.

1.4.3 Advice

Advice is the action you want to take at particular join point, captured by a pointcut. There are three kinds of advice;

- Before
- After
- Around

The *before* advice takes place just before the join point is reached, the *after* advice after the join point has returned, and the *around* around the join point call or execution. The most interesting of these is obviously the *around* advice, as it allows you to possibly bypass the execution of the join point entirely, or to take a completely different action, or to proceed with a different set of parameters, etc.

```
    before(): callGetMessage() {
        // do something
    }

    after(): callGetMessage() {
        // do something
    }

    void around(): callGetMessage() {
        // check something...
        proceed();
    }
```

Figure 1.8: Some Examples of Advice (AspectJ)

1.4.4 Aspects

An *aspect* is a combination of *pointcuts* and *advice*, and potentially methods and fields. In figure 1.9 we see a simple example of an aspect;

```
public aspect AccountLoggerAspect {

    pointcut logPoints() : call(* Account+.*(..));

    before() : logPoints() {
        // do entering logging...
    }

    after() : logPoints() {
        // do exiting logging...
    }
}
```

Figure 1.9: An Example Aspect (AspectJ)

Some AOP implementations allow the Aspect to exist in a single artifact (like AspectJ for example), others allow you to define your Advice separately to your pointcuts (as in Aspectwerkz). But in all cases, the results are the same.

1.5 Summary

What we intend to do is develop a simple J2ME prototype application, and adapt it so that it changes it's behaviour according to changes in it's context, in order to demonstrate both *runtime* and *development-time* adaptation. We will try to do so using an AOP implementation suited to this. To that end we are restricted by the following constraints;

Device Limitations

The choice of AOP implementation will be restricted by the limitations of the J2ME environment. Because of the limited processing power and the low memory footprint of handsets that support J2ME, care will have to be taken to choose an AOP implementation that has minimal overhead.

J2ME Support

The choice of AOP implementation will have to be constrained by it's support for the classes provided by J2ME, or specifically the CLDC. If the AOP implementation makes use of APIs not present in the CLDC (for example, the *reflection* API) it will be discarded in favour of one that does.

Tools Support

The choice of AOP implementation will be dependant on the development tools that are available for it. Of course our evaluation of the suitability of the AOP implementation will need to be based on, amongst other things, the maturity of the tools available for development.

Chapter 2

Research

2.1 Related Work

Not a lot, outside of academic journals, has been written on the subject of Adaptive Software Development with AOP. Even less written in the context of J2ME. I have, however, come across a number of articles of interest;

2.1.1 AdapPE

AdapPE [3] is an architectural pattern that attempts to demonstrate the use of Aspects to cleanly structure Adaptive Software. It breaks the application up into a set of elements;

- The Core Application
- The Adaptability Aspects
- The Auxiliary Classes
- The Context Manager
- The Adaptation Data Provider

Starting from the *core application* it then defines the *adaptability aspects* that define how the application should change according to changes in it's

context. It then defines the *auxiliary classes* that encapsulate the adaptive behaviour. The *context manager* is what analysis the current state and triggers adaptation. Changes in behaviour rely on data defined within the *adaptation data provider*, a complex data structure that allows the application to change it's behaviour dynamically.

The overall pattern is sound, but the adaptation data provider is not practical in the context of J2ME, as it would rely on *reflection* which is not available. In developing my prototype I tried to adhere to the basic structural architecture / separation of concerns as outlined in this pattern, but replaced adaptation data provider with a more J2ME friendly datastructure.

In another paper by the same authors [4], they applied the same pattern to a simple J2ME dictionary application, but here the adaptation data provider was replaced by a combination of network calls and caching (a concern implemented as an adaptation). Although this represented an improvement, insofar as it's applicability in the J2ME domain was concerned, basic issues, for example whether or not AspectJ would work in J2ME out of the box, were hardly addressed. Later we will see that this is indeed a real issue, which will ultimately decide which AOP implementation we will use.

2.1.2 AOP and Design Patterns

In [5], a number of the GoF patterns [6] (specifically the Observer pattern) were re-implemented using AspectJ. The results were interesting and can be summarised as follows;

- Locality
- Reusability

Pattern Locality

All the code for the pattern is *localized* in a single aspect (see figure 2.1). This has many advantages, not least of which is the dependency inversion that results, which means that no longer do you have a situation where

participants are dependant on the pattern, the pattern is now dependant on the participants.

```
public abstract aspect ObserverProtocol {

    protected interface Subject {};
    protected interface Observer {};

    abstract protected void updateObserver(
        Subject subject, Observer observer);

    abstract protected pointcut subjectChange(Subject subject);

    after(Subject subject) : subjectChange(subject) {

        Enumeration enum = getObservers(subject).elements();
        while(enum.hasMoreElements())
            updateObserver(subject,
                (Observer) enum.nextElement());
    }
}
```

Figure 2.1: Extract from ObserverProtocol Aspect

Participants can be incorporated into a pattern, and remain oblivious to the fact (see figure 2.2). Participants can be added and removed seamlessly. The Localized pattern becomes an important artifact, both in terms of modularity and as documentation of patterns being employed.

Another side effect of this locality is that problems that arise when multiple instances of the same pattern are used cease to be an issue. The problem of establishing *who* says *what* to *whom* becomes trivial when the pattern is abstracted out to a seperate module.

Pattern Reusability

Following on form *locality* is *reusability*. If participants are oblivious to the patterns they are associated with, and the pattern itself is localized in a

```
public aspect SpecificObserver extends ObserverProtocol {

    declare parents: FirstObject implements Subject;
    declare parents: SecondObject implements Observer;

    protected pointcut subjectChange(Subject subject):
        call(void FirstObject.changeState(..))
        && target(subject);

    protected void updateObserver(Subject subject, Observer observer) {
        ((SecondObject) observer).setSomeValue(
            ((FirstObject) subject).getSomeValue());
    }
}
```

Figure 2.2: A SpecificObserver Aspect implementation

single aspect, then the same pattern can be reused. This has the effect of simplifying design.

In the context of Adaptive software, reusability has the effect of allowing you to create a set of adaptations that can be re-used accross the application. In particular, the observer pattern stands out as a means of expressing a specific type of adaptation in a modular fashion. We shall see use of this later on.

2.2 The Various AOP Implementations

There are, at this time, many AOP implementations available. Some are quite small and neat, others are monolithic and / or exist as part of larger frameworks. Unfortunately most of them are unsuitable for J2ME development for a variety of reasons. There follows a brief analysis of the "state of the art" with a view to establishing which AOP implementation best suits the task at hand.

2.2.1 AOP Flavours

AOP implementations come in two basic flavours;

1. Bytecode based
2. Proxy based

The bytecode based AOP implementations perform their weaving at compile time, whether they add *hooks* that are evaluated at runtime, or full *weaving* of aspects / advice is performed. The bytecode weaver alters the relevant class file, adding the necessary code as defined in the advice. The bytecode based implementations include the following;

- AspectJ
- Aspectwerkz
- JBoss AOP
- JAsCo

The proxy based AOP implementations perform there weaving at runtime, and make heavy use of Java APIs which are J2SE specific, and don't exist within J2ME (which excludes them from further consideration by that fact alone). The proxy based implementations include;

- Spring AOP
- Dynaop

While the proxy based solutions present elegant, lightweight (in the case of Dynaop) and powerful (in the case of Spring AOP) solutions, owing to their heavy use of *Dynamic Proxies* they are unusable in the J2ME environment, so we can concentrate to the bytecode based implementations.

2.2.2 Comparison

Of the four remaining bytecode based contenders, JAsCo I found to be the least interesting. It has a simple interceptor based model which I found to be quite inflexible, and not appropriate for this project. The JAsCoDT eclipse plugin was quite immature and didn't seem to be actively developed.

There was apparently a subproject intended for J2ME use that comprised 3 jars weighing in with a total of approximately .5MB, but it seems that active development on this subproject has ceased. On top of this it relied very heavily on non-J2ME code (Collections framework). All in all I found it quite disappointing and decided not to investigate it further.

I will now compare the three remaining contenders (AspectJ, Aspectwerkz and JBoss AOP) under the following headings;

- Language
- Dependencies

Language

All three use similar means to describe pointcuts (and in the case of AspectJ and Aspectwerkz, they were virtually identical). The main difference between the three is how they separate the pointcuts and advice. JBoss AOP and Aspectwerkz allow you to use either *annotations* (J2SE 5.0 dependant) or xml to associate your advice with your pointcut definitions.

```
<aop>
  <aspect class="MyAspect" scope="PER_VM" />

  <bind pointcut="call(public void MyObject->myMethod()">
    <advice name="myAdvice" aspect="MyAspect" />
  </bind>
</aop>
```

Figure 2.3: Example JBoss AOP xml

AspectJ requires you to put them in the same aspect artifact, although

you are not required to implement the required advice directly in the aspect. So there is little to separate the implementations in this respect.

Both Apectwerkz and JBoss AOP support *Dynamic AOP*[1]. This means that advice can be associated with join points at runtime, providing the join points have been *instrumented* appropriately[2]. JBoss AOP achieves the same thing by adding, changing or removing the appropriate xml file[3], or alternatively by binding advice at runtime (using the `AdviceBinding` and `AspectManager` objects) as in figure 2.4.

```
AdviceBinding binding =
    new AdviceBinding(
        "call(public void MyObject->myMethod()");
binding.addInterceptor(MyInterceptor.class);
AspectManager.getInstance().addBinding(binding);
```

Figure 2.4: Example runtime advice binding in JBoss AOP

So the choice of AOP implementation will not be influenced by it's specific language; i.e. how it specifies join points, how it associates advice with pointcuts, etc. We move on to the dependencies of each, in the hope that this will help us make our decision.

Dependencies

There follows a comparison of the respective dependencies of the remaining AOP implementations under the following headings;

- Dependency on J2SE
- Size of dependency libraries
- Dependency on external configuration files

[1] In the case of Aspectwerkz, this is called *online weaving*

[2] this amounts to running a compiler over the classes (to insert runtime hooks) before running the code

[3] JBoss AOP requires a further step of *instrumenting* the `ClassLoader` in order for you to be able to make use of *Runtime* weaving

All of the remaining implementations were dependant on J2SE libraries to a greater or lesser extent. By far the framework with the smallest dependency was AspectJ. For simple tasks, the dependencies were minimal, but only became a problem for more complex pointcut definitions (*cflow* based expressions, for example). Both Aspectwerkz and JBoss AOP were dependant on J2SE code for even the most trivial of tasks, and had dependencies on one or more quite large jars to be present in the classpath. In addition, both Aspectwerkz and JBoss AOP needed their respective xml files (where the bindings were defined) to be accessible.

Implementation	J2SE	Libraries	Configuration
AspectJ	minimal	minimal	no
Aspectwerkz	substantial	substantial	yes
JBoss AOP	substantial	large	yes

Table 2.1: Comparison of Dependencies

Taking all these factors into account I decided on AspectJ as my AOP implementation. The factors that influenced my choice are summarized in the table 2.1.

2.3 Tools

We now evaluate the tools available for developing J2ME applications, and for developing with AspectJ.

2.3.1 Wireless Toolkit

Sun's Wireless Toolkit (WTK) supports the MIDP 1.0 / 2.0 and CLDC 1.0 / 1.1 specifications and includes tools to help develop MIDlet suites. Amongst these tools are the following;

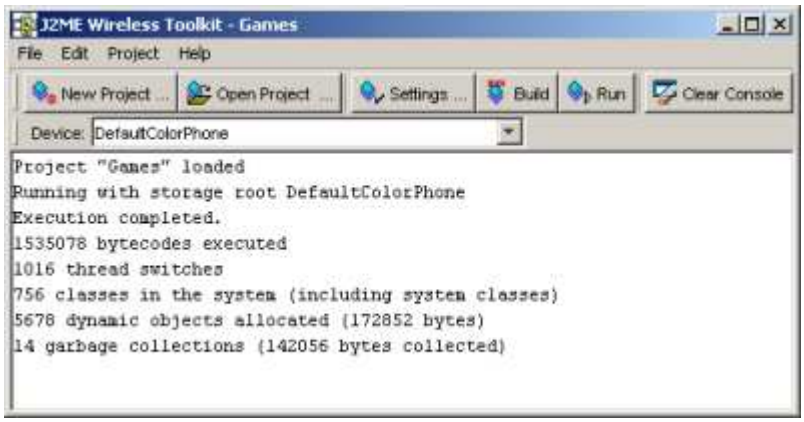

Figure 2.5: WTK: KToolBar

KToolBar The main user interface that comes with the WTK. It allows you, albeit in an unsophisticated way, to manage projects, compile MIDlet suites, package MIDlet suites, and run MIDlets. It is not a full development environment.

Emulator This tool allows you to run MIDlet suites outside of the KToolBar.

In terms of supporting the development process, the WTK is not exactly the best environment. If you do choose to use it you will need to employ your favourite editor in order to write the necessary code, as it doesn't provide one (not a huge problem, more an inconvenience). But at the same time it

Figure 2.6: WTK: Default Colour Phone

does provide you with some useful tools you will need to successfully develop a MIDlet application.

2.3.2 Ant Tasks

We now turn to investigating support for automated building of the project; i.e. do AspectJ and J2ME provide ant tasks suitable for this purpose?

iajc

AspectJ provides a compiler, `ajc`, used for compiling aspects. AspectJ also provides ant tasks that allow you to invoke `ajc` from an ant build script. These tasks are contained in the `aspectjtools.jar`. In order to make use

of them you need to do add a `taskdef` entry to the build script, shown in figure 2.7;

```
<path id="aspectj.classpath">
    <fileset dir="${lib.dir}/aspectj">
        <include name="aspectj*.jar"/>
    </fileset>
</path>

<taskdef
    resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
    <classpath refid="aspectj.classpath"/>
</taskdef>
```

Figure 2.7: Adding support for AspectJ Ant tasks

Once this has been added, you call the task as you would any other task, as demonstrated in figure 2.8;

```
<target name="compile-ajc" depends="jad">
    <iajc destDir="${bin.dir}"
        incremental="false">
        <sourceroots>
            <pathelement location="${src.dir}"/>
        </sourceroots>
        <classpath>
            <path refid="j2me.classpath"/>
            <pathelement location="${lib.dir}/aspectj/aspectjrt.jar"/>
        </classpath>
    </iajc>
</target>
```

Figure 2.8: Using AspectJ Ant tasks

Antenna

Antenna provides a set of Ant tasks to help you in developing J2ME applications. The tasks include;

wtkjad A Jad creation task

wtkpackage A MIDlet suite packaging task

wtkpreverify A MIDlet suite preverification task

wtkrun A task to run a MIDlet suite within a WTK device

As with the AspectJ tasks, in order to run the Antenna ant tasks, they first need to be defined in the ant build file;

```
<path id="antenna.classpath">
    <fileset dir="${lib.dir}/antenna">
        <include name="antenna*.jar"/>
    </fileset>
</path>

<taskdef resource="antenna.properties">
    <classpath refid="antenna.classpath"/>
</taskdef>
```

Figure 2.9: Adding support for Antenna Ant tasks

Once this has been done the tasks can be called as normal;

```
<target name="previrify" depends="package">
    <wtkpreverify jarfile="${midlet.name}.jar"
        verbose="true"
        jadfile="${midlet.name}.jad"
        cldc="false"
        nofloat="false"/>
</target>
```

Figure 2.10: Using Antenna Ant tasks

It is my opinion that arguably the most important thing when setting out on a project is to have the builds scripts prepared and ready. With iajc and Antenna a part of my toolkit I had confidence in proceeding.

2.3.3 IDE Support

We now turn our attention to IDEs. For some people an IDE is an unnecessary convenience, but for others a good IDE is an essential productivity tool (the author admits that he falls into the second camp). We narrowed down the choice of IDE to three obvious contenders;

- Eclipse
- IntelliJ IDEA
- NetBeans

As I had little or no experience using NetBeans at all, and plenty using the other two IDEs, I decided to exclude it, as I did not want the overhead of learning the idiosyncracies of a new tool, as this would have, in my opinion, adversely affected my productivity.

Both Intellij and Eclipse have been around for quite a few years, and both feature a plugable architecture that allows support to be added for different frameworks. IntelliJ seems to be a far more mature IDE than Eclipse, which always seems to be playing catch-up. All of the wonderful features that IntelliJ has eventually find their way to Eclipse (refactorings, ant integration, junit integration) which is not necessarily a bad thing. Because Eclipse is an open platform, there is a proliferation of plugins available for it. So in evaluating which IDE to use for the project I looked to see what support, if any, each had for AspectJ and J2ME.

AspectJ

IntelliJ had support for AspectJ in an earlier version, but for some reason had removed it. Eclipse, on the other hand, had excellent support for AspectJ in the form of the AspectJ Development Tools (AJDT). These tools allow you to develop Aspect Oriented Software within Eclipse in the same manner as you would develop Java code.

Figure 2.11 shows the crossreference view. When an aspect is selected this view details the classes it advises or on which it statically cross-cuts with member and attribute introductions.

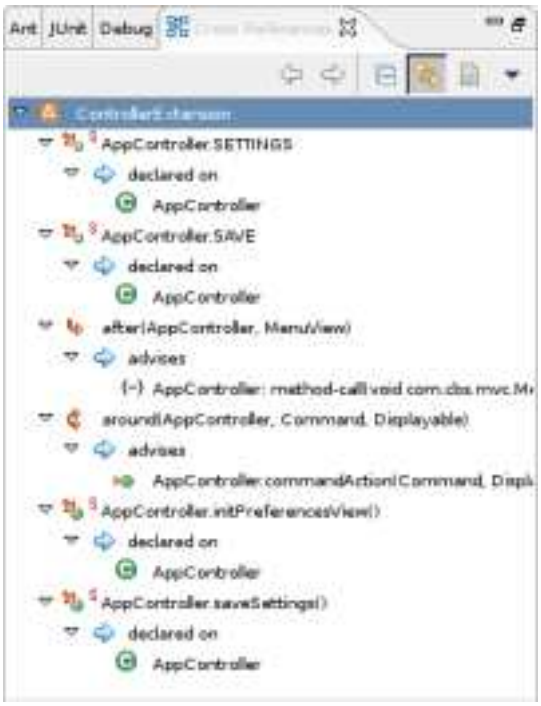

Figure 2.11: Eclipse AspectJ Support - Cross Reference View

J2ME

IntelliJ has currently no support for J2ME development, but there is a plugin available for Eclipse called EclipseME which allows you to develop, build and debug J2ME applications within Eclipse. To set it up you simply point it at the WTK root directory, and it looks after the rest. Among it's features are;

- Support for multiple wireless toolkits
- J2ME project support
- JAD editor
- MIDlet suite launch support
- MIDlet suite debugging support
- OTA (over the air) provisioning support.

Unfortunately EclipseME didn't work well with the AspectJ tools at all. Any aspects contained within the project proved to be a problem for it. It wouldn't pass the preverification phase once there were any aspects present. So a decision had to be made; should I retain EclipseME in order to ease the burden of developing MIDlet suites, or should I retain the AJDT tools and find another way to develop MIDlet suites. As the AJDT eclipse tools were a huge benefit in developing Aspects, and I had a means of automating the building and running of MIDlet suites (Antenna), and because Eclipse had such good support for Ant, I decided to discard EclipseME [4].

[4]as it turns out, EclipseME was pretty flaky when it came to developing and running non-AOP MIDlet projects

Chapter 3

Design / Implementation

3.1 Basic Application

The first thing was to develop a small application for a mobile phone that supported J2ME. The *TourGuide* was a small program that contained a database of locations that could be browsed by the user, detailing information about various locations around a city (in this case Dublin).

Figure 3.1: TourGuide: Menu View

When the application starts up, the initial screen is a list of locations (figure 3.1). Using the arrow keys the user selects a location.

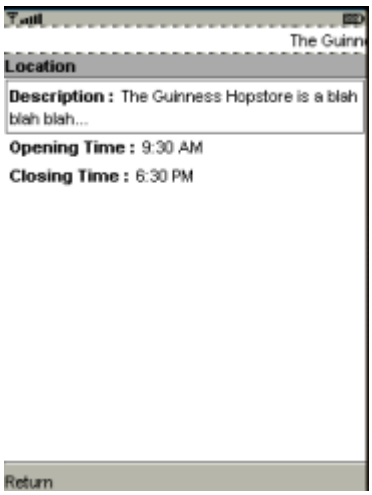

Figure 3.2: TourGuide: Location View

Upon selecting a destination within a city, the user is treated to information regarding the location (figure 3.2); the opening times, a brief description, the type of location, etc.

I saw this as a perfect example of an application that could quite easily be adapted in a useful way; so that it would update it's current location based on it's GPS position, and that it would reflect this change to the user, so that it would tell the user about where he was, and the landmarks, or locations he was close to that would be of interest to the user.

3.1.1 Architecture

The TourGuide architecture was based on a simple MVC (Model-View-Controller) [7] variant. The participants in the MVC architectural pattern are as follows;

Model This is where the state, and the business rules that act on the state, of the application is located.

View The user interface that has the responsibility of rendering the information contained in the model.

Controller This manages the interaction between the view and the model.

In this case, the controller was the MIDlet (`AppController`). There were two views (`MenuView` and `LocationView`) and a single model (`AppModel`).

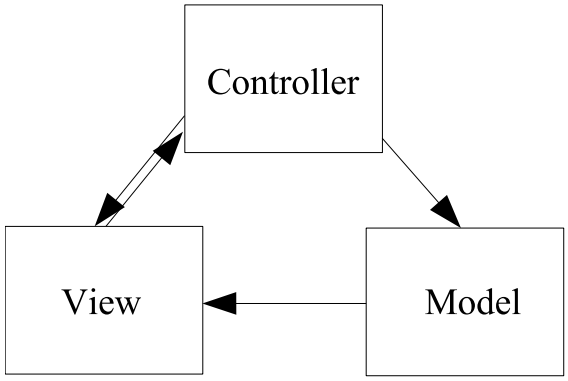

Figure 3.3: The Model-View-Controller Architectural Pattern

The reason for choosing MVC are as follows; The same model code (data and business logic) can be reused across the application. In this case the `MenuView` uses the same data as the `LocationView`. So there is every reason to keep one copy of the data.

Most MIDlets are developed in a single MIDlet class, containing all business logic, data, and gui code. Taking this approach you end up with a large class containing code that's difficult to maintain. The alternative MVC approach results in a slight increase in the complexity of the application with the introduction of extra model and view classes. All the same the choice of MVC as the base application architecture gave us a head start, ensuring a clean separation of concerns from the outset.

3.2 Adaptation

The next step was to take this application and adapt it to change it's behaviour according to it's changing context, specifically it's location. In order to do this a set of Aspects were written along with a set of additional (auxiliary) classes. We can sum up the desired extensions and adaptations as follows;

Preferences We want to add support for user preferences to be set, so that the application could change it's behaviour according to these.

Location Awareness We want to add support for location awareness, so that the application could change it's behaviour according to its location.

3.2.1 The Controller Extension Aspect

The `ControllerExtension` aspect has the responsibility of extending the `AppController`. It has a number of purposes including the following;

- Field Introduction
- Method Introduction
- Method Advising

we will look at these individually and see how they extend the existing application.

Field Introduction

The `AppController` contains a number of attributes, including two `Command` objects (see figure 3.4) which are added to the views at construction time (the `AppController` has responsibility for creating the views in response to user input or application events);

```
public static final Command EXIT
    = new Command("Exit", Command.STOP, 60);
public static final Command RETURN
    = new Command("Return", Command.BACK, 60);
```

Figure 3.4: Command Object Attributes of the AppController

As we want our application to behave differently we need to perform some *static crosscutting* and *introduce* some extra `Command` objects, so they may be added to the relevant view at construction.

```
public static final Command AppController.SETTINGS
    = new Command("Settings", Command.ITEM, 60);
public static final Command AppController.SAVE
    = new Command("Save", Command.ITEM, 60);
```

Figure 3.5: Command Object Attributes introduced to the AppController

Method Introduction

We need to be able to add a new screen where the user can set his preferences (for example, the types of locations the user is interested in). We also want to add some sort of acknowledgement so that the user knows he has successfully set his preferences. To do this we need to introduce two methods to the `AppController`;

```
private void AppController.initPreferencesView() {

    if(getView() != null) getView().destroy();
    PreferencesView preferencesView
        = new PreferencesView(getModel());
    preferencesView.init();

    preferencesView.addCommand(RETURN);
    preferencesView.addCommand(SAVE);
    preferencesView.setCommandListener(this);

    getDisplay().setCurrent(preferencesView);
    preferencesView.updateView();
    setView(preferencesView);
}
```

Figure 3.6: initPreferencesView() method introduced to the AppController

The method in figure 3.6 follows the format of the other view initialisation methods in the controller. It creates an instance of a `PreferencesView`, adds `Command` objects to it and sets it as the current display object.

The method in figure 3.7 is called when the *Save* command is selected in the `PreferencesView`. It notifies the user that the selected preferences have

```
private void AppController.saveSettings() {

    ((List) getView()).getSelectedFlags(
        getModel().getPreferences().getSelected());

    Alert alert = new Alert("Save Alert");
    alert.setType(AlertType.CONFIRMATION);
    alert.setString("Settings Saved!");
    getDisplay().setCurrent(alert);
}
```

Figure 3.7: saveSettings() method introduced to the AppController

been successfully saved.

Method Advice

Next we need to add the `SETTINGS` command to the main screen (the `MenuView`). To do this we need to declare some pointcut definitions, in order to capture the point where we will add the command;

```
pointcut initMenuView():
    execution(void AppController.initMenuView());

pointcut addCommand(MenuView form):
    call(void List.addCommand(Command))
    && target(form);

pointcut addCommandToView(MenuView form):
    cflow(initMenuView())
    && addCommand(form);
```

Figure 3.8: Pointcuts for Capturing a specific Join Point

In figure 3.8 we see three pointcuts; initMenuView pointcut which captures the `initMenuView` method invocation of the `AppController`, the addCommand pointcut which captures an `addCommand` method invocation of a `List` object, and the addCommandToView pointcut that captures the

`addCommand` method within the context of an `initMenuView` method invocation. This uses the cflow construct to restrict the scope to the context of a specified control flow.

```
after(MenuView view):
    addCommandToView(view) && !within(ControllerExtension) {

    view.addCommand(AppController.SETTINGS);
}
```

Figure 3.9: Advice to add Commands to the MenuView

In figure 3.9 we see the advice that adds the new command to the `MenuView`, using the context captured by the pointcut definitions. The `!within(ControllerExtension)` is added to ensure the advised `addCommand` method invocation is not captured by the pointcut, which would result in an infinite loop.

Figure 3.10: TourGuide: Menu View after Command added

In figure 3.10 we see the `MenuView` after the `ControllerExtension` aspect has adapted the `AppController` by adding the new Command to the main view. The next step is to ensure the application takes action when the new control is selected. To accomplish this we add another pointcut definition;

```
pointcut commandAction(
    AppController controller, Command command, Displayable displayable):
    execution(void AppController.commandAction(Command, Displayable))
    && target(controller)
    && args(command, displayable);
```

Figure 3.11: Pointcuts for Capturing Command Selection

In figure 3.11 we see what looks at first to be a fairly complicated pointcut but is actually simple. It captures invocations of the `commandAction` method on the `AppController`, with the `AppController` as the target object and the `Command` and `Displayable` as the arguments of the `commandAction` method. Once this has been declared we can add some around method advice;

```
void around(
    AppController controller, Command command, Displayable displayable):
    commandAction(controller, command, displayable) {

    if(command == AppController.SETTINGS)
        controller.initPreferencesView();
    else if(command == AppController.SAVE)
        controller.saveSettings();
    else
        proceed(controller, command, displayable);
}
```

Figure 3.12: Advice to handle the Settings Command

The advice checks the command to see if it is one of the new commands added in figure 3.5 and takes appropriate action if it is. If not it calls the underlying method (the `proceed(...)` call), passing the context on. In figure 3.13 we see the newly added Preferences view.

Figure 3.13: TourGuide: Menu View after Command added

3.2.2 The Location Customization

The `LocationCustomization` is an aspect who's purpose is to change the behaviour of the `AppModel` if certain conditions are met. It has a single purposes; to intercept the call to `getLocation()` in the `AppModel` (see figure 3.14) and to adapt it if necessary.

```
pointcut getLocation(AppModel model):
    execution(Location AppModel.getLocation())
    && target(model);
```

Figure 3.14: The LocationCustomization pointcut

As can be seen in figure 3.15 the around advice first checks to see if the model is in the correct state to be adapted (the call to `isInAdaptiveState(...)`), and if so it returns a `Location` that reflects it's GPS location. This `Location` is set by the `LocationObserver` which we will cover later.

3.2.3 The Location Monitor

The `LocationMonitor` is a class who's purpose is to constantly monitor it's GPS position. It extends `AbstractMonitor`, a utility class that looks after

```
    Object around(AppModel model) : getLocation(model) {
        if(isInAdaptiveState(model))
            return _location;
        else
            return proceed(model);
    }
```

Figure 3.15: The LocationCustomization advice

the bulk of the work. Sub-classes of `AbstractMonitor` only need to implement the `checkState()` template method to customize it to their particular needs.

```
public class LocationMonitor extends AbstractMonitor {
    //...
    protected void checkState() {
        if(isLocationChanged())
            adaptState();
    }
    // ...
}
```

Figure 3.16: The LocationMonitor class

Any number of *Monitor* classes can be added in this way; for example, you could add a `MemoryMonitor` or a `BatteryMonitor`, and have the application adapt to changes in their states.

3.2.4 The Location Observer

The `LocationObserver`, which extends the `ObserverProtocol` (see figure 2.1), is an aspect who's purpose is to check the `LocationMonitor` to see if it's state has changed, and to notify the `LocationCustomization` of any changes if it has. It is responsible for the runtime adaptation and can be summed up as follows;

Supertype Introduction

In order for the `LocationMonitor` and the `LocationCustomization` to participate in the *ObserverProtocol* they should each implement the `Subject` and `Observer` interfaces respectively. This is achieved through *Static Crosscutting* the classes so that they each implement the relevant interfaces, as demonstrated in figure 3.17.

```
declare parents: LocationMonitor implements Subject;
declare parents: LocationCustomization implements Observer;
```

Figure 3.17: The LocationObserver Aspect: Static Crosscutting

Abstract Pointcut Implementation

Next the abstract pointcut, `subjectChange(...)` which is defined in the `ObserverProtocol` (figure 2.1) needs to be implemented so that it catches calls to the method `adaptState()` of the `LocationMonitor` object, shown in figure 3.18.

```
protected pointcut subjectChange(Subject subject):
    call(void LocationMonitor.adaptState(..))
    && target(subject);
```

Figure 3.18: The LocationObserver Aspect: Implement the Pointcut

Method Implementation

Finally the abstract method `updateObserver(...)` needs to be implemented in order to to define the required changes that need to take place if the `subjectChange(...)` pointcut conditions are met.

As we can see in figure 3.19, the steps taken are to update the `Location` attribute of the `LocationCustomization` and to reset the state of the `LocationMonitor`.

```
protected void updateObserver(Subject subject, Observer observer) {

    LocationMonitor monitor
        = (LocationMonitor) subject;
    LocationCustomization customization
        = (LocationCustomization) observer;

    if(monitor.isAdaptState()) {
        customization.setLocation(getNewLocation(monitor));
        monitor.resetState();
    }
}
```

Figure 3.19: The LocationObserver Aspect: Implement the Method

3.2.5 The Model Extension Aspect

As we wanted the application to change it's behaviour at runtime, we needed to extend the functionality of the `AppModel`. The `ModelExtension` has the following roles;

- Supertype Introduction
- Field Introduction
- Method Introduction
- Method Advising

Supertype Introduction

Currently the `AppModel` responds solely to UI events, and doesn't notify any of the *Views* of any changes to it's state. What was required was for the `AppModel` to change it's behaviour and to start to inform the *Views* of it's changing GPS location.

To overcome this limitation the `AppModel` was made to implement the `Runnable` (see figure 3.20) interface, so that it could run in a separate `Thread`, and regularly update the *Views* of it's changing state if necessary.

```
declare parents: AppModel implements Runnable;
```

Figure 3.20: The ModelExtension Aspect: Type Introduction

Field Introduction

We now needed to introduce a couple of fields (see figure 3.21), one which would be used to tell if the `AppModel` was in *context-aware mode* (running in a separate `Thread`) or not, the other, a `Preferences` object, used to hold the users preferences, and used to decide how the application should adapt.

```
private boolean AppModel._run = false;
private Preferences AppModel._preferences
    = new Preferences();
```

Figure 3.21: The ModelExtension Aspect: Field Introduction

Method Introduction

Next we needed to introduce a number of methods, both to fulfill the `Runnable` contract and to support the application's adaptations.

```
public void AppModel.run() {

    while(_run) {
        updateViews();
        try {Thread.sleep(10000);}
        catch (InterruptedException e) {}
    }
}
```

Figure 3.22: The ModelExtension Aspect: Method Introduction 1

In figure 3.22 we see the implementation of the `run()` method required by the `Runnable` interface. It's implementation is simple, containing nothing more than a `while` loop which continues to loop if the `_run` field, introduced

in figure 3.21, evaluates to `true`. While looping it calls `updateViews()` and then sleeps for 10 seconds.

```
public void AppModel.start() {
    _run = true;
    new Thread(this).start();
}

public void AppModel.stop() {
    _run = false;
}
```

Figure 3.23: The ModelExtension Aspect: Method Introduction 2

In figure 3.23 we see two more introduced methods; the `start()` method which starts the `AppModel` running in a separate `Thread` and the `stop()` method which stops it.

Method Advising

Next we need to add a new `Location`, specifically a `ContextAwareLocation`, to the model, which is used to denote that the `AppModel` is in *context-aware mode*. To accomplish this we declare a pointcut definition as in figure 3.24;

```
pointcut newAppModel(AppModel model):
    execution(AppModel.new(..))
    && this(model);
```

Figure 3.24: The ModelExtension Aspect: newAppModel Pointcut Definition

This pointcut captures the constructor invocation. The corresponding advice, which simply adds the new `ContextAwareLocation` to the `AppModel` is shown in figure 3.25;

```
after(AppModel model): newAppModel(model) {
    model.addLocation(new ContextAwareLocation());
}
```

Figure 3.25: The ModelExtension Aspect: newAppModel Advice

Next we need to intercept the calls to `setIndex(...)` so that we can check to see if this puts the `AppModel` into *context-aware mode*. The pointcut definition to do this is listed in figure 3.26;

```
pointcut setIndexOnModel(AppModel model):
    execution(void AppModel.setIndex(..))
    && this(model);
```

Figure 3.26: The ModelExtension Aspect: setIndexOnModel Pointcut Definition

In figure 3.27 The corresponding advice checks to see if the model is in *context-aware mode* by calling the `isContextAware()` method. It invokes `start()` on the model if it is (see figure 3.23) and `stop()` on the model if it isn't.

```
after(AppModel model): setIndexOnModel(model) {
    if(model.isContextAware())
        model.start();
    else
        model.stop();
}
```

Figure 3.27: The ModelExtension Aspect: setIndexOnModel Advice

3.2.6 The Bootstrap Aspect

Now that we have defined all the required aspects and extensions, we need to connect them all together when the application starts up, and disconnect them when the application shuts down. To achieve this we add a `Bootstrap` aspect which performs the above functions. The pointcut definitions can be seen in figure 3.28;

```
pointcut startApp(AppController controller):
    execution(void AppController.startApp())
    && this(controller);

pointcut destroyApp(AppController controller):
    execution(void AppController.destroyApp(..))
    && this(controller);
```

Figure 3.28: The Bootstrap Aspect: Pointcuts

The `startApp` and `destroyApp` pointcuts capture the application as it starts up and shuts down. The corresponding `startApp` advice can be seen in figure 3.29;

```
after(AppController controller): startApp(controller) {

    _locationMonitor = new LocationMonitor();
    _locationMonitor.start();

    LocationObserver.aspectOf().addObserver(
            _locationMonitor,
            LocationCustomization.aspectOf());
}
```

Figure 3.29: The Bootstrap Aspect: startApp Advice

As can be seen, the advice creates a `LocationMonitor` and starts it running. It then sets up the *Observer* relationship between the `LocationMonitor` object and the `LocationCustomization` aspect. This advice takes place

immediately *after* the application has started. In figure 3.30 we see the `destroyApp` advice;

```
after(AppController controller): destroyApp(controller) {

    LocationObserver.aspectOf().removeObserver(
            _locationMonitor,
            LocationCustomization.aspectOf());

    _locationMonitor.stopRunning();
}
```

Figure 3.30: The Bootstrap Aspect: Advice

This is called immediately *after* the application shuts down. It removes the `LocationMonitor` object and the `LocationCustomization` aspect from the *Observer* relationship, and stops the `LocationMonitor`. This represents a clean shutdown, ensuring no dangling references are left behind, or orphan `Threads` are left running in the system.

3.3 Summary

We have taken a simple J2ME application and added the ability to select preferences, the application can now enter a *context-aware mode* where it's state, and the displayed `Location`, will change according to it GPS position.

Chapter 4

Evaluation

4.1 Comparison of Approaches

In order to evaluate the project another version of the TourGuide application was developed, but this time in "Plain Ol' Java". We will now compare the two different approaches under the following headings;

- The number of artifacts created (class files, aspects, etc.).
- The size of the resultant jar / binary.

4.1.1 Artifacts

As we can see from table 4.1, the AOP approach resulted in 46% more code artifacts when the AspectJ runtime code was included. But if you exclude the AspectJ runtime code this figure drops to 14% more code.

Looking closely at table 4.1 we can see that when we exclude the AspectJ runtime code the AOP approach has two less interfaces than the POJO approach, but six more Aspects (the POJO approach has none). So it would seem the two approaches compare favorably in terms of artifacts.

	AOP (inc. AspectJ runtime)	AOP (exc. AspectJ runtime)	POJO
Classes	28	22	22
Interfaces	7	4	6
Aspects	6	6	0
Total	41	32	28

Table 4.1: Comparison of Artifacts

4.1.2 Jars

As can be see from table 4.2, The AOP approach results in a jar that is more than 71% greater than the POJO implementation. This is an appreciable difference, but I would imagine as a project grows in size, the AspectJ runtime code would play a reduced part in the overall size of the deliverable jar.

	TourGuide	TourGuide : AOP	TourGuide : POJO
Jars (KB)	17.8	45.0	26.3

Table 4.2: Comparison of Jars

4.2 For the AOP Approach

The benefits of the AOP approach can be summarised as follows;

Cleaner Seperation within the Code

The AOP approach ensure that *concerns* remain separate from each other and don't get tangled up. This is important from the point of view of maintenance. With a clean separation of subsystems, it is easier to find problem

code. If you know what you are looking for you will find it more quickly. This is a huge benefit when managing large scale projects containing millions of lines of code, and hundreds of thousands of classes.

Adding Functionality requires no Re-Engineering of Existing Application

Aspects can be added after the fact. In fact, Aspects can be woven into application binaries. Access to source code is not necessary in order to extend the functionality of an existing application.

Allows also for Seperation of Development Concerns

The AOP approach allows you to seperate developers according to their strengths; GUI developers can take care of user interface concerns, security experts can keep to their particular field of expertise, persistence gurus can work on persistence subsystems, etc. and all these individual concerns can be woven together without knowledge of the others.

Useful as a Tool during the Analysis Stage

The kind of critical thought and analysis of an application and it's subsystems can be of great benefit in terms of understanding how an application works. In particular, during this project I initially developed the basic TourGuide, then extended it through AspectJ, but then re-implemented it in plain Java supporting the features added by the AspectJ implementation. The knowledge I had gained from the initial AspectJ implementation was invaluable when re-implementing the plain Java version.

4.3 Against the AOP Approach

The shortcomings of the AOP approach can be summarised as follows;

AOP produces more Artifacts

This is particularly noticeable in a small project, and especially in a J2ME project where the libraries are bundled with the MIDlet application jar. It is important to note that the influence of the AOP artifacts on the overall size of the project reduces as the project size grows.

No AOP Implementation supports J2ME "out of the box"

All of the AOP implementations rely on non-J2ME code to varying degrees, and to get the AOP TourGuide to work with AspectJ, some re-engineering of it's runtime library was required.

Could not take advantage of the full power of AOP

J2ME does not support Java's reflection, or runtime introspection, so a lot of the more powerful features of AOP, and AspectJ in particular, could not be used. This situation may change in the future as mobile devices get more powerful.

Chapter 5

Conclusions

5.1 In Sum

The AOP approach results in improved maintainability, because it allows for clear separation of concerns, or modules. There is no tangled mess of interconnecting concerns to negotiate. This is one of the main problems encountered when maintaining complex applications with a sizeable codebase.

On top of this there is great potential for improvements in the development process. Because functionality can be added or modified after the fact, the cost of change curve can be flattened. There no longer needs to be the same emphasis placed on *big design up front*. Also developers can contribute to a project without needing to get up to speed in technologies they are unfamiliar with.

5.2 Concerns about Testing

Many of the concerns about AOP and testing seem to be largely unfounded. It not only lends itself to be tested (using a unit testing framework such as JUnit or TestNG) but it actually compliments the testing process. AOP can make the creation and use of *Mock Object* trivial.

5.3 AOP and J2ME

It would seem that the only thing standing in the way of AOP being used in J2ME projects is J2ME itself. With a little bit of work an AOP implementation, and in particular AspectJ, can be made to work in the context of J2ME, but with reduced functionality. In future the J2ME specifications, specifically the CLDC, could expand to include *reflection* and *dynamic loading of classes*. This would open the doors for more complex AOP applications.

5.4 The Future

In this project we modified AspectJ so that it would run within a J2ME application. This incentive should be developed further, either as a fork of the current AspectJ development tree, or as a seperate project. As research it would have great value, both academically and in the enterprise.

Appendix A

AspectJ2ME

A.1 Overview

AspectJ had a small number of dependencies on code that fell outside of J2ME, so it required a small amount of re-engineering in order for it to play nice. In this section I will briefly demonstrate one of the code changes that was required.

A.2 Getting the Source

The source for the AspectJ project is available through anonymous cvs. Before we can check out the source we need to issue the following command that sets up the `CVS_ROOT` environment variable;

```
export CVS_ROOT=\
    ":pserver:anonymous@dev.eclipse.org:/home/technology"
```

The following command will check out the entire source tree into the current directory;

```
cvs co org.aspectj/modules
```

as we only want to re-implement the runtime classes, we only need to check out the *runtime* module;

```
cvs co org.aspectj/modules/runtime
```

A.3 Changing the Source

The `ThreadStackFactoryImpl` factory returned different `ThreadStack` and `ThreadCounter` implementations, depending on the version of the VM. These were used in the `CFlowCounter` and `CFlowStack` to keep track of the call stack.

```
public interface ThreadStack {
    public Stack getThreadStack();
}
```

Figure A.1: The ThreadStack interface

We needed to re-implement the `ThreadStackImpl` and the `ThreadCounterImpl`. The current implementations each made use of Java's `ThreadLocal` to keep a copy of a stack and a counter per thread. As `ThreadLocal` is a non-J2ME class, a substitute was required;

```
private static class J2METhreadLocal {

    private static Hashtable _table = new Hashtable();

    protected Object getLocal() {

        synchronized(_table) {
            return _table.get(Thread.currentThread());
        }
    }

    protected void putLocal(Object obj) {

        synchronized(_table) {
            _table.put(Thread.currentThread(), obj);
        }
    }
}
```

Figure A.2: The J2METhreadLocal class

In figure A.2 we see a simple datastructure that stores objects in a *hashtable*, keyed to each thread. This becomes the base class for the `ThreadStackImpl` and the `ThreadCounterImpl` objects;

```
private static class ThreadStackImpl
    extends J2METhreadLocal implements ThreadStack {

    public Stack getThreadStack() {

        if (getLocal() == null) putLocal(new Stack());
        return (Stack) getLocal();
    }
}
```

Figure A.3: The ThreadStackImpl class

In figure A.3 we see the new implementation of the `ThreadStackImpl`. As we can see, when a request for a Stack is made by calling the `getThreadStack()` method, it calls through to the `getLocal()` method. If that return null, it adds a Stack to the current threads local storage and returns it to the requestor. The `ThreadCounterImpl` is implemented similarly.

All that remains is to ensure the new classes are returned by the `ThreadStackFactoryImpl`. In figure A.4 we see this accomplished;

```
public class ThreadStackFactoryImpl implements ThreadStackFactory {

    public ThreadStack getNewThreadStack() {

        return new ThreadStackImpl();
    }

    public ThreadCounter getNewThreadCounter() {

        return new ThreadCounterImpl();
    }
}
```

Figure A.4: The ThreadStackFactoryImpl class

Appendix B

CD

B.1 Contents

The CD bundled with the report contains 5 projects. Although the projects are intended to be opened in the Eclipse development environment, they can be run standalone, providing the Ant build tool is installed, and in the path. A brief description of each project follows;

FYP The basic TourGuide application

FYP-AOP The TourGuide application with *adaptive* behaviour implemented through AspectJ

FYP-GPS A GPS simulator used to demonstrate location awareness.

FYP-POJO The TourGuide application with *adaptive* behaviour implemented through POJOs (Plain Ols Java Objects)

FYP-Spike A project dedicated to AspectJ and J2ME prototypes.

B.2 Opening the Projects

In order to open the projects into an Eclipse workspace you need to do the following;

1. Go to **File - Import**
2. At the Import Dialog select **Existing Project into Workspace** and click **Next**
3. Select the **Browse** button and navigate to the Project directory
4. Click **Finish**

B.3 Using the Ant Plugin

To use the Ant build scripts from within Eclipse you need to take the following steps;

1. Go to **Window - Show View**
2. Select **Ant**
3. In the Ant view select **Add Build Files**
4. In the Choose Location Dialog select the project in the left hand pane
5. In the right hand pane select the build script (build.xml)
6. Select the **ok** button.

When you have completed the above steps you will see the build file listed in the Ant view. You can now run targets by double clicking them. Before you do this it is important that you edit the build.properties file so that it points at the root of the wireless toolkit and the Java Runtime Environment home directory, as demonstrated in figure B.1;

```
#wtk.home=D:/Java/WTK22
wtk.home=/opt/WTK2.2
#jre.home=D:/Java/j2sdk1.4.2_08/jre
jre.home=/opt/sun-jdk-1.4.2.08/jre
```

Figure B.1: build.properties

Bibliography

[1] K Lieberherr. *Adaptive Object Oriented Software The Demeter Method*, pages 77–111. PWS Publishing Company, 1996.

[2] K Lieberherr, I Holland, and A Riel. *Object-Oriented Programming: An Objective Sense of Style*. 1988.

[3] A Dantas and P Borba. *AdapPE: An Architectural Pattern for Structuring Adaptive Applications with Aspects*. World Wide Web, 2003. `http://www.cin.ufpe.br/~sugarloafplop/articles/ww/plop2003_adappe.pdf`.

[4] A Dantas and P Borba. *Developing Adaptive J2ME Applications Using AspectJ*. World Wide Web, 2003. `http://citeseer.ist.psu.edu/borba03developing.html`.

[5] J Hannemann and G Kiczales. *Design pattern implementation in Java and Aspect*. World Wide Web, 2002. `http://citeseer.ist.psu.edu/hannemann02design.html`.

[6] E Gamma, R Helm, R Johnson, and J Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 293–303. Addison Wesley, 1995.

[7] F Buschman, R Meunier, H Rohnert, P Sommerlad, and M Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, pages 125–143. John Wiley and Sons Ltd., 1996.

Index