Using AspectJ to Build a Software Product Line for Mobile Devices

by

Trevor J. Young
B.Math, University of Waterloo, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University of British Columbia
August 2005
© Trevor J. Young, 2005
Abstract

Software development for mobile devices has become very popular in the last few years. There are hundreds of programmable micro devices on the market, each having different properties in terms of memory capacity, screen size, networking protocols, audio/video support, peripheral extensions and software platforms. Despite the technical differences of these devices, people use them in a similar fashion for communicating, organizing and interacting with their personal data. For a mobile software developer, it would save a great deal of time and effort to program an application in a way that allows it to run on all of these devices without modification. However, the variability of hardware in these mobile phones, PDA's or music players has led the vendors to provide additional levels of software support for different devices. For example, with the Java programming language, vendors and standards bodies have introduced optional, device specific libraries (APIs) that only run on certain hardware devices. A software developer can thus not rely on standard programming practices to build an application that can run on multiple platforms and still offer features that take advantage of unique functionality that is only embedded in specific devices. We call this the device API fragmentation problem.

One potential way to ease the development of a common application for a variety of devices is using Aspect-Oriented Programming (AOP) techniques to build a software product line. The basic idea is to modularize features into aspects and to use the weaving technology in aspect-oriented programming to bind features into different instances of the product line. In this thesis, we evaluate an approach for building a software product line for mobile devices using J2ME and AspectJ. We present different implementations of a prototype application to compare the AspectJ product line approach with standard object-oriented techniques that are currently used to build families of mobile Java software in industry. The results indicate that AspectJ provides a good solution to the device API fragmentation problem by maintaining modularity of features.
Contents

Abstract ............................................................................................................................ ii
Contents ......................................................................................................................... iii
List of Tables ................................................................................................................... iv
List of Figures .................................................................................................................. v
Acknowledgements ........................................................................................................ vi
1 Introduction .................................................................................................................. 1
  1.1 Java for Mobile Devices ......................................................................................... 2
  1.2 Implementing a Mobile Product Line using AspectJ ........................................... 3
2 Related Work ................................................................................................................ 5
  2.1 Product Line Practice in the Mobile Industry ...................................................... 5
  2.2 General Product Line Practices ......................................................................... 6
    2.2.1 Multiple Copies and Version Management ..................................................... 6
    2.2.2 Object Oriented Frameworks and Patterns ...................................................... 8
    2.2.3 Code Generation and Template Programming ................................................. 8
    2.2.4 Pre-Compilation Utilities ............................................................................... 9
  2.3 Aspect-Oriented Product Line Practices ............................................................... 9
3 MobilePhoto Prototype Application Design ............................................................... 11
  3.1 The Base Application ......................................................................................... 11
  3.2 The Optional Features ....................................................................................... 12
  3.3 Architecture ....................................................................................................... 14
4 MobilePhoto Implementation using AspectJ ............................................................. 17
  4.1 De-coupling Optional Features using Aspects ..................................................... 18
  4.2 Binding Features to the Application using Advice .............................................. 20
  4.3 Managing Multiple Features Using Precedence ............................................... 23
  4.4 The Build Process ............................................................................................. 24
5 MobilePhoto Implementation using Object Oriented Programming ....................... 28
  5.1 De-coupling Optional Features Using OO Design patterns ................................ 30
  5.2 Binding Features to the Application using a Pre-Processor ................................ 32
  5.3 Managing Multiple Features Using Pure Variants™ ........................................... 34
6 Evaluation .................................................................................................................... 40
  6.1 Quantitative Analysis ......................................................................................... 40
  6.2 Modularity and Flexibility .................................................................................. 42
  6.3 Scalability .......................................................................................................... 44
  6.4 Multi-Platform Support ...................................................................................... 46
  6.5 Complexity of Implementation .......................................................................... 49
  6.6 Tool Support ..................................................................................................... 50
7 Conclusions ............................................................................................................... 53
  7.1 Contributions ..................................................................................................... 54
  7.2 Future Work ....................................................................................................... 54
Bibliography ..................................................................................................................... 55
Appendix A: Acronyms ................................................................................................. 57
Appendix B: ANT Build Script Source Listing ....................................................... 58
Appendix C: SmsAspect Source Code Listing ............................................................ 65
List of Tables

Table 3-1: Basic Features Included in All MobilePhoto Product Instances .................. 12
Table 3-2: Optional Features Available for MobilePhoto Product Instances .................. 13
Table 3-3: Sample of J2ME Devices Indicating the Supported Software Components ...... 14
Table 3-4: Data Store Schema for MobilePhoto Image Data .................................................. 15
Table 3-5: Data Store Schema for MobilePhoto Image MetaData ......................................... 15
Table 4-1: MobilePhoto Product Instances (AspectJ) .......................................................... 17
Table 5-1: MobilePhoto Product Instances (Object-Oriented) ............................................ 29
Table 5-2: Pure Variants Feature Model for MobilePhoto ..................................................... 35
Table 6-1: MobilePhoto Jar Size Comparison ................................................................. 40
Table 6-2: MobilePhoto Heap Memory Comparison ......................................................... 41
Table 6-3: MobilePhoto Source Code Metrics Comparison .............................................. 42
Table 6-4: Modified Classes for Custom Blackberry AspectJ Runtime ......................... 48
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Proposed Mobile Product Line Build Process</td>
<td>4</td>
</tr>
<tr>
<td>2-1</td>
<td>Example Version Control Branching Technique for Mobile Product Lines</td>
<td>7</td>
</tr>
<tr>
<td>4-1</td>
<td>AOP Feature Weaving Applied to Mobile Product Lines</td>
<td>17</td>
</tr>
<tr>
<td>4-2</td>
<td>MobilePhoto Package Diagram</td>
<td>19</td>
</tr>
<tr>
<td>4-3</td>
<td>Class Diagram for SMS Feature (AOP)</td>
<td>20</td>
</tr>
<tr>
<td>4-4</td>
<td>Design Diagram Showing SMS Feature Interaction with Base Application</td>
<td>21</td>
</tr>
<tr>
<td>4-5</td>
<td>Partial Source Listing for SMSAspect of MobilePhoto</td>
<td>22</td>
</tr>
<tr>
<td>4-6</td>
<td>Sequence Diagram to Send SMS (AspectJ)</td>
<td>23</td>
</tr>
<tr>
<td>4-7</td>
<td>Declaring Precedence to Manage Feature Interaction in AspectJ</td>
<td>24</td>
</tr>
<tr>
<td>4-8</td>
<td>MobilePhoto Build Process</td>
<td>26</td>
</tr>
<tr>
<td>4-9</td>
<td>Eclipse View with ANT and Build Configuration Plugins</td>
<td>27</td>
</tr>
<tr>
<td>5-1</td>
<td>OOP Patterns and Preprocessor Applied to Mobile Product Lines</td>
<td>29</td>
</tr>
<tr>
<td>5-2</td>
<td>Class Diagram for SMS Feature (OOP)</td>
<td>31</td>
</tr>
<tr>
<td>5-3</td>
<td>Partial Source Listing of MainUIMidlet Class (OO Version)</td>
<td>32</td>
</tr>
<tr>
<td>5-4</td>
<td>Partial Source Listings of MainUIMidlet Class (OO Version)</td>
<td>33</td>
</tr>
<tr>
<td>5-5</td>
<td>Sequence Diagram to Send SMS (OOP)</td>
<td>34</td>
</tr>
<tr>
<td>5-6</td>
<td>Screen Capture of Pure Variants Plugin with MobilePhoto Feature Model</td>
<td>37</td>
</tr>
<tr>
<td>5-7</td>
<td>Partial Source Listing for Main Build Target (OO Version)</td>
<td>38</td>
</tr>
<tr>
<td>5-8</td>
<td>MobilePhoto Build Process Using Pure Variants</td>
<td>39</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank my supervisor, Gail Murphy for keeping me focused, providing amazing guidance and feedback and showing me the ropes in the world of academic research.

Thanks to all the great friends and colleagues I’ve met in Vancouver for making it such an enjoyable experience.

Thank-you to my family for providing support and encouragement throughout my life, and particularly in the past couple years.

Lastly, thanks to Beverley for learning more about computers than she ever wanted to know, and for being there no matter what.

TREVOR J. YOUNG

The University of British Columbia
August 2005
1 Introduction

Within the past few years, there has been a substantial increase in the amount and diversity of micro devices available on the consumer market. Devices such as phones, organizers, watches and music players can now run Java software such as games or business applications. Much of the current focus has been on creating content for cellular phones and personal digital assistants since there are already millions of users who carry these in their pockets on a daily basis. For instance, in China, there are 300 million mobile subscribers, and this number is expected to grow to 500 million within the next two years.\(^1\) Mobile phone operators such as Sprint and Vodafone report that mobile games and other data services now account for roughly 10 percent of their annual revenue, while mobile-game publishers generated $1.2 billion in global sales in 2004.\(^2\)

For a programmer who wants their application to run on more than one device, managing the complexity of many different hardware and software configurations can be more difficult than anticipated\(^3\)\(^4\)\(^5\). In the mobile phone market alone, there are over 250 device platforms serviced by more than 80 different mobile operators in numerous languages.\(^2\) The major hardware vendors in the market like Research in Motion, Palm, Nokia, Motorola, Siemens and Ericsson have anywhere from 20-50 models of devices each. It should not be surprising that the different models and classes of devices all have different programmable capabilities, platforms and requirements.

At the hardware level, these devices have different screen sizes, memory restrictions, varying audio/video features, multiple input devices (such as stylus, keyboard, keypad, thumbwheel, hard buttons), and numerous peripheral extensions such as MP3 players, digital cameras and wireless connectivity using Bluetooth, CDMA, GPRS or 802.11. The software platform is also not standardized across devices. Many of the phones run on the Symbian operating system\(^6\), while most PDA’s and organizers run some version of Palm OS, RIM OS, Microsoft PocketPC or even Linux. This diversity of device features and software platforms creates a large number of configurations that need to be supported. As an example, imagine you are a mobile game developer and have two games that you would like to release to the masses. If you want to deploy your game on ten different devices, and support two different languages\(^7\) you may need to create as many as \(2 \times 2 \times 10 = 40\) different builds. Now consider this problem from the perspective of a mid-sized mobile game publisher that has thirty game titles and needs to support fifty different devices in five languages. You are now potentially managing \(30 \times 5 \times 50 = 7500\) build configurations. As a concrete example of this scenario, international games publisher

---

\(^1\) China’s Ministry of Information Industry (http://www.china.org.cn/english/BAT/127434.htm)
\(^2\) Mobile Entertainment Analyst Article (http://mobenta.com/analysis/item-2138.html)
\(^3\) Challenges and Opportunities in Mobile Games (http://www-128.ibm.com/developerworks/library/wi-austingameconf.html?ca=dtm-539)
\(^4\) Dealing With a Fragmented Java Landscape for Mobile Game Development (http://www.gamasutra.com/features/20031217/tammenkoski_pvfv.htm)
\(^5\) Creation vs. Fragmentation (http://www.screamingthumbs.com/2005/04/creation-vs-fragmentation.html)
\(^6\) http://www.symbian.com
\(^7\) Multi-language support is often a requirement by game portals and mobile network operators
GameLoft recently released seven hundred different versions of the game Splinter Cell for mobile devices in sixty five different countries.\(^8\)

One way to address this problem is to develop the software as a product line, which supports the development of a family of systems that share a common set of features and that are developed from a common set of core assets, but which can be configured in different ways. In comparison to existing systems built using a software product line approach, mobile devices typically demonstrate greater variability in both the software and the hardware platforms, each of which place rigid restrictions on the developer in terms of available memory or disk resources and access to functionality of the operating system. These restrictions mean that some of the common techniques for developing a software product line for desktop applications do not work with mobile systems. Moreover, instead of building a product line that supports two or three desktop or server platforms, mobile applications need to be portable across dozens or even hundreds of different devices.

1.1 Java for Mobile Devices

To help support the mobile developer community who must deliver applications fast, at low cost, and with high quality, Sun Microsystems provides a version of the Java platform, Java 2 Platform Micro Edition (J2ME)\(^9\) that supports micro devices with different hardware capabilities. J2ME is an environment for running applications on mobile devices such as cell phones, personal digital assistants (PDAs), set top boxes, and embedded devices. It includes Java Virtual Machines (JVMs) and a subset of the standard Java APIs. The architecture of J2ME has been designed to allow developers to choose from a combination of configurations, profiles and optional packages to construct a complete Java runtime environment that closely fits the requirements of a particular range of devices and a target market. This means that a developer could select the MIDP 2.0 profile (Mobile Information Device Profile), the CLDC 1.1 configuration (Connected, Limited Device Configuration) and the WMA (Wireless Messaging API)\(^10\) optional library as a base for their application. However, the application would be limited to devices that support that particular combination of MIDP 2.0, CLDC 1.1 and WMA technology, restricting the functionality that the application can provide and the devices upon which it can run. There are many choices of profiles, configurations, optional and proprietary libraries that fit together in various combinations depending on the operating system and hardware of the device, but building an application across these different configurations and profiles is difficult. As a programmer, you must be aware that using components or libraries that are specific to certain devices, profiles or configurations will make your application less portable.

---


\(^9\) http://java.sun.com/j2me/overview.html

\(^10\) Refer to the appendix for a complete list of acronyms used throughout this document
J2ME offers a partial solution to building portable applications as there exist four critical problems with trying to manage a software product line for mobile devices using this technology. First, there are the physical hardware differences between devices, including screen or canvas sizes, font sizes and types, soft and hard buttons, heap memory, storage space, audio/video capability, and colour depth. Second, there have been many reports of differences in the Java virtual machine implementations between device manufacturers. These differences have resulted in a number of bugs that surface on some platforms but not others, and developers are forced to accept the VM embedded in the device by the vendors. A third problem is that many mobile applications (especially for cell phones) can only be deployed through the telecommunications providers that service the device, and these carriers generally enforce certifications or requirements on the application developer. Often before developers can publish their mobile application to general users, they are required to support multiple languages for the carrier, or may be limited by the size of external Java applications that can be installed, or the amount of wireless data it can send. Finally, many of the device vendors release proprietary Java API’s to take advantage of special functions that are only available on their hardware. This is in addition to the standardized optional API’s from Sun that are similarly only supported by certain hardware or software configurations. For instance, the optional Wireless Messaging API (WMA) for J2ME will only work on devices that support the MIDP 2.0 profile and have wide area wireless connectivity. We refer to this problem as the device API fragmentation problem.

1.2 Implementing a Mobile Product Line using AspectJ

The thesis of this research is that aspect-oriented programming can alleviate the device API fragmentation problem for mobile device applications within the constraints of these kinds of devices, such as limitations on the application footprint size and the memory usage. Using aspect-oriented techniques, the application can be partitioned into base components and optional features; a developer can then selectively include only the components that are supported by a target device. In particular, we investigate the use of AspectJ[1] to support this approach. A high level diagram of this process is depicted in figure 1-1.

11 This problem exists in J2SE/J2EE as well, although in J2ME you generally cannot install a choice of VM
Aspect-oriented programming can help with the expression of the product-line, but approaches are also needed to help with the configuration of the product-line. In this thesis, we also investigate how to keep the product line build process simple and scalable so as to handle many devices and features with minimal maintenance. In particular, we investigate build configuration techniques to ensure that only those components which are supported by the target platform will be included in the final executable of the application.

To evaluate the effectiveness of the aspect-oriented approach, we compare the AspectJ implementation with one that uses standard object-oriented programming paradigms and design patterns. To accurately compare the two techniques, we designed and implemented a prototype application in J2ME and tested the execution and build process on a variety of micro devices and emulators. Details of the implementation and build process are provided for each alternative in Chapters four and five, followed by an analysis of the complexity, modularity, resource usage and scalability of each in Chapter six.
2 Related Work

In this chapter, we discuss current programming practices in the development of mobile applications, research into product line techniques that may be suitable for mobile devices, and existing applications of aspect-oriented programming to solve product line problems.

2.1 Product Line Practice in the Mobile Industry

Commercial J2ME applications for mobile devices are not very abundant because J2ME is a relatively new technology and it has only recently become a standard option deployed with new mobile phones on the market. For example the J2ME virtual machine does not even come pre-installed with the Palm OS or Microsoft’s Pocket PC. In addition, it has taken time for software development kits (SDK’s), toolsets and integrated development environments (IDE’s) to offer developers support in mobile development.

The lack of a standardized platform has led to the industry adopting a number of ad hoc approaches. An interview with Allen Lau, CTO of Tira Wireless suggests that a large number of mobile application providers tend to target only the most dominant device platforms, and consider porting to other devices only if the demand is there. Allen further describes the J2ME (game) industry as being divided into application developers and specialized porting companies. In many cases, publishers of original application content develop one or two generic versions of their J2ME application for representative ‘mass market’ devices (such as the Nokia Series 60 and Series 40 platforms) and then arrange for another company to port, optimize and test the application for all the other handsets it must support. This seems to be a desirable option for small mobile developers who do not have the resources or access to all the hardware to port and test it on their own. In fact, there are a number of companies that have built successful business models around porting J2ME games and business applications across various devices. Unfortunately, there is very little information on the processes these porting companies follow. This indicates a very real problem for programmers or small software companies who want to develop applications for devices in the mobile industry. It is a problem that is likely to get worse before it gets better as the number of phones, PDA’s and other micro devices deployed commercially increases over the next two to three years. Some recent open source projects such as Antenna, J2ME Polish and NetBeans Mobility offer utilities to help alleviate this problem but tend to help with only specific problems such as UI.

---

12 The first KVM for Palm was released in 1999. Standardized MIDP 1.0 specification was released in 2000, and hardware manufacturers began embedding it in devices after that. (http://java.sun.com/j2me/)
15 http://antenna.sourceforge.net/
16 http://www.j2mepolish.org/
17 http://www.netbeans.org/kb/articles/mobility.html
consistency or preprocessing support and do not yet offer a complete solution to building mobile product lines.

2.2 General Product Line Practices

With the advances of wireless technology and reduced costs of micro hardware, the mobile computer industry has only recently become a popular platform for custom application development. As a result, product line techniques in the mobile space are not well established or implemented. The software development environment for this technology has some significant differences compared to desktop computing which means typical product line approaches are not necessarily effective. Outside the mobile application domain, there are a variety of product line implementation techniques that have been used with different degrees of success. Some are used in isolation; others can be used in combination with different product line approaches or complementary component and framework methodologies. This section presents general product line practices that could be considered for mobile application development. In the cases where these general techniques have been tried with mobile development we comment on their applicability to the device API fragmentation problem.

2.2.1 Multiple Copies and Version Management

The ‘brute force’ approach to building product lines is to write the application on a single platform, and then create copies for each additional device you need to support and make platform specific modifications to each copy. This approach is not a generally recommended approach but might be a tempting ad-hoc solution when timing or resources are in short supply. Applying this technique to mobile product families would not be difficult because these applications typically involve less source code than a general-purpose application. Discussions with developers in the J2ME space\(^{18}\) indicate this is often the approach currently used to handle differences in device variability. This might suffice for two or three different devices, but when you need to deploy your application on dozens or hundreds of devices, keeping all the different copies in sync would be very hard to manage. Even a simple change to a common feature might require the developer to update every copy of the application.

This approach can be made easier with the help of version control utilities. Configuration management software such as CVS (Concurrent Version Control System)\(^{19}\) offers branching, versioning and merging techniques that allow you to create sub-branches off of a stable code base to take different copies of your source code in different directions. One technique of this approach is called a Platform Line or Permanent Variant Branch\(^{20}\). This method attempts to track platform specific components and sources by creating a separate code base for each device or platform variant. It is better than simply managing

\(^{18}\) Contacts from mobile and wireless companies Tira Wireless, iAnywhere Solutions, Eontec and others

\(^{19}\) https://www.cvshome.org/

\(^{20}\) http://www.cmcrossroads.com/bradapp/acme/branching/branch-creation.html#PlatformLine
directories and resources at the file system level because most configuration management software provides the ability to automatically merge (push or propagate) baseline code changes to the branch variants. For example, figure 2-1 shows how the main (trunk) code base might contain source code that is compliant with the MIDP 1.0 standard which would run on any J2ME enabled device. Separate branches could then be created for different platforms that use optional or proprietary APIs that are only supported by specific devices. Thus, a separate branch could be created for Nokia devices or Motorola devices, and these can be further sub-branched into specific models of the manufacturer. When changes are made to the common code base, it can be merged back into the branches to keep all the product instances in sync.

Figure 2-1: Example Version Control Branching Technique for Mobile Product Lines

However, with this approach you can end up with a very complex tree structure containing one branch for every individual device class. Depending on how you structure your branches, you may also have a lot of duplicate code. For instance, if both Nokia series 60 and Motorola i-Series phone models support an SMS text messaging feature, you will either have duplicate code in each branch, or have to re-structure your entire tree to keep a single component base for the common SMS code. Once again, this might be an acceptable solution for managing two or three product platforms, but scaling this to dozens of different product instances could be very difficult to manage. Another factor to consider with this approach is that re-structuring your branches after development is not easy because of the static way in which most version control packages handle these features. You must typically know the details of all the platforms you wish to support when you create your branching structure, and you must also know what features or
components each of these platforms support. Developers could take this one step further and branch at a finer granularity (e.g., create branches for each component or feature), but this comes with greater maintenance responsibility for the code bases, and necessitates a more complex build system.

2.2.2 Object Oriented Frameworks and Patterns

The ability to re-use the same software artifact in many different products is an important factor to reduce development time. Object-oriented frameworks[2] and design patterns[3] are an effective way of building re-usable components and sometimes form the basis of product line development practices. Some of these object-oriented frameworks have component models integrated into them that allow plug-n-play behaviour for software applications with many optional features. For example, the Eclipse Integrated Development Environment (IDE)\(^2\) is used as the base for a family of IBM products such as WebSphere Studio, WebSphere Device Developer, Rational XDE\(^2\) along with other variations that provide unique features depending on the target users. The Eclipse software project uses a Java framework that provides a platform for producing different product instances from a set of core assets. This makes it easy to mix and match different plug-ins (with unique features) to form a complete product instance that targets a specific developer domain. A number of software companies have used the base of Eclipse to form their own IDE product families. However, the infrastructure required to enable this easy plug-n-play of optional features increases the size of the core software. The size constraints of mobile devices would make an approach such as this one difficult to deploy because it increases the final executable size. In addition, some of the basic object oriented features required by these frameworks are not available in the limited J2ME libraries. For instance, as of MIDP 2.0 there is no support for things like reflection, JNI (Java Native Interface), or user defined classloading.

2.2.3 Code Generation and Template Programming

There is also an increasing interest in applying generative programming techniques to product line practices.[4] These approaches often rely on modeling tools to define abstract components and artifacts for an application, and then generate product specific source code depending on the target platform selected. Template programming is another technique used in this area. Davis et al. recently conducted research investigating a generative-based approach for application tailoring of mobile devices [5]. They focused on defining a model that can generate web-based content or standard J2ME source code, rather than on the variability of device platforms or models. They tested two approaches: one using XSLT templates to transform user-defined properties into source code, and another that uses a tool to define the meta-model of an application and requires an implementation of a custom interpreter in C++ to generate code from this model. Davis et

\(^2\) http://www.eclipse.org
\(^3\) http://www-306.ibm.eom/software/awdtools/developer/
al. state that "the interpreter must contain transformation code for every artifact needed in the tailoring process" and "an interpreter must be written for each target device" ([5] p.4). This approach thus pushes the work from maintaining multiple source code bases to maintaining multiple design artifact bases, and continuously tweaking the generation or transformation rules and code to support new product instances. Trying to support a large number of product instances using this technique would likely require substantive effort.

2.2.4 Pre-Compilation Utilities

The use of utilities or tools that include build configuration features or pre-processing capabilities is something that is becoming popular among mobile programmers. One of the most common uses of the preprocessor in C is to include code that is conditional on the execution platform. Since Java was designed to be platform independent ("write once, run anywhere"), it is generally thought that a preprocessor should not be needed. However, with the variability in mobile devices and the fragmented API problem inherent in the J2ME platform, this is not the case. Tools such as J2ME Polish, Antenna and NetBeans now provide custom Java preprocessors, which allow a developer to reduce the size of the compiled code by including only platform-specific code for the target device. Build configuration techniques achieve a similar result at a higher-level by excluding entire source files if they are not supported by the target platform. These utilities may be used in combination with other product line techniques, and can be effective at reducing the final application size of mobile applications.

2.3 Aspect-Oriented Product Line Practices

Various approaches to using aspect-oriented technology to develop software product lines have recently started to emerge. Much of this research is focused on using aspects to customize middleware components and servers or desktop applications that have less variability across hardware and software platforms, and allow for a larger application size with more available resources.[6] A study by Murphy et al. explored methods of separating features from application source code using AspectJ [7] which could apply to the design of mobile applications. Loughran, Rashad et al. [8, 9] discuss a hybrid approach using aspect-oriented programming and Frames to support the evolution of software product lines. They provide some simple implementation examples for a generic caching component, but at the time of this writing it has not been tested for mobile applications. Lopez and Batory demonstrate the use of AspectJ to build a product line of graphing algorithms.[10] The application in their case study is built using the standard Java platform which does not suffer from device API fragmentation. Part of their findings identify that AspectJ has no built-in support for handling design rules and feature interaction, thus programmers have to manually perform the complex and error prone task to select the artifacts necessary to create a specific product instance. In more recent work, they evaluate a number of modularization technologies (including AspectJ) in terms of how they support the creation of features as modules. Many of their findings regarding basic properties for feature modularity are technology independent and could
be applied to the mobile domain[11]. Bodkin, Colyer and Hugunin released a practitioner report[12] discussing their experience in industry on a project where they applied aspect-oriented programming to achieve platform independence for an application. Their motivation for using aspect-oriented programming was to target multiple runtime environments with a single source code base for a middleware product line. The implementation was successful at modularizing typical cross-cutting features such as logging, tracing and error handling, but noted concerns with immature tools and difficulty of developers to adopt an ‘AOP mindset’ when programming. Their report indicates that aspect-oriented techniques can help problems with application portability, though issues in mobile development such as device API fragmentation, memory restrictions and footprint optimization were not present in the case study.

We are aware of two groups who have used an aspect-oriented feature weaving approach with Java development on mobile platforms. At the Federal University of Pernambuco, Alves has applied a variation management technique with AspectJ to J2ME game development for mobile phones. They use an incremental approach to re-factor existing J2ME games to isolate commonality from variability within mobile software[13]. Other papers from the same group investigate the use of AspectJ to enable adaptive J2ME applications to behave differently according to changes in its environment.[14], [15]. Anastasopoulos provides a set of specific areas to investigate when evaluating aspect-oriented programming techniques as a product line implementation technology, and provides a case study using J2ME and mobile phones.[16] Similar to the work described in this thesis, the Anastasopoulos case study also looks at J2ME and AspectJ. In contrast, the Anastasopoulos work is intended as a general discussion on the relevance of aspect-oriented programming in product line development and does not address the specific problems related to J2ME and mobile applications, such as device API fragmentation.

Our research extends some of these preliminary findings by focusing on building product lines for mobile applications that use aspect-oriented techniques starting from scratch (as opposed to re-factorimg existing applications). This study encompasses a wider range of micro devices than previous work, that includes various mobile phones, personal digital assistants and pager devices. It compares both the source code and the build process for an aspect-oriented implementation and a straight object-oriented implementation, focusing specifically on the problem of device API fragmentation.
3 MobilePhoto Prototype Application Design

To investigate the J2ME device API fragmentation problem across devices, we designed a mobile application that supports different feature combinations for different devices, depending upon the capabilities of the device. Our prototype application is a photo album (image viewer) appropriate for cellular phones, personal digital organizers or Blackberry devices although it is not limited to these. The requirements we devised for our photo album application include several different features. Some of these features incorporate APIs that can execute on any J2ME enabled device (e.g., anything that supports MIDP 1.0), some can only execute on a single, specific model (e.g., Blackberry 7210), while other features may be supported by a subset of devices (e.g., all mobile phones that support SMS text messaging).

The application was designed from the outset as a product family that needs to run on multiple devices with different variations in API support. The structure of the application is split into two parts: the base application and the optional features. The base application contains the core functionality that allows a user to select photo images from a list and view them on the device screen. The optional features comprise functionality that might only be supported by certain classes of devices. For example, a phone that supports text messaging could have an optional feature to wirelessly send a photo to another user via SMS. Devices such as the Blackberry that support e-mail could similarly mail a photo to a friend over a supported wireless network. However, not all mobile devices support the optional or proprietary APIs that offer this functionality, so it would be inefficient (as well as error prone) to include such a feature on a device that cannot support it.

This approach to building a product line follows a decomposition approach [16] in which the common features are separated from the variant features so that optional components can be attached to the common functionality on demand. An alternate approach is generalization (or data controlled variation) in which generic code is created to encompass many different variations. We chose not to use generalization because we felt the decomposition method would allow for better tool support and more potential for automating the build process.

3.1 The Base Application

The components of the base application are implemented using the lowest common denominator of J2ME to ensure they can run on any J2ME compliant device. For our prototype, this translates to an implementation that uses only MIDP 1.0 and CLDC 1.0 API's, and avoids all optional or proprietary libraries. Therefore all of the base application features listed below will run on any J2ME enabled device.

Table 3-1 shows the five basic features that are included in the base application for MobilePhoto.
Table 3-1: Basic Features Included in All MobilePhoto Product Instances

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Photo Album</td>
<td>Allows a user to define new photo albums to store related categories of pictures on the device. Album information is persisted using RMS (J2ME Record Management System).</td>
</tr>
<tr>
<td>Store Photo</td>
<td>Handles the conversion and persistence of photo image files to the mobile device file system using RMS.</td>
</tr>
<tr>
<td>Add / Delete Photo</td>
<td>Allows a user to permanently delete photos from the device, or add new photos to defined albums.</td>
</tr>
<tr>
<td>Label Photo</td>
<td>Allows a user to label a photo with text. Labels appear in the display list, and could be used for future search functionality.</td>
</tr>
<tr>
<td>View Photo</td>
<td>This displays a selected photo on the device screen.</td>
</tr>
</tbody>
</table>

3.2 The Optional Features

The optional feature components of MobilePhoto are de-coupled from the base application so they can be easily added or removed, depending on the device you are targeting. As a result, the components of the base application are not aware of or dependent on any optional classes. This setup allows the base application to successfully compile when optional features are excluded from the build. Optional features are not standalone components, and require the base application to execute and compile; they only extend the functionality of the base MobilePhoto application. The optional features available are shown in table 3-2.
### Table 3-2: Optional Features Available for MobilePhoto Product Instances

<table>
<thead>
<tr>
<th>Optional Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send Photo via SMS</td>
<td>Allows a user to send a photo to another user via Short Messaging Service.</td>
</tr>
<tr>
<td>Send Photo via Email</td>
<td>Allows a user to send a photo to another user via Email.</td>
</tr>
<tr>
<td>Link Photo with Address Book Entry</td>
<td>Allows a user to associate an entry in their contact list with a photo in their album.</td>
</tr>
<tr>
<td>Display Photo of Incoming Caller</td>
<td>Intercepts incoming phone calls and displays a photo of the associated caller while ringing.</td>
</tr>
<tr>
<td>Play melody of Incoming Call</td>
<td>Intercepts incoming phone calls and plays a custom ring-tone of the associated caller while ringing.</td>
</tr>
</tbody>
</table>

All of the optional features listed above could only be implemented using proprietary libraries, or optional J2ME libraries that will only work on certain mobile devices. Table 3-3 below shows a sample of devices that can support the optional features for MobilePhoto. Note that it lists only six models, from the hundreds of J2ME enabled devices currently available.
### 3.3 Architecture

Following standard practice[17], MobilePhoto utilizes the Model View Controller (MVC) design pattern to update the applications views (UI screens) as the data changes, and to invoke the appropriate actions as the user dictates by pressing buttons or using menu items.

The model represents the photo albums and the photos that the albums contain. Photo albums are just a way to categorize a group of photos stored on the device. Photos are

---

23 MobilePhoto SMS Feature uses the optional WMA API from Sun  
24 MobilePhoto Email feature uses the proprietary email messaging API from RIM  
25 MobilePhoto Address Book feature uses the optional PIM API from Sun  
26 MobilePhoto Incoming Call (Photo and Ringtone) features use the proprietary Telephony API from Motorola
instances of `javax.microedition.lcdui.Image` objects when loaded in the Midlets memory. All of this information is persisted to the devices internal storage using J2ME’s record management system (RMS). RMS is a very crude database implementation that can only store a record Id (assigned automatically by J2ME) and a series of bytes. MobilePhoto creates two record stores for every photo album defined. One store holds the bytes of the images, and a second one holds the metadata of the images. Metadata information is stored as a delimiter separated string of fields and then converted into bytes that are saved in RMS. Tables 3-4 and 3-5 describe the record store ‘schema’ used by MobilePhoto:

**Table 3-4: Data Store Schema for MobilePhoto Image Data**

<table>
<thead>
<tr>
<th>Record Id</th>
<th>Int</th>
<th>Automatically assigned by J2ME RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Data</td>
<td>Byte Array</td>
<td>Images are converted into byte arrays.</td>
</tr>
</tbody>
</table>

**Table 3-5: Data Store Schema for MobilePhoto Image MetaData**

<table>
<thead>
<tr>
<th>Record Id</th>
<th>Int</th>
<th>Automatically assigned by J2ME RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Metadata</td>
<td>Byte Array</td>
<td>The metadata is a string of delimiter separated fields that are converted into a byte array. Specific fields are listed in the next rows.</td>
</tr>
<tr>
<td>Record Id</td>
<td>String</td>
<td>Record Id of this entry (stored as string for easier access later).</td>
</tr>
<tr>
<td>Foreign Record Id</td>
<td>String</td>
<td>Record Id of the corresponding Image Data record in the data store.</td>
</tr>
<tr>
<td>Album Name</td>
<td>String</td>
<td>The name of the photo album this image is associated with.</td>
</tr>
<tr>
<td>Image Label</td>
<td>String</td>
<td>A user defined label.</td>
</tr>
<tr>
<td>Phone Number27</td>
<td>String</td>
<td>A Phone number for an image to associate with address book entries. (Only used for certain features)</td>
</tr>
<tr>
<td>Audio File Name27</td>
<td>String</td>
<td>Name of an audio file for an image to associate with melodies or ringtones. (Only used for certain features)</td>
</tr>
</tbody>
</table>

27 Phone Number and Audio File fields are only used by certain optional features
The view (of the Model View Controller) consists of the actual screens that display the photos, albums and related commands to the user. Since the purpose of this application is to analyze different solutions to device API fragmentation, the user interface was kept very simple and offers only one view for the data model. Screens are basic MIDP forms and lists, and a canvas object is used to display the images. It should be noted that when using the standard MIDP LCD UI widgets, the developer has little control over how the UI is displayed on different devices. The virtual machine of the device automatically attempts to display widgets and map menu items to buttons depending on the physical properties available to it (such as screen size and hard buttons). As a result, items in the UI such as menu commands appear differently on a Nokia phone vs. Palm device vs. Blackberry device. For details on how J2ME handles this, refer to Sun’s MIDP specification.\(^\text{28}\)

A controller class is used to receive user actions and call the corresponding events. Events are typically handled by invoking a separate thread to do some work in the background so as to avoid issues with contention for the device display or interruptions from incoming phone calls or external events.

\[^{28}\text{http://jcp.org/en/jsr/detail?id=037}\]
4 MobilePhoto Implementation using AspectJ

For one implementation of MobilePhoto, we used AspectJ. Our intent was to describe the optional features of the application as aspects. Figure 4-1 describes our approach: the base application was written in standard J2ME using only MIDP 1.0, with the optional features implemented using optional or proprietary J2ME extensions and bound to the base components using aspects. The base application and optional features used in our prototype are concrete implementations of the design and architecture described in chapter 3.

A given configuration of the system (figure 4-1) weaves in only those optional features supported by a device. In this way, we are able to build a product line using aspect-oriented programming.

Using the aspect-oriented implementation of MobilePhoto we built a variety of product instances by combining different combinations of optional features. (Table 4-1)

Table 4-1: MobilePhoto Product Instances (AspectJ)

<table>
<thead>
<tr>
<th>Base Application only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Application + SMS</td>
</tr>
<tr>
<td>Base Application + SMS + Caller Photo</td>
</tr>
<tr>
<td>Base Application + SMS + Caller Photo + PIM</td>
</tr>
<tr>
<td>Base Application + SMS + Caller Photo + PIM + Play Ringtone</td>
</tr>
<tr>
<td>Base Application + Email</td>
</tr>
</tbody>
</table>

29 The final Jar files for these applications are available for download at http://www.cs.ubc.ca/labs/spl/
The aspect-oriented version of our MobilePhoto application was tested on a number of device emulators from different vendors, in addition to testing on selected (real) hardware devices listed below:

- Palm Tungsten C (Hardware device)
- Nokia 6600 (Hardware device)
- Sun Generic Wireless Toolkit Emulator (v. 2.1)
- Sun Generic Wireless Toolkit Emulator (v. 2.2)
- Siemens CF62 Emulator
- Siemens CX65 Emulator
- Siemens SL55 Emulator
- RIM Blackberry 7200 Simulator
- RIM Blackberry 6700 Simulator
- RIM Blackberry 5800 Simulator
- Palm OS Emulator (with Sun MIDP 1.0 KVM for Palm)
- Palm OS Simulator (with IBM J9 KVM for Palm)
- Motorola i730 Emulator
- Motorola i830 Emulator
- Motorola i860 Emulator
- Nokia 3410 Emulator
- Nokia 7210 Emulator
- Nokia Series 60 Emulator

Note that MobilePhoto jars were only tested on emulators or hardware that supported the particular feature-set of a given version.

### 4.1 De-coupling Optional Features using Aspects

To enable the exclusion of optional features we had to ensure there was only one-way communication between the optional components and the base components. As a result, we had the design rule that optional components can call public methods on any class in the base application, but the reverse is prohibited (base components cannot call methods that exist in optional components). This flow of information is referred to as a Class-directional association link, and results in the classes being more re-usable across product instances.[18]

An optional feature in MobilePhoto is contained in a single java package, separate from the components of the base application. The components that make up a feature are a set of standard J2ME java files (that rely on optional or proprietary libraries) and one or more AspectJ aspects (.aj files). Figure 4-2 shows the package diagram for MobilePhoto, indicating the packages that make up the base application components, and additional packages that contain the optional features.
Figure 4-2: MobilePhoto Package Diagram

For example, the feature that allows a user to send photos using SMS messages is contained in the `ubc.midp.mobilephoto.sms` Java package. It includes the four Java source files, and one AspectJ source file displayed in the lower portion of figure 4-3. The `BaseMessaging` class at the top of the diagram is part of a base application package.
No classes in the base application (ubc/midp/mobilephoto/core/* packages) are dependent on classes found in the SMS package. Details on how the optional components are integrated with the base application are discussed in the next section.

4.2 Binding Features to the Application using Advice

In this version of MobilePhoto, AspectJ is used to bind the optional features to the base application using static weaving. In particular, we use inter-type declarations, execution pointcuts and after advice.

In the context of our prototype application, there are two important areas of program flow where optional features need to integrate with the base application. One is the menu initialization, where additional menu items must be added that allow the user to invoke new features. The other is the controller logic, where the application must decide what action to take based on the buttons that were pressed by the user or events received by the system. The controller is part of the base application, but it must be aware of the new actions available to it that are exposed by classes associated with these optional features.

To illustrate how this fits together, consider a version of the MobilePhoto product that contains the optional feature to allow a user to send photos over SMS. There are three points in the design of our base application where the SMS feature must integrate: 1) new
menu items must be added for the user to invoke the feature, 2) the application controller
must know what action to take when a user invokes this feature and 3) the application
must be notified when a new photo is being received from another SMS user. We use
three pointcuts in the SmsAspect (figure 4-4) to identify these join points.

![Design Diagram Showing SMS Feature Interaction with Base Application](image)

The initMenuItems pointcut captures the spot at which the menu items are created
when the application first loads. The SmsAspect uses this pointcut to bind advice that
will execute after the base application menu items are initialized and inserts two new
menu items for this feature: ‘Send a Photo over SMS’ and ‘Receive a Photo over SMS’.
The simple, yet effective source code for this is shown in figure 4-5. A full listing of the
source code for the SMSAspect is in the Appendix.
//Introductions:

//Define new menu commands associated with SMS feature
Command smsSendCommand = new Command("Send SMS", Command.ITEM, 1);
Command smsReceiveCommand = new Command("Receive SMS", Command.ITEM, 1);

//Capture event when UI initializes the menu items
pointcut initMenuItems(List l) : args(l) && execution(* ImageList.initMenu(List));

//Add SMS Specific menu items to the main UI
after(List l) : initMenuItems(l) {
    l.addCommand(smsSendCommand);
    l.addCommand(smsReceiveCommand);
}

Figure 4-5: Partial Source Listing for SMSAspect of MobilePhoto

New menu commands for the SMS feature are defined as inter-type declarations (or introductions) in the aspect. The pointcut identifies the method in the base application where the menu items are initialized, and obtains a handle on the List argument that stores the menu commands. The after advice then adds the additional commands that allow a user to send and receive photos over SMS.

In a similar manner, the SmsAspect also defines a pointcut to capture the spot where the command handler (controller) receives user input and decides what action to take. There is advice defined in the aspect to handle events that are meant for the SMS feature such as pressing one of the SMS menu items. Finally, there is a third pointcut that captures when the SMS listener thread has received a system event indicating an incoming SMS photo from another user has arrived.\(^{30}\) (The diagram of the pointcuts for the SmsAspect is pictured in figure 4-4).

This method of binding with AspectJ pointcuts and advice is used to integrate all of the optional features in the MobilePhoto application. As an example, the UML sequence diagram in figure 4-6 indicates the flow of execution when a user invokes the feature to send a photo over SMS. Note that the handleCommand() method in the BaseController does not contain logic to invoke the SMS command. It is only after the handleCommand() method completes that the SmsAspect pointcut is triggered, and the aspect executes advice to send the SMS message.

\(^{30}\) Full source code for the MobilePhoto application is available at [http://www.cs.ubc.ca/labs/spl/](http://www.cs.ubc.ca/labs/spl/)
Managing Multiple Features Using Precedence

One of the difficulties in managing a large collection of optional features for an application is that some features may conflict with, depend upon or otherwise interact with other features. Effectively managing the possible combinations and dependencies for a complex product line can require a lot of effort, and is beyond the scope of this thesis. AspectJ provides one simple construct that can provide some control over this problem. The `declare precedence` statement allows you to specify the order in which advice is applied.

This technique is especially useful to help manage multiple features that offer similar functionality. For instance, the MobilePhoto application has one feature that will play a custom ringtone when a user receives an incoming call. Another (separate) feature has the ability to pop-up the photo of a person calling during an incoming call. These two features potentially conflict with one another, or could also be combined together to play a custom ringtone and display the caller’s photo. In this scenario, we have used the `declare precedence` statement to control which action happens first. The line of code in figure 4-7 indicates that the ringtone feature has higher precedence over the feature that displays a user photo:
**declare precedence :** Mp3IncomingCallAspect, MotoIncomingCallAspect;

Figure 4-7: Declaring Precedence to Manage Feature Interaction in AspectJ

This specifies that for *after* advice, the ringtone feature executes last. (For *before* advice, it would execute first).

We have not investigated any methods of dynamically preventing an optional feature from executing if it directly conflicted with another feature built into the application. This would have to be done by manually excluding one of the conflicting features during the build process. Section 5.3 discusses the potential of a third party Eclipse plugin that can help (statically) manage these situations during the build process. Using this approach puts the burden of managing features on the developer, because features must now be aware of when they are dependent on or interact with other feature components.

### 4.4 The Build Process

Building a J2ME application is not straightforward even when a product-line is not considered. It requires a few additional steps compared to developing standard desktop Java applications. At a minimum, you must additionally preverify the byte-code, define application descriptor properties in a JAD (Java Application Descriptor) file and package the application as a JAR (Java Archive) file. After these steps are complete, you may elect to obfuscate the code to reduce the size of the application, and might need to use a utility to convert the byte codes into a platform specific format (e.g., .cod for the Blackberry platform or .prc for Palm platform). Devices may also support different external resource formats, such as mp3 versus .wav for audio playback, or place constraints on memory usage, image sizes, size restrictions for sending wireless data etc. The list of constraints to manage increases with the number of devices you need to support, which adds to the complexity in managing the build process. To add to the confusion, most of the major device vendors have released proprietary SDK’s, IDE’s or toolkits to help developers write applications for their specific platforms. This group includes the (generic) Sun Wireless Toolkit, Blackberry Java Development Environment, Nokia Developer’s Suite for J2ME, Motorola iDen SDK for J2ME, Siemens Mobile SDK, Palm OS (J2ME) Developer Suite, and the Qualcomm BREW SDK to name a few.

---

33 [http://www.forum.nokia.com/tools](http://www.forum.nokia.com/tools)
To alleviate some of this complexity, we used Apache’s ANT\textsuperscript{38} and an open source extension to it called Antenna\textsuperscript{39} that offers additional support for J2ME specific build tasks. Although ANT does not eliminate all of the dependencies on installing third party IDE’s, SDK’s or tools, it does let you separate the build process into a simpler chain of tasks and run a single command line script to run the necessary steps.

Because this thesis focuses on the variability of optional and proprietary API support across micro devices, the build system has to make it easy for a developer to select which optional features should be compiled into the final application or product instance. That is, the compilation and build process should enable the inclusion of only those features that are supported by a device. This criterion is important because of the restrictions mobile vendors and operators place on the size of applications that can be installed. As an example, Nokia Series 40 phones place a 64 KB maximum on the size of Midlets compared to some Nokia Series 60 phones that support up to 4MB.\textsuperscript{40} Other manufacturer’s have varying application size restrictions, with some older models allowing as little as 32 KB for your entire application. Having control over the included features would make it easier to keep the application size minimal, define product configurations (combinations of features and base code) and eventually allow one to perform bulk builds for a complete line of dozens of different platforms. For example, if you are running a build for the phone platform then you should be able to easily specify that you want the feature to send photos via SMS. However, if you were building the same application for a Blackberry or Palm device, you want to ensure this feature is not included, because it is not supported on those platforms and its inclusion would cause the application to be larger than necessary, and to most likely throw exceptions about missing classes when the Midlet loads.

The control of feature inclusions for MobilePhoto is done during the build stage by using a configuration file. A list is maintained that includes the names of all source files or packages (directories) to be included in a particular build. This technique also hinges upon the requirement that the base application is not dependent (from a compiler perspective) upon any of the optional features. Thus excluding a feature will not cause the build to fail. All classes for a given feature are also isolated in individual packages. This makes it easier to select a single directory when you want to include a feature in the build. This build configuration compile list, along with a properties file that specifies environment variables such as the classpath, library locations, conversion utilities etc. are used as input to the build process. Figure 4-8 shows the components used in the build process for MobilePhoto.

\textsuperscript{38}http://ant.apache.org/
\textsuperscript{39}http://antenna.sourceforge.net/
\textsuperscript{40}http://www.forum.nokia.com/main.html
Using this procedure keeps the product line builds simple by isolating the variable components to a properties file and a build configuration compile list. Before creating a new product instance, we update the build properties to ensure the classpath settings are correct, and we update the compile list by selecting which features should be included in the build. With the ANT and AJDT plugins for Eclipse, we are able to control the entire build process within the IDE. Figure 4-9 shows the Eclipse workspace for MobilePhoto with the ANT view in the bottom right pane and the compile list view in the top right pane.
Figure 4-9: Eclipse View with ANT and Build Configuration Plugins
5 MobilePhoto Implementation using Object Oriented Programming

Aspect-oriented programming is one approach to tackling the device API fragmentation problem for mobile devices. But the use of aspect-oriented technology in industry is still gaining momentum, and particularly in the mobile space it is not a commonly utilized technique. Hence, to effectively evaluate the AspectJ approach for developing mobile product lines we compared it with current practices used in industry. This chapter discusses the details of how the MobilePhoto application was written using standard object-oriented (OO) programming practices and design patterns [3]. In this section, we discuss an object-oriented approach to handle the variance in product instances due to features that are implemented using optional or proprietary API’s which are not supported by all platforms.

It should be noted that aspect-oriented programming and object-oriented programming are complementary, not competitive technologies. In fact, the AspectJ version of MobilePhoto uses standard object-oriented techniques that are available in Java (J2ME). The intent of our object-oriented implementation is to compare how well known programming techniques handle the device API fragmentation problem with mobile devices, as compared to aspect-oriented techniques.

Many developers are turning to the use of J2ME preprocessors\textsuperscript{41} to handle the different device characteristics in their Java applications. Some common design patterns such as the Chain of Responsibility\textsuperscript{[3]} can also help structure mobile product line development. A given configuration of the system (figure 5-1) uses a preprocessor to bind only those features that are supported by a target device. Each feature implements its own controller to handle actions specific to it.

This chapter discusses how we incorporated these practices in the product line, and briefly comments on a proprietary Eclipse plugin called Pure Variants that is helpful in managing the dependencies and conflicts of optional features that are used in MobilePhoto.

For the object-oriented implementation of MobilePhoto we built the same combination of product instances as the aspect-oriented version. Table 5-1 lists these.

Table 5-1: MobilePhoto Product Instances (Object-Oriented)

<table>
<thead>
<tr>
<th>Product Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Application only</td>
</tr>
<tr>
<td>Base Application + SMS</td>
</tr>
<tr>
<td>Base Application + SMS + Caller Photo</td>
</tr>
<tr>
<td>Base Application + SMS + Caller Photo + PIM</td>
</tr>
<tr>
<td>Base Application + SMS + Caller Photo + PIM + Play Ringtone</td>
</tr>
<tr>
<td>Base Application + Email</td>
</tr>
</tbody>
</table>

The object-oriented version of our MobilePhoto application was also tested on the same set of emulators and hardware that the aspect-oriented version was tested on:

- Palm Tungsten C (Hardware device)
- Nokia 6600 (Hardware device)
- Sun Generic Wireless Toolkit Emulator (v. 2.1)
- Sun Generic Wireless Toolkit Emulator (v. 2.2)
- Siemens CF62 Emulator

42 The final Jar files for these applications are available for download at http://www.cs.ubc.ca/labs/spl/
Note that MobilePhoto jars were only tested on emulators or hardware that supported the particular feature-set of a given version.

5.1 De-coupling Optional Features Using OO Design patterns

As with the aspect-oriented design, the key in allowing optional features to be easily added or removed from an application written using object-oriented programming is ensuring that the base application is not dependent on them for compilation or execution. The package structure for the object-oriented implementation of MobilePhoto is the same as the aspect-oriented version (See figure 4-2). Optional features still needed to add functionality that would let the controller determine what action to take when they are invoked. Our object-oriented approach uses inheritance, the chain of responsibility design pattern and a Java preprocessor to accomplish the de-coupling of components. The preprocessor was only used when necessary to ensure the source code would not become cluttered with non-Java (preprocessor) statements and syntax.

The object-oriented implementation used an MVC design where each feature implemented its own controller. The object-oriented version of the MobilePhoto implementation includes a ControllerInterface interface. This interface is implemented by the BaseController and all ‘feature specific’ controllers extend the BaseController class. Figure 5-2 shows the class diagram for the object-oriented version of the SMS feature.
In figure 5-2 the `BaseController` and `BaseMessaging` classes are part of base application packages and the other classes form the SMS feature. Following the chain of responsibility pattern, on startup all the controllers (`BaseController` plus one controller per feature) are instantiated and each controller will maintain a handle to the next controller in the chain. When a menu item or button is pressed, the first controller in the chain (`BaseController`) calls its `handleCommand()` method to see if it can satisfy the request. If it cannot, then control is passed to the next controller in the chain. The command progresses down the chain until one of the controllers is able to satisfy the request and take the appropriate action. Using this approach allows us to limit the use of preprocessor statements to creating new controllers in the main Midlet class and ordering the chain. A snippet of the source code from this section is in figure 5-3.
rootController = new BaseController(this, model);
prevInChain = rootController;

#ifdef includeSmsFeature
    nextInChain = new SmsController(this, model);
    prevInChain.setNextController(nextInChain);
    prevInChain = nextInChain;
#endif

#ifdef includeBlackberryEmailFeature
    nextInChain = new BlackberryEmailController(this, model);
    prevInChain.setNextController(nextInChain);
    prevInChain = nextInChain;
#endif

Figure 5-3: Partial Source Listing of MainUIMidlet Class (OO Version)

In the current implementation of MobilePhoto there are six optional features, and therefore six sub-classes of the BaseController implementation that are initialized in the chain. It is important to note that the order of the controllers in the chain is significant. This ordering is particularly crucial where two features offer similar functionality, such as ones that take action when intercepting incoming phone calls. In the example where the feature that plays a custom ringtone is built into the application as well as the feature that displays a photo of an incoming caller, their order in the chain will dictate which feature is invoked during an incoming call. With the object-oriented programming implementation, either the ringtone will play, OR the photo will display, depending on which one is first in the chain.

5.2 Binding Features to the Application using a Pre-Processor

A popular (current) object-oriented solution to the device API fragmentation problem is to use a Java preprocessor to help manage device variability. Pre-processing functionality has already been integrated into the most popular open source J2ME utilities and Java IDE’s. The Java preprocessors available are similar to the ones known from other languages like C and C++. A preprocessor is helpful when trying to maintain a single code base for several devices that have unique bugs, API support and hardware or software properties. Using a preprocessor allows for conditional compilation and inclusion of entire source files or small sections of code into another. Although preprocessor usage is often considered poor programming practice by object-oriented enthusiasts, it has been widely accepted by mobile device programmers who want to minimize the size of their J2ME applications.

In the object-oriented version of our MobilePhoto implementation, the developer edits a properties file to indicate which optional features should be included by the preprocessor. For example, to include the feature that allows a user to send photos over SMS, and the
feature to display caller photos for incoming calls the following line must exist in the properties file:

```java
preprocessor.symbols=includeSmsFeature,includeIncomingCallFeature
```

Scattered throughout the source code of the base application are `ifdef` statements that tell the preprocessor to include relevant lines of code if the symbol for a particular feature is defined in the properties file. As mentioned in Chapter four, there are two points in MobilePhoto where optional features are integrated with the base code: extending functionality of the controller, and inserting new menu items to invoke the feature. This means we were able to limit our preprocessor usage to only two classes. We were able to further reduce the amount of preprocessor included code by implementing the Chain of Responsibility design pattern, discussed in the previous section. To illustrate further, the preprocessor statements and corresponding code used by the SMS feature are shown in figure 5-4.

```java
[imports]
#ifdef includeSmsFeature
import ubc.midp.mobilephoto.sms.SmsController;
#ifdef includeSmsFeature
nextInChain = new SmsController(this, model);
prevInChain.setNextController(nextInChain);
prevInChain = nextInChain;
#endif
#endif

[photoListScreen (menu item addition)]
#ifdef includeSmsFeature
static final Command smsSendCommand = new Command("Send SMS", Command.ITEM, 1);
static final Command smsReceiveCommand = new Command("Receive SMS", Command.ITEM, 1);
#endif

[initMenu()]
#ifdef includeSmsFeature
this.addCommand(smsSendCommand);
this.addCommand(smsReceiveCommand);
#endif
```

Figure 5-4: Partial Source Listings of MainUIMidlet Class (OO Version)

Similar sections of the preprocessor tagged lines of code above are needed for each optional feature of the application. The six optional features in MobilePhoto quickly increase the number and complexity of preprocessor statements needed throughout the
code. The UML sequence diagram in figure 5-5 shows the flow of execution when a user invokes the feature to send a photo to another user over SMS. Note that when the BaseController is unable to handle the action to send an SMS message, it passes control to the next controller in the chain. In this scenario, the SmsController is next in the chain. The type of controllers and the ordering of them will depend on which features were included by the preprocessor and the order they are specified in the source code.

![Sequence Diagram](image)

**Figure 5-5: Sequence Diagram to Send SMS (OOP)**

### 5.3 Managing Multiple Features Using Pure Variants™

The focus of this research has been on implementation techniques that make it easy to include combinations of optional features that are supported by a specific device platform. However, it does not address the issue of feature modeling or managing the relationships between these optional features.

To investigate the potential of enforcing relationship rules for features in our model, we used a third party development tool for the Eclipse IDE called Pure Variants[^43]. The Pure Variants Eclipse plugin let us add a layer of abstraction on top of our J2ME object-oriented implementation. The tool allows a developer to define a feature model for your product line and configure attributes, relations and restrictions between the features. From our MobilePhoto design we arranged the features into groups, specified a type for each

feature and defined relationships between features when necessary. The MobilePhoto features were grouped into a category of either ‘Base Features’, ‘Address Book Features’, ‘Incoming Call Features’ or Photo Transfer Features’. We also defined a new group called ‘Build Environment Features’ which does not technically include features in the application but let us capture the variability of the build requirements such as classpath settings, J2ME platform versions and preprocessor usage. The ability to configure these build environment features allowed us to automate the build process more effectively, as we will discuss later. Feature types include: Optional, Mandatory, Or (select at least one from a group) and Alternative (select exactly one from a group). We also defined relationships between features where necessary to indicate dependencies or conflicts.

Table 5-2 shows a summary of the groups, types and relations that make up the MobilePhoto feature model.

**Table 5-2: Pure Variants Feature Model for MobilePhoto**

<table>
<thead>
<tr>
<th>Group</th>
<th>Feature</th>
<th>Type</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>Create Photo Album</td>
<td>mandatory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Store Photo</td>
<td>mandatory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add Photo</td>
<td>mandatory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Delete Photo</td>
<td>mandatory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Label Photo</td>
<td>mandatory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>View Photo</td>
<td>mandatory</td>
<td></td>
</tr>
<tr>
<td>Address Book</td>
<td>Link Photo with Address</td>
<td>optional</td>
<td>&lt;Requires&gt; Include PIM lib</td>
</tr>
<tr>
<td></td>
<td>Book</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Photo Transfer</td>
<td>SMS Photo Transfer</td>
<td>optional</td>
<td>&lt;Requires&gt; Include WMA lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;Requires&gt; Target MIDP 2.0</td>
</tr>
<tr>
<td></td>
<td>Email Photo Transfer</td>
<td>optional</td>
<td>&lt;Requires&gt; Include RIM lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;Requires&gt; Convert to RIM .cod</td>
</tr>
<tr>
<td>Incoming Call</td>
<td>Display Photo of</td>
<td>alternative</td>
<td>&lt;Requires&gt; Link Photo with Address</td>
</tr>
<tr>
<td></td>
<td>Incoming Caller</td>
<td></td>
<td>Book</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;Requires&gt; Include Motorola lib</td>
</tr>
<tr>
<td></td>
<td>Play Ringtone of</td>
<td>alternative</td>
<td>&lt;Requires&gt; Link Photo with Address</td>
</tr>
<tr>
<td></td>
<td>Incoming Caller</td>
<td></td>
<td>Book</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;Requires&gt; Include Motorola lib</td>
</tr>
<tr>
<td>Build Environment</td>
<td>Use Preprocessor</td>
<td>optional</td>
<td>&lt;Conflicts&gt; Use Aspect Weaver</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;Conflicts&gt; Use Preprocessor</td>
</tr>
<tr>
<td></td>
<td>Use Aspect Weaver</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use Obfuscator</td>
<td>optional</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Convert to Palm .prc</td>
<td>optional</td>
<td></td>
</tr>
</tbody>
</table>
There are a number of ways you could define your feature model to give you the desired level of control during the build process. If desired, we could have specified that the ‘Display Photo of Incoming Caller’ and ‘Play Ringtone of Incoming Caller’ features were both optional instead of alternative if we wanted to allow both features to be installed together to display a photo and play the ringtone on an incoming call. We could also create sub-groups under the ‘Build Environment’ feature group to include ‘Compile Options’, ‘Optional Libraries’ and ‘Target Device’. This would allow you to define alternative types within compile options (e.g., Choose exactly one of Preprocessor or Weaver) or obtain finer control of the target devices by defining sub-groups of MIDP 1.0 and 2.0 to include specific device models (e.g., Nokia 6600 could be part of the sub-group ‘Target MIDP 2.0’).

Once the feature model is defined, Pure Variants provides a visual editor within Eclipse to let you select checkboxes for the features you want to include in your product instance. There is a checker included that will validate your selections to make sure the relationship and type rules are satisfied so there are no missing requirements or conflicting elements.

Figure 5-6 shows the MobilePhoto feature model in the Pure Variants Eclipse window.
In figure 5-5 you can see that when the 'Display Photo on Incoming Call' feature is selected, the tool automatically highlights the 'Associate Photo with Address Book Entry' feature because of the Requires relationship we defined.

Using the Pure Variants plugin also enabled further automation of the build process. The plugin includes functionality to generate an XML document from your feature model and allows you to transform it with an XSL style sheet, based on the features you select from the model. We defined an XSL transformation that automatically generates an ANT build file specific to the features you select for a product instance. The transformation is based on a common build.xml file template and uses the selected features as input to generate a custom build file specific for a target device. The selected features help generate the appropriate classpath settings, include the appropriate source files, and insert additional build tasks that are only necessary for certain platforms, such as converting to .pre files for Palm, or using the RAPC compiler for Blackberry.

For example, the main task in the Pure Variants version of our build script template is defined in figure 5-7.
Notice in figure 5-7 that there are three conditional ant calls in the full-build task that will only be generated in the targets build.xml file if certain attributes or features are selected in the Pure Variants configuration editor. The midp-preprocess task invokes the preprocessor, and is only included if the developer checked the preprocess attribute from the 'Build Environment' feature group. Similarly, the rapc-compile task and make-prc tasks are only generated if you are building for the RIM or Palm platforms respectively.

An updated diagram of the build process using Pure Variants is shown in figure 5-8.
Figure 5-8: MobilePhoto Build Process Using Pure Variants

The Pure Variants plugin was only tested using the object-oriented implementation of MobilePhoto, but could also be integrated with the AspectJ version. The only output is a generated build script, so the main difference would be adding AspectJ compile tasks to the build.xml template.
6 Evaluation

To assess the advantages and disadvantages of using aspect-oriented programming to address the device API fragmentation problem for mobile devices, we compare, qualitatively and quantitatively, the two different techniques used to implement the MobilePhoto prototype. Our evaluation builds on categories introduced by [19] to compare an AspectJ implementation with an object-oriented implementation. Our analysis focuses on the effectiveness of each technique to build a product line, and speculates on how each approach might work with more complex or wide-ranging product lines for mobile devices.

Almost all of the source code used in the AspectJ implementation of MobilePhoto is the same as the code in the object-oriented version. Both followed the same architecture and design and adhered to the requirement of maintaining component independence such that base components can be compiled with or without optional components. The key differences in the two implementations were how to achieve de-coupling of optional feature components, how to bind optional features into the core application, and how to manage feature interaction. The rest of this chapter compares these implementations in terms of modularity, scalability, complexity, tool support, platform support and quantitative analysis of size and resource usage.

6.1 Quantitative Analysis

One of the most important factors to consider when choosing an implementation technique for mobile development is the size and performance of your final application. Most J2ME enabled devices have little storage space for data, and have limitations on the size of the application that can be installed. These micro devices also have less heap memory available and slower processors which mean you should avoid using technology that hinders your application in these regards. Our evaluation of the AspectJ implementation and the object-oriented implementation of MobilePhoto shows that the application size will always be larger when using AspectJ because you must include the AspectJ runtime classes with your application jar file. Table 6-1 compares the size of the jar files for each implementation when only the base components are used, and when all the components are compiled into the application.

<table>
<thead>
<tr>
<th></th>
<th>MobilePhoto AOP (Base)</th>
<th>MobilePhoto OOP (Base)</th>
<th>MobilePhoto AOP (All)*</th>
<th>MobilePhoto OOP (All)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final app size [jar]</td>
<td>200 KB jar</td>
<td>165 KB jar</td>
<td>251 KB jar</td>
<td>213 KB jar</td>
</tr>
</tbody>
</table>

The aspect-oriented implementation jars listed in table 6-1 were created by including the classes found in the standard AspectJ runtime without modification. The size of this jar (aspectjrt.jar) is 40 Kb. We were able to reduce this size slightly by removing all of the
AspectJ features that are not supported in J2ME, removing all the comments from the source and running it through an obfuscator. The size of our modified (and obfuscated) AspectJ runtime jar was 29 Kb\(^{44}\).

Table 6-2 shows measurements of heap memory use\(^{45}\) at specified intervals during the MobilePhoto application lifecycle. It compares equivalent product instances for the aspect-oriented and object-oriented implementations after invoking various commands in the application. Each feature listed was executed in isolation, immediately after application launched to ensure that data for one feature invocation is not confounded by data from another.

The maximum heap size for the emulator used was 2 Mb (~2000 Kb), therefore whenever the memory usage reached the 2000 Kb upper limit, garbage collection occurred and cleared the memory back to 0. The data indicates that the memory usage of the aspect-oriented implementation is slightly better than the object-oriented implementation, but not significant enough to make a noticeable difference during use. A minor note is that the object-oriented version used more memory when the application launched, as you can see in every version except the base, it garbage collected before the aspect-oriented version did.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Launch</th>
<th>Select Album</th>
<th>View Photo</th>
<th>Display Caller Photo</th>
<th>List Contacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP Base</td>
<td>1935</td>
<td>1953</td>
<td>166</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OOP Base</td>
<td>1936</td>
<td>1954</td>
<td>167</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AOP Base + SMS</td>
<td>1936</td>
<td>156*</td>
<td>204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OOP Base + SMS</td>
<td>224*</td>
<td>242</td>
<td>291</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AOP Base + SMS + Caller Photo</td>
<td>1936</td>
<td>168*</td>
<td>217</td>
<td>244</td>
<td></td>
</tr>
<tr>
<td>OOP Base + SMS + Caller Photo</td>
<td>275*</td>
<td>294</td>
<td>343</td>
<td>376</td>
<td></td>
</tr>
<tr>
<td>AOP Base + SMS + Caller Photo + PIM</td>
<td>1936</td>
<td>186*</td>
<td>234</td>
<td>264</td>
<td>213</td>
</tr>
<tr>
<td>OOP Base + SMS + Caller Photo + Increase PIM</td>
<td>286*</td>
<td>306</td>
<td>354</td>
<td>389</td>
<td>200</td>
</tr>
</tbody>
</table>

*Indicates garbage collection occurred at 2 Mb, and dropped back to 0 before increasing again

We found that the source code metrics for both implementations were similar. The number of classes (or aspects) used and the total lines of code were either identical, or

\(^{44}\) We used the open source ‘jarg’ (Java Archive Grinder) obfuscator (http://jarg.sourceforge.net)

\(^{45}\) The Motorola iDen SDK Memory Monitor was used to record heap data using the i860 device emulator
only slightly higher in the object-oriented implementation. There was also little noticeable difference in the time to build each executable using ANT. Table 6-3 below provides details of MobilePhoto source code metrics.

Table 6-3: MobilePhoto Source Code Metrics Comparison

<table>
<thead>
<tr>
<th></th>
<th>MobilePhoto AOP (Base)</th>
<th>MobilePhoto OOP (Base)</th>
<th>MobilePhoto AOP (All)</th>
<th>MobilePhoto OOP (All)</th>
</tr>
</thead>
<tbody>
<tr>
<td># build script tasks</td>
<td>7+</td>
<td>6+</td>
<td>7+</td>
<td>6+</td>
</tr>
<tr>
<td># Classes (&amp; Aspects)</td>
<td>16 (no aspects)</td>
<td>16</td>
<td>41 (35 classes + 6 aspects)</td>
<td>41</td>
</tr>
<tr>
<td># Lines of Code</td>
<td>550</td>
<td>565</td>
<td>1481 (227 from Aspect code)</td>
<td>1524</td>
</tr>
<tr>
<td>Build Time</td>
<td>19 Seconds (to run Ant build with ajc)</td>
<td>18 Seconds (to run Ant build with javac)</td>
<td>31 Seconds (to run Ant build, no conversions)</td>
<td>29 Seconds (to run Ant build, no conversions)</td>
</tr>
</tbody>
</table>

The most significant finding of the quantitative analysis is that any J2ME and AspectJ implementation will increase your application by approximately 29 Kb. This is important when developing for devices with tight restrictions on the size of application. In most other quantitative categories we found the difference between the AspectJ and object-oriented implementations to be insignificant.

6.2 Modularity and Flexibility

A likely concern for mobile developers is the maintenance effort involved in managing the source code and build process as a system evolves, given the many different combinations of features that a product instance allows. Better modularity in the application tends to lead to less maintenance.

When updating an existing system with new features, it can be difficult to cleanly isolate all the functionality required by a feature into a separate module. Source code statements associated with a feature implementation may be scattered throughout the rest of the components in the base application.

In the aspect-oriented implementation, AspectJ helped to extend the base application while keeping the logic and code associated with a particular feature in a separate module. In the AspectJ implementation of MobilePhoto, we were able to modularize each of the optional features. The implementation of each optional feature is contained in a few files (classes and aspects) that are grouped into a single java package. If, for example, a change or update is required for the feature that handles the sending and

46 Indicates the application with ‘all’ features was compiled. Note that such a configuration would not run on any current devices since some features conflict. (Ie. No such device that supports the Blackberry email feature and Siemens features together)
receiving of digital photos over SMS, a programmer can focus on classes in the 
ubc.mipd.mobilephoto.sms package. Including or excluding this feature when 
building a product instance simply requires the developer to select a checkbox in the 
Eclipse build configuration plugin for the named package that corresponds to the SMS 
feature. Changing the logic associated with a feature requires updating one or more 
classes or aspects in the package for the optional feature. Changing the way the feature 
integrates with the base application components requires updating the source of the 
aspect. Configuring the build process to include an optional feature in the final product 
requires updating a build compile list to add the features' package. In general, 
maintaining the AspectJ implementation for MobilePhoto is a straightforward process.

The 'pure' object-oriented implementation of MobilePhoto modularizes the programming 
logic associated with optional features, but does not prevent the scattering of source code 
changes needed to integrate the features with the base components. For example, in the 
object-oriented version the majority of the source code associated with the SMS feature is 
still contained in the ubc.mipd.mobilephoto.sms package. However, to integrate this 
feature with the base application it was also necessary to modify classes in the base 
components such as MainUIMidlet and PhotoListScreen with preprocessor 
statements. Using the preprocessor allowed us to maintain the simplicity of the build 
process, so that editing a property file would trigger the inclusion or exclusion of the 
integration code based on preprocessor 'ifdef' tags. Incorporating the Chain of 
Responsibility design pattern improved things further so that instead of having to embed 
the controller logic for a feature directly in the BaseController class, each feature 
maintained its own extension of the controller that was only part of the chain if the 
feature was included in the build process. Therefore in the object-oriented 
implementation of MobilePhoto, changing the logic associated with a feature only 
required updating one or more classes in the feature's package. Changing the way the 
feature integrates with the base application components was slightly more time 
consuming because the developer has to update the necessary base components with the 
source code to integrate optional features that is embedded using preprocessor tags. 
Finally, configuring the build process for the object-oriented version remained 
straightforward by updating the build compile list in addition to the properties file to 
specify which features should be preprocessed.

Another item to consider is the granularity each technique can achieve for the developer 
to change and control the feature interaction and integration at fine grained levels. In the 
object-oriented version, there are few constraints on where new source code related to a 
feature can be added to the base application classes, assuming the developer is willing to 
make substantial modifications at times. A wide range of changes is possible because 
preprocessor statements can be written anywhere to manipulate the actions associated 
with a feature. However, with AspectJ the developer is limited by the pointcut language 
that defines where features can be bound to base components, and what type of advice 
can apply. This lack of fine granularity is particularly evident using AspectJ with J2ME, 
where certain features of AspectJ are not available (see section 6.4 for details on AspectJ 
support with J2ME). One option is to use a preprocessor in combination with AspectJ,
however the increased flexibility and finer granularity would be offset by the loss of modularity due to scattered preprocessor statements.

A final matter that should be noted is the impact of binding time - the point in the development cycle at which the features of your product instance become fixed, and your variability is decided[20] – on flexibility. The later in the development cycle this binding occurs, the greater the flexibility. Allowing an end user to select their optional features for a product during the download or installation stage provides more flexibility than requiring a programmer to pre-code the variability at design or during compilation. The time of binding has an impact on scalability as well, because as more (unique) product instances are offered, more effort is required to update or customize each product instance during the build cycle. In both MobilePhoto implementations, the binding time occurs during compilation and linking. Technically, in the object-oriented implementation it occurs just prior to compilation when the preprocessor is run, and in the AspectJ version it occurs just after compilation when weaving takes place at the bytecode level. However, from the developer standpoint this has no effect on effort or flexibility because in both scenarios the developer merely checks a box to specify which optional features are desired, and then runs the build script. Future work might consider whether or not dynamic weaving techniques in aspect-oriented programming [21] could be used to delay binding further and increase flexibility. With the current size constraints of micro devices, dynamic weaving may not be applicable because dynamically including features implies that all of the potential features one may want to include would have to be deployed to the device initially, making the application larger than necessary. However, dynamic weaving combined with negative variation product line techniques[19] might make sense. Dynamic weaving could make the deployment process more flexible if a developer were able to build an application containing all of the optional components available, and then apply empty advice at runtime to replace or remove features that are not supported by the device it is deployed to.

6.3 Scalability

The flexibility and ease of maintaining an application may not be the highest priority when producing a system that changes infrequently and only needs to support a single platform. But when you are targeting numerous mobile devices with hardware and software that is changing at a rapid pace, scalability becomes more of a concern. It is important to consider how well your product line development process can scale in terms of the number of optional features available to products and the number of devices you must support.

The number of features you can squeeze into a single application instance will be limited by the amount of memory or disk space the hardware vendor allows for third party applications. But as hardware gets cheaper, the number of features you can include will increase. Especially if you are targeting higher end devices that offer larger amounts of resources to play with. For the AspectJ implementation, creating most additional features requires adding additional packages (one per feature) that contain the Java
implementation logic and the aspect that integrates the functionality into the base application. The object-oriented version requires adding new packages for new features, and updating various classes in the base application to integrate these features using preprocessor statements. When dealing with relatively few features, this is not too difficult to manage. But the wide range of hardware peripherals that micro devices offer can lead to a large array of optional features to plug into your product line. This results in many lines of preprocessor tagged source code scattered throughout the components of the base application. The MobilePhoto application has six optional features, which means multiple sections of code in the base components must contain six embedded preprocessor statements in every area you need to integrate these features. For example, in the MainUIMidlet class, there are preprocessor tagged insertions of code for import statements (one for each feature), and code to create new instances of sub-controller objects and add them to the chain of handlers. This is multiplied by six, since there are six optional features. As the number of optional features grows, the source code in the object-oriented version of the base application becomes littered with ifdef statements, making it harder to read and understand how a particular feature interacts with the application, and how it interacts with other features.

Being prepared to handle an increasing number of optional features is good practice, but in the mobile space it is probably not as important as being able to handle an increasing number of hardware devices in your product line. The most popular genre of J2ME applications currently on the market is mobile gaming. In general, there is not a great deal of variability in the gaming features, but there is a large demand to support an increasing number of devices. In our case of building mobile product lines, scalability refers to whether your development and build process can handle dozens or even hundreds of different device models. Given the flexibility provided by the aspect-oriented approach, scalability in this context relates to the level of automation that can be achieved in the build process. The main parameters to the build process vary by device in terms of the libraries needed during compilation (e.g., MIDP 1.0 vs. MIDP 2.0, or including proprietary Blackberry or Nokia libraries), the inclusion or exclusion of source components for optional features, and the addition of build tasks that are specific for a particular device (e.g., converting JAR files to native Palm OS or RIM OS format). The object-oriented implementation and the AspectJ implementation are essentially equal in this regard, since the standard build process is the same for both implementations. The use of ANT XML files to manage the build scripts helps keep things simple and makes it easy to automate tasks.

In the cases we tested, supporting a new device in the build system was trivial and involved adding a new ANT task only if it was necessary to convert the J2ME jar file into a platform specific format (as was the case for both Palm and Blackberry devices) and updating properties such as paths to the IDE or toolkit install directories to the location of the conversion utilities. When new features were added to the application, we needed to update the list of source files in the compile list and update the classpath settings if new optional or proprietary libraries were required during compilation. The task of updating the compilation list was made easier in the Eclipse IDE by using the feature of the AJDT
that scans your source code directory and provides a check-box to indicate which packages should be compiled into the application. The AJDT plugin is intended for AspectJ development, but the build configuration selector generates a file that can be used with standard ANT build tasks and was also used for building the object-oriented implementation of MobilePhoto. By using ANT and the AJDT Eclipse plugin mentioned, executing a build for different devices was as easy as checking off different combinations of features, and double clicking the ant script.

Although we were able to use a single build file for all the product instances, updating the device specific properties each time might become tedious as the number of devices to support grows very large. One option to improve the scalability in this regard is to automatically generate a new build script based on the target platform and features selected. We tested this approach with the Pure Variants plugin (see section 5.3) and it worked well. This frees the developer from having to edit build files or manually invoke additional build tasks that are only required by certain devices. As the number of devices to support grows, the developer can quickly build product instances by checking a few boxes, generating the target build script and executing a single ANT task. This leads to improved scalability in the development cycle because there is less work managing the build process. Automating the build tasks would require the same effort for the object-oriented and AspectJ implementations, so scalability in this regard was equal.

6.4 Multi-Platform Support

One issue we encountered when deploying an AspectJ-based application on mobile platforms is that AspectJ was not designed to work with the core J2ME class libraries. To execute Java applications that have aspects woven in them, the AspectJ runtime must also be deployed to the device and run within the context of whatever virtual machine is installed on your phone or PDA. The AspectJ runtime used in this research (AspectJ v 1.2.0) is intended to be used with a J2SE virtual machine. As a result, there are a number of classes in the runtime that depend upon Java libraries that only exist in the standard Java environment, and that are not available in the micro edition version. You can still deploy and utilize the AspectJ runtime on J2ME platforms but the limitation impacts J2ME and AspectJ development in two areas.

The first area of impact is the extent to which the developer can access information about a join point. Specifically, J2ME does not include support for reflection, and has very limited threading capabilities. Although we have not rigorously tested what parts of the AspectJ runtime will work or fail with J2ME, there are two notable restrictions that we found. The lack of reflection in J2ME means that thisJoinPoint cannot be used in advice. The thisJoinPoint construct of AspectJ uses reflection to allow a programmer to capture static and dynamic context information associated with join points. For instance, you can obtain access to the target object, the execution object and the method

http://eclipse.org/aidt/

http://eclipse.org/aspectj/
arguments using thisJoinPoint. For our prototype application, this would have made it easier to access the List objects during menu initialization, or accessing controller objects after the command handler executes. This limitation was not a major setback, and we were able to work around it by implementing wrapper methods for the join points, and adding additional parameters that could be accessed using the args keyword of AspectJ. For example, in standard AspectJ, we can write the following initMenuItems pointcut:

```java
pointcut initMenuItems() : execution(* PhotoListScreen.initMenu());
```

In our advice, we can update List object that stores the menu items by using:

```java
(PhotoListScreen)thisJoinPoint.getTarget().getList().addCommand(newCommand)
```

However, in J2ME an alternative syntax is needed that exposes explicitly the List to be updated:

```java
pointcut initMenuItems(List l) : args(l) && execution(* PhotoListScreen.initMenu(List));
```

In this case, we have to change the PhotoListScreen.initMenu() method to include the List parameter so that the advice could append new menu items to that object using

```java
l.addCommand(newCommand);
```

The absence of thisJoinPoint was the only limitation we experienced with the pointcut language while implementing the AspectJ version of MobilePhoto. However, there is a similar issue related to cf low pointcuts that we were aware of although it did not affect our implementation. J2ME has very limited support for threading and does not include many of the utility classes found in the standard java.util package. Unfortunately, parts of the AspectJ runtime related to cf low pointcuts rely on many of these thread, stack and utility classes. We read about this limitation from an FAQ entry on the AspectJ website.

The second major limitation we experienced using AspectJ to implement MobilePhoto was a difference in behaviour for java enabled RIM devices. For most emulators and hardware that we tested, the AspectJ version of our application executed without any problems as long as we avoided using thisJoinPoint constructs. However, Research In Motion’s Blackberry platform was an exception. The Blackberry virtual machine has strict dependency checking when a Java application launches on the device. When we first tried running MobilePhoto on the RIM simulators, the application crashed with a series of exceptions indicating it could not find many of the classes used in the AspectJ runtime. We did not encounter this problem on other devices, as long as the runtime was preverified against some J2SE libraries, and as long as we did not use any features of

---

47

49 http://dev.eclipse.org/viewcvs/indextech.cgi/aspectj-home/doc/faq.html?rev=1.9&q=aspectjandj2me

50 We tested using Blackberry JDE v 3.7 and v 4.0, and the standalone Simulator v 3.7.0.52
AspectJ that relied on unsupported APIs.

But for the Blackberry we had to make several modifications to the source code of the AspectJ runtime before it would work. Specifically, we had to remove any references or usage of classes that were not supported by the standard J2ME library. These changes did not affect the AspectJ usage in MobilePhoto, and slightly reduced the size of the runtime. Table 6-4 provides a summary of the AspectJ runtime source files that were modified, and notes the non-J2ME references that had to be removed.

Table 6-4: Modified Classes for Custom Blackberry AspectJ Runtime

<table>
<thead>
<tr>
<th>Modified AspectJ Runtime Class</th>
<th>Non-J2ME references removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.aspectj.lang.SoftException</td>
<td>java.io.PrintWriter</td>
</tr>
<tr>
<td>org.aspectj.runtime.internal.CFlowCounter</td>
<td>System.getProperty(x,x) not supported in J2ME</td>
</tr>
<tr>
<td>org.aspectj.runtime.internal.CFlowStack</td>
<td>System.getProperty(x,x)*</td>
</tr>
<tr>
<td>org.aspectj.runtime.internal.cflowstack.ThreadCounterImpl1</td>
<td>java.util.Iterator, java.util.List, java.util.ArrayList</td>
</tr>
<tr>
<td>org.aspectj.runtime.internal.cflowstack.ThreadStackFactoryImpl1</td>
<td>java.lang.ThreadLocal (inherited from this class)</td>
</tr>
<tr>
<td>org.aspectj.runtime.reflect.Factory</td>
<td>java.lang.ClassLoader</td>
</tr>
<tr>
<td>org.aspectj.runtime.reflect.InitializerSignatureImpl</td>
<td>java.lang.reflect.Modifier</td>
</tr>
<tr>
<td>org.aspectj.runtime.reflect.StringMaker</td>
<td>java.lang.reflect.Modifier, Class.getComponentType*</td>
</tr>
</tbody>
</table>

* listed method calls are not available for the named class in J2ME
** double and float primitive types are not supported by RIM J2ME implementation

After we replaced the standard AspectJ runtime jar file with our modified jar file the application successfully launched on all Blackberry simulators and the MobilePhoto application worked correctly. Some of these limitations could be fixed by updating the AspectJ runtime source code to use J2ME only classes to implement the same functionality. However, re-writing reflective classes and utility classes like Iterator, ThreadLocal or ArrayList would likely increase the size of the jar file and it would be difficult to guarantee the same behaviour as the standard J2SE implementations.
There is one other issue to note when compiling AspectJ aspects against the J2ME platform. Sun introduced an additional preverification phase in the development and deployment process that is required to run Java applications on mobile devices. Preverification performs certain checks on the Java bytecodes ahead of runtime. If this first verification pass is ok, the preverifier annotates the class files (using standard Java bytecode attributes), and then saves the annotated class files or passes them along to the next tool in the compile chain. When the J2ME virtual machine (VM) on a device attempts to execute a J2ME CLDC-based application, it checks the Java class files for these preverification annotations. Proper annotations in the class files guarantee that certain compile-time checks were made, and so the VM can pass through its own verification and security checks much faster and start executing the application more quickly. Sun provides a (command line) utility that will preverify your J2ME application code prior to deployment. However, when writing a J2ME application with AspectJ you must include the runtime classes within your jar file, but the AspectJ runtime classes will fail the preverification step because they utilize classes from the J2SE that are not available within the J2ME core classes. You can work around this issue by including the required J2SE classes on the classpath during the preverification step. Even though these classes will not be available at runtime, as long as you are not using them (e.g., thisJoinPoint and cfFlow) your application should run without any problems. For this reason, we extracted the specific Java classes from the J2SE library that were required by the AspectJ runtime to include in the preverification classpath. This technique worked with all of the available emulators and hardware tested, with the exception of the Blackberry platform, which was discussed above.

### 6.5 Complexity of Implementation

The complexity of applying the product line implementation technology affects how easy it would be for developers to build an application with the technology. For example, how easy is it for a J2ME programmer to implement their own product line using a similar strategy? This analysis of implementation complexity looks at the design and implementation of the application framework and components in the two approaches taken, and the ease with which (optional) variant components can be reused across product instances.

Both versions of our prototype application were programmed using Java 2 Micro Edition, but using the AspectJ technique likely requires a little more effort if a developer is not familiar with aspect-oriented programming concepts. However, the use of AspectJ is somewhat simplified when applied to mobile devices because some of the more complicated functions in AspectJ are not available when deploying to mobile devices (see Section 6.4). Therefore a developer who is not familiar AspectJ only has to learn some basic pointcut syntax, and become familiar with the new AspectJ build tasks available for ANT. By comparison there is probably less of a learning curve for programmers to use

---

51 The preverify tool is included with the Sun Wireless Toolkit (http://java.sun.com/products/sjwttoolkit/)
the preprocessor approach taken in the object-oriented implementation, since there is no new language to use except some basic ‘ifdef’ tags.

Both MobilePhoto implementations would be classified as positive variability techniques[19], using a decomposition (or module replacement) model[16]. Positive variability adds functionality to create new product instances, while negative variability removes it. Our decision to use positive variability with module replacement simplifies the development process, but is only applicable when you are designing your application from scratch with variability in mind. When you must re-factor an existing application into a family of mobile products, a negative variation approach might be necessary. The use of AspectJ in a negative variation approach is discussed elsewhere [22], [13].

Another issue of complexity in the AspectJ approach is the ease of testing the correctness of the implementation. Standard object oriented unit and integration testing approaches may not apply to due to the fact that aspects behave differently than common classes, and the side effects may not always be apparent to the developer by reading the source code[23]. Understanding the testing constraints for an aspect-oriented implementation may extend the learning curve for new programmers during the development cycle. However, the area of testing aspects is being explored further, and helpful tools are starting to emerge to facilitate this and make it easier.52

6.6 Tool Support

Tool support for developing J2ME applications is still in its infancy. Although there is an abundance of SDK’s, programming tools and development environments available from vendors, most of these tools are aimed at developing Java applications for a specific hardware platform. Almost every hardware manufacturer that supports Java on their device has also released a proprietary development environment to make it easier to build applications specifically for their platform. A few of these vendors plan to release versions of their toolkits that integrate with existing IDE’s like Eclipse or Netbeans. But at the moment, there is still a significant amount of installation, setup and configuration required just to get your development workspace to the point where you can program, test and debug applications for multiple devices. You can reduce this list of tools if you want to sacrifice the functionality offered in proprietary API’s and rely on only the generic emulators for testing. But you still need to install some of the utilities from Palm and RIM that convert your Java application into native file formats.

All of the vendor specific development environments available come bundled with emulators for their device models, optional and proprietary libraries that are available on the devices, and additional tools necessary to test and deploy mobile applications in markets where they are available. Before we could use all of the required libraries, emulators and utilities necessary to build the relatively small product line for the MobilePhoto prototype we needed to install all of the applications listed below:

52 http://www.aunit.org/ is a unit testing framework for AspectJ that follows the xUnit/JUnit framework.
- Eclipse IDE (Could be any Java editor of choice)
- AspectJ Development Toolkit (AJDT plugin for Eclipse)
- Sun J2ME Wireless Toolkit
- Apache Ant + optional tasks for J2ME and AspectJ
- J2ME Preprocessor (Antenna)
- Blackberry Java Development Environment
- Nokia Developer Suite for J2ME
- Motorola iDEN SDK for J2ME
- Siemens Mobility Developer Suite
- Sun PRC Conversion Tool for Palm OS
- IBM J9 Virtual Machine for Palm OS (+ utilities)

Some open source projects might alleviate these problems with compatibility. The EclipseME\textsuperscript{53} open source plugin helps connect a variety of wireless toolkits and emulators to the Eclipse development environment so mobile programmers can focus on the application instead of the development workspace. The latest release of the Netbeans Java IDE includes a Mobility Pack\textsuperscript{54} that specifically tries to solve the problems with device fragmentation. It includes a preprocessor for J2ME, and extends the graphical interface to allow the developer to define configurations that will only display the source code associated with a target configuration. For example, we could define a blackberry configuration for MobilePhoto and configure Netbeans so that only the blackberry preprocessor statements are visible, while the lines of code associated with other devices (or features) are hidden from view. This would certainly help the readability of source code for projects that contain lots of preprocessor statements and many targeted device platforms. Although this benefit is only available as long as you are using the NetBeans IDE to program because the underlying source code still contains all of the preprocessor tagged code in the files they are used in. IBM has also released a branded version of Eclipse (part of their Eclipse ‘product line’) called Websphere Device Developer (WSDD) that provides automatic integration for a variety of J2ME emulators and utilities available. WSDD requires a purchased software license to use, and at the time of this writing it did not offer any solution to the API fragmentation problem.

Despite the wide array of development environments available (and necessary), we found that none of them met all of our needs to facilitate easier mobile product line development. The vendor provided tools were generally lacking in basic Java IDE features that many developers have come to expect, such as advanced search and navigation, re-factoring and customizable views. These vendors IDE’s also included no support for aspect-oriented programming. Most of the IDE’s (with the exception of Netbeans) did not directly support a preprocessor, which meant that the object-oriented version of MobilePhoto could not have been built using the built-in IDE compiler. When we attempted to use EclipseME with the AJDT plugin for Eclipse to compile the aspect-oriented version of MobilePhoto it resulted in conflicts in the IDE. Specifically, the AspectJ runtime needs to be packaged into your final J2ME application, but it fails the

\textsuperscript{53} http://eclipseme.org/
\textsuperscript{54} http://www.netbeans.org/kb/articles/mobility.html
J2ME preverification step because it relies on classes that do not exist in the MIDP or CLDC libraries. We were not able to find a work-around for this with the EclipseME plugin. To get around this using ANT, we had to manually extract the required classes from the J2SE library and override the preverification classpath for the AspectJ runtime files. Overriding this classpath was not always possible to do with the built-in IDE plugin editors. There were also additional classpath conflicts that had to be resolved since the Aspect source files are built using the AspectJ compiler and may need access to standard (J2SE) libraries, while regular J2ME source files should only be compiled against the J2ME core libraries. None of the third party J2ME IDE's currently facilitate (or encourage) component or feature based development, and have no way of managing feature interaction the way that Pure Variants did. There are some generic toolsets available that support feature oriented programming for product line development. The AHEAD Tool Suite (ATS)\footnote{http://www.cs.utexas.edu/users/schwartz/ATS.html} is a collection of Java based tools that treats features as the core building blocks of a system. Programs in ATS are unique compositions of algebraic operations that correspond to a feature module. We did not attempt to build J2ME and AspectJ programs using AHEAD since it focuses heavily on the algebraic domain and feature management but is lacking in general Java programming and IDE functionality.
7 Conclusions

We evaluated the use of aspect-oriented programming, and in particular AspectJ, to alleviate problems with fragmented API support across micro devices when implementing a family of mobile Java applications.

Many existing approaches to product line development are not applicable for mobile devices because of tight restrictions on application resources and because the volume of devices to support is much greater than typical desktop or server platforms.

To evaluate our strategy, we compared two different implementation techniques for a mobile prototype application written in J2ME. One implementation uses standard object-oriented practices, design patterns and preprocessor utilities that are used by many J2ME application developers in industry. A second implementation uses aspect-oriented practices, including pointcuts and advice, and the AspectJ weaver. We compared these implementations with respect to the complexity of the development, modularity, scalability, device support, tool support and quantitative measurements of size and resource usage.

We found that aspect-oriented programming, as supported by AspectJ, provides an effective way to bind optional components to your mobile application during the build process. It allows a developer to isolate J2ME API's that are only supported by specific device types and keep them in a single module instead of being scattered throughout the application classes. However, there is minimal support in AspectJ to manage dependency relationships between different optional features. Build configuration is also required to minimize the application size and exclude unnecessary resources. The biggest drawback is that including the AspectJ runtime with your application increases its size between 30-40 KB, which is significant for many micro device platforms. Most of the other metrics compared such as memory utilization, number of classes, lines of code, duration of builds etc. revealed no significant difference between the object-oriented programming and aspect-oriented programming implementations. Scaling the product line build process to include many optional features and many supported devices is difficult using only object-oriented techniques and a preprocessor. As the optional features increase, the number of embedded preprocessor statements scattered throughout your code base also increases, making it difficult to manage without some type of tool support. At the moment, tool support for building J2ME product lines in general is still lacking, and tool support for using aspect-oriented programming with J2ME is almost non-existent.
7.1 Contributions

This research makes several contributions:

- It demonstrates the feasibility of using AspectJ on a wide variety of simulators and real hardware devices that support J2ME (including Symbian based phones from Nokia, Motorola and Siemens, Palm OS devices, and RIM Blackberry devices.)
- It documents specific classes and methods in the AspectJ runtime that are not supported on current J2ME platforms due to the absence of reflection and minimal threading support in the MIDP and CLDC libraries.
- It documents limitations of using AspectJ on certain devices, notably current Blackberry simulators and devices by Research In Motion can only execute a custom modified version of the AspectJ runtime.
- It provides a sample application implemented in an aspect-oriented and object-oriented version that may be used for education or research purposes.

7.2 Future Work

Aspect-oriented programming has potential to optimize development practices for software product lines, and particularly for mobile devices. The work described in this thesis focused on the use of aspects to vary the functionality available in a particular configuration of a prototype application. It would also be interesting to see if aspects could alleviate other problems with porting J2ME applications, such as variations in device hardware, operating constraints on networking usage or multi-language support, and VM specific bugs. This work has also focused on aspect-oriented programming as provided by AspectJ. It would also be useful to broaden the range of implementation languages (e.g., C/C++, AspectWerkz\textsuperscript{56} or HyperSpaces\textsuperscript{57}) and device platforms (e.g., Microsoft PocketPC\textsuperscript{58} or Qualcomm BREW\textsuperscript{59}) to see if similar advantages are obtained.

Another avenue of future work is creating a fully functioning version of the AspectJ runtime that is optimized for J2ME. This work might involve eliminating AspectJ's dependency on the J2SE API's, and removing unsupported (or unnecessary) classes from the runtime. Having a J2ME optimized version of the AspectJ runtime could limit the increase in size that all mobile applications incur when using AspectJ.

\textsuperscript{56} http://aspectwerkz.codehaus.org/index.html
\textsuperscript{57} http://www.research.ibm.com/hvperspace/
\textsuperscript{58} http://www.microsoft.com/windowsmobile/
\textsuperscript{59} http://brew.qualcomm.com/brew/en/developer/resources/dev_resources.html
Bibliography


Appendix A: Acronyms

The following list details the acronyms used in this thesis.

J2ME  Java 2 Platform, Micro Edition
API   Application Programming Interface
AOP   Aspect Oriented Programming
AOSD  Aspect Oriented Software Development
OOP   Object Oriented Programming
OO    Object Oriented
PDA   Personal Digital Assistant
RIM   Research In Motion
(J)VM  (Java) Virtual Machine
KVM   Kilobyte Virtual Machine
WMA   Wireless Messaging API
CLDC  Connected Limited Device Configuration
CDC   Connected Device Configuration
J2EE  Java 2 Enterprise Edition
J2SE  Java 2 Standard Edition
JAD   Java Application Descriptor
JAR   Java Archive
MIDP  Mobile Information Device Profile
PIM   Personal Information Manager
RMS   Record Management System
SMS   Short Message Service
WORA  Write Once Run Anywhere
SDK   Software Development Kit
IDE   Integrated Development Environment
CTO   Cheif Technology Officer
(G)UI  (Graphical) User Interface
CVS   Concurrent Version System
MVC   Model View Controller
CDMA  Code Division Multiple Access (Wireless data transmission protocol for mobile telephone networks)
GPRS  General Packet Radio Service (Wireless data transmission protocol for mobile telephone networks)
802.11 Family of specifications for wireless LANs developed by the IEEE
LCD   Liquid Crystal Display
ORB   Object Request Broker
ANT   Another Neat Tool (Build Tool)
XML   eXtensible Markup Language
XSL(T) eXtensible Stylesheet Language (Transformation)
UML   Unified Modeling Language
FAQ   Frequently Asked Questions
Appendix B: ANT Build Script Source Listing

<?xml version="1.0" encoding="iso-8859-1" ?>
<project name="MobilePhoto AOP Version" default="full-build" basedir=".">

<!-- Define all relevant properties here. -->
<property name="workspace.dir" location="${basedir}"/>
<property file="laptopBuild.properties"/>
<property name="antenna.jar" location="${antenna.home}\antenna-bin-0.9.12.jar"/>
<property name="antenna.adapter" value="org.aspectj.tools.ant.taskdefs.AjcllCompilerAdapter"/>
<!-- For Blackberry, use the modified AspectJ Runtime Jar file -->
<property name="aspectjtools.jar" location="${aspectj.lib.dir}/aspectjtools.jar"/>
<!-- Define core MIDP libraries -->
<!-- MIDP 1.0 CLDC 1.0 -->
<property name="midpapi10.jar" location="${wtk.home}\lib\midpapi10.jar"/>
<property name="cldcapi10.jar" location="${wtk.home}\lib\cldcapi10.jar"/>
<!-- MIDP 2.0 CLDC 1.1 -->
<property name="midpapi20.jar" location="${wtk.home}\lib\midpapi20.jar"/>
<property name="cldcapi11.jar" location="${wtk.home}\lib\cldcapi11.jar"/>
<!-- Define Third Party and optional MIDP Libraries -->
<!-- The motorola and siemens's zip files are just archives of the class directories for the respective devices. The contained files can be found in the install directory of the respective SDK or Toolkit. -->
<property name="moto730.zip" location="${motorola.home}/lib/i730/Motorola730.zip"/>
<property name="wma.jar" location="${wtk.home}/lib/wma.jar"/>
<property name="mmapi.jar" location="${wtk.home}/lib/minapi.jar"/>
<property name="jsr75.jar" location="${wtk-beta.home}/lib/jsr75.jar"/>
<property name="blackberryLib.jar" location="${blackberry.jde.home}/lib/net_rim_api.jar"/>
<property name="siemensLib.zip" location="${siemens.home}/SiemensJ2MElib.zip"/>
<!-- Set the classpath to be used -->
<property name="build.classpath" value="${midpapi20.jar};${cldcapi11.jar};${wma.jar};${mmapi.jar};${jsr75.jar};${moto730.zip};${aspectjrt.jar};${siemensLib.zip};${blackberryLib.jar};${antenna.jar}"/>

<!-- Builds for Blackberry should only use MIDP 1.0 and CLDC 1.0 -->
<!-- <property name="build.classpath" value="${midpapi10.jar};${cldcapi10.jar};${aspectjrt.jar};${blackberryLib.jar}"/>

<!-- Directories created or used during the build process -->
<property name="rapc-compiled.dir" location="${workspace.dir}/rapc-compiled"/>
<property name="prc-compiled.dir" location="${workspace.dir}/prc-compiled"/>
<property name="midp-compiled.dir" location="${workspace.dir}/midp-compiled"/>
<property name="src.dir" location="${workspace.dir}/src"/>
<property name="aspect-source.dir" location="${workspace.dir}/ubc/aspectj/photoalbum"/>
<property name="ajc-compiled.dir" location="${workspace.dir}/ajc-compiled"/>
<property name="preverified.dir" location="${workspace.dir}/preverified"/>
<property name="jar.dir" location="${workspace.dir}/jars"/>
<property name="palm.dir" location="${workspace.dir}/palm"/>
<property name="rim.dir" location="${workspace.dir}/rim"/>
<property name="images.dir" location="${workspace.dir}/images"/>

<!-- This resource is required for Antenna based tasks -->
<taskdef resource="antenna.properties" classpath="${antenna.jar}" />

<!-- Full Build (Default Target) -->
<target name="full-build">
  <antcall target="clean" />
  <antcall target="midp-compile" />
  <antcall target="aspectj-compile" />
  <antcall target="copy-ajruntime-classes" />
  <antcall target="copy-image-files" />
  <antcall target="midp-preverify" />
  <antcall target="make-jar" />
</target>

<!-- Blackberry Build Part 1 (Needs to split in two because the aspectJ compile exits ANT, even though the build is valid) -->
<target name="bberry-1">
  <antcall target="clean" />
  <antcall target="aspectj-compile" />
</target>

<!-- Blackberry Build Part 2 (split in two because the aspectJ compile exits ANT, even though the build is valid) -->
<target name="bberry-2">
  <antcall target="copy-ajruntime-classes" />
  <antcall target="copy-image-files" />
  <antcall target="midp-preverify" />
  <antcall target="make-jar" />
  <antcall target="rapc-compile" />
</target>
<!-- Clean all targets -->

<target name="clean" description="clean and create classes/jar dir, .ajesym files">

<!-- Delete all the directories and their contents -->
<delete quiet="on" dir="${ajc-compiled.dir}"/>
<delete quiet="on" dir="${preverified.dir}"/>
<delete quiet="on" dir="${midp-compiled.dir}"/>
<delete quiet="on" dir="${rapc-compiled.dir}"/>
<delete quiet="on" dir="${prc-compiled.dir}"/>
<delete quiet="on">
  <fileset dir="${jar.dir}" excludes="**/*.jad"/>
</delete>
<delete quiet="on">
  <fileset dir="${workspace.dir}" includes="**/*.ajesym"/>
</delete>

<!-- Recreate the directories we just deleted -->
<mkdir dir="${ajc-compiled.dir}"/>
<mkdir dir="${preverified.dir}"/>
<mkdir dir="${midp-compiled.dir}"/>
<mkdir dir="${rapc-compiled.dir}"/>
<mkdir dir="${prc-compiled.dir}"/>
<mkdir dir="${jar.dir}"/>
</target>

<!-- Compile MIDP classes using standard midp (javac) compiler -->

<!-- For Motorola Build, only update jad so it includes property: iDEN-Midlet-Phone: MainUIMidlet10 -->
<wtkjad jadfile="${jar.dir}/${jad.name}" jarfile="${midp-compiled.dir}/${jar.name}" name="${midlet.name}" update="true" vendor="Trevor Young" version="1.0.0">
  <midlet name="${midlet.name}" class="${main.class}"/>
</wtkjad>

<!-- Compile everything, but don't preverify (yet). Use bootclasspath to override MIDP 1.0/CLDC 1.0 apis-->
<wtkbuild srcdir="${src.dir}" destdir="${midp-compiled.dir}" preverify="false" bootclasspath="${build.classpath}" />
  <includesfile name="MobilePhotoCompileList.lst"/>
</wtkbuild>

<!-- Package the compiled classes into a JAR and JAD file -->
<wtkpackage jarfile="${midp-compiled.dir}/${jar.name}" jadfile="${jar.dir}/${jad.name}" obfuscate="false" preverify="false">
  <fileset dir="${workspace.dir}" includes="${main.class}.class"/>
</wtkpackage>
</target>
<!-- Some Initialization tasks required for AspectJ Compiler Target-->
<!-- ============================================================== -->
<target name="aspectj-init">
    <!-- Check the required libraries -->
    <available file="${aspectjtools.jar}" property="aspectj tools.jar.available"/>
    <available file="${aspectjrt.jar}" property="aspectj rt.jar.available"/>

    <!-- Sets name of new task to iajc, old task to ajc -->
    <taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
        <classpath>
            <pathelement path="${aspectjtools.jar}"/>
        </classpath>
    </taskdef>
    <property name="taskdefs.init" value="true"/>
</target>

<!-- ============================================================== -->
<!-- Compile AspectJ Sources and weave with MIDP compiled Jar -->
<!-- ============================================================== -->
<target name="aspectj-compile" depends="aspectj-init" description="Uses AspectJ compiler to compile and weave">
    <echo message="Compiling with ajc...(Disabled Import Errors, Will Proceed On Error)"/>
    <iajc destdir="${ajc-compiled.dir}" fork="true" proceedOnError="true" noimporterror="true"
        forckclasspath="${aspectjtools.jar}" classpath="${build.classpath}";
        argfiles="${workspace.dir}/MobilePhotoCompileList.1st" inpath="${midp-compiled.dir}" verbose="true"/>
</target>

<!-- ============================================================== -->
<!-- Unjar the AspectJ Runtime Classes so they can be preverified -->
<!-- ============================================================== -->
<target name="copy-ajruntime-classes">
    <unzip src="${aspectjrt.jar}" dest="${ajc-compiled.dir}"/>
</target>

<!-- ============================================================== -->
<!-- Copy Image Files so they are bundled with the jar file -->
<!-- ============================================================== -->
<target name="copy-image-files">
    <mkdir dir="${preverified.dir}/images"/>
    <copy todir="${preverified.dir}/images">
        <fileset dir="${images.dir}"/>
    </copy>
</target>
</target>
<!-- Preverify MIDP classes, Aspects and the AspectJ Runtime Library-->
<target name="midp-preverify">
  <!-- Note we have to include some classes from J2SE in order for the AspectJ runtime classes to preverify. For now, the required classes are in the a temp directory -->
  <wtkpreverify srcdir="${ajc-compiled.dir}" destdir="${preverified.dir}" classpath="${build.classpath};${j2se.partial.zip}" />
</target>

<!-- Generate final JAR file for MIDP devices and update JAD-->
</target>

<!-- Compile/convert Jar file for BlackBerry using rape.exe -->
</target>
<!-- Convert Jar file for Palm OS using Sun MIDP converter -->
<target name="make-prc">
  <exec executable="${palm.midp.home}/converter.bat">
    <arg line="-Xdescriptor:${jar.dir}/${jad.name}"/>
  </exec>
</target>

<!-- Run the Midlet in the standard Phone Emulator -->
<target name="run-default-phone-emulator">
  <wtkrun jadfile="jars/${jad.name}" device="DefaultColorPhone">
    wait="true"/
  </wtkrun>
</target>

<!-- Run the Midlet in the Motorola Phone Emulator -->
<target name="run-motorola-i730-phone-emulator">
  <exec executable="&quot;${motorola.home}/bin/emulator&quot;">
    <arg line="-Xverbose:monitors -Xdevice:i730 -
      Xdescriptor:&quot;${jar.dir}/${jad.name}&quot;"/>
  </exec>
</target>

<!-- Run the Midlet in the Siemens Phone Emulator -->
<target name="run-siemens-phone-emulator">
  <exec executable=""${Siemens.home}/emulators/CX65/bin/emulator">
    <arg line="-Xverbose:monitors -Xdevice:i860 -
      Xdescriptor:&quot;${jar.dir}/${jad.name}&quot;"/>
  </exec>
</target>
</target>

<!-- == RUN THE MIDLET IN THE WTK 2.2 BETA PHONE EMULATOR -->
<!-- == RUN THE MIDLET IN THE WTK 2.2 BETA PHONE EMULATOR -->
<target name="run-beta-default-phone-emulator">
  <exec executable="${wtk-beta.home}/bin/emulator">
    <arg line="-Xdescriptor:"
      Xdescriptor:="${jar.dir}/${jad.name}"/>
  </exec>
</target>

<!-- Info Target: Echo usage and information about this build script-->
<target name="info">
  <echo>
    Insert any relevant help information here...
  </echo>
</target>

</project>
Appendix C: SmsAspect Source Code Listing

```java
public aspect SmsAspect {
    Command smsSendCommand = new Command("Send SMS", Command.ITEM, 1);
    Command smsReceiveCommand = new Command("Receive SMS", Command.ITEM, 1);
    Command acceptPhotoCommand = new Command("Accept Photo", Command.ITEM, 1);
    Command rejectPhotoCommand = new Command("Reject Photo", Command.ITEM, 1);
    NetworkScreen networkScreen = new NetworkScreen("Reciever Details");
    String selectedlmageName = "null";
    BaseController controller;
    byte[] incominglmageData;

    //Define pointcuts
    pointcut initMenuItems(List l) : args(l) && execution(*
            PhotoListScreen.initMenu(List));

    pointcut handleCommandAction(BaseController sc, Command c,
            Displayable d) : args(sc, c, d) && execution(*
            BaseController.postCommand(BaseController, Command,
            Displayable));

    pointcut imageReceived(byte[] imageData) : args(imageData) &&
            execution(* SmsReceiverThread.imageReceived(byte[]));

    //Define advice
    after(List l) : initMenuItems(l) {
        l.addCommand(smsSendCommand);
        l.addCommand(smsReceiveCommand);
    }

    after(byte[] imageData) : imageReceived(imageData) {
        if (imageData==null)
            System.out.println("imageData was NULL!");
        if (controller==null)
            System.out.println("controller was NULL!");

        Alert alert = new Alert("New Incoming Photo");
        alert.setString("A MobilePhoto user has sent you a Photo. Do you want to accept it?");
        alert.addCommand(acceptPhotoCommand);
        alert.addCommand(rejectPhotoCommand);
        alert.setCommandListener(controller);
        controller.setCurrentScreen(alert);
        incominglmageData = imageData;
    }

    after(BaseController sc, Command c, Displayable d) :
```
handleCommandAction(sc, c, d) {

    // Set the aspect's controller variable to the midlet one
    controller = sc;
    String label = c.getLabel();

    if (label.equals("Send SMS")) {
        selectedImageName = sc.getSelectedImageName();
        networkScreen.setCommandListener(controller);
        controller.setCurrentScreen(networkScreen);
    } else if (label.equals("Send Now")) {

        // Get the data from the currently selected image
        ImageAccessor accessor = new ImageAccessor(sc.getModel());
        ImageData ii = accessor.getImageInfo(selectedImageName);
        byte[] imageBytes = accessor.loadImageBytesFromRMS(ii.getParentAlbumName(), ii.getImageLabel(),
            ii.getForeignRecordId());

        // Get the destination info
        String smsPort = "1000";
        String destinationAddress = "5550001";
        String messageText = "Command goes here";

        smsPort = networkScreen.getRecPort();
        destinationAddress = networkScreen.getRecPhoneNum();

        SmsSenderThread smsS = new SmsSenderThread(smsPort, destinationAddress, messageText);
        smsS.setBinaryData(imageBytes);
        new Thread(smsS).start();
    } else if (label.equals("Receive SMS")) {

        // Start receiver thread
        SmsReceiverThread smsR = new SmsReceiverThread(sc);
        System.out.println("SMSAspect::Starting SMSReceiver Thread");
        new Thread(smsR).start();
    } else if (label.equals("Cancel Send")) {

        // TODO: If they want to cancel 'sending the SMS message, send them back to main screen
        System.out.println("Cancel sending of SMS message");
    } else if (label.equals("Accept Photo")) {

        Image image = Image.createImage(incomingImageData, 0, incomingImageData.length);
        PhotoViewScreen canv = new PhotoViewScreen(image);
        controller.setCurrentScreen(canv);
    }
}
} else if (label.equals("Reject Photo")) {
    //TODO: Go back to whatever screen they
    were previously on
    System.out.println("Reject Photo command");
}