REGRESSING OBJECT-ORIENTED PRINCIPLES TO ACHIEVE PERFORMANCE

GAINS ON THE JAVA PLATFORM, MICRO EDITION

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

_____

Sean Christopher Cook

Certificate of Approval:

_____	_____
Richard Chapman Associate Professor Computer Science and Software Engineering	David A. Umphress, Chair Associate Professor Computer Science and Software Engineering
_____	_____
John A. Hamilton, Jr. Associate Professor Computer Science and Software Engineering	Joe F. Pittman Interim Dean Graduate School

REGRESSING OBJECT-ORIENTED PRINCIPLES IN ORDER TO ACHIEVE PERFORMANCE GAINS THE JAVA PLATFORM, MICRO EDITION

Sean Christopher Cook

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

December 15th, 2006
Auburn, Alabama

REGRESSING OBJECT-ORIENTED PRINCIPLES TO ACHIEVE PERFORMANCE

GAINS ON THE JAVA PLATFORM, MICRO EDITION

Sean Christopher Cook

_____
Signature of Author

_____
Date of Graduation

THESIS ABSTRACT

REGRESSING OBJECT-ORIENTED PRINCIPLES TO ACHIEVE PERFORMANCE GAINS ON THE JAVA PLATFORM, MICRO EDITION

Sean Christopher Cook

Master of Science, December 15, 2006
(B.S. Software Engineering, Auburn University, 2005)

92 Typed Pages

Directed by David Umphress

Object-Oriented Programming is a software design method that models the characteristics of abstract or real objects using classes and objects [Sun Microsystems 2006b]. The Java language is intrinsically object-oriented; in fact Sun Microsystems' definition of Java contains the phrase "object-oriented" [2006]. It would then be assumed that the Java Platform, Micro Edition would be optimized such that correctly-implemented OO code will run, unmodified, faster than incorrectly-implemented code. This is not the case. Code exhibiting "good" OO design actually runs slower than equivalent code written in a functional fashion. In short, a Java ME MIDlet which adheres to accepted standards of "good" object-oriented design can have its execution speed increased by regressing its design.

ACKNOWLEDGEMENTS

I would like to sincerely thank my major professor, Dr. David A. Umphress, for providing the funding which made my graduate studies possible. His guidance, feedback, and sense of humor have all made this undertaking a wonderful experience.

I would also like to thank my committee members, Dr. Richard Chapman, and Dr. John A. Hamilton, Jr., for their participation on my committee, as well as their tutelage throughout my undergraduate and graduate studies.

Thanks are also due to Brad Dennis and William “Amos” Confer for their outstanding work on the MobileEdition project. In addition, I would like to acknowledge the employees of Rocket Mobile for their comments, suggestions, and code that were used throughout the development of Mobile Edition.

This research was supported in part by the National Science Foundation (NSF 0311339) and by Rocket Mobile, Inc.

Finally, I would like to express my utmost gratitude to my parents.

Style manual or journal used: ACM Computing Review

Computer software used: Microsoft Office System 2003

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 Area of Research

Software developers have recently witnessed a rapid increase in the speed and capacity of hardware, a decrease in its cost, and a proliferation of hand-held consumer electronics devices such as mobile phones and Personal Data Assistants (PDAs). In turn, this has resulted in an increased demand for software applications, outpacing developers' ability to produce them, both in terms of their sheer numbers and the sophistication demanded of them [Mikic-Rakic 2002].

The design and implementation of software for mobile devices is intimately tied to the constraints of the device. In order to overcome the constraints of hand-held devices, mobile software challenges old assumptions and demands novel software engineering solutions including new models, algorithms, and middleware [Roman et al 2000]. While recent and anticipated technological advances in wireless computing will permit users to compute anywhere, mobile platforms are unlikely to have the computational resources to solve even moderately complex problems that users routinely solve on workstations today [Drashanksky, et al 1996]. One mobile platform which has demonstrated continued success and adoption in the mobile marketplace, but is still hindered by its available computational resources, is the Java Platform, Micro Edition

(Java ME). Java ME provides a robust, flexible environment for applications running on consumer devices, such as mobile phones, PDAs, TV set-top boxes, printers and a broad range of other embedded devices [Sun 2006]. It is a collection of technologies and specifications that implementers and developers can choose from and combine to construct a complete Java runtime environment that closely fits the requirements of a particular range of devices and target markets. Each combination is optimized for the memory, processing power, and I/O capabilities of a related category of devices [Sun 2006].

The adoption of Java ME-capable devices in the marketplace has helped to drive the need for Java ME software. In a survey of mobile developers, 40% reported that they were using the Java ME platform for wireless development, and another 24% were evaluating the platform for future use [Evans Data Corporation 2004]. These developers are faced with the daunting task of developing quality software that meets the performance expectations of the marketplace while still operating within the constraints of the mobile device, and in turn, the mobile platform, Java ME.

In order to meet the needed or expected performance goals of a Java ME application, developers must implement “novel software engineering solutions” [Roman et al 2000]. This thesis presents a method by which mobile software developers can achieve performance gains in Java ME by relaxing the object-oriented constraints of the software architecture.

1.2 Statement of Problem

Object-Oriented Programming is a software design method that models the characteristics of abstract or real objects using classes and objects [Sun Microsystems 2006b]. The Java language is intrinsically object-oriented; in fact Sun Microsystems’ definition of Java contains the phrase “object-oriented” [Sun Microsystems 2006]. As a result, Java developers must create OO code in order to solve problems using the language.

It would then be assumed that a Java platform would be optimized such that correctly-implemented OO code could run, unmodified, faster than incorrectly-implemented code. This is not the case. Code exhibiting “good” OO design actually runs slower than equivalent code written in a functional fashion. In short, a Java ME MIDlet which adheres to accepted standards of ‘good’ object-oriented design can have its execution speed increased by regressing its design. This regression includes relaxing object-oriented design principles such as encapsulation, polymorphism, and inheritance. As a side-effect of relaxing these design principles, the resulting MIDlet will typically contain fewer physical files, and will be more difficult to maintain.

Since regressed OO code is much more difficult to maintain than well-formed code due to its lack of structure, it is beneficial to create a well-formed MIDlet as a baseline and then apply optimizations to it to achieve a final, optimized build of the application. Thus, this is the approach we take in our experimentation. The alternative to this approach would be to implement optimizations throughout the design process, ending with a single, optimized build; however, this optimized build will not lend itself easily to future maintenance.

While Java ME developers understand that these optimizations are possible, no research has been performed to establish how much OO design is too much in a Java ME application. This thesis presents a top-down approach for melting a well-formed OO Java ME software design into a more-efficient variant which breaks OO accepted design practices to increase performance. To support this approach, we derive empirical statistics of the effects of key OO design principles, including inheritance, polymorphism, and encapsulation, on the efficiency of a Java ME MIDlet.

1.3 Motivation for Research

In the spring of 2004, Auburn University was approached by Rocket Mobile of Los Gatos, California to develop a Java ME version of an existing news reader for mobile devices. Throughout 2004, Brad Dennis and William “Amos” Confer, both graduate students at Auburn University, designed, tested, and implemented a version of the news reader for Java ME.

The initial version of MobileEdition was a well-formed Java ME application but did not perform well enough to be considered a marketable product [Rocket Mobile 2005]. Rocket Mobile engineers tested and profiled the initial application on various handsets and then used this data to pinpoint performance bottlenecks in the application. They then re-wrote and consequently re-structured selected components of the application which resulted in an intermediate version which performed considerably better but still suffered reliability issues. This intermediate version was then returned to the graduate student developers at Auburn University along with general comments on the code base. These comments, as well as extensive handset testing results, were then used in the development of the final version of MobileEdition.

The final version of MobileEdition addressed the functional requirements of the application, as well as non-functional requirements, such as performance and responsiveness. Handset-specific errors, such as stalling and crashing, were solved through an extensive testing phase utilizing a Nokia 3650, which was the target device for the application.

As MobileEdition's performance increased, it became clear to the developers that the object-oriented characteristics of the application were slowly deteriorating. The maintainability of the application was clearly decreasing, and the overall program structure lost its rigidity. A post-mortem examination of the final delivered version of MobileEdition revealed three core types of object-oriented regressions which had been applied to the initial version in order to effect a more efficient Java ME application. These regressions fell into three distinguishable categories which were termed canonical, algorithmic, and architectural.

CHAPTER 2 LITERATURE SURVEY

2.1 Literature Overview

A survey of the literature reveals the key challenges of developing Java ME applications, as well as solutions to these challenges. These challenges, which all result from the limited resources available to Java ME, have been approached with various techniques, or “optimizations”. These optimizations include high-level, software-based solutions, and low-level, compiler-based solutions.

2.1.1 Mobile Software Constraints

An under-lying theme of mobile software literature is that “pervasive mobile devices have extremely limited memory and storage spaces, requiring us to minimize both the storage and runtime footprints of the application” [Yuan 2004].

2.1.2 Java ME Constraints

Sun Microsystems states that the primary limitation in many Java ME systems is the amount of memory available to store and run applications. Many current MIDP devices, for example, limit application sizes to 50K or less [2002].

Kochnev and Terkhov [2003] present four difficulties of developing Java ME software. First, the size of the Java ME application and its data (many business-class mobile devices with more memory exist, but size is still a problem for an overwhelming majority of phones— including mass-market phones currently produced for entertainment purposes and "legacy" phones still widely used). Second, intermittent network connections with lower bandwidth present the added challenge of limited communications. Third, small display sizes, which can cause problems for creating an acceptable user interfaces, plague the market. And fourth, primitive facilities for inputting text information further disconnect users from applications.

2.1.3 Portability

"Write once—run anywhere" does not work for a Java ME platform [Kochnev and Terkhov 2003]. Frequent application, porting and tweaking on many devices is still a way of life for Java ME developers because the platform does not correctly address the "device fragmentation" problem [Yuan 2005]. Tira Wireless has created a commercial software package which automates the portability process [2006].

2.2 Known Performance Enhancement Techniques for Java ME

The known performance enhancements for Java ME target the constraints of the platform, and most are aimed at the limited amount of memory available to the platform. Multiple optimizations are frequently combined to produce a final efficient MIDlet.

2.2.1 High Level Optimizations

High level optimizations for Java ME should always be considered before attempting low-level optimizations. Examples of high-level optimizations include pre-calculating values, loop optimizations, and using StringBuffer[]'s in place of Strings [Shivas].

2.2.2 Low Level Optimizations

Mike Shivas states that low-level optimizations easily plug into existing code but tend to degrade readability as much as they improve performance. Strength reduction, or replacing a relatively slow operation with a fast one, is a common optimization technique. The most common example is using the bit shift operators, which are equivalent to multiplication and division by a power of two [2005].

2.2.3 Class Refactoring

Eric Giguere states that the first step in reducing size in a Java ME application is to remove unnecessary classes by pruning the application's functionality [Sun Microsystems 2002]. One approach suggests that after creating the application's initial version, you must transform it by merging classes. Technically speaking, this is performed by applying the inline class refactoring pattern. The transformation's main goal is to remove all user-defined classes from the program so that the final version will contain only standard classes that deal with the user interface, display, timer, and so forth—that is, only the indispensable minimum. The changes amount basically to merging the user-defined classes with standard ones, creating new interfaces, and moving methods to the new class, which results in a "flat" class model [Kochnev and Terkhov 2003].

2.2.4 Garbage Collection Optimizations

Michael Yuan explains that on mobile devices, due to the small amount of available memory, the garbage collector must run more often. When the garbage collector runs, its thread takes up precious CPU cycles and slows down all other application processes. For effective Java ME applications, the developer must minimize object creation and quickly dispose of objects that are no longer in use. Yuan provides five steps to avoid unnecessary garbage collections in Java ME. First, carefully examine design patterns in early stages of the development cycle. Second, concisely reuse existing objects at the implementation level. Third, use arrays and StringBuffers. Fourth, close network connections, file handlers, and Record Management System (RMS) record

stores quickly after use. And finally, free resources when using native libraries, especially since native libraries are not subject to garbage collection [2004].

2.2.5 Obfuscation

In Java ME, obfuscation can help protect applications that are deployed to millions of devices. Importantly, but often forgotten, obfuscation can also help developers with some other equally important issues—namely application size and performance [White 2004].

2.2.6 Optimize the Packaging Process

In the packaging process, developers should include only the classes they actually use. They can do this manually for smaller libraries or use automatic tools bundled with some Java ME IDEs for large libraries. If they want to further reduce the binary application size, they can use a bytecode obfuscator to replace long variable names and class names with shorter, cryptic ones.

Since the MIDP runtime loads classes only as needed, applications can be partitioned into separate parts to reduce the runtime footprint. For MIDP applications, the MIDlet suite can contain several relatively independent MIDlets [Yuan 2004].

2.3 Software Tools for Java ME Optimization

The complexity of existing optimization solutions led to the development of many automated software tools which target Java ME optimizations. Innaworks' mBooster is an "automated optimizer for Java ME applications." At the core of mBooster is

proprietary optimizing Java compiler [2006]. Another automated tool, jPRESTO performs obfuscation, code reduction, class amalgamation, and image optimization [S5 Systems].

CHAPTER 3 SOLUTION

3.1 Approach

Our solution to inefficient Java ME applications involves first developing a well-formed object-oriented MIDlet. Once this MIDlet meets all of its functional requirements, its developers can then address non-functional requirements, such as required performance standards, through optimizations to the code base. The developers should start by regressing the core object-oriented features of the system—encapsulation, inheritance, and polymorphism. Next, the developer should identify easily comprehensible code segments which can be replaced with a more efficient segment which may not be as easy to comprehend. Finally, the developer should consider entire components of the application which can be replaced by more efficient variants.

3.2 Canonical Regressions

Canonical regressions are the simplest changes that can be made to a well-formed Java ME application in order to achieve performance gains. These changes involve modifying encapsulation, inheritance, or the polymorphism in an application. These changes to an application's structure greatly affect the maintainability of the application while flattening, or simplifying the overall structure. Due to their simplicity, target areas in code where these regressions can be applied are immediately recognizable to developers. Each type of canonical regression was shown through experimentation to increase the performance of a test application.

3.2.1 Encapsulation

An example of a canonical regression is the removal of getters and setters in favor of implementing publicly-accessible members in a class. Once this canonical regression has been applied to a class, every other class which utilizes the members of the class must be modified. The negative effects of changing the code base are minimal from a code maintenance standpoint since developers can understand "x=object.x" to mean "x=object.getX()".

Figure 3.1 presents a normal, unoptimized Java ME class which requires getters and setters to access and modify its members. Figure 3.2 presents a regressed version of the same class which allows public access to its members, as opposed to getter and setter functions. In addition to the speed increase resulting from removing function calls, the smaller optimized class will help to achieve a smaller overall JAR size.

```
class Circle{
 private int x;
 private int y;
     private double radius;
     public Circle(int _x, int _y, int _radius){
               this.x=_x;
               this.y=_y;
               this.radius=_radius;
     }
     public getRadius(){ return radius; }
 public int getX(){ return x; }
 public int getY(){ return y; }
}
```

FIGURE 3.1 – Class Pre-Regression

```
class Circle{
        public int x;
        public int y;
        public double radius;
        public Circle(int _x, int _y, int _radius){
                this.x=_x;
                this.y=_y;
            this.radius=_radius;
        }
}
```

FIGURE 3.2 – Class Post-Regression

To test the performance gains brought about by regressing encapsulation, we created a simple class with three members that were accessed via getters and setters. Next, we made an identical class but set each of the three members to public visibility. The source code for our classes is available in Appendix B. We then ran the two classes through identical loops were each member had its value set and read. The results are presented in Figure 3.1.2. The average time between runs for the normal test case was 6.5 milliseconds, while the optimized test case averaged 0.2 milliseconds. The results of testing revealed that the canonically-regressed code segment performed 32.5% faster than the normal code segment.

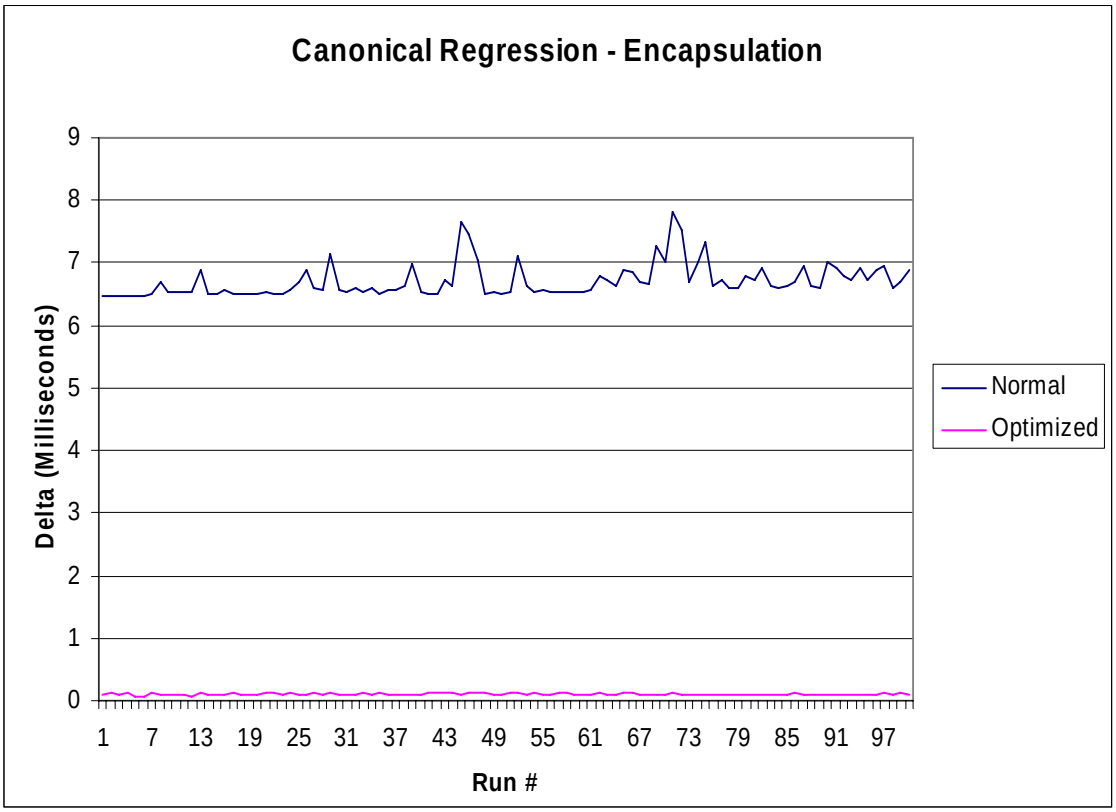

FIGURE 3.3 - Comparison of an initial class vs. a canonically-regressed class

3.2.2 Inheritance

The canonical regression of combining super and subclass functionalities as one requires an application-wide change in how each affected object is handled. For instance, method signatures involving the original sub-class have to be changed to accept the new super-class.

To test the effects of regressing inheritance in a Java ME application, we created three classes (see Appendix B). The first two classes are named SuperClass and SubClass and their respective names distinguish their role in a simple inheritance tree.

SuperClass provides 10 functions and a single member, x, which are all inherited by SubClass. SubClass defined its own version of function6(), which overrides SuperClass's function6(). The third class, NoInheritanceClass, has all 9 functions provided by SuperClass and one function, function6(), is the same version provided by SubClass. Figure 3.4 provides a UML diagram of the class structures.

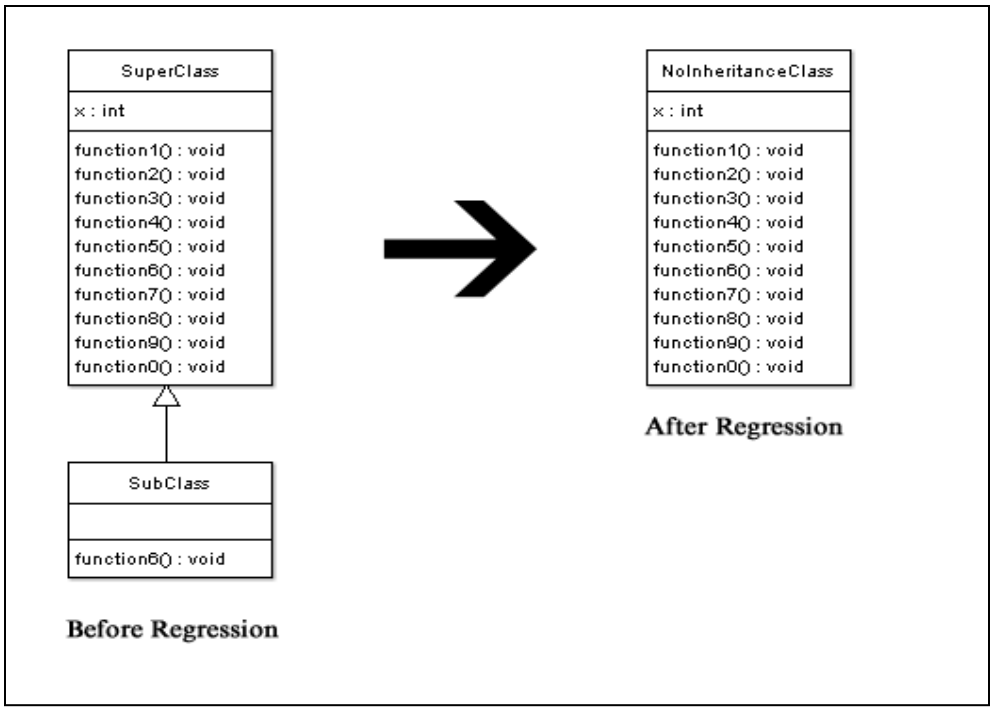

FIGURE 3.4 - Inheritance Regression

To test the three classes, we first measured the amount of time required to create an instance of SubClass and then call each of its 10 members. This time was then averaged across 1000 runs and the average delta time in milliseconds was recorded. We repeated this process 100 times. Next, we measured the amount of time required to create an instance of NoInheritanceClass and then call each of its 10 members. This time was averaged across 1000 runs and once again, this average delta time was recorded. This process was repeated 99 more times and average delta times were recorded.

Figure 3.5 presents a graph of the results of the test. The average delta running time of the SubClass was 0.62ms. The average delta running time of the NoInheritanceClass was 0.0005ms. Thus, for this simplistic example regressing inheritance provided a roughly 1200% speed increase.

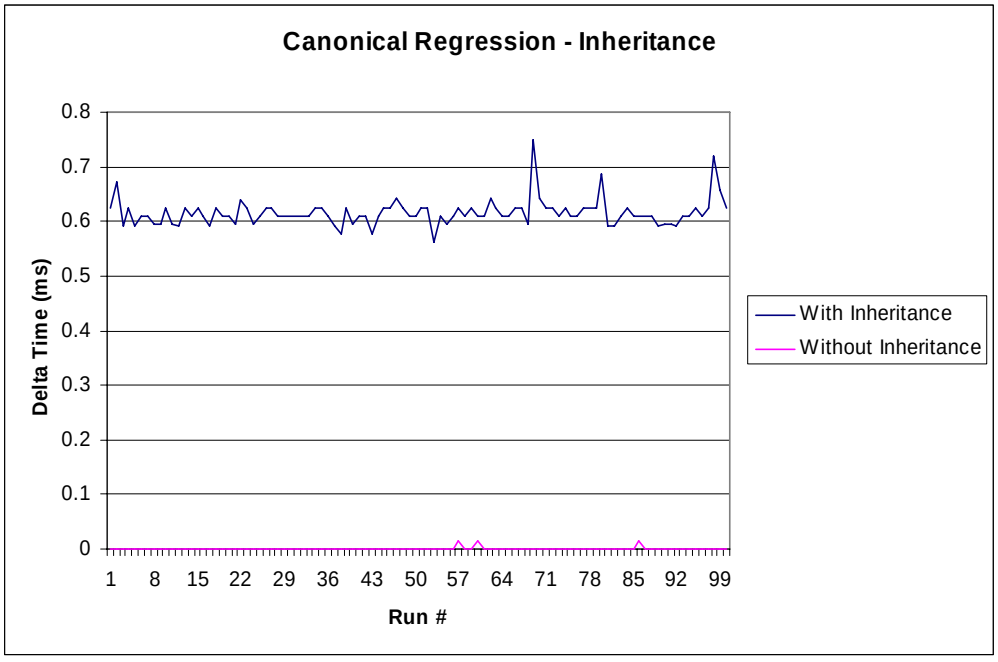

FIGURE 3.5 Canonical Regression - Inheritance - Results

3.2.3 Polymorphism

Modifying polymorphism in an application through canonical regressions helps to eliminate run-time decisions which can greatly affect the performance of a Java ME application. Care must be taken to ensure that all prior contracts or interfaces between classes are still upheld. The resulting application trades code elegance for execution efficiency, which can adversely affect the maintainability of the application.

To test polymorphism, we created three classes: Widget, SmallWidget, and BigWidget, where SmallWidget and BigWidget both extended the Widget class. Figure 3.6 presents the class diagram for SmallWidget and BigWidget. To handle these three classes polymorphically, we created a WidgetWorker function which takes an argument of type Widget. To handle these three classes without polymorphism, we created two functions, SmallWidgetWorker and BigWidgetWorker, which operate on their respective object types.

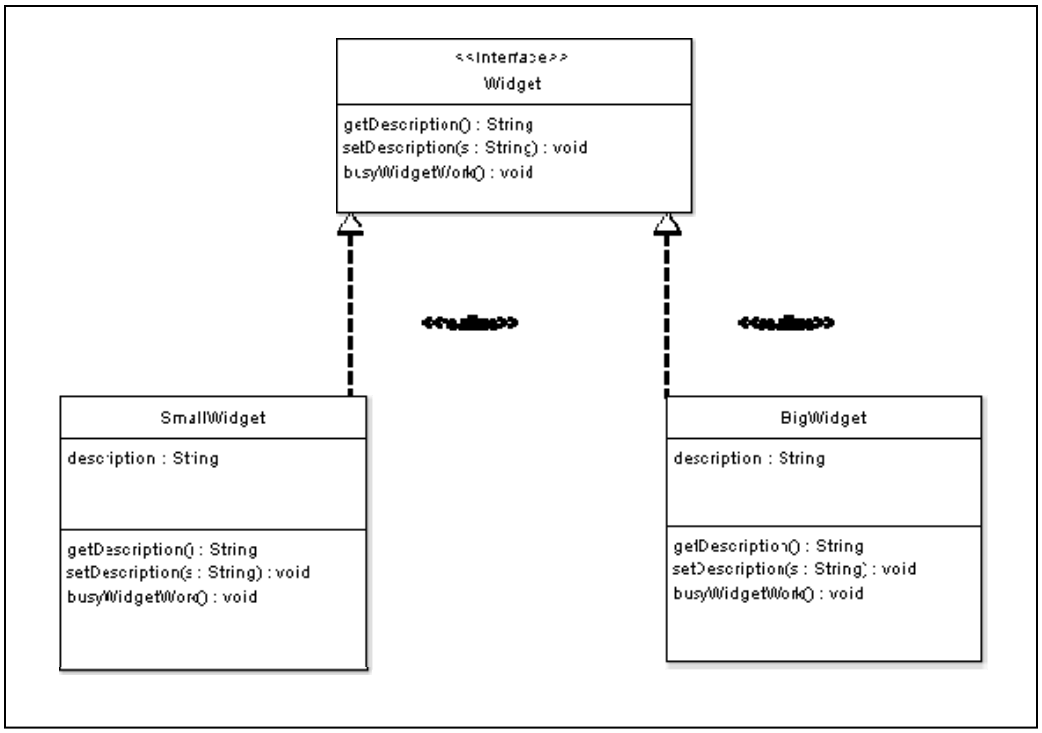

FIGURE 3.6 – Inheritance test case class diagram

```
private void
widgetWorker(Widget w) {
        w.getDescription();
        w.setDescription("foo");
        w.busyWidgetWork();
}
```

```
private void
smallWidgetWorker(SmallWidget w)
{
        w.getDescription();
        w.setDescription("foo");
        w.busyWidgetWork();
}

private void
bigWidgetWorker(BigWidget w) {
        w.getDescription();
        w.setDescription("foo");
        w.busyWidgetWork();
}
```

FIGURE 3.7 – Regressing a polymorphic function

For our polymorphic test case, we first create a SmallWidget and a BigWidget. Then for 100 iterations we average the amount of time it takes to call WidgetWorker with both classes 1000 times. For our test case without polymorphism, we perform the same procedure except we call the SmallWidgetWorker and BigWidgetWorker functions respectively. The average delta time for the polymorphic test case was 1.7ms, while the average delta time for the regressed test case was 0.002ms. Thus, for this simple example the performance increase was 970%. Figure 3.8 charts the difference in average delta times between the two test cases over their 100 iterations.

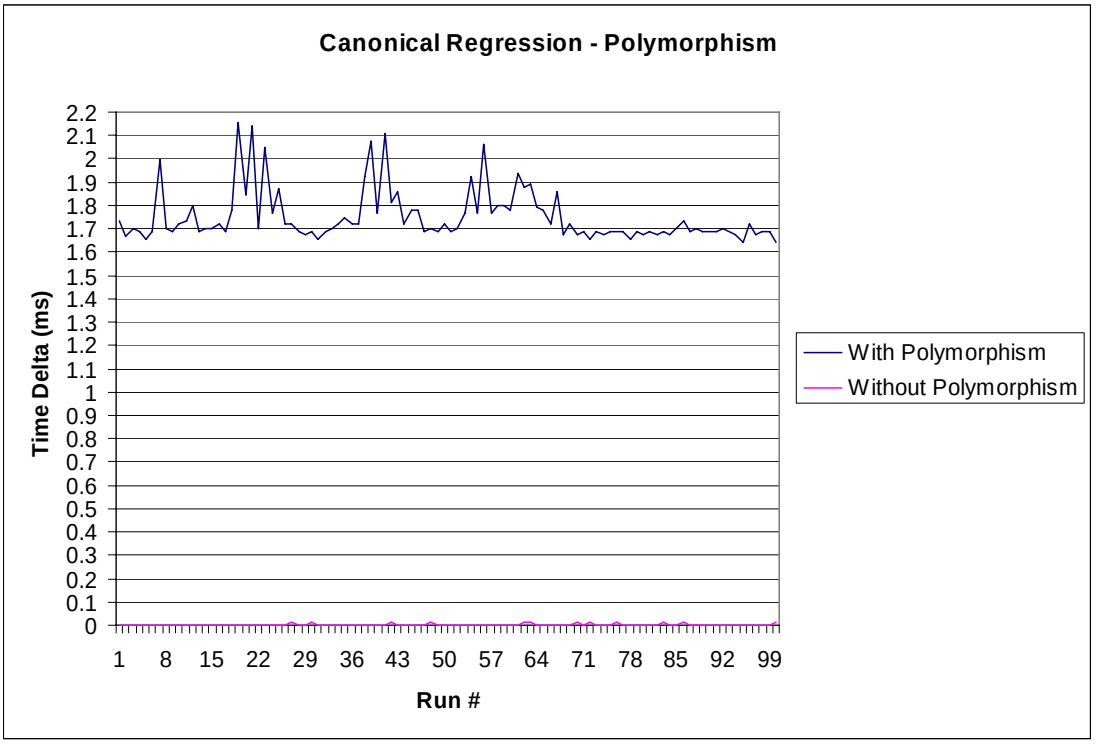

FIGURE 3.8 – Canonical Regression - Polymorphism

3.3 Side Effects of Canonical Regressions

The most obvious side effect of canonical regressions is the invalidation of core OO principles. For example, by removing getters and setters from the classes in a MIDlet, the developer is invalidating the principle of encapsulation. If an application is initially developed following traditional "data-hiding" techniques, replacing calls to simple getters and setters will not affect data integrity in the application. This point emphasizes one of the reasons why it is necessary to apply OO regressions after a well-formed application has been created, and not during the development process.

A positive unintended side effect of removing getters and setters is smaller JAR file sizes. The JAR file size is frequently mentioned as a starting point for optimizing Java ME code [Sun 99]. By removing two functions per every member in a class throughout an application the developer is effectively trimming the size of the JAR.

The effect of canonical regressions on the maintenance of a MIDlet is substantial. As a result, maintenance changes will need to be analyzed by the developer before implementation so that he can determine whether or not the regressed MIDlet or the baseline MIDlet should be the target of the maintenance update. If he decides to return to the baseline MIDlet, a new optimization process will need to be applied after maintenance is complete.

3.4 Algorithmic Regressions

The purpose of an algorithmic regression is to replace a slower, easily comprehensible code segment with a more efficient segment which may not be as easy to comprehend. An example of an algorithmic regression in Java ME is an object pool. An

object pool instantiates a group of related objects at a pre-determined time in an application rather than allowing the application to instantiate each individual object as it is needed. The efficiency of pooling objects compared to creating and disposing of objects is highly dependent on the size and complexity of the objects [Muir 2002].

To understand the functionality of an object pool, consider the following scenario: A developer wishes to develop a mobile web browser which downloads page content, parses the content, and then displays the content. For each image that is found in the content, the developer will need to store the image content into an image object and then display that object. To do this, he can either create a new image object each time from the heap or re-use a previous image object that is stored in an object pool.

```
//download and parse content
//create image objects to display
for( i=0; i<imageContent.length; i++ ){
        canvas.addImage( new Image( imageContent[i] ) );
}
```

FIGURE 3.9 – Without Algorithmic Regression

```
//initialize a pool of 100 images
ImagePool pool = new ImagePool( 100 );
//download and parse content
//add images to the canvas
for( i=0; i<images.length; i++){
        img = pool.get();
        if (img != null){
        canvas.addImage( img.setContent(imageContent[i]) );
        }
}
```

FIGURE 3.10 – With Algorithmic Regression Applied

3.4.1 Testing Object Pools

To test object pools, we created a simple class with three members and three methods (see Appendix B). For a test case without algorithmic regression, we measured the average amount of time it required to instantiate the object and then call each of its three methods over 100 runs. This process was repeated 10 times which resulted in 10 average times in milliseconds. The number of runs was limited to 10 because on the desktop computer used for experimentation test runs greater than 10 would often result in out-of-memory errors.

For a test case involving algorithmic regressions, we repeated the 10 by 10 runs, but instead of instantiating objects during the loops, we created an object pool of 100 pre-instantiated objects before-hand. As a new object was needed, it was retrieved from the

pool and then the three methods were called. The average delta time for non-regressed code was 17.8ms, while the average time for the object pool was 0.04ms. Thus, for our simple example a performance increase of 445% was achieved using an algorithmic regression. Figure 3.11 graphs the test data for the two test cases.

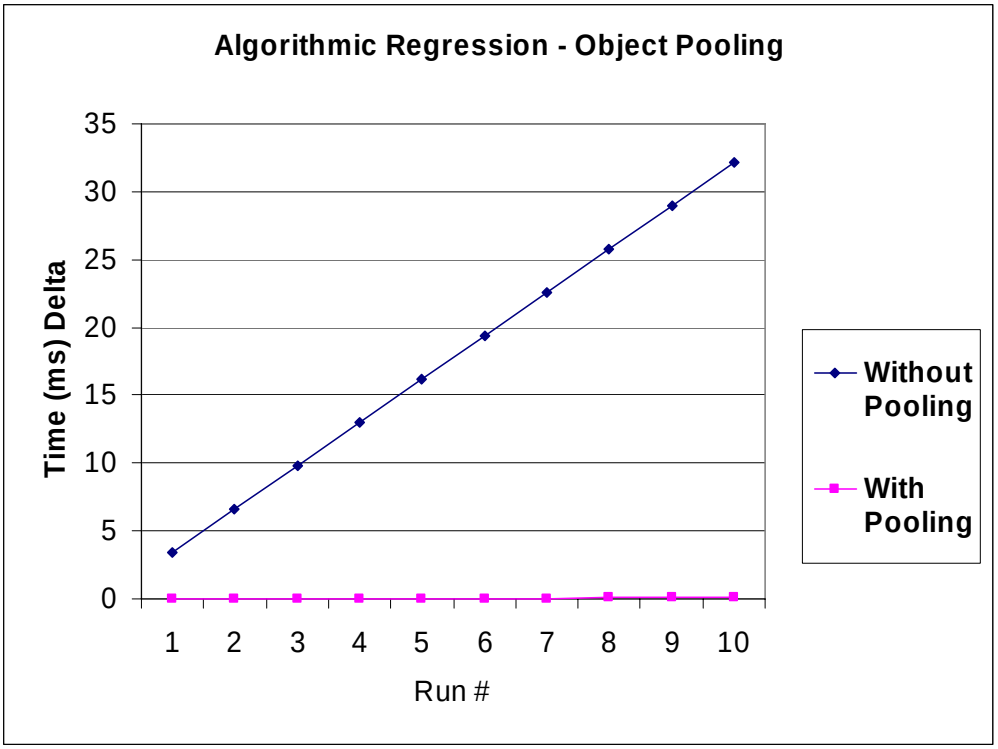

FIGURE 3.11 – Algorithm Regression via Object Pooling

3.4.2 Side Effects of Algorithmic Regressions

The positive side effect of algorithmic regressions on an application is improved performance. Object pools have deterministic access and reclamation costs for both CPU and memory, whereas object creation and garbage collection can be less deterministic [Bialach 2002]. In addition to improved execution speeds, algorithmic regressions such as object pools result in improved heap usage in a MIDlet. This side effect helps to overcome the problems associated with the amount of memory available to Java ME applications.

A negative side effect of algorithmic regressions is that object instantiation is a core aspect of an application. By changing an object's lifetime, care must be taken to ensure that the original interactions of the object, as well as any code clean up needed for the object, remain in place. So, just as for canonical regressions, the developer must make a conscious decision to either apply updates to the regressed version of the MIDlet or the baseline version.

3.4 Architectural Regressions

An architectural regression is a restructuring of an entire component in such a way that it significantly alters the overall design of the software. These regressions are the most intensive of the three regression types as they involve changing multiple classes which work together to achieve a common functionality. The developer must identify and retain the communication paths between the target component and the rest of the application while maintaining the original essence of the component. An astute developer will use profiling data combined with Pareto's Principle to identify the architectural regression candidate components of his MIDlet.

An interesting property of architectural regressions is that they can be aggregates of other regressions. That is, the collection of classes which replace an inefficient component can implement canonical regressions and algorithmic regressions. Unlike the previous regressions, the self-contained nature of a component can effectively hide internal regressions to outside classes.

Since an algorithmic regression is an aggregate of the previous tested and proven regressions, no specific test cases were constructed. Instead, an example of this regression is provided.

3.5.1 Architectural Regression Example

A Java ME web browser must implement a parsing component to parse web pages into viewable content. If a baseline implementation of the web browser provides sub-par performance, a developer can decide to perform an architectural regression on the parsing component. To do so, he would first identify the data paths between the

component and the rest of the application. Next, he can re-design the component to achieve better performance, including canonical and algorithmic regressions in the development of the component classes. Finally, he can replace the component in the system architecture, ensuring that the component's interface to external classes remains intact.

3.5.2 Side Effects of Architectural Regressions

Since architectural regressions involve self-contained components it is possible to achieve better performance without greatly affecting the maintainability of a MIDlet. In fact, in some scenarios a lone architectural regression of a component may provide the required increase in the overall speed of an application. The difference then between the baseline MIDlet and the regressed MIDlet is only one component. Thus, the regressed MIDlet will still easily accommodate future maintenance activities.

CHAPTER 4 CASE STUDY

4.1 Overview

This case study presents the Java ME web browser "MobileEdition" by Rocket Mobile as an example of an industry-level application which employs the object-oriented optimizations presented in this thesis. The intended purpose of the application is to download, parse, and render well-formed XHTML documents on Java ME-enabled devices. The application was chosen because its initial version was a well-formed, object-oriented Java ME application whose performance was unsuitable for a handheld device. Then, through revisions which utilized numerous industry-standard performance enhancements, including those presented in this thesis, MobileEdition matured into a deployable Java ME web browsing solution.

4.2 Version History

MobileEdition progressed through two major versions. For the purpose of this case study, these versions will be referred to as the initial, or pre-regressed version, and the final or post-regressed version. Both versions were targeted for and tested on the Nokia 3650 handset.

4.2.1 Initial Version

The initial version of MobileEdition was developed by Auburn University graduate students Brad Dennis and Amos Confer. This version provided the required core functionalities required by Rocket Mobile, but did not perform well enough on the Nokia 3650 handset to be considered a marketable product [Rocket Mobile 2005]. The structure of the application was very thoroughly defined and included 82 source files. In order to maintain its rigid object-oriented structure, this version of the application utilized numerous object-oriented design patterns, including mediators and facades.

4.2.2 Intermediate Revisions

After submitting the initial version to Rocket Mobile for review, engineers at Rocket Mobile returned an optimized version which employed numerous optimizations but was still affected by stalls and crashing on the handheld device. This version contained only 44 source files, but it still maintained the essence of the initial version. The most extensive change to the application was in the core rendering component, which was changed from a tag-based scheme to a cell-parsing scheme which rendered pages significantly faster.

4.2.3 Regressed Version

The final version of the application corrected numerous problems seen in the pre-regressed version of the application, including the stalls and crashes, while achieving the needed level of performance on the Nokia 3650 handset. It was the result of contributions from engineers at Rocket Mobile, as well as Brad Dennis and Sean Cook from Auburn University.

4.3 Optimizations Performed

There were many optimizations applied to the initial version of MobileEdition which contributed to the final version's performance on the target handheld device. Some of these optimizations included modifying object-oriented aspects of the application, while others included utilizing methods and data structures which have lower overheads associated with their use.

Closer examination of the object-oriented optimizations which were applied to MobileEdition revealed that there were three significant types of regressions applied to the MIDlet. There were canonical regressions involving the regression of inheritance, polymorphism, and encapsulation. There was also a significant algorithmic regression which resulted in a collection of Worker threads. And finally, there was a complete replacement of the core component of the application which handled parsing and displaying XHTML. Each of these regressions had a different, distinct effect on the overall maintainability of the application once it was applied.

4.3.1. Canonical Regressions

There were multiple canonical regressions applied to the initial version of MobileEdition, both as stand alone regressions, and as essential elements of the core architectural regression. These regressions illustrated a key difference between industrial or practical Java ME development and theoretical or academic Java ME development.

4.3.1.1 Encapsulation

The original version of MobileEdition implemented standard data hiding techniques typical of object-oriented applications; that is, each class was responsible for maintaining the current state of its members, and access to members was typically controlled by getter and setter functions. In contrast, the final version of the application utilized public data members to speed up access times.

One such example is the RM_JTonzMIDlet class, which serves as the main MIDlet of the application. In the initial version of the application, a getter method named "getDisplay()" is used to retrieve the display member. In contrast, the final version of the application declares display as a public member which can be accessed throughout the application using standard dot notation.

Original:	**Optimized version:**

private Display display = null;
public getDisplay(){
return display;
}
	public Display display = null;

FIGURE 4.1 – Canonical Regression Example from MobileEdition

4.3.1.2 Inheritance

In the initial version of the application, two classes existed for every XHTML tag which must be parsed: one for an opening tag and one for a closing tag. Each of these 33 classes extended an RM_Tag superclass which served as the base type for polymorphic calls during the rendering process. Figure 4.2 illustrates the initial tag class hierarchy. In the modified version, only 19 classes extended a Cell superclass. The idea of a "type" of tag no longer existed as an object; instead, it existed as a condition in a much more complex CellParser object.

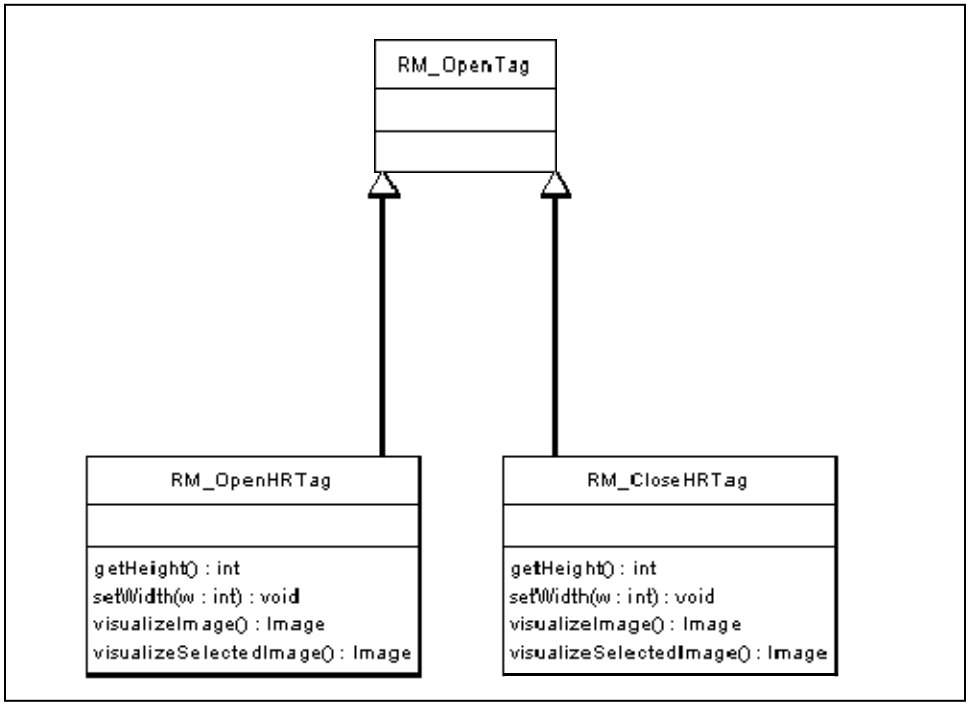

FIGURE 4.2 – Inheritance in pre-regressed Mobile Edition

The benefits of canonically regressing the tag structure's inheritance were three-fold. First, the changes resulted in fewer class files, which in turn means a smaller JAR size. Second, when a new tag is now added you only have to add a condition in a file instead of create two new objects. And finally, the change facilitated the updated, more efficient parsing component to parse pages.

4.3.1.3 Polymorphism

Polymorphism is utilized throughout the initial application using interfaces to help define listeners which handle different events that an object may encounter. For instance, an "RM_PageDownloadListener" serves to define the contract required to capture a page download event. Figure 4.3 illustrates the original RM_PageDownloadListener.

```
RM_PageDownloadListener.java:
public interface RM_PageDownloadListener {
        void rmPageNotify(RM_Page page, int response);
}
```

FIGURE 4.3 – Interface example from MobileEdition

While the initial implementation allowed for potentially more listeners, all page listening was actually maintained in a central "RM_PageManager" class. Thus, all listeners that were previously passed as parameters, and would thus be determined dynamically at run-time, were replaced with the RM_PageManager type.

```
public void registerPageListener(final RM_PageDownloadListener listener) {
        if (listener == null) {
                deregisterPageListener();
        } else {
                pageListener = listener;
        }
}
```

FIGURE 4.4 – Original registerPageListener

```
public void registerPageListener(final RM_PageManger listener) {
                if (listener == null) {
                        deregisterPageListener();
                } else {
                        pageListener = listener;
                }
        }
```

FIGURE 4.5 – Modified registerPageListener

The benefits of regressing polymorphism in the MIDlet once again include lowering the number of class files compiled into the JAR. Furthermore, the dynamic lookup required for parametric polymorphism can have a significant impact on performance. Thus, by removing these calls a developer is at least some what increasing his application's performance.

4.3.2 Algorithmic Regressions

Throughout the initial version of MobileEdition, new threads are created as new pages are downloaded. One of the initial responses we received back from Rocket Mobile was the need to create Worker threads which could handle "Work" tasks, such as downloading images in the background [Rocket Mobile 2005]. Prior to implementing the Worker threads, MobileEdition simply used new() to create a new Thread class when one was needed to perform a task in parallel with the main execution of the program.

The post-regressed version of the application featured a collection of already-instantiated Worker threads in the mediator. As a class needed a task executed in a separate thread, it would give the mediator an instance of Work, and then continue while the WorkerThread worked in the background. Figure 4.6 presents the source code for a WorkerThread object.

```
public class WorkerThread extends Thread {

 RM_HttpMediator mediator;
 private boolean timeToDie = false;
 private Work work;

 public WorkerThread(RM_HttpMediator mediator) {
  super();

  this.mediator = mediator;
  start();
 }

 public void run() {
  while(!timeToDie) {
    work = mediator.getWork();
         if ((!timeToDie) && (work!=null)) {
      work.run();
      work = null;
    }
  }
  }

 public void die() {
    timeToDie = true;

    if (work!=null) {
      work.die();
      work = null;
    }
  }

  public void cancel() {
    if (work!=null) {
      work.die();
      work = null;
    }
  }

 }
```

FIGURE 4.6 – WorkerThread Source Code

The implementation of WorkerThreads in MobileEdition is an example of an algorithmic regression because it took an easily comprehensible segment of code that creates threads and turned it in to a more efficient, yet more complex segment.

4.3.3 Architectural Regressions

One of the most significant changes made by Rocket Mobile engineers to the initial version of MobileEdition was to the parsing facility of the application. The logic behind parsing changed, as did the way each element was interpreted-- instead of seeing a

page element as an object, the new parser viewed the element as a condition. Canonical regressions played a role in the improved efficiency of the new component, and the resulting parser was implemented in the existing design with very minimal changes to other classes. Thus, the addition of the new parser constituted an architectural regression.

The initial page rendering facility operated by opening an InputStream to a web page, passing the InputStream to a ParserFacade along with a Vector to receive the parsed tags, and then iterating over the parsed tags to display the page. The code was well-formed and worked as intended on the Java ME emulator, but was very sluggish when executed on a handset. In fact, Rocket Mobile's profiling of the application showed that the parseStream() function was one of the application's biggest bottlenecks [Rocket Mobile 2005].

The replacement renderer was based on the idea of cells instead of page elements, with the difference being that many elements comprise a single cell. With this parser, as soon as a cell is through being populated with its designated children it can be displayed to the screen. Thus, the new Cell parser displays content to the screen much faster than the original version. The Cell Parser implements canonical regressions, such as when it declares

```
public static ProgressListener listener = null;
```

which allows an outside class to update the current progress listener for the Cell Canvas using dot notation.

The change in the page rendering component brought new life to the MIDlet, but at the cost of complexity in its design. The affect on maintenance outside of the component is minimal since the new component's interactions with the rest of the system are still primarily along the same communication paths as the initial version. The developers at Auburn University did see, however, that maintenance inside the component was much more complex due to the relaxation of OO principles.

4.3.4 Other Optimizations Performed

Other optimizations applied to MobileEdition included using data constructs and method alternatives with faster operation times. For example, using index operators to work with elements in vectors instead of using enumerators greatly sped up numerous sections of the code where the application iterated over a group of objects. These performance enhancements require a familiarity with the Java ME language, and should be implemented during development, unlike the 3 types of object-oriented regressions.

4.4 Result of Optimizations

The ultimate result of the optimizations that were performed on the initial version of MobileEdition is that the application was accepted as being capable of performing within the required performance standards on the Nokia 3650 handset. In addition to this result, there were other benefits, both positive and negative, which should be considered.

4.4.1 Positive Benefits

The most noticeable positive benefit of these optimizations was a gain in performance, both in the emulator and, more importantly, on the handheld device. Another benefit of these changes was that the application contained fewer classes, and in turn had a smaller compiled footprint. This simplification of the code structure made understanding the behavior of the application easier, but also had unwanted, yet unavoidable side effects.

4.4.2 Negative Benefits

The negative effect of these optimizations is that they degrade the maintainability of the code base. The initial version's code leant itself to future modifications or feature additions. However, as the application's development continued, benefits of OOP were gradually traded for the benefit of performance. Furthermore, MobileEdition suffered from numerous deadlock issues which were directly attributed to the unprotected access of members of a class.

4.5 Conclusion

MobileEdition provided first-hand evidence that encapsulation, inheritance, and polymorphism all play a role in the performance of Java ME applications. Modifying member variable's access rights provided a simple yet effective way to boost the application's performance, especially for members which were accessed frequently. The reduction of inheritance and polymorphism in the code base directly reduced the number of class files needed for the application, which resulted in an overall smaller JAR file.

MobileEdition's final version also provides insight into the difficulty associated with maintaining Java ME code whose structure has been regressed. Any future changes or additions to the application would need to be performed on the original code base, and then the same object-oriented performance enhancements performed on the final version would need to be made.

The exact measurements of the effects of each optimization technique were not measured during development, so the overall performance of the final application must be considered. Figure 4.7 provides a comparison of execution times for the initial version of MobileEdition and the final spring 2006 version of MobileEdition. Two sites, Yahoo Mobile and BBC Mobile, were selected from the initial wall of links and allowed to download all content and images. To achieve consistency between measurements on the emulator and on the handset, a simple stop watch was used to measure execution times.

	Emulator		**Nokia 3650**	
Version	Load Site 1	Load Site 2	Load Site 1	Load Site 2
Initial MobileEdition	12 seconds	7 seconds	22 seconds	10 seconds
Final MobileEdition	8 seconds	3 seconds	13 seconds	6 seconds

FIGURE 4.7 - Comparison of MobileEdition execution times

The Yahoo Mobile site (Site 1) contained 15 small icons and minimal text content, while the BBC Mobile site (Site 2) contained multiple paragraphs of text and only 3 large images. These sites were chosen because two of the performance issues noted by Rocket

Mobile (2005) included the downloading of multiple images during page loads and the parsing of pages with large amounts of textual content. Thus, these two comparison factors provide evidence that the OO improvements, coupled with the other improvements made to the MIDlet, resulted in a more efficient Java ME application.

CHAPTER 5 CONCLUSION

5.1 Overview

A close examination of the development life cycle of a real-world Java ME MIDlet revealed many surprising effects of object oriented principles on the efficiency of the application. These characteristics, observed extensively during the optimization phase of development for MobileEdition, revealed techniques which can be performed to increase the performance of a Java ME MIDlet. Figure 5.1 provides empirical data supporting the effectiveness of these techniques. Furthermore, these observations revealed Java ME development generalizations which can be used to optimize applications throughout the development process.

Regression Type		**Normal (average delta time)**	**Optimized (average delta time)**
Canonical	Encapsulation	6.8ms	0.11ms
	Inheritance	0.6ms	0.0004ms
	Polymorphism	1.71ms	0.04ms
Algorithmic	Object Pooling	17.79ms	0.001ms

FIGURE 5.1 – Table of Experimentation Results

5.1.1 The OO Paradox

Although the entire Java ME platform is built using the object-oriented paradigm, a MIDlet's performance can be increased by regressing the very properties which make it object-oriented. These properties-- encapsulation, inheritance, and polymorphism-- are core to all Java ME applications. This paradoxical behavior was witnessed during the initial development and optimization stages of MobileEdition and was then duplicated during the testing and research phase of this thesis.

It is important to remember that this behavior is a result of the limitations of handheld devices and not a Java ME implementation flaw. For example, the initial version of MobileEdition performed well enough on desktop emulators to be used as intended, but the application was virtually impossible to utilize on a handheld device. Thus, it is the developer's responsibility to understand the effects of OO principles on a Java ME application.

5.1.2 Optimize a Well-Defined Application

The pre-optimization version of MobileEdition was implemented using proper object-oriented and Java ME platform design principles. Each class implemented data encapsulation standards and the benefits of polymorphism and inheritance were utilized throughout the application. Design patterns such as singletons and mediators were also used throughout the application, which resulted in a clear, concise program structure. The resulting MIDlet was a solid application which easily leant itself to optimizations.

We observed that the application of those optimizations which we termed canonical regressions to a well-formed MIDlet was a trivial exercise. For example, we found that encapsulation optimization techniques are easy to implement when the initial application utilizes proper data hiding. If a class only allows private member access via getters and setters, the conversion of all get() and set() functions to dot notation accesses is a straightforward process.

Our experience with MobileEdition revealed that the maintenance of an optimized Java ME application is best performed on the initial version. After maintenance has been performed, previous optimizations to the code can then be re-applied. The structural regressions mentioned in this thesis often involve collapsing inheritance trees and other significant alterations to the core of the application, which in turn decreases the ease with which a developer can maintain the application. Additionally, by maintaining a baseline Java ME MIDlet which is not optimized, developers are allowing for the possibility of porting the application to another Java platform such as the Java 2 Platform, Standard Edition.

5.2 Future Work

This research can be extended to a number of different research topics and fields, including software engineering, compilers, and artificial intelligence. The dynamic identification of troublesome object-oriented regions of code would take the effects of OO principles presented in this thesis and use them to identify areas which the developer should target for optimizations. A compiler-level approach to optimizing Java ME based on compilation properties of the OO principles could be used to further investigate their

effects on the physical size of a MIDlet. Finally, an artificial intelligence agent could be developed to find a MIDlet's optimum object-orientation; that is, the point at which any more or any less OO traits would result in a lower level of performance than the current level.

BIBLIOGRAPHY

Barowski, L, Cross, J., Jain, J., Meda, N., and Umphress, D. 2004. Bringing J2ME industry practice into the classroom. *Proceedings of the 35th SIGCSE technical symposium on Computer science education.* 301-305.

Chen, H., Kazi, I., Lilja, D., and Stanley B. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys*. 32, 3, 213-240.

Couton, P., Rashid, O., Edwards, R., Thompson, R. 2005. *Computers in Entertainment*. 3, 3, 3.

Dean, J., Grove, D., Chambers, C. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. *Proceedings of the 9th European Conference on Object-Oriented Programming*. 77-101.

Drashansky, T., et al. 1996. Software architecture of ubiquitous scientific computing environments for mobile platforms. Mobile Networks and Applications. Vol. 1, Issue 4. December. pp421-432.

Grove, D. 1995. The impact of interprocedural class analysis on optimization. *Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, 25.

Janecek, A., and Hlavacs, H. 2005. Programming interactive real-time games over WLAN for pocket PCs with J2ME and .NET CF. *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games.* 1-8.

Keshav, S. 2005. Why cell phones will dominate the future internet. *ACM SIGCOMM Computer Communications Review.* 35, 2, 83-86.

Kochnev, D.S., and Terekhov, A.A. 2003. Surviving Java for Mobiles. *Pervasive Computing, IEEE.* 2, 2, April-June 2003, 90-95.

Mikic-Rakic, M., Medvidovic, N. and Jakobac, V. 2002. Middleware for Software Architecture-Based Development in Distributed, Mobile, and Resource-Constrained

Environments." Technical Report USC-CSE-2002-501, Center for Software Engineering, University of Southern California, February 2002.

"jPRESTO," S5 Systems, 2006. http://www.s5systems.com/jPresto.htm.

"J2ME For Wireless Development Surges 33% In Six Months, New Evans Data Survey," *Evans Data Corporation,* 2004. http://www.evansdata.com/n2/pr/releases/04_02Wireless.shtml

Knudsen, Jonathan. 2001. "Performance Tuning." Wireless Java: Developing with Java 2 Micro Edition. p 132.

J2ME Tech Tips: February 26 ,2002. Sun Microsystems. http://java.sun.com/developer/J2METechTips/2002/tt0226.html

Rocket Mobile, personal communication, 2005.

Roman, G., Picco, G., and Murphy, A. 2000. Software engineering for mobility: a roadmap. Proceedings of the Conference on the Future of Software Engineering. (Limerick, Ireland). pp 241-258.

Shivas, Mike. 2005. "J2ME Game Optimization Secrets." http://www.microjava.com/articles/techtalk/optimization,

Shultz, U., Burgaard, K., Christensen, F, and Knudsen, J. Compiling java for low-end embedded systems. 2003. *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems.* 42-50.

"The Java Language: An Overview", Sun Micrososytems. http://java.sun.com/docs/overviews/java/java-overview-1.html.

"The Java Platform, Micro Edition (Java ME)," *Sun Microsystems*, 2006, http://java.sun.com.

APPENDIX A

Canonical Regressions – Polymorphism

	With Polymorphism	Without Polymorphism
1	1.734	0
2	1.671	0
3	1.703	0
4	1.687	0
5	1.657	0
6	1.687	0
7	1.999	0
8	1.703	0
9	1.687	0
10	1.718	0
11	1.734	0
12	1.797	0
13	1.687	0
14	1.703	0
15	1.703	0
16	1.719	0
17	1.687	0
18	1.782	0
19	2.156	0
20	1.843	0
21	2.141	0
22	1.703	0
23	2.047	0
24	1.765	0
25	1.874	0
26	1.718	0
27	1.719	0.016
28	1.688	0
29	1.672	0
30	1.688	0.016

Canonical Regressions - Polymorphism Continued

31	1.656	0
32	1.688	0
33	1.704	0
34	1.719	0
35	1.75	0
36	1.718	0
37	1.719	0
38	1.922	0
39	2.078	0
40	1.766	0
41	2.109	0
42	1.814	0.015
43	1.86	0
44	1.718	0
45	1.781	0
46	1.781	0
47	1.687	0
48	1.703	0.015
49	1.688	0
50	1.719	0
51	1.687	0
52	1.704	0
53	1.766	0
54	1.922	0
55	1.766	0
56	2.063	0
57	1.765	0
58	1.797	0
59	1.797	0
60	1.782	0
61	1.937	0
62	1.875	0.016
63	1.891	0.016
64	1.796	0
65	1.781	0
66	1.719	0
67	1.86	0
68	1.672	0
69	1.719	0
70	1.672	0.016
71	1.687	0
72	1.656	0.015
73	1.688	0
74	1.672	0
75	1.687	0

Canonical Regression -- Polymorphism Continued

76	1.687	0.016
77	1.688	0
78	1.656	0
79	1.688	0
80	1.672	0
81	1.688	0
82	1.673	0
83	1.687	0.015
84	1.672	0
85	1.703	0
86	1.734	0.016
87	1.687	0
88	1.703	0
89	1.687	0
90	1.687	0
91	1.687	0
92	1.703	0
93	1.688	0
94	1.672	0
95	1.64	0
96	1.719	0
97	1.672	0
98	1.687	0
99	1.687	0
100	1.641	0.016

Canonical Regression – Inheritance

	With Inheritance	Without Inheritance
1	0.625	0
2	0.671	0
3	0.593	0
4	0.625	0
5	0.593	0
6	0.609	0
7	0.61	0
8	0.595	0
9	0.594	0
10	0.625	0

Canonical Regression -- Inheritance Continued

11	0.594	0
12	0.593	0
13	0.625	0
14	0.609	0
15	0.625	0
16	0.609	0
17	0.593	0
18	0.625	0
19	0.61	0
20	0.609	0
21	0.594	0
22	0.64	0
23	0.625	0
24	0.594	0
25	0.609	0
26	0.625	0
27	0.625	0
28	0.609	0
29	0.61	0
30	0.61	0
31	0.609	0
32	0.609	0
33	0.609	0
34	0.625	0
35	0.625	0
36	0.609	0
37	0.593	0
38	0.578	0
39	0.625	0
40	0.594	0
41	0.609	0
42	0.61	0
43	0.578	0
44	0.61	0
45	0.625	0
46	0.625	0
47	0.642	0
48	0.625	0
49	0.61	0
50	0.609	0

Canonical Regression -- Inheritance Continued

51	0.625	0
52	0.625	0
53	0.562	0
54	0.609	0
55	0.594	0
56	0.609	0
57	0.625	0.016
58	0.609	0
59	0.625	0
60	0.61	0.016
61	0.609	0
62	0.641	0
63	0.625	0
64	0.609	0
65	0.61	0
66	0.625	0
67	0.625	0
68	0.595	0
69	0.75	0
70	0.641	0
71	0.625	0
72	0.625	0
73	0.609	0
74	0.625	0
75	0.61	0
76	0.609	0
77	0.625	0
78	0.625	0
79	0.625	0
80	0.688	0
81	0.593	0
82	0.593	0
83	0.609	0
84	0.625	0
85	0.61	0
86	0.61	0.016
87	0.609	0
88	0.61	0
89	0.593	0
90	0.594	0

Canonical Regression -- Inheritance Continued

91	0.594	0
92	0.593	0
93	0.609	0
94	0.61	0
95	0.625	0
96	0.61	0
97	0.625	0
98	0.719	0
99	0.657	0
100	0.625	0

Canonical Regression -- Encapsulation Continued

	Normal	Optimized
1	6.484	0.109
2	6.485	0.125
3	6.47	0.109
4	6.484	0.125
5	6.469	0.078
6	6.454	0.079
7	6.516	0.125
8	6.704	0.109
9	6.531	0.109
10	6.531	0.11
11	6.531	0.109
12	6.531	0.079
13	6.891	0.141
14	6.515	0.094
15	6.516	0.109
16	6.578	0.109
17	6.516	0.125
18	6.516	0.094
19	6.499	0.11
20	6.516	0.109
21	6.531	0.125
22	6.5	0.125
23	6.5	0.109
24	6.562	0.125
25	6.688	0.094

Canonical Regression -- Encapsulation

26	6.875	0.094
27	6.609	0.125
28	6.562	0.109
29	7.156	0.125
30	6.562	0.094
31	6.532	0.093
32	6.594	0.093
33	6.546	0.125
34	6.61	0.093
35	6.499	0.125
36	6.563	0.109
37	6.562	0.109
38	6.625	0.094
39	6.984	0.109
40	6.547	0.094
41	6.5	0.125
42	6.5	0.125
43	6.719	0.125
44	6.625	0.125
45	7.641	0.109
46	7.453	0.125
47	7.031	0.125
48	6.501	0.125
49	6.531	0.093
50	6.516	0.094
51	6.547	0.14
52	7.095	0.14
53	6.625	0.109
54	6.532	0.14
55	6.563	0.093
56	6.531	0.109
57	6.532	0.125
58	6.531	0.125
59	6.531	0.095
60	6.532	0.109

Canonical Regression -- Encapsulation

61	6.562	0.109
62	6.782	0.125
63	6.734	0.109
64	6.641	0.093
65	6.876	0.125
66	6.844	0.125
67	6.688	0.094

68	6.672	0.094
69	7.265	0.093
70	7	0.109
71	7.813	0.124
72	7.516	0.094
73	6.688	0.109
74	7	0.11
75	7.328	0.11
76	6.64	0.094
77	6.719	0.109
78	6.61	0.093
79	6.61	0.109
80	6.797	0.11
81	6.718	0.11
82	6.907	0.109
83	6.641	0.094
84	6.609	0.109
85	6.625	0.109
86	6.703	0.125
87	6.953	0.11
88	6.625	0.109
89	6.61	0.095
90	7.015	0.11
91	6.921	0.11
92	6.797	0.094
93	6.719	0.109
94	6.907	0.109
95	6.719	0.109
96	6.875	0.109
97	6.953	0.141
98	6.609	0.094
99	6.703	0.125
100	6.875	0.109

APPENDIX B – Source Code

ResearchMIDlet.java:

```
/**
 *          Purpose:  Driver for various test cases.
*           Fall 2005, Spring 2006, Summer 2006, Fall 2006
 */
package edu.auburn;


import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;


import edu.auburn.data.Profiler;
import edu.auburn.inheritance.NoPolyClass;
import edu.auburn.inheritance.SubClass;
import edu.auburn.optimized.CircleNormal;
import edu.auburn.optimized.EncapsulationNormal;
import edu.auburn.polymorphism.Ball;
import edu.auburn.polymorphism.RedBall;
import edu.auburn.polymorphism.Shape;
import edu.auburn.polymorphism2.BigWidget;
import edu.auburn.polymorphism2.SmallWidget;
```

```
import edu.auburn.polymorphism2.Widget;

import edu.auburn.pool.ArrayPoolManager;

import edu.auburn.pool.PoolTestObject;

import edu.auburn.unoptimized.CircleOptimized;

import edu.auburn.unoptimized.EncapsulationOptimized;


;


/**

 * @author Sean

 *

 */

public class ResearchMIDlet extends MIDlet {

	EncapsulationNormal normalEncapsulation = null;


	EncapsulationOptimized optimizedEncapsulation = null;


	CircleOptimized optimizedCircle = null;


	CircleNormal normalCircle = null;


	PoolTestObject testPoolObject = null;


	ArrayPoolManager poolManager = null;
```

```
Profiler normalProfiler = null;

Profiler optimizedProfiler = null;

private static final int NUM_TEST_CASES = 1000;

/**
 *
 */
public ResearchMIDlet() {
        super();
        // TODO Auto-generated constructor stub
        normalEncapsulation = new EncapsulationNormal();
        optimizedEncapsulation = new EncapsulationOptimized();

}

protected void startApp() throws MIDletStateChangeException {

        // testEncapsulation();

        // testObjectPool();

        // testInheritance();
```

```
        // testPolymorphism();


}

private void widgetWorker(Widget w) {
        w.getDescription();
        w.setDescription("foo");
        w.busyWidgetWork();
}

private void smallWidgetWorker(SmallWidget w) {
        w.getDescription();
        w.setDescription("foo");
        w.busyWidgetWork();
}

private void bigWidgetWorker(BigWidget w) {
        w.getDescription();
        w.setDescription("foo");
        w.busyWidgetWork();
}

private void testInheritance() {
        for (int r = 0; r < 100; r++) {
```

```
normalProfiler = new Profiler(NUM_TEST_CASES);

for (int e = 0; e < NUM_TEST_CASES; e++) {
        normalProfiler.start(System.currentTimeMillis(),
                        "With     Inheritance: ");

        SubClass subClass = new SubClass();
        subClass.function1();
        subClass.function2();
        subClass.function3();
        subClass.function4();
        subClass.function5();
        subClass.function6();
        subClass.function7();
        subClass.function8();
        subClass.function9();
        subClass.function0();

        normalProfiler.end();
}

normalProfiler.endTest();

for (int j = 0; j < NUM_TEST_CASES; j++) {
        optimizedProfiler = new Profiler(NUM_TEST_CASES);
```

```
                    optimizedProfiler.start(System.currentTimeMillis(),
                                    "Without Inheritance: ");

                    NoPolyClass noPolyClass = new NoPolyClass();
                    noPolyClass.function1();
                    noPolyClass.function2();
                    noPolyClass.function3();
                    noPolyClass.function4();
                    noPolyClass.function5();
                    noPolyClass.function6();
                    noPolyClass.function7();
                    noPolyClass.function8();
                    noPolyClass.function9();
                    noPolyClass.function0();

                    optimizedProfiler.end();
                }
                optimizedProfiler.endTest();

        }
}

private void testPolymorphism() {
        for (int r = 0; r < 100; r++) {
```

```
normalProfiler = new Profiler(NUM_TEST_CASES);
optimizedProfiler = new Profiler(NUM_TEST_CASES);

BigWidget bigWidget = new BigWidget();
SmallWidget smallWidget = new SmallWidget();

for (int e = 0; e < NUM_TEST_CASES; e++) {
        normalProfiler.start(System.currentTimeMillis(),
                        "With Polymorphism: ");

        widgetWorker(bigWidget);
        widgetWorker(smallWidget);

        normalProfiler.end();
}

normalProfiler.endTest();
for (int j = 0; j < NUM_TEST_CASES; j++) {
        optimizedProfiler = new Profiler(NUM_TEST_CASES);

        optimizedProfiler.start(System.currentTimeMillis(),
                        "Without Polymorphism: ");

        bigWidgetWorker(bigWidget);
        smallWidgetWorker(smallWidget);
```

```
                optimizedProfiler.end();
            }
            optimizedProfiler.endTest();
        }
    }

    private void shapeTester(Shape s) {
        s.details();
    }

    private void testObjectPool() {
        normalProfiler = new Profiler(NUM_TEST_CASES);
        optimizedProfiler = new Profiler(NUM_TEST_CASES);

        for (int r = 0; r < 100; r++) {
            // create our Pool and initialize it
            poolManager = new ArrayPoolManager(100);
            for (int i = 0; i < 100; i++) {
                testPoolObject = new PoolTestObject();
                poolManager.putObject(testPoolObject);
            }

            for (int e = 0; e < 100; e++) {
                normalProfiler
```

```
                                        .start(System.currentTimeMillis(), "Without Pool");
                        // loop and use new to create an object
                        for (int i = 0; i < 500; i++) {
                                testPoolObject = new PoolTestObject();
                                testPoolObject.doAction();
                        }
                        normalProfiler.end();
                }

                // normalProfiler.dump();
                normalProfiler.endTest();

                for (int e = 0; e < 100; e++) {
                        optimizedProfiler
                                        .start(System.currentTimeMillis(), "With Pool");
                        testPoolObject = (PoolTestObject) poolManager.getObject();
                        testPoolObject.doAction();
                        optimizedProfiler.end();
                }
                optimizedProfiler.endTest();
                System.out.println("--------   END TEST # " + (1 + r));
        }
}

private void endTesting() {
```

```
            try {
                    destroyApp(true);
            } catch (MIDletStateChangeException e) {

            }

}

private void testEncapsulation() {
            for (int c = 0; c < 100; c++) {
                    normalProfiler = new Profiler(NUM_TEST_CASES);
                    optimizedProfiler = new Profiler(NUM_TEST_CASES);

                    for (int a = 0; a < NUM_TEST_CASES; a++) {
                            normalProfiler.start(System.currentTimeMillis(),
                                            "Normal Encapsulation Run#" + (a + 1));
                            normalEncapsulation.setIntNumber(100);
                            normalEncapsulation.setLongNumber(1234.234);
                            normalEncapsulation.setName("Foobar");
                            normalEncapsulation.doOperation();// busy work

                            String s = normalEncapsulation.getName();
                            int jj = normalEncapsulation.getIntNumber();
                            double dd = normalEncapsulation.getLongNumber();
```

```
                normalProfiler.end();
                // normalProfiler.dump();
            }
            normalProfiler.endTest();

            for (int a = 0; a < NUM_TEST_CASES; a++) {
                optimizedProfiler.start(System.currentTimeMillis(),
                                "Optimized Encapsulation Run#" + (a + 1));
                optimizedEncapsulation.intNumber = 100;
                optimizedEncapsulation.longNumber = 1234.234;
                optimizedEncapsulation.name = "Foobar";
                optimizedEncapsulation.doOperation();// busy work

                String s = optimizedEncapsulation.name;
                int jj = optimizedEncapsulation.intNumber;
                double dd = optimizedEncapsulation.longNumber;

                optimizedProfiler.end();
                // optimizedProfiler.dump();
            }
            optimizedProfiler.endTest();
        }
}

/*
```

```
	 * (non-Javadoc)
	 *
	 * @see javax.microedition.midlet.MIDlet#pauseApp()
	 */
	protected void pauseApp() {
		// TODO Auto-generated method stub
		notifyDestroyed();
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see javax.microedition.midlet.MIDlet#destroyApp(boolean)
	 */
	protected void destroyApp(boolean arg0) throws MIDletStateChangeException {
		// TODO Auto-generated method stub

	}

}
```

NoPolyClass.java:

```
package edu.auburn.inheritance;


public class NoPolyClass {


	int x;


	public NoPolyClass() {
		super();
		// TODO Auto-generated constructor stub
	}
	public int function1(){
		return x;
	}
	public int function2(){
		return x;
	}
	public int function3(){
		return x;
	}
	public int function4(){
		return x;
	}
	public int function5(){
		return x;
```

```
	}
	public int function6(){
		int j=0;
		for(int i=0; i<1000; i++){
			i+=j;
		}

		return x;
	}
	public int function7(){
		return x;
	}
	public int function8(){
		return x;
	}
	public int function9(){
		return x;
	}
	public int function0(){
		return x;
	}
}
```

SubClass.java:

```
package edu.auburn.inheritance;


public class SubClass extends SuperClass {


	public SubClass() {

		super();

		// TODO Auto-generated constructor stub

	}


	public int function6(){

		int j=0;

		for(int i=0; i<1000; i++){

			i+=j;

		}

		return x;

	}

}
```

SuperClass.java:

```
package edu.auburn.inheritance;


public class SuperClass {
	int x=0;
	public SuperClass() {
		super();


	}


	public int function1(){
		return x;
	}
	public int function2(){
		return x;
	}
	public int function3(){
		return x;
	}
	public int function4(){
		return x;
	}
	public int function5(){
		return x;
	}
```

```
	public int function6(){
		return x;
	}
	public int function7(){
		return x;
	}
	public int function8(){
		return x;
	}
	public int function9(){
		return x;
	}
	public int function0(){
		return x;
	}

}
```

Circle.java:

```
package edu.auburn.interfaces;


public interface Circle {
	public double getRadius();
	public double getX();
	public double getY();
	public double calcArea();
}
```

CircleNormal.java:

```
package edu.auburn.optimized;


public class CircleNormal extends Point {


	private double radius;


	public CircleNormal(double x, double y, double radius) {
		super(x, y); // Call Point (double x, double y).


		this.radius = radius;
	}


	public double getRadius() {
		return radius;
	}


	public double calcArea() {
		return 3.14 * radius * radius;
	}
}
```

EncapsulationNormal.java:

```
package edu.auburn.optimized;

public class EncapsulationNormal {

	private String name = "";

	private int intNumber = 0;

	private double longNumber = 0.0;

	public EncapsulationNormal() {
		super();

	}

	public void doOperation(){
		for(int i=0;i<10000; i++){
			intNumber = intNumber*1;
		}
	}

	public int getIntNumber() {
		return intNumber;
	}
```

```
	public void setIntNumber(int intNumber) {
		this.intNumber = intNumber;
	}

	public double getLongNumber() {
		return longNumber;
	}

	public void setLongNumber(double longNumber) {
		this.longNumber = longNumber;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

}
```

BigWidget.java:

```
package edu.auburn.polymorphism2;


public class BigWidget implements Widget {
	private String description;


	public BigWidget() {
		super();
		description = "I'm a big widget";
	}


	public String getDescription() {
		return description;
	}


	public void setDescription(String d) {
		description = d;
	}


	public void busyWidgetWork() {
		int k = 0;
		while (k < 3000) {
			k += 1;
		}
```

```
        }
    }
```

SmallWidget.java:

```
package edu.auburn.polymorphism2;

public class SmallWidget implements Widget {
	private String description;

	public SmallWidget() {
		super();
		description = "I'm a small widget";

		// TODO Auto-generated constructor stub
	}

	public String getDescription() {
		return description;
	}

	public void setDescription(String d) {
		description = "[SMALL WIDGET]: " + d;
	}

	public void busyWidgetWork() {
		int i = 0;
		while (i < 3000) {
```

```
                    i += 1;

                }

            }

        }
```

Widget.java:

```
package edu.auburn.polymorphism2;


public interface Widget {


	public String getDescription();


	public void setDescription(String s);


	public void busyWidgetWork();
}
```

ArrayPoolManager.java:

```
package edu.auburn.pool;

//TAKEN FROM http://www.microjava.com/articles/techtalk/recycle?PageNo=2


/**
 * The ArrayPoolManager maintains a set of objects, rather then creating and
 * destroying objects as they are required. The objects can be taken from and
 * returned to the object pool, effectively removing the problem of memory
 * fragmentation.
 * <p>
 * There should be a separate pool for each type of object to be pooled.
 * <p>
 * The ArrayPoolManager is not synchronized.
 */
public class ArrayPoolManager {
	/**
	 * Array to hold objects that are ready for use.
	 */
	private Object[] m_Pool;


	/**
	 * The capacity of the object pool.
	 */
	private int m_Capacity;
```

```
/**
 * The next available object; -1 means that the pool is empty.
 */
private int m_NextAvail = -1;


/**
 * Creates a pool of objects, with an initial capacity.
 */
public ArrayPoolManager(int capacity) {
        m_Capacity = capacity;
        m_Pool = new Object[m_Capacity];
}


/**
 * Gets an object from the pool if one is available; null otherwise.
 *
 * @return An object from the pool if one is available; null otherwise.
 */
public Object getObject() {
        /* Check to see if an object is available to be returned */
        if (m_NextAvail >= 0) {
                // makes sure that the object is released from the array
                // so that only the caller has a reference, in case it is
                // not put back into the pool
                Object t = m_Pool[m_NextAvail];
```

```
                m_Pool[m_NextAvail--] = null;
                return t;
            }

            return null;
        }

        /**
         * Puts an object back into the pool, if there is room for it.
         *
         * @param obj
         *          The object to be put back into the pool.
         */
        public void putObject(Object obj) {
            if (m_NextAvail < m_Capacity) {
                m_Pool[++m_NextAvail] = obj;
            }
        }
}
```

PoolTestObject.java:

```
package edu.auburn.pool;


public class PoolTestObject {


	private int m_int = -1;


	private String m_str = "";


	public PoolTestObject() {

		m_int = 99;

		m_str = "Foobar";

	}


	public void doAction() {

		String result = m_str + Integer.toString(m_int);

	}


}
```

CircleOptimized.java:

```
package edu.auburn.unoptimized;


import edu.auburn.interfaces.Circle;


public class CircleOptimized implements Circle{


	private double x;


	private double y;


	private double radius;


	public CircleOptimized(double x, double y, double radius) {

		this.x = x;

		this.y = y;

		this.radius = radius;

	}


	public double getRadius() {

		return radius;

	}


	public double getX() {

		return x;
```

```
        }

        public double getY() {
                return y;
        }

        public double calcArea() {
                return 3.14 * radius * radius;
        }
}
```

EncapsulationOptimized.java:

```
package edu.auburn.unoptimized;


public class EncapsulationOptimized {
	public String name = "";


	public int intNumber = 0;


	public double longNumber = 0.0;


	public EncapsulationOptimized() {
		super();
		// TODO Auto-generated constructor stub
	}


	public void doOperation() {
		for (int i = 0; i < 100; i++) {
			intNumber = intNumber * 1;
		}
	}
}
```