Thesis Software Engineering
One Year Master Course Software Engineering

'Variability through Aspect Oriented Programming
in J2ME game development'

Sannie Kwakman
Universiteit van Amsterdam
15-08-2006

One Year Master Course Software Engineering
Thesis Supervisor: Jurgen Vinju
Internship Supervisor: Ferdy Blaset
Company or Institute: Gamica
Availability: Confidential

I. Preface

This thesis is part of the graduation project of Sannie Kwakman, currently studying the Master Software Engineering course at the Universiteit van Amsterdam.

The project was done under supervision of Jurgen Vinju of the Universiteit van Amsterdam.

Research described in this thesis was performed by order of Gamica, a mobile game development company based in The Hague, Netherlands. The primary contact and supervisor within this company was its main developer and founder, Ferdy Blaset.

Acknowledgement
During this project, support was received from several persons, who had a positive influence on the quality of the research. Here I like to thank Mr Vinju and Mr Blaset for their unwavering support and cooperation during this project. Additionally, I would like to thank Jeffrey Kamermans, a programmer at Gamica for his assistance during several phases of the research.

II. Summary

Problem description
The subject of this thesis is to find a solution to the problem of dealing with variability in the field of J2ME mobile game development. This variability is required because mobile games are expected to run on a wide variety of mobile devices. Additionally, games are often highly optimized to smoothly run within this strict environment, because of limitations of these devices regarding processing speed, heap memory and disk space. However, introduction of variability often introduces a certain amount of overhead.

In order to introduce variability in mobile games effectively, a variability solution is required that minimizes any introduced overhead and still keeps the targeted game maintainable and it's source code readable.

The research described in this thesis concerns with finding this solution. Not only is a structural and technical solution explored, but it should also fit in the process of developing a mobile game. To obtain more detailed requirements and a basis for a test case, the knowledge and experience of an actual game development company was utilized. The company in question is Gamica, specialized in J2ME mobile game development founded in The Hague, Netherlands.

Additional requirements detailed by Gamica were that the solution should work within or alongside Eclipse. Furthermore, a highly used debugging and code tweaking technique called hot code-replacement should be supported as well.

Aspect Oriented Programming
In finding a viable variability solution, focus was put on an Aspect Oriented Programming (AOP) approach. Although AOP is originally intended for other purposes, it can be utilized to introduce variability in which the targeted application stays oblivious to these changes. The assumption was made that no predetermined variability points may be required, and no code tags or other kinds of code inserts should be needed to implement this variability.

Determining required transformations
By utilizing several informal interviews with Gamica's developers and an existing knowledge base at Gamica, a common set of variations was determined. By applying these variations to existing source code of one of Gamica's games and in-house developed hardware abstraction libray, a list of required transformations were extracted. Lastly, these transformations then resulted into a set of variability operations which the variability solution should support.

Existing implementations and transformation approach
For the implementation of the variability solution several existing Aspect Oriented Programming implementations were assessed. However, these implementations either weren't efficient enough, or provided lacked required features.

These solutions were based on changing bytecode to implement any changes to a program. During the research it was revealed that changing program logic and features at a bytecode level has several limitations regarding the handling of field constants. As these constants are used extensively in Gamica's games and library, this method of program transformation could not be used.

As a result of these findings, the earlier determined variability operations were to be implemented using transformations on a source code level.

Proof-of-concept
A proof-of-concept was developed, which would enable Gamica's developers to apply the earlier determined variability operations on their games and library. This implementation consisted of an Eclipse plug-in which utilizes Eclipse's built-in source transformation features. Furthermore, a language was defined in which developers can target operations to specific parts of game and library source code.

Case study
The proof-of-concept was then used to evaluate the framework's effectiveness, ease-of-use and related source code maintainability. This was done by partially porting a game and library for a significantly limited mobile device. Experiences and opinions of developers responsible for the porting were recorded using informal interviews and a questionnaire.

Results
Although the framework required several additional features to completely implement all required variations, the approach of the framework was declared a success. Developers were able to implement the variations effectively. Furthermore, developers found that code readability was increased when compared with earlier used variability techniques.

Using this framework, Gamica is now able to support a wide range of devices for their games by introducing variability using the framework that was implemented in this thesis.

Reflection

The solutions provided in this research were based on code styles and the development process of one mobile game development company. The solution itself became very dependent on this same code style. This renders the solution not immediately usable for scenarios with different coding practices.

Furthermore, the assumption that AOP can be used to introduce variability whilst keeping the targeted program oblivious to any changes, was proven to be incorrect. Because the variability operations required certain low-level changes to game logic, related game code was required to be isolated in the program code itself. This removes the earlier mentioned obliviousness of the targeted program.

Table of contents

1. Problem Description

The subject of this thesis is to find a solution to the problem of dealing with variability in the field of J2ME mobile game development. This variability is required because mobile games are expected to run on a wide variety of mobile devices. Additionally, games are often highly optimized to smoothly run within this strict environment, because of limitations of these devices regarding processing speed, heap memory and disk space. However, introduction of variability often introduces a certain amount of overhead. As there often are strict limits on how large a game build may become (sometimes smaller than 100 kilobytes), any overhead introduced by a variability framework should be kept at a minimum.

In other cases, introduction of variability decreases maintainability and source code readability. Examples of this are listed in section *'Current solutions to these challenges'.*

Finding an efficient and maintainable variability solution
In order to introduce variability in mobile games effectively, a variability solution is required that minimizes any introduced overhead and still keeps the targeted game maintainable and it's source code readable.

The research described in this thesis concerns with finding this solution. Not only will a structural and technical solution be explored, but it should also fit in the process of developing a mobile game. To obtain requirements and a basis for a test case, the knowledge and experience of an actual game development company will be utilized. The company in question is Gamica, specialized in J2ME mobile game development founded in The Hague, Netherlands.

1.1. Challenges of mobile game development

Mobile game development is a challenging field. Typical issues are limited hardware and a wide variety of mobile device specifications requiring an extreme amount of variability.

In addition to hardware related variability, mobile games are often asked to support a wide range of languages as well. Furthermore, game distribution channels sometimes enforce certain requirements as well.

More details and several examples of these challenges are listed in the following chapters. The last section titled *'Implications for mobile game development'* provides a summary of these challenges and their implications to the development of mobile games.

1.1.1. Limited hardware

Although a lot of progress has been made in increasing the capabilities and features of mobile devices, they still impose a severe limiting factor when creating games for them. Some of the more restricting factors are:

- Limited amount of available memory
  Depending on the mobile device, games sometimes don't have more than a few kilobytes of heap memory available for use.
- Limited processor capacity
  To preserve battery power and their small form factor, processing power of mobile devices are relatively low (compared to desktop computers)
- Limited file size
  Most mobile devices impose a maximum file size for game packages. Some game publishers also impose such a restriction in their contracts.

1.1.2. Varying models

There are large numbers of different mobile devices in the market, which all have different form factors, hardware specifications and other differences. These include:

- Varying processing power
- Varying memory and storage capacities
- Varying device-specific capabilities
  Capabilities like camera's, localization units (like GPS), gyroscopes etc.
- Varying screen sizes

- Varying input methods
- Device-specific specific bugs
- Differences between supported Java profiles
  Java profiles are discussed in more detail in chapter '*Current solutions to these challenges*'.

1.1.3. Languages

Apart from device variations, mobile games also require the support for multiple language sets. A language set contains a group of languages targeted at a specific region. For example, a game targeted for western Europe could contain English, Dutch, French, German and Spanish language texts. While an Asian build should support various forms of Chinese, Taiwanese, Japanese etc.

1.1.4. Distribution channel specific requirements

As a bonus, some game publishers have certain requirements as well. For instance, a publisher could require that its logo must be displayed at the beginning of the game.

1.1.5. Implications for mobile game development

The challenges mentioned in the previous chapters are all related to supporting variations in hardware, channel distributor requirements and languages.

Regarding languages, theoretically it is possible to include support for all possible languages in one game build. However, as storage and memory capacity is limited, game developers are forced to create separate game builds for each language set.

All these variations lead to a large number of separate builds per game. For example, when a game should support 10 language sets, 10 devices for 3 different publishers, 300 different builds of one single game are required.

Supporting manageable variability
The key factor here is the support of variability. Because of all these differences, a game should be developed to support the mentioned variations. All without imposing extra restrictions on the already limited storage, memory and processing capacities.

Additionally, the variability should also be manageable. As will be described in chapter '*Current solutions to these challenges'*, certain variability solutions can decrease the readability and maintainability of source code. This in turn reduces the effectiveness of such a solution.

Development process requirements
Furthermore, if any solution regarding the introduction of variability is to be used in a real-world environment, it should fit inside the process of developing mobile games. In this research, these requirements are directly related to Gamica's own development process. These requirements are detailed in chapter '*A new aspect oriented approach'*

Current solutions
The next chapter describes several current solutions to the mentioned challenges. Thereby focusing on solutions that Gamica already utilizes to (partly) solve issues related to the challenges.

1.2. Current solutions to these challenges

1.2.1. Abstracting hardware differences through Java profiles

In an attempt to ease the creation of applications and games for a mobile environment, Sun Microsystems [32] provides a Java [4] version specifically targeted for mobile devices. The Java 2 Mobile Edition (or J2ME) [22] provides a common ground which enables developers to focus on the actual game instead of the varying hardware platforms. Although this doesn't eliminate the previously mentioned challenges, it does prevent developers having to manage each and every varying hardware interface.

To manage the variations between device capabilities, J2ME can be used in conjunction with a profile. Such a profile provides a common interface to devices with similar capabilities. In theory, this means that developers can develop for a certain profile without having to worry about the devices themselves. Examples of these profiles are MIDP 1.0 [6], MIDP 2.0 [7] and DOJA [8].

Issues
Unfortunately, because of the earlier mentioned device specific bugs and capabilities, an absolute usage of these profiles is not possible. This means that for each game, a separate build must be made for each device. Furthermore, when a game is to be ported from a MIDP 2.0 to a MIDP 1.0 device, changes are required to the game as well.

Abstracting profile from game
To fix this issue, Gamica has developed a custom built library in which all profile specific calls are abstracted from the game. This library, called FALCON, also provides several convenience methods and other functions in order to separate device specific issues from the game code itself.

This means that with every game Gamica releases, a version of the library specifically built for a targeted is included.

1.2.2. Introducing variability through standard Object Oriented structures

An object oriented programming language such as Java offers a set of features which can be used to introduce variability in an application. Through separation of functionality in classes and methods, super classes, interfaces and abstract classes, certain variable areas (or variability points) can be defined.

However, in an environment where every byte counts, the creation of extra interface and abstract classes becomes too costly regarding total game size. Therefore, other methods are required that introduce variability in a more size-efficient way.

Gamica uses several techniques to achieve efficient variability in code. However, these techniques have certain drawbacks. These techniques are described in the following chapters.

1.2.3. Introducing variability through manual code changes

One simple method to introduce variations in source code, is to manually edit source code for every required game build. Listed below are several advantages and disadvantages when using this method.

Advantages

- No extra variability overhead: high efficiency in memory, storage and cpu usage of game
- Large amount of freedom regarding changes that can be made in the source code

Disadvantages

- Very inefficient in terms of time and effort
- High amount of build management required
- Higher chance of introducing bugs

Manual editing each build costs a lot of time, especially in a situation where lots of different builds are required. Also, when a bug has been found in a generic piece of game code, all the other builds need to be adjusted as well.

Within Gamica, manually editing source code is currently done only in a small number of cases. The negative aspects of this method makes it unusable when dealing with a high number of builds.

1.2.4. Introducing variability through preprocessing

As was mentioned in section *'Abstracting hardware variations through Java profiles'*, Gamica uses an in-house developed hardware abstraction library called FALCON that handles most of the device-specific variations. Per device, a specific version of the library is built and used when developing games. To accommodate all the variations in devices, the library code contains certain preprocessing directives (partly formatted using XML/XSL data) that define which pieces of code are compiled and which omitted from compilation.

The actual insertion of code related to these directives depends on the currently selected device and certain properties of this device. These properties are stored in a device capability database, maintained by Gamica.

Advantages

- Variations can be defined on every location within the code.
- No added overhead to support variability.
- Adjustments can be made according to tag values, which can be linked with a central device capability repository.
- Existing variability can easily be reused for new mobile device releases, because of the device capability database.
-

Disadvantages

- Decreased code readability. Tags are placed everywhere, lessening the ability to read which code is executed and which is omitted from compilation.
- Development environment doesn't understand the preprocessing directives
  Gamica uses Eclipse as a development environment. When using this technique, most of Eclipse's features (like hot code replacement, debugging, code highlighting etc.) are disabled because of the non-Java nature of the tag descriptions.

This tag based preprocessing technique eliminates most variability problems, without adding any overhead. However, it does increasingly reduce the readability and maintainability of source code. Because of all the code inserts and surrounded directives, it becomes increasingly difficult to add new functionality to existing code. Furthermore, it is hard to see which code is actually executed.

The example below illustrates how complex source code can become when using this method extensively.

```
//#ifdef DOJA
public final void paint (com.nttdocomo.ui.Graphics graphics)
//#elifdef JAL
public final void update (java.awt.Graphics g)
{
          paint (g);

          synchronized (m_paintLock)
          {
                    m_maintLock.notify();
          }
}

public final void paint (java.awt.Graphics graphics)
//#elifdef MIDP
          //#ifdef MIDP_2_0
          public final void specialPaint (javax.microedition.lcdui.Graphics graphics)
          //#else
          public final void paint (javax.microedition.lcdui.Graphics graphics)
          //#endif
//#else
!ERROR!
//#endif
{
```

Example of source code utilizing preprocessing

These problems with preprocessing are similar to preprocessing techniques used in combination with C. More details about these problems in the context of the C programming language are provided in [23].

1.2.5. Resolving efficiency and maintainability issues

Because the earlier mentioned techniques had issues with either efficiency and/or maintainability, a new approach is being researched. This approach is described in the next section: '*A new, aspect oriented approach'*.

1.3. A new, aspect oriented approach

Gamica is interested in a fairly recent technique called Aspect Oriented Programming (AOP) [26].

1.3.1. What is Aspect Oriented Programming

The original concept of Aspect Oriented Programming (or AOP)[26] is to provide a solution for so-called *cross cutting concerns.* These kinds of concerns are certain responsibilities of application logic that creates repeating code on multiple locations within an application's source code structure. A typical example of this is logging functionality. When a log file is being maintained in which several application activities are stored, it usually means that the writing to this log file is described in several different locations within an application.

Although conventional object oriented structures are capable of solving this issue in a number of cases, adjusting the logging features commonly requires adding and altering source code at various parts of an application. Aspect Oriented Programming provides a method to define (or describe calls to) such a feature on one location, after which this code can be placed at multiple locations automatically.

More information about AOP can be found in chapter *'Background and context'*, section *'Aspect Oriented Programming'*.

1.3.2. Using Aspect Oriented Programming to introduce variability

Apart from it's original intent of solving *cross cutting concerns*, AOP can also be used to change source code for the purpose of introducing variability.

Other techniques (like traditional object oriented structures) require that source code is specifically structured to certain variability points. When applying AOP it might be possible to create variability points without changing the source code beforehand.

AOP utilizes a 'descriptive' method of defining variations, in which the targeted source code is oblivious to any changes that may be done to it. Because of this obliviousness it becomes possible to introduce variations virtually anywhere within the source code, without the need of creating special variability points beforehand.

Possibility of efficiency
As was mentioned before, traditional object oriented techniques usually introduce some overhead to create a variability points. With AOP, any place in the source code can be altered to serve as a variability point, possibly without adding any additional overhead.

Because of the descriptive nature of defining AOP operations, Gamica wants to know if this method can be utilized to introduce variability within its games and library.

1.3.3. Requirements

In order for AOP to be successfully used by Gamica to introduce variability, the technique should adhere to the following requirements:

- Minimal overhead
  As there are strict limits on how large a game build may become (sometimes smaller than 100 kilobytes), any overhead introduced by a variability framework should be kept at a minimum.
- Should fit within Gamica's development process
  Gamica uses the Eclipse [16] integrated development environment, of which several features are heavily used by developers. Gamica perceives some of these features as an important factor in terms of developer efficiency, and would not like to give them up. Some of the more important features are:
  - Hot code replacement
    When using this feature, developers can change various pieces of source code while the code is being executed. Certain changes can therefore be directly viewed and debugged without restarting the application. This feature is most commonly used to tweak coordinates of various graphical elements within a game.
  - Debugging
    Furthermore, the ability to debug a piece of game or library source code should still be available when using the variability solution.

  Although it isn't required that the solution itself works from within the Eclipse environment, it is preferred. The solution should at least work nicely alongside Eclipse, without interfering with the development process when using Eclipse.
- Support all required variability operations in order to change functionality of both library and games.
- Provide an improvement regarding readability of source code over the previously used preprocessing methods within Gamica

1.4. Research Questions

In short, the research question which will be answered in this thesis is formulated as: '**Can Aspect Oriented Programming be applied to introduce variability in J2ME games?'**

To answer this question, the following subquestions are to be answered as well:

1. 'Which exact variations should be supported by the AOP solution?'
2. 'Can required variations be implemented using AOP and how?'
3. 'Is there an efficient enough AOP implementation, or can one be created?'
4. 'Can the AOP solution be used within Gamica's development process, or what changes are required to this process to make it possible?'

2. Background and Context

In this chapter, several concepts and techniques are described which are frequently mentioned in this thesis.

2.1. Aspect Oriented Programming

The concept
The original concept of Aspect Oriented Programming (or AOP)[26] is to provide a solution for so-called *cross cutting concerns.* These kinds of concerns are certain responsibilities of application logic that creates repeating code on multiple locations within an application's source code structure. A common example of this is logging functionality. When a log file is being maintained in which several application activities are stored, it usually means that the writing to this log file is described in several different locations within an application.

Conventional object oriented structures are capable of solving this issue in a number of cases. However, adjusting the logging features commonly requires adding and altering source code at various parts of an application. Aspect Oriented Programming provides a method to define (or describe calls to) code for such a feature on one location, after which this code can be placed at multiple locations automatically.

Common elements within AOP
Aspect Oriented Programming instructions are defined in so-called *aspects*, a file in which code and the target location where the code should be placed is described.

A typical aspect contains the following elements:

- Special well-defined points in the program flow (where alterations can be made) are described through *joinpoints.* Examples of *joinpoints* are method calls, field assignments, exception execution etc.
- *Pointcuts* specify which *joinpoints* are relevant for a certain alteration.
- One or more *advices*, in which the actual inserted/changed code is described. These advices are applied when a certain *pointcut* is reached within the program flow.

When the targeted application is executed or compiled, the created logging advice will be *woven* onto the location(s) defined in the pointcuts.

Common elements within implementations of AOP
The actual descriptions of aspects are somewhat different per implementation. The definition described in this chapter are similar to AspectJ [1], one of the more popular AOP implementations currently available. More information about this description can be found in references [1], [27] and [28]. An alternative AOP implementation like JbossAOP, uses similar terms and descriptions but manages the locations of these descriptions in a different way.

Both implementations utilize bytecode instrumentation to apply variations to an application. More information about this technique is described in the next section *'Bytecode instrumentation'*.

2.2. Bytecode Instrumentation

Overview
Java source files are compiled to bytecode. This bytecode contains generic instructions which can be translated to platform-specific instructions using a Java Runtime Environment [30]. Bytecode instrumentation [29] is the act of changing a Java program's logic through the alteration of its bytecode.

Alteration of bytecode is mainly done through the use of specific bytecode instrumentation libraries. Two well known libraries for this purpose are Javassist [10] and BCEL [11].

Purposes
Changing Java source code and bytecode are two very different concepts and have different purposes. Source code is mainly meant as a human readable interface for developers to develop applications. While bytecode is mostly read by other programs and is therefore a lot less readable for regular humans.

Bytecode instrumentation is mainly done to make relatively small changes to a program after compilation. Possible reasons for this can be because the related source code isn't available. Or the source code is not supposed to be altered while changes to a program are still required.
Several Aspect Oriented Programming implementations utilize bytecode instrumentation in order to implement changes to a program defined by aspects.

For more information about the internal structure of Java bytecode, see reference [31] and [9]. A detailed knowledge of internal bytecode structure however, is not required for reading this thesis.

2.3. The Eclipse development environment and platform

Overview
Directly quoting from its website [16]:

“Eclipse is an open source community whose projects are focused on providing an extensible development platform and application frameworks for building software.”.

The main product of this community is the Eclipse platform. This platform is mainly used as a development environment for developing Java applications. But due to its extensibility and flexibility it can be used for other languages as well.

Within Gamica's development process, the Eclipse environment is used to develop J2ME games and Gamica's FALCON library.

Projects
Software development in Eclipse is done in so-called projects. Each project contains several types of resources required for a developed application (or game) to build and run. Resources can be files containing source code, images, libraries, xml data files etc. These resources can be ordered through the use of folders. Whereas source code files which are actively edited and compiled are placed in 'source folders', other non-source files are placed in 'regular folders'.

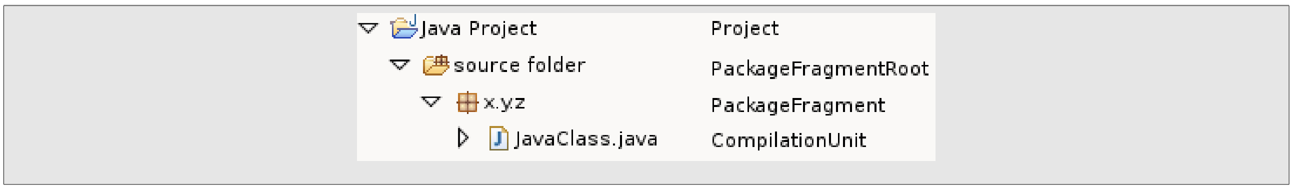
Project natures
Certain languages require language-specific elements within projects. For instance, elements related to Java include packages, .jar libraries and a Java compiler. To support these elements, certain 'project natures' can be linked to a project through the plug-in system. A project nature can define many things, including how a project's builder queue should be set up (which defines which compilers are used, and in what order), how the project's basic structure should look like, which files are required and which project settings should be set.

Eclipse Java Development Tools and the Abstract Syntax Tree
The mentioned 'project natures' and 'project builders' utilize Eclipse's extension points. Eclipse provides several extension points for which plug-ins can be written that provide additional functionality at these points.

One example of a set of plug-ins that extend Eclipse's functionality, is Eclipse's own JDT (Java Development Tools) [20]. The JDT contains a set of Eclipse plug-ins which make it possible to use Java specific tools for debugging, code analyzing, code completion, automated code refactoring, error detection etcetera.

Another feature provided by the JDT, is the capability of converting a Java source file to an Abstract Syntax Tree and back again. As explained in [17] and [18], an Abstract Syntax Tree (or AST) is a abstract representation of source code. Similar to the Document Object Model [19], an AST provides a tree of elements, which in the case of Abstract Source Trees are specifically used to analyze source code structures. It is also possible to make changes to this structure and convert an AST back to regular source code. This feature enables changing source code programmatically, without having to deal with syntax specific issues such as placing semicolons behind statements, putting curly braces around method bodies etc.

From within Eclipse, ASTs can be generated from so-called compilation units. A compilation unit is an element within an Eclipse project which can and should be compiled by a compiler. These compilation units can be extracted from packages, which in turn reside inside a source folder.



Project structure

Usage of the Abstract Syntax Tree
Currently, the AST is used for a number of Eclipse's internal tools. This includes certain refactor tools in which certain code changes triggers an event that updates every reference to these changes as well. For example, when a developer changes the name of a certain class, all references to this class can be updated through the refactor tools. These tools utilize the AST to implement the refactoring.

3. Research Plan

In order to answer the questions described in the problem description, the research will be done in several phases:

1. **Gather detailed requirements**
   This phase focuses on summarizing a set of variation requirements and process requirements of the variability framework. In this phase, it is determined which kind of variations should be supported by the framework and how those variations should be organized. This will result in a common variability model, in which all game and library variations (in the form of variability operations) can be placed.
   In this phase, answers will be found to the subquestion '*'Which exact variations should be supported by the AOP solution?'.*

   This phase is divided into the following steps:

   1.1 Determine typical variations
   Using informal interviews with developers and variations encountered in previous projects, several typical variations will be determined. For example, a typical variation could be 'No audiosupport', which deals with situations where a mobile device doesn't support the playback of audio samples and music.

   1.2 Determine transformation operations
   Using the variations determined in the previous step, a number of program transformation operations are distilled. These operations describe how and where both game and library builds should be changed in order to apply the variation.

2 **Assessment current implementations**
   An assessment was performed to determine if existing Aspect Oriented Programming implementations are usable within the field of J2ME game development. Focus on this assessment was on the introduction of overhead regarding bytecode size and supported functionality. The results of this evaluation determined if the variability solution will utilize an existing AOP solution or a custom made solution should be developed.

   During this phase, the question '*Is AOP efficient enough'* from the problem description is answered, regarding efficiency of current AOP implementations.

3 **Proof-of-concept**
   Using the requirements assembled in the previous steps, a technical implementation is constructed. This implementation will serve as a testbed in order to determine if the requirements are technically feasible and if the suggested solution actually works in a real world environment.
   In this phase, answers will be found for subquestions '*Can these variations be implemented using AOP and how?*' and '*Can the AOP solution be used within Gamica's development process'*

   This phase is divided into the following steps:

   3.1 Determine implementation of variability operations
   In this step, it is determined how the operations are executed. In most of the current AOP frameworks, these operations are done on a bytecode level, after compilation. But it is also possible to apply the required operations on a source code level. Both approaches will be evaluated on feasibility, taking into account the positive and negative side effects of each approach. Whether or not the observed side effects are significant or negligible is determined by the organization's demands and wishes. The result in this step is a blueprint of techniques and methods which will be used to implement the framework.

   3.2 Implement the framework
   In this step the proposed framework is constructed, solving various issues surrounding project structuring, designing a operation declaration language etc.

4 **Test and evaluate the framework**
   To test the framework's compliance with the requirements, a case study will be done in which several variability operations will be developed using this framework. These operations will then be applied to one or more game projects that are completed or currently in development within the organization.

   During and after this test case, informal interviews and a survey are performed. Questions asked in these surround subjects such as usability, readability and maintainability regarding the framework.

4. Gathering detailed requirements

Activities described in this chapter will attempt to answer the following subquestion from the problem description: *'Which exact variations should be supported by the AOP solution?'*.

First, certain variations that are common within Gamica's mobile development process are determined. The results of this analysis is then used to form a set of common variability operations which should be supported by the variability solution.

4.1. Determining typical variations

To determine a set of common variations which must be supported by the variability solution, one of the company's games currently in development was analyzed. The game in question is called *Battleships*, which combines a slotmachine with the battleship boardgame. When completed, *Battleships* will be made available to many different mobile handsets. Covering a wide range of models which differ greatly in terms of hardware capabilities, available amount of memory, screen resolutions and other variations. These characteristics made *Battleships* a suitable target for analysis of this subject.

4.1.1. Sources of information

In order to determine the mentioned set of variations, several sources of information were analyzed.

Common variations between devices
Utilizing official specifications and informal interviews with the development team, a cross-section was made of handsets that are to be be supported by *Battleships.* Focus in this cross-section was put on the most different mobile handsets within that range.

Because of time constraints and that some of the listed criteria below aren't described in phone specifications, the cross-section was limited to the phones which were known by the development team. This lead to the selection of a set of four handsets. A matrix containing the variations between these handsets can be found in A*ppendix A*.

Key differences between between these handsets, such as screen resolution or maximum game file sizes, were recorded.

Existing knowledge bases
Additionally, the existing knowledge base of Gamica was utilized was well. As was mentioned in the problem description, Gamica maintains a game library in which a large number of variations are already implemented. These variations, combined with existing developer experiences gathered through informal interviews were used as well to create the variation set.

The resulting variation set can then used to determine which variations are to be analyzed for determining required variability operations.

4.1.2. Results

Most of the gathered variations were related to device-specific differences. However, the informal interviews revealed that several other, non-device specific variations were to be accounted for as well. These two types of variations are listed below.

Device-specific variations

- GraphicsLocation
  Supporting different resolutions means different locations for graphics per build. Some resolutions can be grouped together. For instance, games on phones with resolutions that are slightly larger than the reference resolution, could compensate with the difference by drawing a black border around the game's graphics display.
- MidiPlayback
  The development team has learned that the method of playing MIDI files (and in some cases other audio files as well) differs between mobile phone models. And the required behavior to restart MIDI playback after a pause event varies as well.
- ResourceLoading

The required calls to load resources (images, audio etc.) varies between Java profiles. Model-specific bugs also influence the required operations.

- ImageFormat
  Because of the difference in supported image formats, different kinds of files need to be supplied with certain builds.
- DisabledConnectivity
  Especially the 1st generation of the Nokia series 40 mobile phones cause problems regarding making connections to the Internet. Protocol errors and memory leaks are the most common problems in this area. This means that in some cases the developers would rather not support online features than risking a game crash.
- JavaProfile
  Variations between supported Java profiles (like MIDP 1.0, 2.0 and DOJA) require different kinds of method calls and object classes.
- SimultaneousAudioSupport
  The support of playing back multiple audio streams at once (2 MIDI files, MIDI combined with WAV/MP3) varies between mobile phone models.
- AudioFormatSupport
  As seen in the capability matrix, support for audioformats varies between mobile phone models.
- AnimationImplementation
  Most MIDP phones can implement animation using a 'filmstrip', a sequence of animation frames contained within one graphics file. But some other phones (mainly those which support DOJA 1.5) aren't able to 'cut up' the individual frames from a film strip. These models have to resort to loading a graphics file for each frame, which is a relatively slower operation.
- PauseEventHandling
  A 'pause event' is an externally called event which requests (or demands) that the game should set itself to a paused state and try to minimize consumed resources. Examples of pause events are incoming phonecalls, low battery warnings or any other warnings issued by the phone's operating system.

Other, non-device specific variations
Apart from device specific variations, there are also issues surrounding language support and distribution channel requirements. The former could require a variable language implementation, combined with an optional language menu. The latter depends on what requirements a distribution channel demands from a game. Most of the time the requirements are limited to displaying an extra image (the logo of the distributor for example) when starting/loading/playing the game. Some distributors however, uphold certain usability guidelines. One example of such a guideline is that the right softkey is always used to go back to a previous (options) menu or deleting a character from a text input field.

For this research, the most common variation regarding the distributor requirements was chosen.

All these considerations resulted into the following additional variations:

- Language
  Certain game builds, localized for a specific region, could support one or more languages.
- LanguageMenu
  When more than one language is supported by a game build, display a language menu.
- DistributorImage

Complete results
The compiled list of variations is by no means a complete listing of all possible variations that could be required when porting a game. Instead it is a list of most common variations, which applies for most mobile game ports. It is assumed that the manipulations that are required to support these variations represents a fairly complete set that will also support any future variations that aren't listed.

This assumption is however difficult to be proven correct. On the other hand, one could state that it isn't possible to guarantee that such a list is 100% complete. By supporting the most common variations, the resulting variability framework should be applicable for most games. Further verification of this assumption will be done in the '*Proof-of-concept*' stage of the research, described in the chapter '*Proof-of-concept*'.

The next step
In the next step the list of variations will be used to determine which manipulation operations will be required by the variability framework.

4.2. Determine transformations for variability

Using the list of variations compiled in the previous step, each variation has been evaluated to determine which transformations are required to implement them. This analysis was performed for both Gamica's in-house developed library and the *Battleships* game. This was done by analyzing the source code of the library and the game, describing which pieces of code needed to be added, removed or replaced. These manipulations were then translated into operations which should be supported by the variability framework. How these operations were to be implemented depended on the chosen implementation technique. This was determined in the next step, which is described in chapter *'Proof-of-concept'*.

4.2.1. Variability strategy

Variability can be applied in several stages of development. When variability is applied after development, variability operations are required to scrape off any unsupported features. However, variability can also be designed before actual development of games. In this scenario all features could be structured in various modules which are combined in variations forms for each game build.

This variability strategy has an impact on what kind of variability operations are required. To define which strategy should be chosen, a choice ranging between two extremes were considered. These extremes were:

- Variability by removal (apply variability 'after the fact')
  A game could contain all possible features whereas the variability framework only removed the features that could not be supported for a certain device.
- Variability by composition (apply variability beforehand)
  A game is comprised of several pluggable variability operations which are put together in a generic game architecture. Every operation deals with a certain feature and/or representation of this feature, and can be enabled or disabled depending on a device profile.

Strengths and weaknesses of both strategies are listed below.

Strengths of 'variability by removal'

- Clear separation of concerns between source code and variability operations. Code stays in the original source, only change operations are defined in the variability operations.
- 'Normal' development can be done in the initial stages, whereas the variability operations are defined and applied after a successful development cycle. This speeds up release of the first few builds.

Weaknesses of 'variability by removal'

- Variability is more of an afterthought. The process doesn't force variability-friendly code, and it can be expected that certain changes must be made to the code to make it possible to be adjusted later on.
- Problems with conflicting functionality. When all kinds of functionality are to be applied in a single implementation, conflicts can arise between certain functions, which makes this method less useful and applicable.

Strengths of 'variability by composition'

- Variability is directly implemented for a project, which makes code immediately variability-friendly and faster to port.
- Possible re-use of game and library code, as code is strongly modularized.

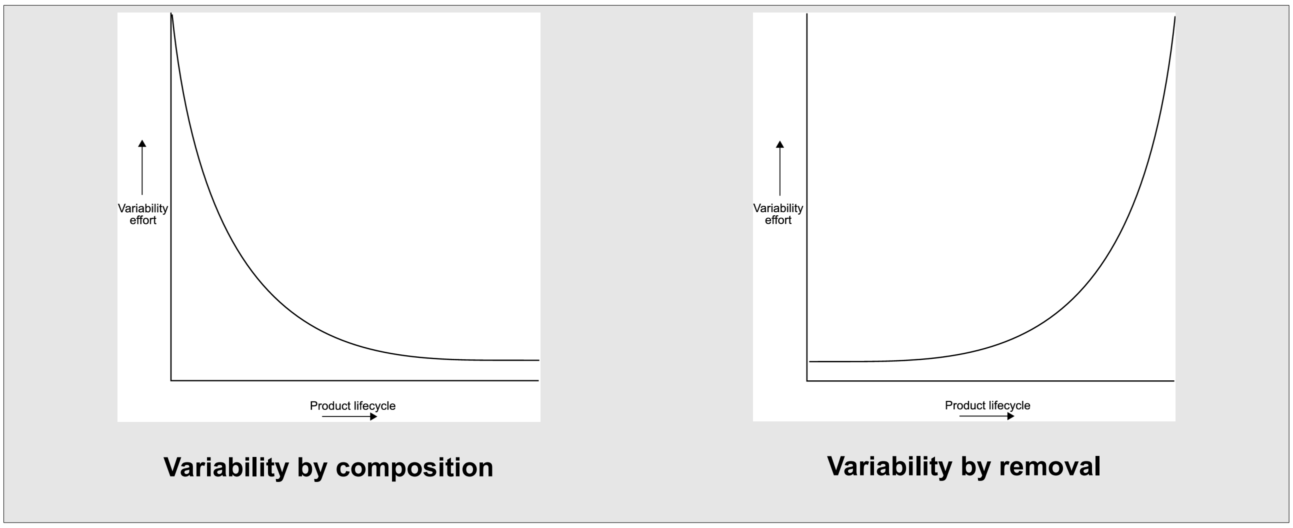
Weaknesses of 'variability by composition'

- Game and library code spread out over variability operations.
- Development of first build of game takes more time. Careful variability operation structure design is needed in order to keep the game maintainable.
- Because of the pluggable structure, more overhead can be expected than the 'variability by removal' model.

Strategies and required variability within product lifecycle
When analyzing the mentioned extremes in terms of required effort to implement variability using these strategies, a curve exists related to amount of effort required and the time within the product lifecycle.

For the 'variability by composition' strategy, the required effort to implement the variability peaks at the beginning of development. This is because a skeleton of variability points and modules must carefully be constructed and designed before any actual game code is built.

The 'variability by removal' strategy requires more effort later on, as completed code needs to be carefully removed without breaking any other features. When this strategy is used more, it becomes more difficult to remove any other features without breaking something.
These relations between variability effort and the product lifecycle are displayed in the graphs below.



Variability curve of different strategies

Choosing between extremes
Because of each strategy's weaknesses, both of these strategy extremes aren't applicable for the project. A pure removal strategy will create conflicts between features that are mutually exclusive. Whereas a pure composition strategy should handle an impossible amount of modularity in order to guarantee support for all current and future variability operations. Furthermore, a composition strategy has a high probability of increased overhead (in terms of memory, cpu and storage requirements) because of its modularity.

The program transformation strategy for the variability framework should therefore be somewhere in the middle of these extremes. To find this middle ground, it's important to know how the game development process within Gamica looks like in its starting phase.

Finding a middle ground through development process requirements
Game development at Gamica is usually done by order of a game publisher. When Gamica starts developing a game for such a publisher contracts usually state that in the first run, the game should be compatible for limited number of devices. Development is therefore initially focused on making the game work on those devices. Porting issues will be dealt with after the first batch of working game builds are delivered.

In light of this development process, it is required that the first build of a game is finalized quickly in order to adhere to the terms of the contract. This means that the chosen strategy should not slow down the development of a initial build of a game.

When comparing this conclusion with the earlier described variability/lifecycle analysis, the ''variability by removal' strategy is preferred when it comes to game development. However, because of the problems with functionality conflicts weakness of the removal option, a pure version of this strategy is not realistic. To remedy this, a more agile strategy has been chosen, in which operations both add and remove code from a source base.

An add/remove strategy
In this add/remove strategy, game code consists of a set of functionality usable for a certain build, whereas variability operations will change its code to be compatible with other devices.
Advantages of this strategy are:

- First source code version is relatively fast to create, as little variability-specific structuring is enforced (or required) beforehand

- Conflicting features can be placed in separate modules

This strategy however, has also its weaknesses:

- Separation of concern becomes an issue: where will the code be placed? When is removal or addition of code preferred?
- Variability is more of an afterthought. The process doesn't force variability-friendly code, and it can be expected that certain changes must be made to the code to make it possible to be adjusted later on.

In the listed weaknesses of this strategy several questions are raised. Answers about where actual inserted code will be placed, is answered at the implementation stage. More details about this can be found in chapter *'Proof-of-concept'.*

To determine when removal or addition of code is preferred, certain selection criteria of manipulations are defined. These are discussed in the next section *'Selection criteria'*.

4.2.2. Selection criteria

The execution of a certain program transformation can be done in several ways. When selecting a method of transformation (hereby called, a 'transformation operation'), the choice would depend on the following criteria:

- Efficiency
  The transformations should introduce a minimum amount of introduced variability overhead, regarding an increased size of compiled code (related to total game size).
- Utilize 'descriptive manipulation'
  Descriptive manipulations change code without any pre-insertions inside the original source code. This way, the original source code can stay oblivious to any future variability changes. When determining the manipulations this descriptive method is, when possible, preferred.
- Number of required changes to code and code structure
  Some manipulations could require prior changes to related source code and its structure before they are applied. When these changes are significant and spread out through the game's code, they could introduce new bugs in otherwise stable code. Complex and wide ranging manipulations could also conflict with other manipulations. Therefore, the manipulation which requires the least amount of changes to the original source code, and/or upholds most of the source code's original structure has the highest preference.

As said, the manipulations and the resulting variability operations are done through analysis of source code and required changes to this code in order to implement the variations. The resulting set of operations will be used to evaluate if a certain technical implementation technique supports the required variations.

Because these operations are based on source-code manipulations, it is very possible that techniques which rely on bytecode manipulations work better with operations which are implemented differently. The resulting operations from this analysis will therefore be mainly used as a guideline for the technical evaluations. If a certain operation doesn't work well with a certain technique, an alternative operation will be determined whilst still holding to the requirements and issues surrounding the original operation.

4.2.3. Analysis of variation points

On the next pages are the aforementioned variations and the required transformation operations in order to implement the variation. Per variation a number of elements are listed:

- Previous method
  The method previously used in order to support the variation, and the problems that have been observed using this method
- Required source code transformations
- Proposed improvements
- Proposed transformation operations.
  The ultimate goal is to create a limited set of transformation operations, in which all variations can be implemented.

The results of this analysis are two sets of data, which are displayed in tables in chapter *'Summary of* *required operations'*. The first table maps variations and required operations to implement these variations. A second table provides a summary of all required transformation operations including names, preconditions, postconditions and required parameters of each operation.

Variation A GraphicsLocation

Previous method

Locations of various graphics were previously done by generating a so-called ProjectStub java source file, using a custom made ANT [5] buildscript. In this buildscript, a large list of variable names and values were manually inserted and maintained in XML notation. The ant script then generated a java source file which contained the source for a static class. This ProjectStub class contained all variable names and values listed in the ant script, typed as constant values. In Java, this means the variables had a `static final` prefix. This method had the following advantages:

- Easily to find and change values
  All graphics locations were defined on one location, making them easy to locate and to change.
- Optimized bytecode by use of `static final` prefix.
  Because all variables were defined as a constant, the Java compiler can then apply a optimization technique called 'constant inlining'. This process removes the reference to the projectstub variable, replacing it with the actual value of the variable. For example, consider the following source code:

  ```
  a = ProjectStub.LOCATION_X;
  ```

  In this piece of code, a variable named 'a' gets the value of the constant LOCATION_X of the ProjectStub class. In this class, the constant variable has the value 10. After the Java compiler has compiled this piece of code, the code was compiled as if it was like this:

  ```
  a = 10;
  ```

  As the example illustrates, the compiler has removed the reference to ProjectStub, and replaced it with the actual value that was referenced. Using other optimizers like ProGuard [24], it becomes possible to completely remove the ProjectStub class from the class files when a game is distributed, as it is no longer being used. As well as maintaining a central location for placing varying values of a game between build targets, this method also contributes in minimizing the distributed game file size because of Java's constant inlining optimization.
- Tweakable values through hot code replacement
  Java's runtime environment supports a technique called 'Hot code replacement', in which code changes can be applied at run-time. Changes to code are compiled into new .class files, and these .class files can be reloaded into an already running environment. The changes can then be directly applied to the running application, where the developer can observe the effects of the changes. Although this method has some limitations (for example, a class can only be reloaded if the signatures the class' methods and members aren't altered), Gamica makes frequent use of it. The ProjectStub class can be used in combination with hot code replacement, to test and tweak various values at run-time.

But this ProjectStub method also has its drawbacks:

- Tedious management of multiple ProjectStubs
  The current method works pretty well when working with one single ProjectStub. But when implementing several ProjectStubs, the ant script increases in size or gets different versions. When increasing or decreasing the number of variables defined in the ProjectStub, all other ProjectStubs need to be changed as well.
- Changes made to the ProjectStub aren't directly applied to the create-script of the class
  When a change has been made to the ProjectStub's source, the changes must be applied to the ant script that created the ProjectStub as well. Otherwise the changes have the risk of being overwritten by another ant ProjectStub build.

Proposed improvements

Combining efficiency and ease of use
In normal circumstances, extending the reference ProjectStub with another class using traditional object

oriented methods could suffice. But as this method creates some additional overhead in the form of extra class definitions, this traditional method does not meet the requirements. Other methods which include a generic class containing getter methods doesn't apply either, as the newly introduced methods will also generate a greater bytecode size.

The most efficient method is already used, as a single ProjectStub file containing constants combined with constant-inlining removes the need for additional classes, methods etc. The problem here is managing multiple versions of the ProjectStub. As a developer tweaks and sets the right values when running and debugging a game, the resulting values should later be re-inserted in an ant script.

What is required, is a method in which any changes made to the ProjectStub are directy applied and saved (without any need to reapply the values somewhere else), while still being able to use a single ProjectStub in a game build to reduce overhead.

Optimized inheritance
The proposed solution for this, is to create a single ProjectStub file (the reference) which can be extended by multiple versions. These versions only describe which values are changed in relation to the reference. Any newly introduced values are described in the reference and then changed in one of the replacing versions.

This method is very similar to traditional object oriented inheritance. In order to optimize it for minimal overhead, a new ProjectStub could be generated at build time which contains all variables of the reference, and the values of the required version.

With these improvements, the ant script generating the ProjectStub isn't necessary anymore. All changes to a ProjectStub reference and versions are applied and saved directly, without having to revert them back to the ant script. Furthermore, changing and editing different ProjectStub versions becomes easier to do and to maintain, while still able to utilize hot code replacement.

<u>Proposed transformation operations</u>

This 'overriding' of fields and their values of a reference class by another class, resulting into a single class could be implemented in a manipulation hereby called OverrideFieldsOperation. As the overriding of fields is expected to be used for multiple fields per class, the operation will be executed per class, instead of on a per-field basis.

OverrideFieldsOperation
Parameters Victim class (class of which its fields will be overridden) Invader class (class of which its fields will be placed in the victim) *Preconditions* Both Victim and Invader class should exist. For the operation to be meaningful, the Invader class should contain several field members. *Postconditions* Every field which is declared in the Invader class is inserted into the Victim class. When a Victim class' field name matches with one of the Invader class, its value is replaced by that of the Invader class field. After all other operations are completed, the Invader class is discarded. This depends on whether the Invader class is placed among the reference source code or not (this decision is made in 'determine technical approach')

Note that this operation is different from traditional object oriented inheritance. Whereas inheritance still makes it possible to access the original values of the Victim class (which is called a 'superclass' in object oriented terms), with the OverrideFieldsOperation all original field values are lost.

Variation B: ResourceLoading

<u>Previous method</u>

Code for loading of resources (images, audio files etc.) tends to differ between Java profiles, used (proprietary) APIs and in some cases even between devices. Most of these operations are executed within a single method of a Loader class within Gamica's in-house developed generic library. In this method, a loop iterates through all types of resources. Per type, a switch statement handles every type of resource encountered. To accommodate the various ways of loading a resource, preprocessing directives denote the variations that occur within the switch cases. Depending on the current build settings, code related to a specific tag are inserted into the source code just before compilation.

This method has the following advantages:

- Small, fast alterations possible
  It is relatively easy to add a new variation to source code. Inserting a new tag with new code is a more direct approach then describing a variation using aspects.
- Variations can be defined everywhere, using the tags.
- Small footprint
  Code for loading all the various types of resources could be separated by using methods, but this adds overhead because of method descriptions. Keeping it all inside a switch statement therefore decreases the total size of compiled bytecode.

There are also some problems observed when using this method:

- Disabled Eclipse functionality
  Because of the tags, most of Eclipse's java editor functionality is disabled. Eclipse does apply code highlighting, but any code completion, refactoring and debugging isn't possible. This is mainly because the tags are against standard Java conventions.
- Decreased maintainability
  Within a single switch case, multiple versions of code are placed underneath each other. This is done at various points within the method as well. This makes it difficult to manage, as it becomes increasingly unclear which code will actually be applied at compile-time.

Required source code transformations

Any variability operations in for this variation will require changing code within a case body. Changing the entire method in which the mentioned switch block is located will not do, as code around the switch is generic enough to be applied for many variations.

When considering replacing the entire switch statement, it was found that this method isn't efficient as well. This is because device specific APIs and capabilities sometimes have both similarities as differences in the required calls to load resources. For example, a certain device could require a specific order of API calls to load an audio file (to adhere to its specification, or to work around a known bug), but could very well use generic image loading calls which are similar to other device builds.

To prevent placing duplicate code in various variation descriptions (which could lead to editing several variation descriptions in order to fix a bug in a set of generic code), the variability framework should be able replace code within a switch case.

Proposed improvements

Staying within context
Code within a switch case could reference class members, methods, imports and any variables declared within the method in which the case was defined. These elements could be defined as 'the context' in which the code of the switch case is placed.

When a change is made to this context, there's a good chance the code of the switch case should be changed as well. When the switch case's code can be defined by one of several variability-operations, it is very likely that more than one of said operations are subject to change. Determining which variability operations are relevant to a certain context changes and making sure that the operations still work after a context change can become an increasingly difficult and time consuming process.

Changing blocks of code
A proposed transformation operation should address a code block which is as self contained as possible. When dealing with changing case bodies, it means the code within the bodies should have a minimized relation to the code's context. While that is the case in this variation as only calls to certain APIs are made, in other situations applying the variability operation at this level could require additional restructuring of the reference source code.

Locating target code
Another issue is locating the code that is to be changed. For this, a certain unique identification should be used. For example, when changing code of a method, the signature of the method (name and parameter types) can be used. When finding a case body, the following information is needed: method signature in which the switch statement is used, the switch statement's expression and the switch case's expression. When creating a variability operation which depends on these three factors, a change to any of these factors

would require a change to the operation as well.

Isolating code for transformation
To circumvent these issues, the following solution is proposed. As a small restructuring effort, the case body's code is placed within a separate method (using method parameters to handle any relations to the case body's context). The method's body can then be replaced by using an method-specific version of the previously described OverrideFieldsOperation, in which methods are overridden. This means that when a certain operation defines a method and issues a (hereby named) OverrideMethodOperation on a certain reference class, the defined method's body will be placed in the method of the reference class that matches its signature. This technique is very similar to the Template design pattern [25].

Optimizing code isolation
But like the Template design pattern, this technique introduces additional overhead because of the newly introduced method. The design pattern itself doesn't function well either, as the introduction of an abstract class will add overhead as well. To counter this, a new operation could be introduced, which uses 'method inlining'. Similar to constant inlining, method inlining places the body of the relevant method to the place where it is called. This way the method definition can be removed and no additional overhead is introduced.

While this variation mainly addresses changing code within a case body, the proposed solution can be applied to just about every location. This method can thereby account for any out-of-the-ordinary situations where pieces of code should be changed without being inside a clearly defined block.

__**Proposed transformation operations**__

An new type of override operation called OverrideMethodsOperation will be able to override methods within a targeted class, in the same way as the OverrideFieldsOperation does with fields.

OverrideMethodsOperation
Parameters
Victim class (class of which its methods will be overridden)
Invader class (class of which its methods will be placed in the victim)
Preconditions
Both Victim and Invader class should exist. For the operation to be meaningful, the Invader class should contain several methods.
Postconditions
Every method which is declared in the Invader class is inserted into the Victim class. When a Victim class' signature (a combination of name, parameter types and return value type) matches with one inside the Invader class, its body is replaced by that of the Invader method. After all other operations are completed, the Invader class is discarded. This depends on whether the Invader class is placed among the reference source code or not (this decision is made in 'determine technical approach').

Additionally, another operation is introduced which provides method inlining for specific methods. This operation will be called InlineMethodOperation

InlineMethodOperation
Parameters
Name of the class in which the targeted method is located
Signature of the method
Preconditions
For the operation to be executed, the targeted class should contain a method that matches the given signature. To avoid duplicate code, this operation should only be applied in situations where methods are called once, for example when creating a variation for a certain block of code.
Postconditions
The body of the targeted method will be placed at the location(s) where the method was called inside the class. After the operation is completed the method calls and the method definition itself is removed from the class.

Variation C: JavaProfile

Previous method

Differences between Java profiles are abstracted from game code by Gamica's own in-house developed library. This library provides several generic methods and classes which in turn make the profile-specific calls. Most profile specific code is located in separate classes, but in some cases several variable calls are placed within one class or method and are marked and separated by tags. One particular example of this, is the library's main class: FalconApp, which serves as the execution point of every one of Gamica's games which use the library. To be an execution point within a J2ME environment, the related class should extend either MIDlet (for MIDP profiles) or IApplication (for DOJA profiles). Next to this variability, the class contains several profile specific methods and members, as well as generic methods and members that apply to every profile.

Although a variable profile has a big influence on the library itself, most of these can be dealt with by exchanging classes in which profile specific operations are isolated. Here, the focus lies on changing the parts where both generic and profile specific variations are found the most. In this case, that will be FalconApp.

The already mentioned positive and negative points surrounding using the preprocessing directives can be applied in this situation.

Required source code transformations

To be able to exchange the usage of classes, certain classes should be removed from the library when they are not used. This can be done by using an optimizer (such as Proguard [24]), but in some cases a more direct approach is required in which a variability operation explicitly removed an entire class.

Apart from that, several profile specific methods and members need to be introduced into classes which contain both generic as profile specific code.

To support the mentioned superclassing of either the MIDlet or IApplication class, the 'extends' parameter of a class definition should be able to be changed as well.

Proposed solution

Most of the required source code manipulations can be done using the already defined OverrideMethodsOperation and OverrideFieldsOperation. Profile specific code is easy to isolate in this situation, and can therefore be placed inside a separate class, which will be used to override an already existing library class if it is required to combine generic code with profile specific code.

Managing imports
One extra problem with this is the used parameters at the 'import' field of a class. When a profile specific class uses a profile specific version of an Image object, this usually is referred to within the import directives. In this situation, a profile-specific class which is going to override a library class, should also override any import directives. This means that if a reference source imports `java.awt.Image` and an overriding class imports `javax.microedition.lcdui.Image`, then the latter should overwrite the former within the reference source. Any additional imports that weren't already defined inside the reference source will be added as well.

Removing obsolete classes
In other cases, classes who are not required (classes that contain code for another profile) should be removed. In this case, it is assumed that all profile-specific classes are included in the library's source tree. This hasn't been decided yet at this stage though, but for completeness sake, removal of classes will be added to the list required manipulations. When more details of the technical implementation are being determined, which happens later in the research, will the usage of this manipulation be re-evaluated.

Proposed transformation operations

For this variation, the OverrideMethodsOperation and OverrideFieldsOperation can be utilized extensively. To account for any conflicting or unknown type names, the `import` directives of the Victim class are to be overridden by these operations as well.

To account for changing the superclass definition, an OverrideExtendsOperation could be introduced. This operation will change the superclass definition to that of another class. This operation can only be used in a limited amount of situations however, because replacing any existing superclass definitions in a Victim class could invalidate certain code that relies on the original superclass relation. In some cases this can be averted by replacing the invalidated code through OverrideMethodOperations, if the relation can be easily broken.

As this operation is expected to be done in conjunction with other class-wide operations, the OverrideExtendsOperation is done on a class level as well.

OverrideExtendsOperation
Parameters
Victim class (class of which its superclass relation will be overridden)
Invader class (class of which its superclass relation will be placed in the victim)
Preconditions
Both Victim and Invader class should exist. For the operation to be meaningful, the Invader class should have a superclass relation. For the end result to be valid, the Victim class should not have a superclass definition on which its non-overridden code depends on.
Postconditions
The superclass relation of the Victim class is now the same as that of the Invader class. After all other operations are completed, the Invader class is discarded. This depends on whether the Invader class is placed among the reference source code or not (this decision is made in 'determine technical approach').

To remove any unused classes, another new operation called RemoveClassOperation is introduced. Although this operation is still under advisement, as the actual requirement of this operation depends on the technical implementation and design.

RemoveClassOperation
Parameters
Targeted class name
Preconditions
The targeted class should exist.
Postconditions
The class is removed. In order for the resulting build to be valid, the issuer of this operation should make sure that the class isn't relied upon.

Variation D: DisabledConnectivity

Previous method

Disabling connectivity for a certain build of a game requires changes to both game and library code. In the library only certain connectivity related classes are to be removed. For game, features that require connectivity should be removed. Also, any menus, options or other elements that link to those features should be altered to remove the links.

This variation is currently implemented in the library by using the aforementioned preprocessing directives. The game specific changes are done manually, but the tag method could also be applied. Again, the same strong and weak points are related to this method, as were mentioned in previous variations.

Required source code transformations

Changing the library is a relatively simple process, as only the removal of certain classes is required to minimize unused code in the library, further minimizing the total file size of the game.

Changing the game however, is a total different matter. The locations of connectivity-related code can vary greatly between games. As an base for analysis, the earlier mentioned game *Battleships* is used.

Removal types
Battleships contains a highscore feature in which the player's scores can be posted and synchronized with an online scores list. When removing this feature from the game, two kinds of removals should be done:

- Removing access to said functionality
- Removing code that implements the functionality

Access to the connectivity features are given through option menus. In *Battleships*, the contents of option menus are defined by a multi dimensional array. Using this array as input, standard library functionality is used to create the visuals and interaction handling of the menus.

Removing access
So to remove access to the connectivity features, the array containing an option to synchronize the highscore list should be altered. This means that a certain value from the array should be removed and that an entire option menu contents should be emptied. This last transformation poses a problem however, because it causes displaying an empty menu which would only confuse players. To counter this, the call which displays the related option menu should be replaced as well.

Removing inaccessible code
Removing the connectivity code itself is somewhat more tricky. *Battleships* (as all other Gamica games) uses several game states to define how the game should behave and what graphics should be displayed. These states are checked in a switch statement, of which it's case bodies contain the code that should be executed when the related game state is active. Every game state is associated with an integer constant which is defined as a class member. This class member is therefore only used to determine if the current gamestate matches the value of the member. This gives the opportunity to locate the related game state switch case which matches the class member to a game state, independent of the case's context. When a case defined by the gamestate member can be found, the body is to be removed. This is a relatively safe operation, when all access to the setting of that particular game state has been removed.

Some code however uses exceptions to this rule, introducing more complicated if statements to execute code that is relevant to multiple game states. The code itself cannot be removed in these cases, as they're still relevant to other cases. Part of the if statement's evaluation expression could be removed, if the implementation technology supports it.

Removing game states
When all access to the state has been removed, and all evaluations that include the state member value are removed as well, it becomes possible to actually remove the state value altogether. This will depend heavily on the used code style, as the game state member should only be used in a small number of ways, if it is to be removed completely.

The analysis of determining which code grants access to which game states and what code can be safely deleted, lies in the hands of the developer him/herself. Although determining access to game states is relatively easy because of the straightforward state handling within the game code, certain knowledge about the code is preferred when altering the game code at his level. Again, strict code styles can help in structuring these kinds of variations. For now, the developer should thoroughly check if a certain variation creates errors or other problems. By limiting him/herself to only changing game states and access to these states, human errors can be prevented as these operations are relatively simple and easy to check.

These issues do open up another discussion about how this code style should look like in order to work best with these kinds of variability operations. This topic is further discussed in chapter *'Further research'*.

Obsolete classes
Lastly, *Battleships* has two classes containing methods specifically handling the parsing of externally imported highscore data. As only the game states related to the connectivity features used these methods, one of these classes can be safely removed. The other class contains a combination of both connectivity features as 'offline' highscore features, that are still required when no connectivity is provided. To remove the connectivity aspect of this class, several methods should be removed from it. As no other code within this class required these methods, removing the methods is sufficient.

Proposed solution

This variation requires several operations manipulating small pieces of code on a low level within the source code. In previous variations, efforts have been made to handle blocks of code instead of individual parts, but in this case tinkering with individual statements and expressions cannot be avoided.

Changing arrays
The first required manipulation in which this becomes apparent, is removing access to the connectivity feature of *Battleships*. As mentioned in the previous paragraph, certain values of an class field array need to be removed. One easy way to do this is to replace the entire array altogether using the aforementioned OverrideFieldsOperation. But this method can easily conflict with other similar operations, as the operation itself defines what other values the array should contain. When a similar operation overrides the array values again, any previous changes will be lost.

Therefore a new operation is required in which an array value can be changed, only in the locations where the change is required. This new ChangeArrayOperation should be able to replace certain parts of a single or multidimensional array, without breaking the array's structure.

Altering single statements
Also mentioned in the previous paragraph, a single statement has to be removed in order to disable access to the game's connectivity features. This can be done using a combination of the OverrideMethodsOperation and the InlineMethodOperation. By placing the offending statement in a separate method, the method can be overridden with an empty method. After inlining the method, the statement is effectively removed from the game's flow. Another approach would be to remove the method using a new RemoveMethodOperation, which also removes any calls made to the method. This technique is a bit more efficient, as it only requires the execution of one, possibly simpler to implement operation, instead of two. It also will not require any additional classes which define the emptied method that should be used in conjunction with the OverrideMethodsOperation.

The RemoveMethodOperation will have some limitations. In situations where a return value of the method is used as part of a formula or as part of an evaluation, removal of the method call will most certainly break game logic. And possibly syntax as well. This operation should therefore only be applied in cases where the method can be safely removed without it being used for the described purposes.

Removing case bodies
Removing a complete case body from a switch statement block hasn't been done in any of the previous variations. Therefore a new operation is required, which is named RemoveCaseBodyOperation. In this variation the related case body can easily be located because the bodies are marked using a constant class field value, of which every occurrence of this value should be removed.

In the previous paragraph, it was mentioned that removing a class field that relates to a certain game state, and removing all elements that accesses this field could also be considered. One problem with this are if statements and expressions where the field value is used in some sort of mathematical formula or array identifier. Because the removal of field access from these kind of uses is a relatively complex operation, which has a high probability of breaking code syntax and structure, a possible 'RemoveFieldOperation' is left out of the manipulations. If a technical solution can be found to solve or circumvent these complexities, it will be reconsidered.

Removing obsolete connectivity code
Finally, removing the connectivity classes from both *Battleships* and the library, can be done by a RemoveClassOperation.

For the special case of one class in *Battleships* that combined 'offline' and 'online' features, the removal of a select set of methods is required. This can be done by either a set of RemoveMethodOperations, or by seperating the offline and online features and use OverrideMethodsOperation and OverrideFieldOperation to include them when connectivity is possible. From a maintainability perspective, the latter seems the most appropriate, as further changes to the separated classes won't intervene with the variability operations.

<u>Proposed transformation operations</u>

For this variation, the following operations are proposed:

A ChangeArrayOperation to replace or remove array values from an array field.

ChangeArrayOperation
Parameters Class name in which the array is defined Array name Needle string (the value or values that are to be replaced) Replacement string (the new values of the needle string) *Preconditions* For this operation to be applied, both class, array and needle string should match. *Postconditions* The array values are changed. In order for the resulting code to be valid, the replacement string should contain a string which should generate a valid array definition.

A RemoveCaseBodyOperation to remove a case (including its body) from a switch statement.

RemoveCaseBodyOperation
Parameters The ideal usage of this operation is to only require two parameters, class name and case expression. This leads to the removal of all case bodies which evaluate the given case expression. The alternative is to include method name and switch statement expression. But this will introduce a stronger dependency on the context of the case body. As the variation described above only requires a case expression to be successful (in which the case expression matches a state field name) the current operation parameters will only include classname and case expression. If other uses of this operation requires the inclusion of context specific parameters, the definition of this operation is subject to change. So for now, the operation parameters are: Class name Case expression *Preconditions* In order to apply this operation, both class name and case expression should match. *Postconditions* The matched case expressions are removed. Their bodies are also removed in case these are only related to the given case expression. Access and other relations to this case body should be removed or disabled as well, in order for the game's logic and code to remain valid.

A RemoveMethodOperation which removes a method from a class.

RemoveMethodOperation
Parameters Class name in which the array is defined Method signature *Preconditions* For this operation to be applied, both class and method signature must find a match *Postconditions* The method definition, its code and all calls to it are removed. Except when the method is called from within a formula or evaluation expression, as this most certainly will break application/game logic and possibly code syntax.

Variation E: Language

Previous method

Supporting multiple languages is currently implemented by defining a class containing a large number of static string array members. For every piece of text (varying from the text of an 'ok' button, to complete character dialog) a new string array is constructed, in which every element contains the text in a different language. To display the texts, game classes directly reference these arrays using the element number defined as the currently selected language. The class containing the string array members is generated from an ant script. In this ant script, every language text is described using XML. When generating the class source, the ant script converts any non-ASCII characters to a Unicode definition which will be placed in the member declaration.

This method has no observed problems at this point. Inserting new language definitions is done within the XML file, without having to handle any Unicode specific markups. Hot code replacement to enable runtime tweaking, as with the GraphicsLocation variation, isn't required for this variation. Translated text strings are delivered by a specialized translation company. Gamica doesn't have enough domain knowledge to manually abbreviate texts when they are too large. This removes the need for runtime tweaking, making an aspect-based solution for this variation unnecessary. Therefore, this variation will not be implemented using the aspect framework.

Variation F: LanguageMenu

Previous method

When multiple languages are supported, a language menu should be inserted into the game code. This enables the player to manually select his/her preferred language. Currently, this language menu is implemented using a standard menu framework implemented in Gamica's library. The contents and availability of such a menu would depend on the number of languages defined in the language definition class. This ensures a minimized amount of introduced overhead.

The currently used method for implementing a language menu doesn't have any significant drawbacks. This leads to the conclusion that an aspect-based implementation of this variation isn't required.

Variation G: ImageFormat

Previous method

Handling differences between image format support (.gif, .png and others) is mainly done on a resource level. For certain builds, a special version of a resource file needs to be created in which the standardly used .png files are switched with .gif or other versions.

The code to load these resources is not different for each image format however. Supported image formats varies mostly among J2ME profiles (MIDP, DOJA etc.). Such a variation will fall under *Variation C: JavaProfile*.

In some cases device specific bugs prevent the usage of .png, in which the build should fall back to .gif support. But in these cases the actual calls to load the image will stay the same.

Because this variation doesn't require any additional variability that is not handled in other variations, no additional manipulations are required.

Variation H: SimultaneousAudioSupport

Previous method

This variation is actually comprised of two parts. One parts concerns playing multiple audio streams simultaneously, while the other concerns playing different kinds of audio (WAV, MIDI) simultaneously. The first part will be dealt with in *Variation I: MidiPlayback*. The second part is dealt with in this variation.

Preparation of the FALCON library for playing different kinds of audio simultaneously, is done based on the loaded resources. Within the resource file's structure, every piece of audio data contains a flag determining whither the audio is used for background music or sound effects. When creating this resource file, sound effects and/or background music resources are created according to device specifications. As with the ImageFormat variation, no library changes are required here.

There are however some changes expected for games, as they will require a separate volume slider option when multiple forms of audio is supported.

Required source code transformations

Within the *Battleships* game, sound volume options are placed inside a special option menu. As was mentioned in V*ariation D: DisabledConnectivity*, the contents of this option menu is determined by an array filled with option description constants. For this manipulation, this array needs to be altered to add or remove the necessary menu options.

As the user is browsing through the option menu, the menu maintains several 'option states' in which the current possible input and reactions to this input is determined per option. These states are, like the earlier mentioned game states, located in a switch statement. The state which handles input during an option selection, needs to be added or removed to the state switch statement as well.

Lastly, drawing the menu slider needs to be added or removed. This code is placed within a certain method and needs to be altered.

Proposed solution

It is proposed to include the additional volume slider inside the original game source, and isolate the related code using a separate methods. As with *Variation D: DisabledConnectivity*, a combination of ChangeArrayOperations and RemoveCaseBodyOperations can be executed to perform the required manipulations. To remove the slider drawing, the draw code should be isolated in a method, and be removed using RemoveMethod and InlineMethodOperation.

Proposed transformation operations

For this variation, no new types of operations are required. The already mentioned InlineMethodOperation, RemoveMethodOperation and ChangeArrayOperation can be utilized.

Variation I: MidiPlayback

Previous method

For most of Gamica's games, MIDI files are used to provide background music for games. As these files utilize a device's internal MIDI sequencer, MIDI files only contain data describing when and which note of which internal instrument should be played. Because of the relatively small amount of data required to play several minutes of music, the format is fairly suitable to use in a restrictive environment of mobile game development.

In J2ME, playing back a MIDI file consists of creating a Player object in which the MIDI file's contents are 'prefetched'. When the object has finished prefetching, the object can start playing back the file. When an object is subject for removal to save up memory or in situations where the game is certain not to play the MIDI file for some time, the player object can be disposed.

Midi playback device types
Although this process sounds relatively simple, its implementation sadly isn't. Several devices have widely different problems, bugs and other issues regarding playing back MIDI files. In the current version of Gamica's FALCON library, mobile devices are ranked in three different categories regarding MIDI playback:

- Type A
  Type A devices have the most problematic MIDI playback issues. For devices of this type, every time a MIDI file is played, the mentioned player object needs to be disposed, recreated and prefetch the MIDI file. This procedure is highly unwanted however, as it creates a delay between calling for playback and actual playback. For games as *Battleships* where MIDI files are also used for playing back sound effects, this introduces a delay for several user actions.
- Type B
  Type B devices can handle multiple active player objects, but only one can be playing back a file at the time. This means that additional checks are required in the library to make sure a certain MIDI file can be played back, and other player objects have stopped playing.
- Type C
  Type C devices have the best implementation of midi playback, as they support playing back multiple MIDI streams.

In its current form, Gamica's FALCON library manages these differences by adding a high number of preprocessing directives within the related source code. Because multiple versions of code exist in several parts of this code, and the versions of code need to communicate with other variable code at other locations, reading the source file becomes increasingly difficult.

Required source code transformations

Playing back MIDI files is contained within three methods. The create method creates player objects when the midi playback class is constructed. As this pre-construction isn't required for Type A devices (as these objects are rebuilt per playback action) parts of this method are to be removed.

Another method, play(), is used to actually play back the midi file. As this code is the same for all types of devices, this method will remain unaltered.

Lastly, there is a stop() method, in which a midi file is stopped. For Type A devices, a stopped file also means it should be disposed. Type B and C devices however, will only be disposed when it is explicitly stated it should be disposed (by using a method parameter). The related code is resided in an if statement, which should be moved outside the statement for Type A devices.

Proposed solution

Most of the required manipulations can be done using combinations of OverrideMethod and InlineMethod. Disabling the dispose if statement will require several additional methods however. The case study will determine if this doesn't affect readability and maintainability too much. If this is the case, a new kind of operation will be required.

Proposed transformation operations

For this variation, no new types of operations are required. The already mentioned InlineMethodOperation and OverrideMethodOperation can be utilized.

Variation J: PauseEventHandling

Previous method

Handling differences between pause events ultimately boils down to differences between Java profiles, as the events only differ between these profiles. Namely pause events between DOJA and MIDP differ greatly from each other.

On the library level, these differences are dealt with by replacing code within several generic methods using preprocessing directives, and introduce some DOJA specific classes within the library.

Variations in pause events aren't handled within games. Gamica's current code style for games require a specific game state in which a pause event is handled. In this, the type of pause event doesn't make any difference.

Required source code transformations

Required source manipulations are restricted to replacing code of several methods and introducing some DOJA specific classes inside the library.

Proposed solution

The source manipulations can be handled through OverrideMethodOperations

Proposed transformation operations

For this variation, no new types of operations are required. The already mentioned OverrideMethodOperation can be utilized.

Variation K: AnimationImplementation

Previous method

Variations in the implementation of loading and displaying an animation depend on the available Java profile. MIDP 1.0, MIDP 2.0 and DOJA profiles all provide different means to implement animations. This means that this variation is actually part of the JavaProfile variation described earlier in this chapter.

Currently, variations between implanting these differences are done by a set of preprocessing directives within a single class responsible for image animation preparation and displaying. Whereas this project is trying to remove the dependency on these code polluting tags, another method is devised.

Required source code transformations

In one single class, the contents of several methods are entirely replaced. As the methods' signature stay the same, calls to these methods will not need any alterations.

Proposed solution

A set of OverrideMethodOperations can implement these manipulations.

Proposed transformation operations

For this variation, no new types of operations are required. The already mentioned OverrideMethodOperation can be utilized.

Variation L: DistributorImage

Previous method

Displaying a distributor-specific image is done inside the game's code. For this, a special game state is constructed in which the image is displayed. Changing the contents of the image is done on a resource level.

When no distributor image is required, the code is changed manually or using preprocessing directives.

Required source code transformations

In cases where a special distributor image is required, the game's source code should contain the required game state. Furthermore, the game state should be included in the game's flow (in other words. the state should be reached through the call of a setState method). When no such image is required however, the gamestate and its access can be removed.

Proposed solution

For this variation, changes are only required when there is no specific distributor image to be displayed. In such a case, a size reducing optimization can be done. This is possible by removing the related game state case block and replacing the call to the game state.

These kind of manipulations have been already been discussed in previous manipulations, and were solved using a combination of OverrideMethodOperations , InlineMethodOperations and a RemoveCaseBodyOperation.

Proposed transformation operations

For this variation, no new types of operations are required. The already mentioned OverrideMethodOperation, InlineMethodOperation and RemoveCaseBodyOperation can be utilized.

4.3. Summary of required operations

The analysis of required manipulations on existing source code, resulted in a set of common source operations, which are used as a requirement for the variability framework.

In the next pages, two tables display the required operations per variation and a summary of descriptions of these operations.

Variation	***Operations***	***Rationale***
Variation A: GraphicsLocation	OverrideFieldOperation	Change values of static constants
Variation B: ResourceLoading	OverrideMethodOperation InlineMethodOperation	Replace code within a switch case body
Variation C: JavaProfile	OverrideMethodOperation OverrideFieldOperation OverrideExtendOperation RemoveClassOperation	Insert profile-specific code into methods which also contain generic code. Introduce profile-specific code. Define profile-specific superclass. Remove classes which are not required for certain profiles.
Variation D: DisabledConnectivity	ChangeArrayOperation RemoveCaseBodyOperation RemoveMethodOperation InlineMethodOperation OverrideMethodOperation	Remove array values to remove menu options. Remove code related to game and option menu states. Often found in switch case bodies. Remove calls to these game and option menu states. Remove connectivity related methods and calls to these methods.
Variation E: Language	none	
Variation F: LanguageMenu	none	
Variation G: ImageFormat	none	
Variation H: SimultaneousAudioSupport	RemoveCaseBodyOperation ChangeArrayOperation RemoveMethodOperation InlineMethodOperation	Remove array values defining game option regarding an unused volume slider. Remove related game and option states and their code from switch statement bodies and methods bodies.
Variation I: MidiPlayback	InlineMethodOperation OverrideMethodOperation	Place code within certain parts of method bodies.
Variation J: PauseEventHandling	OverrideMethodOperation	Replace code of several methods
Variation K: AnimationImplementation	OverrideMethodOperation	Replace code of several methods
Variation L: DistributorImage	OverrideMethodOperation InlineMethodOperation RemoveCaseBodyOperation	Remove game state related code (from a switch statements) and paths to the game state.

Required operations per variation

OverrideMethod	*Description* Places a method in a so-called 'victim class'. When this victim class already has a similar method it is overwritten. *Parameters* Victim class (class of which its methods will be overridden) Invader class (class of which its methods will be placed in the victim) *Preconditions* Both Victim and Invader class should exist. For the operation to be meaningful, the Invader class should contain several methods. *Postconditions* Every method which is declared in the Invader class is inserted into the Victim class. When a Victim class' signature (a combination of name, parameter types and return value type) matches with one inside the Invader class, its body is replaced by that of the Invader method.
OverrideField	*Description* Places a field in a so-called 'victim class'. When this victim class already has a similar field it is overwritten. *Parameters* Victim class (class of which its fields will be overridden) Invader class (class of which its fields will be placed in the victim) *Preconditions* Both Victim and Invader class should exist. For the operation to be meaningful, the Invader class should contain several field members. *Postconditions* Every field which is declared in the Invader class is inserted into the Victim class. When a Victim class' field name matches with one of the Invader class, its value is replaced by that of the Invader class field.
OverrideExtends	*Description* Places a superclass directive in a so-called 'victim class'. When this victim class already has a superclass, it's directive is overwritten. *Parameters* Victim class (class of which its superclass relation will be overridden) Invader class (class of which its superclass relation will be placed in the victim) *Preconditions* Both Victim and Invader class should exist. For the operation to be meaningful, the Invader class should have a superclass relation. For the end result to be valid, the Victim class should not have a superclass definition on which its non-overridden code depends on. *Postconditions* The superclass relation of the Victim class is now the same as that of the Invader class. After all other operations are completed, the Invader class is discarded. This depends on whether the Invader class is placed among the reference source code or not (this decision is made in 'determine technical approach').
InlineMethod	*Description* Copies a method's body to the place(s) where the method is being called. Afterwards, the method and calls are removed. *Parameters* Name of the class in which the targeted method is located Signature of the method *Preconditions* For the operation to be executed, the targeted class should contain a method that matches the given signature. *Postconditions* The body of the targeted method will be placed at the location(s) where the method was called inside the class. After the operation is completed the method calls and the method definition itself is removed from the class.
ChangeArray	*Description* Replaces values within an array *Parameters* Class name in which the array is defined Array name Needle string (the value or values that are to be replaced) Replacement string (the new values of the needle string) *Preconditions* For this operation to be applied, both class, array and needle string should match.

	Postconditions The array values are changed. In order for the resulting code to be valid, the replacement string should contain a string which should generate a valid array definition.
RemoveCaseBody	*Description* Removes a case body block *Parameters* Class name Case expression *Preconditions* In order to apply this operation, both class name and case expression should match. *Postconditions* The matched case expressions are removed. Their bodies are also removed in case these are only related to the given case expression. Access and other relations to this case body should be removed or disabled as well, in order for the game's logic and code to remain valid.
RemoveClass	*Description* Removes a class *Parameters* Targeted class name *Preconditions* The targeted class should exist. *Postconditions* The class is removed. In order for the resulting build to be valid, the issuer of this operation should make sure that the class isn't relied upon.
RemoveMethod	*Description* Removes a method and all calls to the method. *Parameters* Class name in which the array is defined Method signature *Preconditions* For this operation to be applied, both class and method signature must find a match *Postconditions* The method definition, its code and all calls to it are removed. Except when the method is called from within a formula or evaluation expression, as this most certainly will break application/game logic and possibly code syntax.

Summary of operations

After the operations were determined, work began on a proof-of-concept in which the operations were implemented.

5. Assessment current implementations

In this chapter, two of the currently most popular existing AOP implementations are analyzed. During this analysis, the implementations are evaluated in terms of efficiency and the support of required functionality. The results of this analysis will party answer two of the subquestions from the problem description, relating to current AOP implementations: *'Can required variations be implemented using AOP and how?'* and *'Is there an efficient and functional enough AOP implementation, or can one be created?'*

5.1. Overview of implementations

The AOP implementations that were analyzed are:

- Eclipse's AspectJ[1]
- JbossAOP[2]

This section provides a short overview of both implementations.

5.1.1. AspectJ

AspectJ is an Aspect Oriented Programming implementation developed and maintained by the Eclipse community. AspectJ provides plug-ins for the Eclipse development environment, which ease the creation and management of aspects. AspectJ also provides command-line tools for applying and running aspects outside the Eclipse environment.

Definition of aspects
AspectJ's definition of aspects comprises mainly of advices, pointcuts and joinpoints defined in a custom language. This language is somewhat similar to Java. Regardless of this partial similarity, it is possible to use regular Java code alongside aspect definitions.

An example of how AspectJ's custom language look like is displayed below.

```
aspect SimpleTracing {

    pointcut tracedCall():
        call(void FigureElement.draw(GraphicsContext));

    before(): tracedCall() {
        System.out.println("Entering: " + thisJoinPoint);
    }
}
```

Example of an aspect definition using AspectJ

Applying aspects
AspectJ applies aspects through bytecode instrumentation. An AspectJ compiler is provided which applies (or *weaves*) the changes defined in the aspects to previously compiled bytecode.

5.1.2. JBossAOP

JBossAOP is part of the JBoss application framework. Like AspectJ, it provides a plug-in for integration in Eclipse. Also, it offers a set of command-line tools which can be used separately from Eclipse and JBoss' application framework.

Definition of aspects
JBossAOP defines aspects in two separate locations and language definitions. Whereas AspectJ both defines advices and pointcuts in the same definition, JBossAOP uses XML for pointcuts and regular Java code for the rest.

An example of how JBoss' pointcut definition looks like is displayed below.

```
<bind pointcut="public void com.mc.BankAccountDAO->withdraw(double amount)">
  <interceptor class="com.mc.Metrics"/>
</bind >
```

Example of an pointcut in JBossAOP

Advices in JBossAOP can be defined by providing Interceptor classes or regular classes containing specifically crafted methods. These classes contain regular Java code, utilizing JBossAOP's API.

An example of an Interceptor class is displayed below.

```
public class Metrics implements org.jboss.aop.advice.Interceptor
{
  public Object invoke(Invocation invocation) throws Throwable
  {
    long startTime = System.currentTimeMillis();
    try
    {
      return invocation.invokeNext();
    }
    finally
    {
      long endTime = System.currentTimeMillis() - startTime;
      java.lang.reflect.Method m = ((MethodInvocation)invocation).method;
      System.out.println("method " + m.toString() + " time: " + endTime +
"ms");
    }
  }
}
```

Example of an Interceptor in JBossAOP

Applying aspects
Like AspectJ, JBossAOP utilizes bytecode instrumentation for applying aspects. An aspect compiler is provided which weaves these aspects into targeted bytecode.

5.2. Implementation efficiency

Within game development, great care is given to the visual and audio representation of games. This often asks for several space consuming resources such as images and audio files. This leaves out a relatively small amount of space for game code. And because of the already strict space requirements existing in mobile game development, a variability solution used in mobile gaming should have a minimal impact on total game bytecode size.

In this chapter, both mentioned AOP implementations are assessed on their impact on total game bytecode sizes.

5.2.1. AspectJ

When applying aspects on bytecode using AspectJ, an increase of bytecode size can be expected. However, the code that is being inserted or changed isn't the only element that increases bytecode size.

In order to accommodate for a wide range of possible scenarios, AspectJ can include additional elements in bytecode that aren't defined in the related aspects themselves. It also adds a dependency to several AspectJ API classes. The full set of API classes has a size of approximately 112 Kb. However, when optimized it's size can be decreased down to 651 bytes.

Impact on bytecode size
In [3] a measurement of bytecode sizes has been made when handling variability using traditional object oriented structures and AspectJ. It mentions that a 15% increase in bytecode size was measured when using AspectJ, compared with an object oriented solution (which was downsized to 10% after using optimizers).

Traditional object oriented solutions were discarded in the problem description earlier in this thesis, because of its inefficiency. A solution that creates a higher amount of overhead should thereby not be acceptable as well. However, as the overhead is not very significant, it may be still possible to further optimize the output of AspectJ.

Should this solution be chosen for further examination however, it should at least support the required functions listed earlier. This part of the assessment is described in section 'Implementation functionality'.

5.2.2. JBossAOP

With JBossAOP, similar increases of class files have been recorded in self conducted smaller scale tests. However, JBossAOP creates dependencies in the targeted bytecode. After using JBossAOP, the targeted bytecode becomes dependent to a large amount of API classes. Unoptimized, these classes require approximately 2,1 Mb of disk space. When using the Proguard optimizer[24] a decrease to +/- 500Kb was achieved.

While this optimization is significant, the size of these classes is still way too much for it to be of any use within mobile game development. This leads to the conclusion that in its current state, JBossAOP is not usable for introducing variability in mobile games.

5.3. Implementation functionality

In this chapter an assessment is made regarding if AspectJ contains the functionality required. The functions which were used in this assessment are listed in a previous chapter *'Gathering detailed requirements'*. Key features assessed were:

- Inserting and replacing code inside method bodies
- Removing methods and related calls
- Changing superclass definition
- Replacing values of fields
- Removing case bodies

5.3.1. Inserting and replacing code inside method bodies

AspectJ supports inserting code at the beginning or the end of a method. Alternatively, it can replace the entire body of a method as well. However, AspectJ cannot place code at an arbitrary location within the method body.

Although, the proposed inlineMethod operation can be implemented if the method's code is defined in the aspect itself. By using a '*call*' pointcut, code can be inserted at the location where a certain method is called. If this code can implemented inside the aspect itself (because of a total replacement of a method body) the InlineMethod operation can be realized.

5.3.2. Removing methods and related calls

Within AspectJ it is possible to remove the contents of a method body. Secondly, it is possible to target all calls to a certain method, and replace them with nothing. When using these techniques in combination with an optimizer, the empty methods are removed as well.

5.3.3. Changing superclass definition

Using the *declare parents* declaration of AspectJ, it is possible to change the superclass definition of a class.

5.3.4. Replacing values of fields

AspectJ supports changing values of fields. However, it has a significant limitation when such a field is defined as a constant. AspectJ is unable to target a field constant in a pointcut, and is therefor not able to make any changes to it.

This is mainly because the Java compiler uses a technique called 'constant-inlining'. In this technique, the value of a constant is moved to the location where it is read. This makes it impossible for other bytecode-level frameworks to find references to a constant value.

Usage of constants
As mentioned before, mobile games are heavily optimized and a heavy usage of constant values is one of these optimizations. One of the prime examples of using constants is mentioned in section '*Variation A: GraphicsLocation'* in chapter *'Gathering detailed requirements'*. In this variation, a separate class filled with constant values is used for defining locations of graphic elements. Because all these values are defined as constants, the entire class can be removed from the game at build time. This is because of the compiler's constant-inlining.

Furthermore, the usage of constant values for graphics locations makes very effective when used with hot code replacement. Because all values are placed directly at the locations where they are needed, hot code replacement immediately applies any changes made to them. These changes are directly displayed at run-time.

Consequences
When these constant values cannot be used anymore, both an important optimization and a tool for efficient tweaking of field values is lost. This decreases the applicability of AspectJ for use in a variability solution targeted at mobile game development. Further discussion regarding this issue is described in section *'Conclusions'* of this chapter.

5.3.5. Removing case bodies

It is currently not possible to target specific case bodies for removal in AspectJ. Furthermore, case bodies are labeled through a constant value. This can be a literal number or a reference to a constant.

As the operation listing in section *'Summary of required operations'* shows, removing of case bodies is often done for the purpose of removing game states. Game states are referred to as constant values.

As is determined in the previous section, constant values cannot be targeted or influenced by AspectJ. Therefor, this operation cannot be implemented in AspectJ as well.

5.4. Conclusions

Based on the analysis and results presented in the previous sections, it can be concluded that none of the analyzed implementations are fit for usage as a variability solution.

JBossAOP introduces a significant amount of kilobytes to the size of a game. This makes it impossible for game developers to create a game that fits inside 200 or even 100 kilobytes.

AspectJ is more efficient in this regard, although it does produce more bytecode than a traditional object oriented solution. Furthermore, it has been determined that AspectJ doesn't support all required functions in order to be used as a variability solution. Functions regarding placing code at arbitrary locations within a method and influencing constant fields are limitations that are not acceptable for this research project.

Alternative solution
As current implementations are lacking in efficiency and features, an alternative solution is required. This solution should apply manipulations directly without having to introduce additional methods and other classes. This to minimize any introduced variability overhead. Furthermore, it should be able to support the features mentioned in previous sections and chapters.

This customized solution is to be targeted specifically at J2ME game development, also taking Gamica's specific process requirements into account. The design and implementation of this domain-specific solution is further detailed in the next chapter: '*Proof-of-concept'*.

6. Proof-of-concept

In this phase, a proof-of-concept was constructed which will support the previously mentioned variability operations. While developing this proof-of-concept certain issues surrounding implementation of the operations are explored and discussed.

The goal of the proof-of-concept is two-fold:

1. Determine if the operations are technically possible.
   Answering subquestion '*Can these variations be implemented and how*?'
2. Determine if the resulting implementation really does work within Gamica's game development process when porting its games to different devices.
   This will answer subquestion '*Can the solution be used within Gamica's development process, or what changes are required to this process to make it possible?*'

The first goal will be dealt with while implementing the framework. After a first implementation is completed, the results will be used as a case study. In this case study, Gamica's developers will use the implementation to port *Battleships* and Gamica's library to several different devices. Experiences and issues regarding the usage of the implementation will then be recorded and analyzed. Using the results of this case study it should be possible to determine if the framework actually works within the process of game development and if there are any early signs of maintainability issues.

Implementation issues
In this chapter, various issues regarding the implementation of the variability framework are discussed. These issues are discussed in the following sections:

- Implementing variability operations
  Describes which techniques are to be used for implementing the required variability operations.
- Designing the program transformation language
  Details design decisions regarding how developers will apply the variability operations
- Implementing the framework
  Describes how the framework is implemented within the targeted environment

6.1. Implementing variability operations

As was stated in chapter *'Assessment of current implementations'*, bytecode instrumentation was deemed too limited to be used for the variability framework. As a result, the framework will apply changes to game and library functions at a source code level.

Dangers of source code transformations
One of the dangers of programmatically transforming source code, is the source parser and transformer. Because source code can be structured in many different ways, source code parsing becomes a complex task, which can be error prone.

Currently there are several software project dedicated to source code parsing and manipulation, including JavaCC [12] (which can be used to generate source code parsers) combined with JTB [13], BeautyJ [14], SableCC [15], Eclipse's internal source parser based on Abstract Syntax Trees [17, 18] and more.

Because it is currently very difficult if not impossible to determine which one of these methods will generate the least errors, the answer to which one is going to be used is decided from the development process perspective.

All development within Gamica is currently done using Eclipse, and Eclipse itself extensively uses its own source parser for debugging and source information displays. Because the variability framework is to work with or alongside Eclipse, the decision was made to create a plugin for Eclipse which utilizes Eclipse's source analysis and manipulation capabilities.

6.1.1. Source code manipulation in Eclipse

As was mentioned in chapter *'Background and Context'*, certain features of the Eclipse development environment can be used to do program transformations at a source code level. The variability framework can utilize this functionality in order to implement the required operations.

Creating a custom plug-in
When creating a variability framework that works alongside the usage of Eclipse as a development environment, extensions and changes are required to that environment. As said, Eclipse has a flexible structure for which several extensions (or 'plug-ins') can be developed. For the variability framework, this means that a new 'project nature' and 'builder' needs to be inserted which, using Eclipse's AST API, makes adjustments to certain pieces of source code prior to regular Java building and compilation.

This custom builder will require certain data to determine which source files should be altered, what kind of alterations should be applied and where to place the results. This requires a language in which the variability operations are described. The design of this language is discussed in the next section, *'Design of the program transformation language'*.

6.2. Designing the program transformation language

The currently defined operations are mainly designed for doing one or a very restricted set of transformations at once. For a description of variability operations, it will be required to define sets of operations, similar to 'aspects' in current AOP solutions. In these 'sets', several operations are described, which will account for a certain piece of variability.

Required elements
To determine how the various variability operations are declared, it is required to determine what common information is needed to execute the sets of operations. This information should be described in addition to operation-specific parameters, which were already determined in chapter *'Gathering detailed requirements'*. This required additional data is as follows:

- A unique identifier (ID), which the set of operations can be called or referred to
- A description determining *when* or *if* the operation set should be applied.
- A description of location(s) within source code (class name, method signature, field name, statement signature etc.) *where* the operation or operation set should be applied in the source code. This information is (partly) stored in an operation's parameters.
- A description of *what* kind of operations are applied.

In the following chapters, the language describing *when* (or *if*) and *where* the operation sets should be applied is discussed. The *when/if* is discussed in chapter '*Capabilities and requirement based variability*', whereas the *where* and *what* will be described in '*Designing operation sets*'.

6.2.1. Capability and requirement based variability

As was briefly mentioned in the problem description, Gamica maintains a database of device specific capabilities, specifications, bugs and other issues, which is used to apply certain device specific alterations to games and its library. The variability platform can utilize this information to determine which operations should be applied for a certain build. For example, when a device has a display resolution of 200 by 400 pixels, the relevant variation operations should be applied that implement the relevant graphics locations for such a resolution. Example content of these database are listed in A*ppendix B: Device properties and channel requirements databases*.

Relating variations to capability and requirements
A relationship between a device capability database, a channel requirements database and sets of variability operations could create an environment in which variability is determined by specifically described requirements and device capabilities.

When new mobile devices are released or channel operators change their requirements, changes made to the relevant databases should automatically change or add new builds for games and the library. But only if enough already created operation sets were available to support any new device properties or changed requirements.

This means that the creation of a new game build based on new device specifications, could be done by only changing the device database if the changes aren't radically different from other devices. In such a case, the creation of several new operation sets is required.

Structuring the relations
This relation between capabilities, requirements and operations can be roughly done in three ways:

1. Capability and requirement databases contain Ids of operation sets, whereas these sets are

oblivious to the databases

2. Capability and requirement databases remain oblivious to operation sets, whereas the sets themselves contain references to capabilities and requirements for which they should be applied. In this structure, databases are oblivious to operation sets.
3. Make both databases and operation sets oblivious to each other, linking the relation in a separate description.

A decision between these options would depend on how the databases and operation sets are expected be used within the game development process.

Considering databases that refer to operation sets (option 1)
As was mentioned in an earlier chapter, development of Gamica's library and games are done separately. Both elements however, do require a high level of variability. The same capability and requirements databases are used for both library and games, but the operations required to implement any variations regarding capabilities and requirements are different between library and each game.

This means that any relationship between databases and operations should be oblivious to the databases, as the operations differ between development projects. This rules out option 1.

Considering total separation of operations and databases (option 3)
Option 3, in which relations between databases and operation sets are declared separately, is a bit overkill for this project. When a new requirement or capability is introduced which requires a new operation set to be implemented, both the link description and the operation set need to be maintained.

Option 3 does offer effective reuse of operation sets, but as these sets are expected to be different for each development project, option 3 only adds an extra maintenance level with no added benefits. Reuse of operation sets could become possible when Gamica enforces a strict code style and structure policy for each game. With such a policy, certain operation sets can be applied to different games because of enforced similarities in locations and structures for certain common features. But as this is not yet the case, option 3 is ruled out for the moment.

Considering operations that refer to relevant database values (option 2)
This leaves out option two, in which the related requirements and device capabilities are listed in the operation sets themselves. Using this structure, it becomes possible for Gamica to manage device capabilities and distribution requirements separately from variability operation sets. This enables a certain freedom when defining a base set of features and capabilities for a game and determining how variations should be and structured for each game.

From a technical standpoint, the above conclusions create the requirement for an extra parameter for operation sets, in which the relationship with a certain capability or distributor requirement is described. The query syntax and underlying technology is discussed in the next section '*Querying device capabilities and channel requirements*'.

Querying device capabilities and channel requirements
Operation sets are required to use the data stored in the XML formatted databases (see *Appendix B: Device properties and channel requriements databases* for examples) to determine if the set should be applied. For this, a specialized query technique will be used, called XPath [21]. XPath enables simple querying of XML formatted data, using a special query language. By using XPath queries, operation sets can determine the existence of certain currently applicable device property, capability or channel requirement. When the query matches, the operation set should be applied. If no matches have been found, the operation set is disabled.

For example, when a certain operation set should only be applied when a device's resolution is between 240x300 and 250x350, the following XPath query is required.

```
//capability[@refid = 'j2me_screen']/property[@name = 'width' and @value > 100]
```

Example XPath query, in which a screen width of more than 100 pixels is matched

As this query will match multiple devices, the variability framework should prepend the query to make sure the query will try to match with values related to the currently selected device. For example, when a developer is currently creating or testing a build for a Nokia 6230, the XPath query will be internally formatted as follows:

```
/gdr/device[@id = 'nokia_6230']/capability[@refid =
'j2me_screens']/property[@name = 'width' and @value > 100]
```

Example XPath query, in which a screen width of more than 100 pixels is matched, combined with a match with a certain device

When the above query matches, the operation set will be applied to the base source code of a game or library.

Determining *what* and *where* apply operations
It is hereby determined how the *when* or *if* of a variability operation set description will be defined. What is left is to design a language in which is described *what* and *where* operations should be applied. This will be discussed in the next chapter *'Designing operation sets'*.

6.2.2. Designing operation sets

In this chapter, a language is proposed which can be used by developers to target and apply variability operations using the variability framework.

Determining language requirements

Developer preferences and requirements
Using informal interviews, several preferences and requirements were discussed to which the description should adhere to from a developer's perspective. One of the most important issues that was raised was that it should have simple description. 'Simple', meaning that the number of possible commands, tags or structures should be as low as possible. The main developer therefore requested a low entry-barrier for developers to use the variability framework and its descriptions.

To this end, it was preferred that the description would match a standard language convention, which uses or works similar as language conventions which were already known to the development team. The description should also have a low probability of errors. And if it is possible, any automated refactoring made to the original source code in Eclipse, should also be applicable to the description itself. This would prevent any additional maintenance when the original source code undergoes some structural refactoring.

Keeping variability code in context
It became apparent that a method using standard Java conventions and language structures was preferred. This preference stemmed from a wish to develop source code which is inserted by a variability operation, while still being able to do context specific lookups, error checks etc.

One problem with the operation language of AspectJ for example, is that most added source code is placed inside an AspectJ specific context. Any relations with the location(s) where this code ends up is non existent. This is mainly because the code could be placed into several different contexts. In this variability framework however, most operations are applied at it's own specific location. It should therefore be possible for developers to develop the code while working inside the context of the targeted location.

Focus on highly used operations
By analyzing the manipulations required for various variations, the operations which are expected to be used most and which define newly inserted code, are both OverrideFieldsOperation and OverrideMethodsOperation. It is also expected that both these operations will often be used in conjunction with each other. Because of these expectations, efforts were made to make the declarations of these operations as easy and simple as possible.

To this end, it was decided that a conventional Java structure was used to describe the operations and operation sets. Annotations were utilized to define certain parameters and other elements which cannot be defined using normal Java language conventions.

Utilizing conventional Java language structures
In the earlier mentioned example of the ProjectStub, the OverrideFieldsOperation is used multiple times to define new values for a high number of class fields. To ease the creation of this operation for the developer, the definition of these values should therefore be simple and straightforward. This is accomplished by defining the OverrideFieldsOperations as regular field definitions in a normal Java class. An annotation at the class definition level then describes for which class these new fields should be overridden, and when the variation should be applied (using an earlier mentioned XPath query). The name of the class definition functions acts as a identifier which carries the name of the operation set.

```
@Variation (class="com.gamica.anygame.AnyGame"
query=”//capability[@refid = 'j2me_screen']/property[@name = 'width' and @value
> 100]”)
public final void class ExampleVariation
{
      int STATE_RETRIEVE_HISCORES = 10;
      int STATE_POST_HISCORES = 10;
}
```

Operation set in which two fields are overridden in class AnyGame

The same is done for the OverrideMethodsOperation.

```
@Variation (class="com.gamica.anygame.AnyGame" query=”//capability[@refid =
'j2me_screen']/property[@name = 'width' and @value > 100]”)
public final void class ExampleVariation
{
      public void showMessage()
      {
            System.out.println(“Message”);
      }
}
```

Operation set in which a method is overridden in class AnyGame

Consequences and limitations

This definition has several consequences. Firstly, the mixture of a variation description and class definition makes it impractical to create an operation set in which multiple classes are altered. Although multiple class definitions can reside within one source file, the operation set's readability and maintainability will suffer when doing so. Also because of this strong linkage, it becomes impossible to reuse any code that was defined in a certain operation set, to be used in another set.

Methods to counter the side effects
These side effects can be countered by separating the Invader class code from the variability descriptions. The Invader class code could be placed in a separate source file, whereas the operation set only describes the OverrideFieldsOperation and/or OverrideMethodsOperation itself, instead of also defining the overriding code. Reuse will become possible with such a construction as well.

The description should be both easy to create, as to provide a conveniently arranged structure in which already entered data can be easily be found, read and understood. Although the last mentioned description method does sound the most sensible at first, making a decision about how to define and structure variability descriptions will depend on how the variability framework is expected to be used.

The necessity of counter measures
As said, the negative side effects of the currently described method of variability description, is that it doesn't support handling multiple classes per description and doesn't support reuse of invader classes. Question is, are these properties really required? An analysis of the earlier described variations didn't indicate any scenarios where invader classes can be reused. Furthermore, the number of classes which are required to be altered per variation is relatively low, for both the FALCON library as the *Battleships* game. No more than a maximum of 4 different classes are required to change per variations.

When the mentioned separation is introduced, developers have to maintain two sets of data. One set in which all descriptions are placed, and another set in which invader classes are coded. As the relationship between these two types of data is one-to-one (as no reuse is expected), it only creates an additional layer of obscurity for the developer. Both while implementing the variability and reviewing it.

This leads to the conclusion that although the described separation of class definition and variability description can be seen as a common practice, it isn't required in this situation. A higher preference is given to ease of development, in which variability description and invader class definition are combined.

Other operations

As the other operations which were defined in chapter *'Gathering detailed requirements'* are expected to be used a lot less than the earlier described operations, most of them are described in the form of annotations. Examples of these descriptions are displayed in the following examples.

```
@RemoveCaseBody(case=”STATE_SETAUDIO”)
```

RemoveCaseBodyOperation

```
@ChangeArray(arrayName=”AVAILABLE_OPTIONS”,
             needle=”OPTION_AUDIO”,
             replacement=””)
```

ChangeArrayOperation

Because of certain limitations in Java's annotation features, it isn't possible to use multiple annotations of the same type within one description. To solve this, the mentioned annotations can be stacked using a parent annotation type, in which the actual annotations are placed within an array value of the parent annotation. An example of this is displayed below.

```
@RemoveCaseBodies({
@RemoveCaseBody(case=”STATE_SETSFX”),
@RemoveCaseBody(case=”STATE_SETMUSIC”),
})
```

Stacking of multiple RemoveCaseBody operations

Other operations can take advantage from the Java language based description, by using normal Java constructions as parameters. For the RemoveMethodOperation and the InlineMethodOperation, this can be used as such:

```
@Remove public void showConnectionStatus(){}
```

RemoveMethodOperation

```
@Inline public void showVolumeSlider(){}
```

InlineMethodOperation

Issues when using multiple similar annotations
However, a problem occurs when the above annotation-based operation descriptions are used. As was mentioned in chapter *'Gathering detailed requirements'*, a combination of OverrideMethod and InlineMethod operations are expected to be commonly used. In the annotations described previously, this combination isn't possible. This is because the description can only describe if a method is inlined or overridden.

While it is possible to simply add code to the method declaration annotated with @inline, this will conflict in cases where only an inline is needed. So, in order to support a combined Inline and OverrideMethod operation, a new kind of annotation is introduced: @InlineAndOverride.

```
@InlineAndOverride public void showMessage()
{
  System.out.println("message");
}
```

Combined Inline and OverrideMethod operation

Differences between variability framework languages and other AOP implementations

Features

The main difference between the earlier mentioned AOP implementations and the described operation declaration language, is that the number of supported transformations is more limited. Because of the focus on minimizing overhead, only a limited number of operations are supported in which detailed transformations can be done without introducing this overhead.

Language structure

On the language side, this means that there are no distinct *joinpoints* in the language definition if the variability framework. Instead of offering a set of operations that can be targeted at a wide range of joinpoints, these operations are limited to certain code structures. These limitations operate within Gamica's code style practices and can therefore offer the required optimizations.

When finding similarities between the language definitions, AspectJ's *aspects* can be seen as *variations* within the custom variability framework's language. *Advices* are a bit more difficult to isolate however, as they're mostly intwined within transformation operations.

Towards implementing the framework

In the next section, the implementation of the framework is described. Internal procedures of the framework will be mentioned. Additionally, the features of the variability framework offered through the Eclipse plug-in interface are discussed.

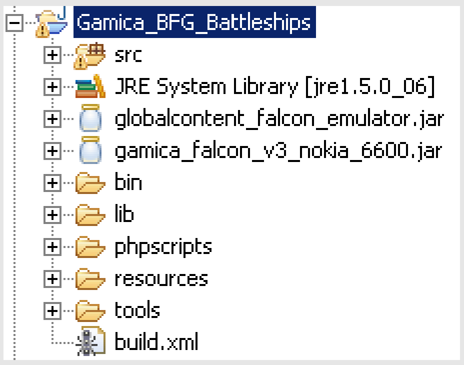
6.3. Implementing the framework

As was mentioned earlier, the framework will be implemented as an Eclipse plug-in. The Eclipse plug-in will provide a new project nature and builder, which can be linked to an existing Eclipse project. The project nature will mark and setup a project to be used to implement variability, and the builder will add the variability operations to the project's build process.

6.3.1. Project layout

Standard project layout

Gamica maintains a strict policy on project layout, as their ANT build scripts depend on a certain project folder structure. A typical project layout for a game looks like this:


```
Gamica_BFG_Battleships
  src
  JRE System Library [jre1.5.0_06]
  globalcontent_falcon_emulator.jar
  gamica_falcon_v3_nokia_6600.jar
  bin
  lib
  phpscripts
  resources
  tools
  build.xml
```

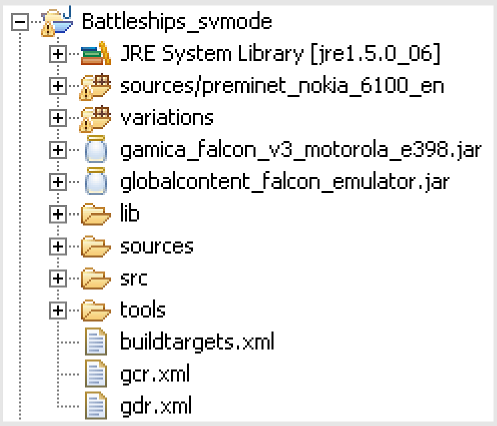
Project tree of regular Gamica game project

In this layout, all source code of a game (or FALCON library for that matter) is placed inside a source folder called 'src'. Files that are created when building the source, are placed in a 'bin' directory. This directory contains the various builds per device, language and distributor, separated in different subfolders.

This project can be built in two ways. An Eclipse build will only compile the source to be able to run it through a J2ME emulator. There is also an ANT build script, in which a generic resource file is compiled (in which all audio, images etc. are stored) and all compiled .class files are obfuscated and optimized using ProGuard [24]. Libraries which are required to compile and run the game are stored in the 'libs' folder.

Variability project layout
This project tree is used when a game is being developed for its first release, mostly supporting a very limited amount of devices. When this development phase is completed, the variability framework comes into play, in which variations of the game's source are created.

For this purpose, the variability framework's plug-in will slightly change the project tree structure. This is done to accommodate the variability operation definitions and different builds. As such:



```
Battleships_svmode
   JRE System Library [jre1.5.0_06]
   sources/preminet_nokia_6100_en
   variations
   gamica_falcon_v3_motorola_e398.jar
   globalcontent_falcon_emulator.jar
   lib
   sources
   src
   tools
   buildtargets.xml
   gcr.xml
   gdr.xml
```

Project tree of Gamica game project using the variability framework

In this new tree, some elements are added to the project tree. A new source folder is created called 'variations'. In this folder all variation descriptions (in the form of operation sets) are stored as java source files. Operation sets which are related to each other are grouped in packages.

Separating sources
Another alteration to the original project tree is the type of the 'src' folder. Whereas this folder was previously a 'source folder' (containing files which are compiled in every Eclipse build), this folder is now turned into a regular folder. The folder which contains the source files which will be compiled, is one of the subfolders of the 'builds' directory. In 'builds', all variations of the original source code are stored. The currently 'active' one (which is compiled and can be run for testing) is set as source folder.

Capability and requirement databases
There are also some .xml files added to the project tree. In gdr.xml (which stands for Global Device Repository), all device specific properties are stored. As mentioned in section *'Capability and requirement based variability*", this file is used to determine which variations are to be applied.

Also, the file gcr.xml is added to the project tree. This file (of which its name stands for Global Channel Repository) contains descriptions of channel specific requirements.

Buildtargets
Lastly, a third 'buildtargets.xml' file is added. This file contains the supported devices, language sets and channels for a project. Additionally, any project specific variation properties which could not be described in the gcr.xml and gdr.xml, can be listed in buildtargets.xml as well. As these last two files are designed to be used for every game, buildtargets.xml will handle any project specific properties, if any. This means that for every project, a different buildtargets.xml is to be defined.

Example content of this buildtargets file can be found in *Appendix C: Example buildtargets.xml*.

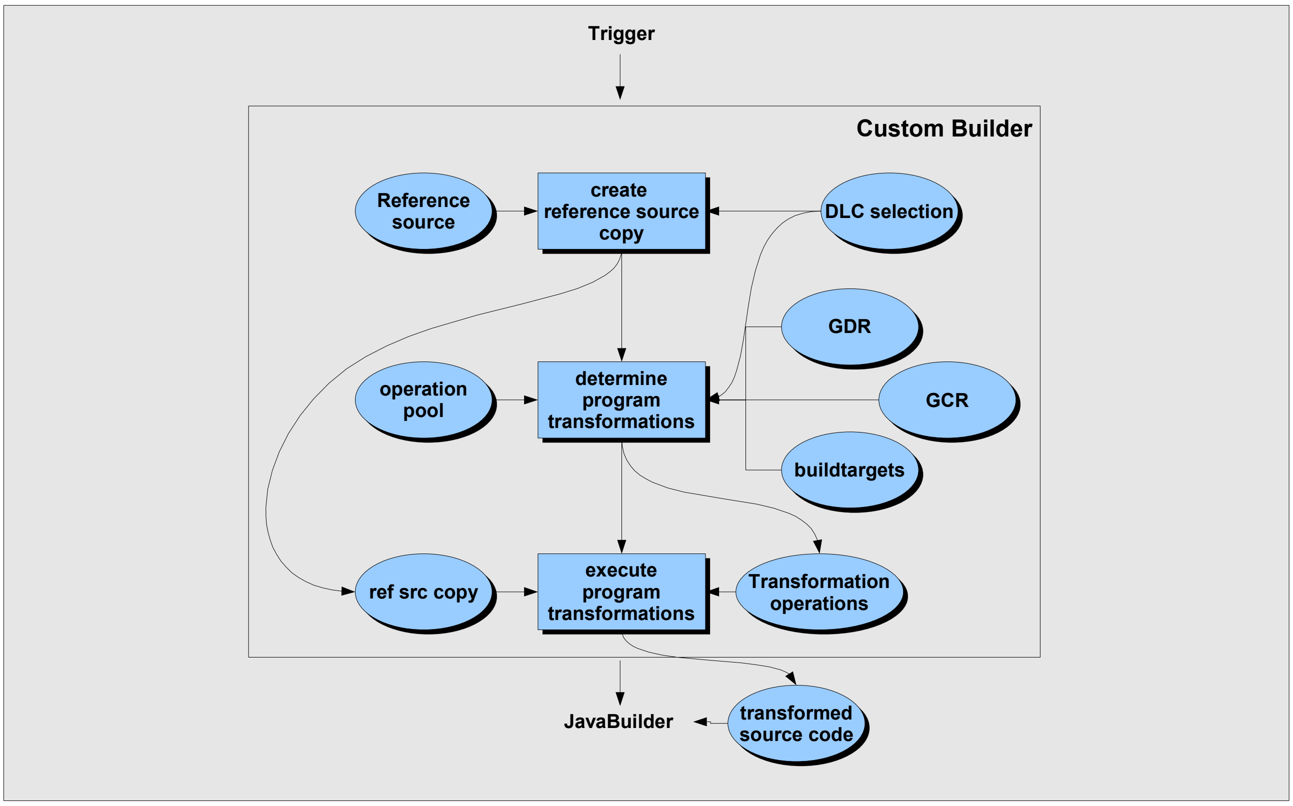
6.3.2. Build process

Within Eclipse, certain user actions can trigger a 'build', depending on current settings. When a setting called 'automatic build' is turned on for example, alterations to files that reside in source folder will trigger a build. When this option is turned off, users need to click on a build button to trigger a build.

A 'build' in Eclipse will activate 'builders', which are associated with a project. This association is usually created by the earlier mentioned project natures. In normal Java projects, only one builder is present: the Java builder. One of the tasks of this Java builder, is to compile .java source files to .class bytecode and display any error messages when something went wrong.

A custom builder
For the variability framework a custom made builder is required, which alters .java source files prior to compilation. This means that the custom made builder should precede the Java builder in the project's build process.

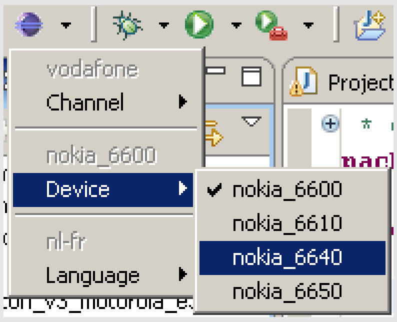
This custom builder will have to do several tasks, which are outlined in the figure below.



Build activities variability framework

Although the collected data and described activities from the above figure are required for each build, the process can be optimized. As Eclipse maintains change delta's in which every change since the last build are recorded, some of the data can be cached if the related files haven't been subject to change.

Selecting device, language and distribution channel (DLC)
The determination of the currently set device, language and channel combination, is evaluating a value set by the developer currently working with the platform plug-in. To let developers change this value, the plug-in provides a special menu, in which all available devices, channels and languages from the GDR and GCR can be selected.



Selecting current device, language and distributor channel

6.3.3. Target for case study

In order to test this framework's effectiveness, a case study was performed in order to test its effectiveness. Details regarding this case study are described in the next chapter: *'Case study'*.

7. Case study

In order to evaluate the effectiveness and usefulness of the variability framework, a case study was preformed. In this study, the following issues were focused on:

- Does the framework indeed supported enough operations to implement the required variability?
- Can the framework be effectively used to decrease bytecode size?
- Are there any hurdles regarding ease-of-use, presentation and code readability when developing variations using the framework.?

For the case study, the framework was used to implement variability in Gamica's FALCON library and *Battleships*, for the purpose of porting both products to other devices. The implementation of the variations was performed in cooperation with two of Gamica's developers. Reactions, experiences, problems and other issues were recorded through personal sit-throughs and a questionnaire.

The results of this case study are used to answer the following questions from the problem description: '*Can required variations be implemented using AOP and how?'* and *'Can the AOP solution be used within Gamica's development process, or what changes are required to this process to make it possible?'.*

This chapter will first describe a couple of changes that were made prior to this case study, suggested by Gamica's main developer. Following, the case study scenario and activities are described. Finally, the questions raised above are discussed and answered in sections *'Support of operations*', *'Decreasing bytecode size'* and *'Ease-of-use, presentation, readability'*. A short summary of this can be found in section *'Summary'*.

7.1. Changes to operations

When discussing the previously listed operations with Gamica's main developer, some changes to the operations were proposed.

Removing game states
One of these changes was combining the RemoveCaseBodyOperation and ChangeArrayValueOperation into one operation in which a field is removed. These operations are mainly used to remove (access to) a certain game or menu option state. As these states are defined by a constant integer field defined at class level, this field can be removed as well. When removing the field, all elements that use the field should be removed as well. This means that switch cases that use the field to define code for a certain state is removed, as well as array values which determine menu options related to a certain state.

Problem with this approach is that when the field is used in other elements such as IF statements, it cannot be removed safely in some cases without breaking the logic of the if statement. Also, when the field is used in certain statements or in formulas, removing it also isn't possible without breaking game code syntax and logic. This problem is seen by the main developer as signs that the reference source code requires restructuring, in which the field shouldn't be used in other situations than switch cases and array values.

This will mean a new operation is required in which a field definition is removed, and any related array values and switch case bodies with it. This new operation is named RemoveFieldOperation.

Lessening complexity
Another issue brought up by the main developer, is added complexity of introducing new methods to reference source code to isolate variable code. This complexity stems from management of both OverrideMethod and InlineMethod operations. Although this is acceptable in some cases, when this isolated code only consists of one statement which requires changing, this is considered overkill. For these cases, a new operation is required in which the single statement is isolated and replaced or removed.

Locating a single statement through an operation description creates a situation in which an operation becomes very context sensitive. When something changes in the code surrounding the statement, the operation becomes invalid. As this is highly possible in such an operation, a different method is required.

Thus, for this special case, a special marker requires to be placed within the reference code. Although this is against the descriptive nature of the variability framework, it is required for the cases where a purely descriptive method becomes too complex and context sensitive. This new operation (ReplaceAtMarker) will search for a certain marker in the code (in the form of a label), and will replace the associated code.

7.2. Activities

Taxonomy of targeted source code
A full sized FALCON library specifically targeted to MIDP 2.0 consists of 10 Java classes, containing 4704 lines of source code. For *Battleships,* two of these classes aren't used. The standard implementation of the *Battleships* game has 8 classes, comprised of 8501 lines of source code. The number of source code lines mentioned include commentary.

The reference build of *Battleships* was targeted for the Nokia 6600, and was about 113 kb in total file size.

Scenario
The test case is based on a real-life scenario in which a game is to be ported to a more limited mobile device. Whereas *Battleships* was originally targeted for the Nokia 6600, the test case will focus on porting this game to an older Nokia 6100 model. Key differences between these devices are listed in the following table.

Capabilities	***Nokia 6600***	***Nokia 6100***
Model series	Series 60, second edition	Series 40, first edition
Screen resolution	176x208	128x128
Heap memory	Approx. 3 mb	200 kb
Java Profile	MIDP 2.0	MIDP 1.0
Multimedia support	MMAPI	Nokia specific (Nokia sound API, Nokia UI)
Max. file size	-	64 kb

Key differences between Nokia 6600 and Nokia 6100 devices

Two main issues in this porting process, are downsizing the game's distribution binary and memory usage in order to fit inside the Nokia 6100's maximum file size limit and heap space. Because of time constraints, the test case was focused on minimizing the game's file size.

Additionally, Gamica's FALCON library was required to support MIDP 1.0 and include code using Nokia specific API's.

Applied game variations
In order to support the Nokia 6100 with its limited maximum file size, certain non-essential elements of *Battleships* needed to be downsized or removed. These changes were implemented through variations using the framework.

The variations applied to *Battleships* were:

- Remove audio support
  To minimize file size, audio resources and code were to be removed.
- Change graphics locations
  Because of the smaller resolution, smaller graphics with different screen locations were required.
- Remove scrolling and interpolated paths
  Battleships utilizes a specific library class which creates interpolated paths used for smooth scrolling and sprite movement. When removing dependencies on this class, the class itself can be removed, thus further minimizing file size.
- Simplify graphics
  Examples: create a grid graphic manually using code, instead of using an image.
- Remove special effects
  Certain background animations were removed to speed up the game and lower the game's file size.

Some of these variations were anticipated in the earlier mentioned variation list in chapter *'Gathering detailed requirements'*. However, other variations were not previously been anticipated. This scenario presented a test to see if the previously determined operations were enough to implement these unexpected variations.

A complete port would also remove any unnecessary code from the FALCON library. Although some unused code is automatically removed through the usage of optimizers such as ProGuard[24], the creation of variations for the library's code would would result in a more thorough optimization.

Applied library variations
Variations applied to the FALCON library were limited to create support for J2ME profile MIDP 1.0, instead of the default supported MIDP 2.0. Because of time constraints this variation could only be implemented party, mainly dealing with converting functions related to displaying graphics. The focus here lied on converting the library's functionality, rather than decreasing it's size.

As was mentioned in the problem description, the current method to introduce variability into the FALCON library was based on preprocessing directives combined with xml/xsl transformations. In this part of the test case it was examined if the variations provided by the variability framework were any improvement over the usage of these directives, regarding code readability and maintainability.

Evaluation
During and after the test case, several informal interviews took place in which the experiences of the developers using the framework were evaluated.

Additionally, developers were asked to fill in a questionnaire after the test case. This questionnaire contained questions regarding the usability of the framework and the maintainability and readability of related source code when the framework is used. The questions asked in the questionnaire can be found in *Appendix D: questionnaire*.

The results of these activities can be found in the next sections, in which every issue mentioned at the beginning of this chapter is discussed. These sections are named: *'Support of operations*', *'Decreasing bytecode size'* and *'Ease-of-use, presentation, readability'*.

7.3. Support of operations

7.3.1. Results

During the test case it became apparent that the operations implemented in the variability framework weren't enough to implement all listed variations in the previous section *'Activities'*. Certain changes were required to the set of supported operations in order to implement the variations successfully.

However, this issue was relatively simple to solve by adding (additional) parameters to existing operations and adding new operations. These changes are further discussed in the next section *'Observations'*.

7.3.2. Observations

The following changes to the framework's operations were required in order to implement the variations listed in section *'Activities'*.

RemoveFieldOperation
The RemoveFieldOperation originally removes a field from a class and all references to this field within the same class. However, when removing references to resources, certain field references need to be removed across several classes. An additional parameter was required in which the range of the field removals could be determined.

Although it seems logical to always remove all field references across different classes, the relatively high complexity of this operation introduces an increased duration of source code processing. To optimize the framework's source code processing, developers can determine the range in which a field reference can be removed.

RemoveMethodOperation
Furthermore, for the implementation of the 'Remove scrolling and interpolated paths' variation, it was required that the number of parameters for a certain method were decreased.

This can be implemented by removing the original method and introducing a method with different parameters into the source code. The calls to the method were altered through a RemoveFieldOperation to make them compatible with the new method parameters.

However, a removal of a method includes removing all calls to this method. This means that the calls to the method should not be removed along with the method declaration.

Thus, the implementation of this variation required a new parameter for the RemoveMethodOperation, in which the removal of related method calls can be toggled.

SetFieldValueOperation

In order to determine the current screen resolution, mobile game developers can utilize certain J2ME features that provide this information. However, these functions do not always work correctly. For example, there are certain mobile devices which return a resolution of -1*-1 when retrieving this information using standard J2ME functions.

Because of this, the FALCON library usually places the device's resolution in hard-coded variables. When using the variability framework, this would normally mean that a specific OverrideFieldOperation is required for each device resolution. This in turn requires an increasing amount of variations, which can become increasingly difficult to manage.

To solve this, a new SetFieldValueOperation was introduced. This new operation can target a class field and providing it a new value. The contents of this value is retrieved from the earlier mentioned Global Device Repository (GDR) by using earlier mentioned XPath queries.

OverrideImplements
For certain variations regarding Java profiles and event handling, it became necessary to change the inheritance definition ('implements..'), similar to the OverrideExtends operation.

InsertBefore and InsertAfter operations
During the implementation of Java profile related variations for the FALCON library, it became apparent that the contents of a certain constructor required multiple changes throughout different variations.

As the standard OverrideMethodOperation only supports completely replacing a method's body, other techniques were required. For instance, adding new method calls to the constructor, filling these methods with variable source code and inlining them later on is a possible way of implementing such a variation. However, this required the introduction of several new methods and a separate variation which always inline these methods regardless of the actions of other variations. This implementation would therefor require several variations, new methods and other elements which increase the effort of managing the variations.

A more elegant and easier to use operation was required in order to add new contents to a method without having to introduce new methods and multiple new variations. A solution to this was proposed in the form of InsertBefore and InsertAfter operations. These operations can be targeted at methods, which insert a certain piece of code at the beginning or the end of a method.

InsertBefore and InsertAfter: similarities with AspectJ
The InsertBefore and InsertAfter operations are very similar to the before() and after() pointcuts of AspectJ. Additionally, AspectJ defines an around() pointcut, in which an entire jointpointpoint can be overridden. When using around(), an optional proceed() call is provided to return to a joinpoint's normal execution.

This around() pointcut is partly emulated by the OverrideMethodOperation. Most other constructions can be done by the earlier mentioned InsertBefore and InsertAfter operations.

Operation complete
Efforts have been made to create a relatively difficult scenario in which a game is to be severely downsized. Such a scenario requires a wide range of variations, which would test the operation-completeness of the variability framework. However, it is difficult to determine if the (newly) listed operations will be enough for future variations. On the other hand, it is possible to implement new operations into the framework if it is required.

7.4. Decreasing bytecode size

7.4.1. Results

Variations applied to *Battleships*
The table below displays the sizes on total game size after variations were applied.

Variation	*Operations*	*Effect*
Change graphics and locations	- Swap graphics with lower resolution versions - Change locations of these graphics	32,887 bytes smaller
Remove audio support	- Remove audio resources - Remove references to these resources - Remove calls to audio methods - Remove related menu options and game states	6,876 bytes smaller
Remove special effects	- Remove method and call to method in which a background animation is generated - Remove code which let lights blink in intro screen	1,059 bytes smaller
Remove scrolling and interpolated paths	- Isolate and replace all usage of InterpolatedPath object with static coordinates - Remove several if statements regarding waiting for scrolling to end	1,639 bytes smaller
Simplify graphics	- Isolate and replace image usage with several calls to graphics-draw methods.	3,554 bytes smaller

Applied variations and their influence on total game size

The listed variations managed to downsize *Battleships* to 70,312 bytes. While this is still not enough to support the Nokia 6100, results so far indicate that the game could be fit inside the 64 kb limit by applying more variations.

7.4.2. Observations

From the results given in the previous section it can be concluded that the framework is able to introduce variability, while at the same time decrease total bytecode size. Because virtually no overhead is introduced, bytecode downsizing can be performed effectively.

These same results show that variations which only change source code don't have a significant effect. Variations which influence other resources such as images and audio seem to have the most effect. However, in a situation where every saved kilobyte counts, variations that only remove code can still be considered relevant.

Missing resource management features
Variations which require the actual removal of several game resources from the game's distribution binary revealed a missing feature of the framework regarding resource management.

As was mentioned in chapter *'Gathering detailed requirements'*, game resources such as audio and image files are stored in a single binary file. When porting a game to another device requires a different set of resources, this resource file needs to be regenerated specifically for that port.

It was assumed that Gamica's current solution (generation of the resource file by using an Ant script) could be slightly altered outside the framework to support these different resource files. During the test case this assumption was proven to be incorrect.

The assumption was based on the idea that game resources are solely dependent on screen resolution. However, the variations applied within the test case were also weren't only related to screen resolution, but also to memory specific issues.

For instance, a device with the same resolution as the Nokia 6100 could very well have a higher maximum file size, allowing certain additional graphics files to be included. This means that the addition or removal of these files aren't related only to the screen resolution, but to other variability factors as well.

Proposed solution
Currently, variations can only be apply changes to Java source code. In order to improve support for variations which require changes to game resources, the framework should also be able to influence the creation of the earlier mentioned resource file.

In order to provide some consistency when creating variations, an ideal solution would be to implement the resource changes in the same way as the regular source code variations. To implement this, the generation of the resource file could be done through specifically formatted Java source code. This Java code could be changed through regular variations, and later be interpreted by the framework to generate the resource file.

This solution however has not been implemented or tested during this research because of time constraints. Because of this, the solution remains target for further research.

7.5. Ease-of-use, presentation, readability

7.5.1. Results

Although the developer's experiences with the framework was limited, the experiences with the framework thus far resulted in the following opinions.

Debugging, finding bugs, fixing bugs
Because all existing Eclipse functionality regarding debugging was still in place, debugging wasn't seen as a problem. Finding and fixing problems created by variations weren't seen as problematic as well.

There are however some issues with hot code replacement, in which code can be replaced while running a game (which can be used as a debugging effort). Both developers have observed a relatively slow compile process when using the framework, which makes hot code replacement less effective.

Additionally, one developer had problems with seeing the difference between a variation, reference source file and generated source file. It was suggested that creating small differences in the presentation of these types of files could resolve this issue.

Readability game source
According to the developer who was in charge of creating variability for *Battleships*, readability of the reference source code was slightly decreased because of added methods and markers to achieve variability. Especially methods that are inlined later on and markers that reside within the reference source code are found to cause some confusion.

One developer proposed a policy in which these added methods and markers are specifically commented. This to mitigate any confusion and to ensure that future developers understand their existence.

Readability library source
The developer who created variability for the FALCON library had a somewhat different view. The implementation of variability in this library was previous achieved by using several preprocessing directives. These directives severely decreased the readability when a high number of variations and code insertions were applied.

From this previous experience, the developer saw a great improvement in readability when using the variability framework. It was found that the framework provided a clearer overview of variations and better options to structure the variations and related operations. Furthermore, basic Eclipse features that were previously disabled because of using the tags, like debugging, code highlighting, error detection etcetera were again possible with the framework. This was seen as a great improvement over the previous situation.

Managing variations
Managing variations was found to be relatively easy, but only if a decent naming of variation sets (ordered in packages) was applied. One developer noted that an overview of which variation was actually applied for a combination of language-device-channel was preferred.

Another developer cited that it was difficult to tell the different types of files apart (generated sources, reference sources, variation descriptions). The previously mentioned presentation issues and possible solution apply here as well.

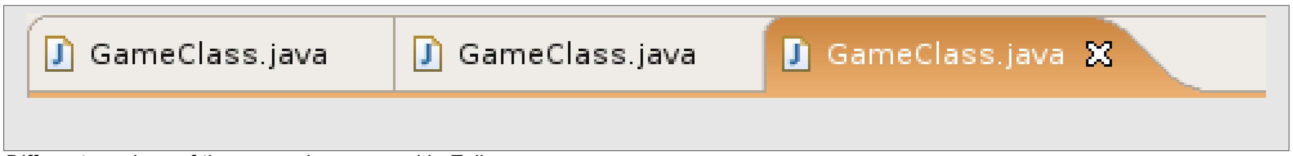
Modularity
Source code responsible for key input and displaying graphics were previously done in the same class, mainly to minimize bytecode size and optimize execution time. By structuring the required operations into variation sets, the responsible developer was able to introduce a certain modularity in the code without needlessly increasing the total bytecode size. Additionally, the developer found that managing the different pieces of code became easier.

Encountered issues and problems
There were also some issues noted regarding the usage of the framework.

Display applied variations
One missing feature that frustrated the developers, was that the framework didn't supply a list of variations that were applied for a certain combination of device, channel and language set. Although this list could be obtained manually, an automatic display of applied variations was preferred.

Telling apart different types of source code
Another issue that came up was that developers had trouble telling apart different parts of source code. For instance, when a developer had several versions of the same class opened in Eclipse it was difficult to determine which version came from the reference source, generated source or variability descriptions. This

scenario is shown in the image below.



Different versions of the same class opened in Eclipse

One developer proposed color coding the background of source file presentation elements, in order to tell these elements apart.

Speed and instability
Because of the prototypic nature of the framework implementation, little attention was given to speed and memory optimization of the framework itself. This lead to a significant slow-down of compiling and building source code. And because the framework currently requires large amount of memory for certain variations, the overall stability of Eclipse was negatively influenced as well.

Handling of 'outside elements'
For several variations it was required that code introduced by a variation consisted of relations with objects and methods that were defined outside the variation definition itself. This presented a problem as Eclipse's source viewer generated errors because the referenced methods and fields (the 'outside elements') weren't known in the context of the variation.

This issue was partly solved by adding special @Ignore annotations to these 'outside elements', which were ignored by the variation operation parser. A more ideal solution would be to always include all methods and fields of a targeted class inside the variation, but making them invisible for the developer. When such an element is mentioned in an operation (like an override or remove), the element could be automatically removed from the hidden part of the variation.

Overall
In overall, the questioned developers found that the framework was easy to use and effective in its usage. One developer found that the framework did have a steeper learning curve when compared to the earlier mentioned preprocessing technique. Although using preprocessing was found to be easier to learn, it's effectiveness and readability decreases when introducing a certain amount of variability. According to the developer, the steeper learning curve of the framework earned itself back in terms of clear structuring and readability of variations and related source code.

7.5.2. Observations

The overall opinion of developers regarding the framework's ease-of-use, presentation and maintainability were positive. There were however, some differences in opinion between both developers regarding readability.

Readability
The developer who worked on variations of *Battleships*, found that the readability of its source code was slightly decreased when using the framework. On the other hand, the developer responsible for library variations saw an increase of readability.

This can be explained by the different types of source code of the game source and library source. Source code of *Battleships* was constructed using traditional object oriented structures. Whereas the library contains numerous preprocessing directives, significantly decreasing its readability.

From this difference, it can be stated that the framework's readability lies below traditional object oriented structures. However, it is found to be an improvement over previously used preprocessing directives. Additionally, the availability of Eclipse's code presentation and debugging features further increased code readability.

Presentation issues
Although the overall opinion was positive, some issues were raised regarding presentation and other issues.

Because these issues have a negative impact on the efficient and effective usage of the framework, they should be solved in later development cycles.

7.6. Summary

In summation, it can be determined that the framework does have the potential to meet the challenges described in the problem description. Although the variations targeted in the test case do not represent the complete spectrum of possible variations, it does prove that the framework can be used effectively in a real-life scenario.

However, care must be taken when implementing the variations. Because variations can not only influence the original source code but the effects of other variations as well, variations are required to be structured and implemented carefully. This to prevent variations to impede on each others territory and creating invalid or incorrect code.

Furthermore, additional work is required to increase the effectiveness of the framework. Especially regarding resource management and presentation of source code and variations.

8. Conclusions

The research described in this thesis has tried to find a balance between introducing variability in a mobile J2ME application while minimizing the introduction of additional overhead to maintain this variability. Additionally, keeping source code readable and maintainable was also a focus point on this research.

In this chapter, the conclusions regarding this research are described by answering the questions asked in the problem description.

8.1. 'Is AOP efficient enough?'

Current implementations of Aspect Oriented Programming introduce a significant amount of overhead mainly regarding total file size, which makes it unusable in the context of mobile game development. The main cause of this, is that these implementations attempt to apply variability operations on a wide variety of situations, which in most cases requires the inclusion of several additional methods and classes.

However, this doesn't mean that Aspect Oriented Programming itself is inefficient. By creating an implementation which focuses on requirements enforced by mobile game development this inefficiency is greatly minimized.

This efficiency is achieved by only supporting a limited set of variability operations, which are applicable for a specific number of cases. These operations do however have a certain dependency on code style and structure, for which the operations are tailored.

Thus, answering the question *'Is AOP efficient enough?'*: not in current implementations, but it can be made efficient by specifically creating an implementation which focuses on efficiency.

8.2. 'Which exact variations should be supported by the AOP solution?'

A list of variations were derived from known common variations that exist between mobile devices, and certain issues the Gamica development team has been confronted with in the past.

This list of variations can be found in chapter *'Determining typical variations'.*

8.3. 'Can required variations be implemented using AOP and how?'

Operations
A set of common operations was created which can be used to implement the variations. These are mostly similar to standard object oriented solutions, however their implementations are optimized so that the end result doesn't introduce any additional classes and methods. The list of operations can be found in chapter *'Summary of required operations'.*

During the test case however, it became apparent that the operations listed in the previously mentioned chapter weren't enough for certain variations. These additional operations are described in section '*Further required changes to operations'.*

Technical approach
Because bytecode instrumentation was deemed too restrictive in order to execute the operations, program transformations are done at a source code level. Using Eclipse's features to alter source code using Abstract Syntax Trees, a custom plug-in was created in which developers can manage the operations and the resulting source code.

More details on the actual implementation of the resulting variability framework can be found in chapter *'Proof-of-concept'.*

8.4. 'Can the AOP solution be used within Gamica's development process, or what changes are required to this process to make it possible?'

As the implementation could be used in conjunction with Eclipse and supports hot-code replacement, technical requirements regarding Gamica's development process were met.

Questions regarding if the implementation could actually implement the variations and if the end result remains easy to manage for developers were answered through a case study. In this study, a proof-of-concept implementation was used to introduce variability on both a game currently in development and Gamica's in-house developed library.

Case study
This case study revealed that the framework did fit within Gamica's development process regarding source code transformations. Because transformations were directly related to device and channel properties, it became possible to create and manage generic variations which can be reused when new devices are released. This made it easier for Gamica to manage different builds and variations of their games and library.

Shortcomings
However, the framework did fall short regarding the management of varying resources. When certain game builds require a different set of resources (such as low-detailed images for limited devices), the framework didn't provide any functions to implement this. This decreased the effectiveness of the framework. More details regarding this and other issues are described in section *'Decreasing bytecode size'* of chapter *'Case study'*.

Additionally, certain presentation issues prevented optimal usage of the framework. These issues are further detailed in section *'Ease-of-use, presentation, readability'* of chapter *'Case study'*.

8.5. 'Can Aspect Oriented Programming be applied to introduce variability in J2ME games?'

The main question asked in the problem description, can be answered as such: 'Yes, but carefully'.

Granularity
As was previously discussed in this thesis, the solution provided by this research is strongly related to code style and structuring. The reason from this originates from the strict requirements regarding minimizing introduced overhead. the applicability of the variability framework was lessened. This requirement makes it necessary to include transformations on a low level within the source code. These 'low level' transformations include affecting switch statements, specific lines of code within a method, if statements etc. Because of this level of granularity, the effectiveness of the solution becomes depended on code style. A detailed discussion of this topic can be found in the discussions section: *'The illusion of obliviousness'*.

Structuring
Furthermore, variations need to be implemented carefully to prevent any unintended effects to the original source code and transformations of other variations. This requires the programmer to carefully create variations that don't impede on each others territory or create invalid or incorrect code.

Readability and maintainability
One challenge regarding the introduction of variability, as was described in the problem description, was readability and understandability of the variations and their related source code. The method used previously by Gamica (preprocessing directives combined with xml/xsl data) introduced a significant decrease of code readability. One of the goals of the variability framework was to improve this situation.

The test case revealed that the perceived readability and maintainability was increased in this regard. But observations to include that the variations are required to be applied cautiously. As it is very easy to implement variations that break code logic or introduce errors, developers are required to use caution in creating and managing these variations.

Because the test case was done in a limited amount of time, any effects of long term usage of the framework could not be measured. Further research would be required to determine if the proposed solutions still maintain an acceptable level of of readability and maintainability in a longer timespan.

8.6. Contributions

The research described in this thesis provide an alternative method of achieving variability whilst keeping any introduced overhead to a minimum. Although the solutions provided in this research are very dependent of a certain code style and structuring, the described framework could be utilized for another custom set of operations which can be applied for other code styles.

Furthermore, the proposed solutions provides a way of structuring these variations in a maintainable way, whilst providing a relatively high degree of readability amongst related source files.

This thesis also mentions some of the shortcomings of Aspect Oriented Programming and it's current implementations. Also, certain shortcomings regarding the usage of bytecode instrumentation are discussed.

9.Further research and discussions

Using addition
In this research, a decision was made to base variability operations on existing code because this matched Gamica's game development process. But it could be interesting to see how the variability framework would look like if variations were based on empty class skeletons, in which code is inserted through different modules. Although this might introduce overhead in order to link these modules together, further research in this area could improve reuse of certain variable code and code structures.

This method was partially used when creating the library variations described in the chapter *'Case study'.* But a more thorough experimentation using this method could shed more light on how effective the method can be utilized.

Defining a code style
Mentioned in previous chapters and further discussed in section '*The illusion of obliviousness*', successful usage of the described variability platform relies on code style and structuring. Further research into how this code style would look like and what kind of structuring works best with the variability framework could be utilized to further enhance the usage of the framework.

9.1. Discussion: The illusion of obliviousness

The theory
The research described in this this thesis has tried to find a balance between introducing variability in a mobile J2ME application while minimizing the introduction of additional overhead to maintain this variability. Additionally, keeping source code readable and maintainable was also a focus point on this research.

Through the use of specific manipulations structured similarly to aspect oriented programming, an attempt was made to achieve this balance. The main assumption based on this attempt was, that aspect oriented programming could be utilized to introduce variations in an application, without the need of preconditioning the application to support the variations.

As most other variability techniques require a certain application structure (or variability points) to apply variations, these methods are limited in use because all variation points need to be known in advance prior to development time. Using AOP, it might have been possible to keep the application code oblivious to any introduced variability, by describing variation operations separately from the application itself. By inserting and removing pieces of code, an application could support variability without any predefined structuring.

The reality
Because of the strict limitations in which mobile game are developed, fine grained manipulations are required to introduce any meaningful variability within a game. It is this granularity which makes some of these manipulations very context sensitive. This means that certain manipulations are heavily related on surrounding code structures and style. When something in this context changes later on, the manipulation quickly becomes invalid and can break game code syntax or intended logic.

To minimize manipulation dependency on context, manipulations should only be done on isolated structures, which can be altered with a minimum amount of context relations. In current Java language structures, the body of a method is ideally suited for this purpose. But determining which pieces of code should be isolated within a method to support variability, requires information which mostly isn't available at development time.

While the purpose of using AOP like structuring for introducing variability is to keep applications oblivious to changes and not having to predefine any variability points, in practice the application must become aware of them anyway. Although using certain operations to optimize the result of the manipulations minimizes any introduced overhead, the application doesn't stay oblivious to these changes. When isolating a certain piece of code for variability while the code structure itself doesn't really require a separate method, the application (and developer of the application) loses obliviousness to variability.

Statement
To summarize: in order to apply variability 'virtually everywhere' using AOP, certain low-level changes require isolating pieces of code that normally aren't required to be isolated in traditional development. This means that a traditionally developed application should explicitly isolate these pieces of code, in order to apply variability later. This defeats the assumption where code can stay oblivious to any later introduced variability.

This kind of variability framework can therefor not be seen as a silver bullet which can be used on any form of application, but requires a certain code style (depending on the required operations) to be successfully used. This makes the method not that different from other techniques that require preconditioning of source code. But the techniques mentioned in this thesis does has some methods to optimize the result, without introducing any variability overhead in terms of compiled byte size.

10.Evaluation

This chapter evaluates the conclusions and the process of the research described in this thesis.

10.1. Successes

Variability framework
The research described in this thesis resulted in an implementation of a variability framework, in which variability can be introduced in an application with a minimal amount of introduced overhead. Although this implementation is a bit rough around the edges (as it is mainly meant as a proof-of-concept), it has proven to be usable to introduce variability to games and an in-house developed library for Gamica.

As was mentioned in earlier sections of this thesis, the developers' overall response to the framework was positive. Although there are some improvements to be made at certain points, Gamica's main developer is confident that the framework can be of great value for Gamica's porting activities.

Communication
Because of the relatively small size of Gamica's organization, lines of communication between researcher and developers at Gamica were short. Ad-hoc discussions about issues surrounding manipulations, variations and code style were possible and small non-technical evaluations of variation descriptions weren't a problem as well. Additionally, Gamica's main developer manage to provide a considerable amount of time for whiteboard discussions and other talks regarding the research. These informal discussions did speed up the research and improved communications between researcher and developers. This also lead to a growing awareness of the functions and capabilities as well as acceptance of the framework within the company, which eased the execution of the test case considerately.

10.2. Misconceptions

In the initial phase of the research, it was assumed that the variability framework would be based on bytecode instrumentations. Possibly through existing AOP solutions (like AspectJ), or through manual instrumentations using specialized libraries such as BCEL or Javassist. A large amount of time was consumed in finding solutions to certain limitations encountered with these methods (these limitations are described elsewhere in this thesis), but ultimately they were proven to be unavoidable. Primarily, the requirement in which any additional overhead should be minimized invalidated the use bytecode instrumentation.

The decision to implement variations through descriptive source manipulations did offer the granularity required to manipulate games and the library using the defined operations. While not completely without bugs, the Eclipse platform's source manipulation framework did provide a decent ground for introduction of variability.

Not all goals and hypothesis are met in this framework, as a pure descriptive method of variability operations was found to be too restrictive and too context dependent in certain cases. Also, the assumption that descriptive operations can be applied to an application that stays oblivious to the variations was found to be incorrect as well (as is explained in section *'The illusion of obliviousness'*).

As an end result, the implemented variability framework doesn't have much in common with AOP as it was originally intended. It isn't used to place recurring code in various parts of an application, and the targetted source code doesn't stay completely oblivious to these changes. Therefor, it can be stated that AOP isn't that usable for porting games and introducing variability in a library, but a derived technique provided by the variability framework does offer some options.

10.3. Hindsight

In hindsight, far too much time was spent on implementation-specific issues, finding solutions to limitations of techniques and other technical difficulties. While these activities were required to implement a proof-of-concept and to become familiar with bytecode instrumentation, AOP and AST concepts, it did put a lot of stress and pressure on the research time table. The end result became very specifically targeted at Gamica's development process and code policies. To generate a more generic solution, more details about how code should be structured for the variability operations is required. As there wasn't any time left to research compatible code styles, this question remained unanswered.

11.References

[1] **AspectJ**
A seamless aspect-oriented extension to the Java^{tm} programming language
http://www.aspectj.org

[2] **JbossAOP**
JBoss AOP is a 100% Pure Java aspected oriented framework
http://labs.jboss.com/portal/jbossaop/index.html

[3] **Developing Adaptive J2ME Applications Using AspectJ**
Ayla Dantas, Paulo Borba
Informatics Center, Federal University of Pernambuco
Recife, Pernambuco, Brazil

[4] **Java Technology**
http://java.sun.com

[5] **ANT**
A Java based build tool.
http://ant.apache.org

[6] **Mobile Information Device Profile (MIDP 1.0)**
Sun Microsystems
http://java.sun.com/products/midp/

[7] **What's new in MIDP 2.0**
Sun Microsystems
http://java.sun.com/products/midp/whatsnew.html

[8] **About DOJA**
http://www.doja-developer.net/about/

[9] **The Java^{TM} Virtual Machine Specification, Second Edition**
Tim Lindholm
Frank Yellin
Sun Microsystems

[10] **Load-time structural Reflection in Java** (An overview of Javassist)
Shigeru Chiba
ECOOP 2000 -- Object-Oriented Programming, LNCS 1850, Springer Verlag, page 313-336, 2000.

[11] **BCEL**
The Byte Code Engineering Library is intended to give users a convenient possibility to analyze, create, and manipulate (binary) Java class files (those ending with .class).
http://jakarta.apache.org/bcel/

[12] **JavaCC**
A parser/scanner generator for Java
https://javacc.dev.java.net

[13] **Java Tree Builder (JTB)**
A syntax tree builder to be used with the Java Compiler Compiler (JavaCC) parser generator.
http://compilers.cs.ucla.edu/jtb/jtb-2003/

[14] **BeautyJ**
A source code transformation tool for Java source files
http://beautyj.berlios.de

[15] **SableCC**
An object-oriented framework that generates compilers (and interpreters) in the Java programming language.
http://sablecc.org/

[16] **Eclipse**
An open source community whose projects are focused on providing an extensible development platform and application frameworks for building software.
http://www.eclipse.org

[17] **'What is an Abstract Source Tree?'**
Eclipse FAQ
http://wiki.eclipse.org/index.php/FAQ_What_is_an_AST%3F

[18] **Higher-order abstract syntax**
F. Pfenning
C.Elliot
Carnegie Mellon University, Pittsburgh, PA

[19] **Document Object Model (DOM)**
http://www.w3.org/DOM/

[20] **Eclipse Java Development Tools**
http://www.eclipse.org/jdt/

[21] **XML Path Language (XPath) - W3C Recommendation**
http://www.w3.org/TR/xpath

[22] **Java ME Technologies**
http://java.sun.com/javame/technologies/index.jsp

[23] **#ifdef Considered Harmful, or Portability Experience With C News**
Henry Spencer
Zoology Computer Systems
University of Toronto

Geoff Collyer
Software Tool & Die

Summer '92 USENIX 1992

[24] **ProGuard**
Java class file shrinker, optimizer, and obfuscator
http://proguard.sourceforge.net/

[25] **Wikipedia: Template Design Pattern**
http://en.wikipedia.org/wiki/Template_method_pattern

[26] **Aspect-Oriented Programming**
Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes,
Jean-Marc Loingtier, John Irwin.
Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[27] **Aspect-Oriented Programming wth AspectJ**
Kiselev
Sams Publishing

[28] **AspectJ In Action**
Laddad
Manning Publications

[29] **Instrumenting Java Bytecode**
Jari Aarniala
Seminar work for the Compilerscourse, spring 2005
Department of Computer Science
University of Helsinki, Finland

[30] **J2SE platform at a glance**
Sun Microsystems
http://java.sun.com/j2se/1.5.0/docs/index.html

[31] **Java bytecode: Understanding bytecode makes you a better programmer**
Peter Haggar
Senior Software Engineer, IBM
IBM DeveloperWorks

[32] **Sun Microsystems**
http://www.sun.com

12.Appendix A: Device capability matrix

Phone model	*Profile*	*Heap memory*	*Max. file size*	*Proprietary APIs*	*Screen resolution*	*Color depth*	*Graphics file support*	*Audio file support*	*Number of simultaneous streams*	*Method of implementing animation*	*Handling of pause event*
Nokia series 40 1st gen: 6610	MIDP 1.0	195 kb	63 KB	NokiaUI	128x128	12 bit	.PNG	Own format, limited MIDI	1	Filmstrip	No pause event available
Motorola E398	MIDP 2.0	800 kb	128 KB	-	176x220	16 bit	.GIF .PNG	WAV MIDI MP3	2	Filmstrip	Standard Java impl.
Nokia Series 40 3rd gen.: 6270	MIDP 2.0	2 Mb	500 KB	-	240x320	18 bit	.GIF .PNG	WAV MIDI MP3	unknown	Filmstrip	No pause event available
DOJA 1.5 handset: NEC n341i	DOJA	800 kb	30 KB for classes, 100 KB for other resources	-	162x180	16 bit	.GIF	Own format (MFI 3.2, SMF 0i)	1	Separate graphics files	DOJA specific, 'forced' pause *

* MIDP based pause events send a pause event to an application, requesting to minimize its resource consumption. DOJA based pause events cut off several IO channels, forcing the application to minimize its resource consumption.

13.Appendix B: Device properties and channel requirements databases

As was mentioned in other chapters of this thesis, Gamica maintains a device capability database in which certain properties of devices are stored. The device capability database and distribution channel requirement database are based on this work, and were slightly refined for use in the variability framework.

Data regarding the capabilities and requirements are stored in XML formatted text files. Below are two simple examples of the formatting of each of these databases.

```
<gdr version="0.1"> <!-- GDR -->
      <capabilities>
            <capability id="j2me_screen"> <!-- CapabilityDefinition -->
                  <property name="width"/> <!-- Property -->
                  <property name="height"/>
                  <property name="colors"/>
            </capability>
      </capabilities>

      <device id="nokia_6600"> <!-- DeviceDefinition -->
            <capability refid="j2me_screens"> <!-- DeviceCapability -->
                  <property name="width" value="176" /> <!-- Property -->
                  <property name="height" value="208" />
            </capability>
      </device>
      <device id="nokia_4000"> <!-- DeviceDefinition -->
            <capability refid="j2me_screen"> <!-- DeviceCapability -->
                  <property name="width" value="1076" /> <!-- Property -->
                  <property name="height" value="208" />
            </capability>
      </device>
</gdr>
```

Example layout GDR.xml

```
<gcr version="0.1"> <!-- GCR -->
    <requirements-definitions>
     <requirement id="max_filesize"> <!-- RequirementDefinition -->
        <property name="max_filesize" /> <!-- Property -->
     </requirement>
      </requirements-definitions>

      <requirements>
            <requirement id="MAX_60K" refid="max_filesize">
                <property name="max" value="60000" /> <!-- Property -->
            </requirement>
      </requirements>

    <channels>
        <channel id="vodafone"> <!-- ChannelDefinition -->
            <device id="nokia_6600"> <!-- ChannelDevice -->
                <requirement refid="MAX_60K"/> <!-- RequirementReference -->
            </device>
        </channel>
    </channels>
</gcr>
```

Example layout GCR.xml

14.Appendix C: Example buildtargets.xml

```
<buildtargets version="0.1" id="battleships"> <!-- BuildTargets -->
      <channels>
        <channel id="vodafone"> <!-- ChannelReference -->
            <device id="nokia_6600"/> <!-- ChannelDevice -->
            <device id="nokia_6610"/>
            <device id="nokia_6640">
                  <requirement refid="MAX_60K"/>
             </device>
            <device id="nokia_6650"/>
        </channel>
        <channel id="preminet">
            <device id="nokia_6100"/>
            <device id="nokia_6600"/>
        </channel>
    </channels>

    <languages>
       <languageset id="nl-fr">
          <language id="nl"/>
          <language id="fr"/>
        </languageset>

        <languageset id="ch-tw">
          <language id="ch"/>
          <language id="tw"/>
        </languageset>

        <languageset id="en">
          <language id="en"/>
        </languageset>

    </languages>
</buildtargets>
```

Example content of buildtargets.xml

15. Appendix D: Questionnaire

Note that the proof-of-concept of the variability framework was codenamed 'SVMode' within Gamica.

This survey is a short questionnaire in which the opinion of application programmers is asked in terms of code maintainability and readability. To this end, several questions are asked relating to a software project *after* SVMode is applied to it (such as *Battleships* and the FALCON library).

Each statement listed below can be answered by highlighting the level of agreement with the statement. For each statement it is asked to give an explanation to why you agree or disagree with it. If possible, please explain the main factors which influenced your opinion regarding the statement.

Filling in this form shouldn't take longer than 15 minutes.

Any further questions about the statements can be directed to Sannie Kwakman, email: sankwak@instantstuff.net or mobile phone: 0623435902.

When a bug is encountered in the reference build, it is easy to find and fix the problem. Why?	Completely disagree	Somewhat disagree	Somewhat agree	Completely agree
When a bug is encountered in a variation build, it is easy to find and fix the problem. Why?	Completely disagree	Somewhat disagree	Somewhat agree	Completely agree
It is difficult to let someone else take over this project. Why?	Completely disagree	Somewhat disagree	Somewhat agree	Completely agree
Letting someone take over this project became harder after SVMode was applied tot the project. Why?	Completely disagree	Somewhat disagree	Somewhat agree	Completely agree

Game/Library code was better readable before using SVMode Why?	Completely disagree	Somewhat disagree	Somewhat agree	Completely agree
Extending the features of the game/library became easier after applying SVMode to the project Why?	Completely disagree	Somewhat disagree	Somewhat agree	Completely agree
Managing different variations is difficult using SVMode. Why?	Completely disagree	Somewhat disagree	Somewhat agree	Completely agree

My experience with SVMode was:
Why?

What elements of SVMode do you wish to see improved? And in what direction?