

Techniques for De-fragmenting Mobile Applications: a Taxonomy

Damith C. Rajapakse

School of Computing, National University of Singapore
damith@comp.nus.edu.sg

Abstract

Fragmentation, in the context of mobile applications, is the inability to "write once and run anywhere". Fragmentation increases the effort required in all aspects of application development. This paper analyzes various aspects of fragmentation, and presents a taxonomy of techniques used to combat such fragmentation. Our aim is to establish a set of useful terminology for the benefit of researchers and practitioners working in this area.

1. Introduction

Fragmentation is the term used in the industry to describe the inability to "write once and run anywhere", often resulting in multiple versions of an application. More formally, we define fragmentation as the "inability to develop an application against a reference operating context and achieve the intended behavior in all operating contexts suitable for the application". Further, we define the *operating context* (OC) for an application as the "external environment that influences its operation". Therefore an OC is defined by the hardware/software environment in the device, the user, and the environmental constraints introduced by various other stakeholders such as the network operator. While fragmentation can affect any type of application, this paper focuses on the fragmentation of mobile applications. Note that by "mobile applications" we mean *installed* applications (an application installed on the mobile device itself), and not server-side applications such as SMS applications¹ and Mobile web applications².

Fragmentation is caused by the diversity of OCs (see Figure 1 for an illustration). In Section 2 we describe how one OC could differ from another, resulting in fragmentation. While users, developers, distributors, carriers and device manufacturers are all affected by fragmentation, this paper looks at fragmentation from the point of view of an

organization developing mobile applications. In section 3 we describe how fragmentation affects various aspects of mobile application development. As fragmentation is a big problem in the industry today, a number of techniques have emerged to combat it. We call them *de-fragmentation* techniques. Section 4 presents a taxonomy of de-fragmentation techniques, based on the basic approach each one uses to tackle the problem. This taxonomy was inspired by the work of practitioners [3] and later refined based on further feedback from practitioners (as acknowledged in Section 7). Where appropriate, we refer to industry tools to illustrate each approach. Comments about related work, conclusions, and future directions are given at the end of the paper.

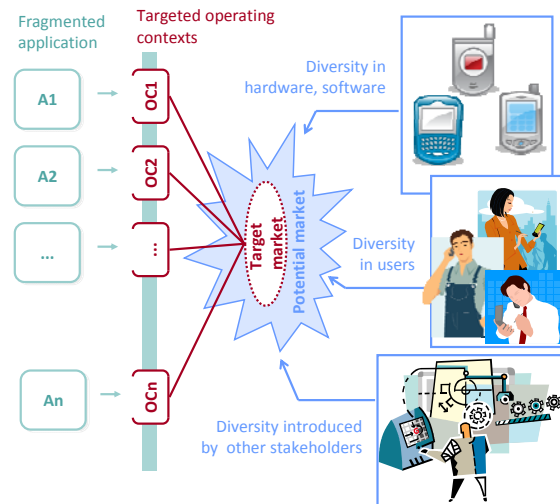


Figure 1. Fragmentation overview

2. Causes of fragmentation

By definition, fragmentation is caused by the diversity of operating contexts (OCs). One operating context may differ from another in the following reasons:

- **Hardware diversity** of the device, such as differences in screen parameters (size, color depth, orientation, aspect ratio), memory size, processing power, input mode (keyboard, touch screen, etc.), additional hardware (camera, voice recorder), and connectivity options (bluetooth, IR, GPRS, etc.).

¹ A server-side application accessed using a mobile device, using SMS as the mode of communication

² A web application accessed over the Internet, using a web browser in a mobile device.

- **Software diversity**, which may be a result of platform diversity or implementation diversity:
 - **Platform diversity** is caused by factors such as differences in platform/OS (Symbian, Nokia OS, RIM OS, Android, BREW, etc.), API standards (MIDP 1.0, MIDP 2.0, etc.), optional/proprietary APIs, variations in access to hardware (e.g., full screen support), maximum binary size allowed, etc.
 - **Implementation diversity** is caused by factors such as quirks/bugs in implementing standards.
- **Feature variations**, such as light version vs full version
- **User-preference diversity**, in aspects such as the language, style, etc., or accessibility requirements
- **Environmental diversity**, such as diversity in the deployment infrastructure (e.g., branding by carrier, compatibility requirements of the carrier's back-end APIs, etc.), locale, local standards.

As we can see from the above, one OC can differ from another due to many factors. Let us call these factors *fragmentors*. i.e., a fragmentor is a factor, diversity of which causes fragmentation. The fragmentation of mobile applications is often referred to as *device* fragmentation, because most of the fragmentors can be traced to a particular device model. However, this is a misnomer, as factors outside the device (e.g., branding by carrier) too can cause fragmentation.

Since it is the diversity that drives fragmentation, a closer look at diversity may provide us with clues as to how to deal with fragmentation. It is our opinion that diversity can be either *essential* or *accidental*.³

- **Essential diversity** is the diversity that differentiates a product/service in some useful manner. Such diversity is intentional and often unavoidable. For example, users will continue to differ in their preferred size for a device, and the device manufacturers will continue to differentiate the devices in terms of size.
- **Accidental diversity** is the diversity that - does not serve any useful purpose, is often introduced unintentionally, and is often avoidable. For example, diversity due to API implementation bugs/quirks is unintentional, avoidable, and does not serve any useful purpose

Fragmentation is often associated with JavaME (Java Mobile Edition) applications, but it is also applicable to non-JavaME applications. Theoretically, a JavaME application is able to run on any Java-enabled mobile device. This means a JavaME

application can target a much wider range of OCs as compared to non-Java applications, exposing it to more diversity. As non-JavaME applications (e.g., native applications for Symbian platform) are created for a smaller range of devices, they are exposed to less diversity. While a JavaME application has to run on platforms developed by many vendors, a typical non-JavaME application will run on a platform implemented by a single vendor or a small number of vendors (e.g., Symbian). This means JavaME applications have to face more implementation diversity, as compared to non-JavaME applications. However, developers may still have to develop a JavaME equivalent as well, if a wider range of OCs is to be targeted.

3. Effects of fragmentation

Fragmentation, and the subsequent de-fragmentation, complicates all disciplines⁴ of a mobile application project. Some examples are given next.

- **Business modeling:** Business analysts have to determine the optimum set of OCs for the application to target. Questions to be answered include “Is operating context OC1 suitable for application A1?” and “Is it worth porting A1 to OC1?”
- **Requirements management:** If the interaction between the actor and the application is OC-dependent, it complicates the use case specification by introducing a vast number of exceptional/alternate flows.
- **Analysis and design:** The system architecture, and the detailed design, should be able to accommodate not only all the variations demanded by different OCs targeted at the time, but also future OCs the application will be exposed to during its lifetime.
- **Implementation:** Implementers need to optimize the application to all the targeted OCs. Questions to answer include “What do I have to do to fit application A1 to fit operating context OC1?”, “How does OC1 differ from OC2?”, and “Which OCs can be served by a single version of the application?”
- **Testing:** The application need to be tested for all targeted OCs. It is usually not enough to test on device emulators, as real devices on a real network sometimes behave differently from the emulators.
- **Project management:** Having to accommodate new (and unexpected) OCs in the middle of a project complicates project scheduling.
- **Configuration and change management:** Having multiple versions of an application (to suit multiple OCs) clearly impacts this discipline. New devices entering the market will increase the version count, while evolution of the platform software may require substantial changes to the existing versions.

³ This classification borrows from Fred Brooks' seminal book *The Mythical Man-Month*, which discusses “essential difficulties” and “accidental difficulties” of software development

⁴ *disciplines* as defined in the IBM Rational Unified Process

- **Environment:** The software process has to be augmented to cater for additional complications introduced by fragmentation. For example, additional tools will have to be brought into the process, to tackle various fragmentation issues.

As a result of these complications, fragmentation increases the required effort in almost all aspects of the software life cycle, driving up the cost, and lengthening the time-to-market. Other side-effects are:

- It could reduce the quality of the product - The additional complexity of maintaining a large number of versions could increase the probability of bugs. Cost considerations may tempt developers to release applications that behave in sub-optimal ways for certain OCs (E.g., an application may work well for certain screen sizes, but may appear distorted in certain other screen sizes).
- It could narrow the target market - Cost considerations may force the application vendors to target a smaller market than the actual potential market it could target otherwise (see Figure 1).
- It hinders the growth of the mobile application market, by acting as a barrier-to-entry for new entrants - This is because creating a mobile application to fit a wide variety of OCs requires a much higher effort and a better expertise, compared to a desktop/web application.

4. A taxonomy of de-fragmentation techniques

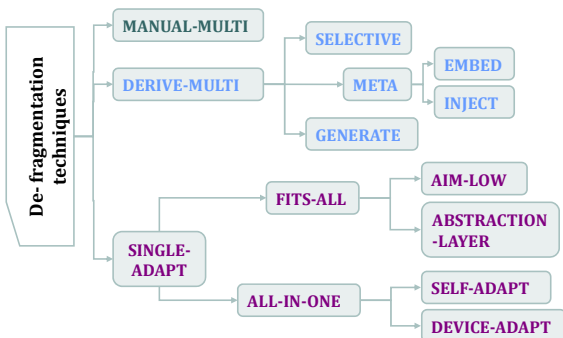


Figure 2. The complete ontology

One obvious way to reduce fragmentation is by eliminating accidental diversity. Measures such as better standardization (e.g., less optional APIs, more detailed specifications), stricter enforcing of the standards (e.g., using API verification initiatives, Technology Compatibility Kits) can help in this regard. Major players in the mobile application industry such as platform vendors, device manufacturers, and carriers have a critical role to play in this front of the war against fragmentation. One such effort in the JavaME arena is the Mobile Service Architecture [7].

On the other hand, essential diversity will be much harder, if not impossible, to avoid. The pragmatic response here is to find ways to reverse the resulting fragmentation. This is called *de-fragmentation* [3]. Note that de-fragmentation is NOT eliminating

diversity. Rather, it is the process of making the application behave as intended, on all target OCs.

In this section, we present a taxonomy of de-fragmentation techniques. Figure 2 illustrates this taxonomy in its current state. Each technique will be explained in detail in the subsequent subsections. A combination of the above approaches can be used within a single application too, using one of the approaches to manage each OC-specific variation.

4.1 The MANUAL-MULTI approach

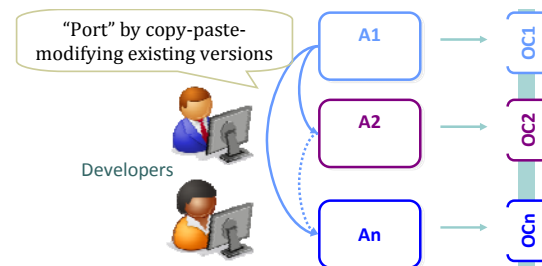


Figure 3. The MANUAL-MULTI approach

The most primitive way of de-fragmenting is to manually develop distinct versions of the application to suit different OCs. We call this approach *MANUAL-MULTI*. Figure 3 illustrates this approach, where A_1, A_2, \dots, A_n are different versions of the application A , customized to fit operating contexts OC_1, OC_2, \dots, OC_n respectively. These distinct versions will be largely similar, but also different in subtle ways, as a result of subtle variations in the OCs. Copy-paste-modify techniques are commonly used to “port” the application to various OCs. *MANUAL-MULTI* approach results in duplication of work in many aspects of software development (e.g., fixing the same bug in hundreds of different versions). The following two alternative approaches try to minimize such extra effort:

1. Derive OC-specific versions from a single code base (we call this approach *DERIVE-MULTI*)
2. Use a single version to serve multiple OCs (we call this approach *SINGLE-ADAPT*)

4.2 The DERIVE-MULTI approach

In the *DERIVE-MULTI* approach, we derive OC-specific versions of the application from a single code base. While this still results in multiple versions of the application, there is only one code base to work on, and therefore the effort required may be less than in the *MANUAL-MULTI* approach. In particular, we no longer need to manually maintain duplicate copies of the same source.

An example tool that supports the *DERIVE-MULTI* approach is the NetBeans Mobility Pack [8] (a JavaME mobile application development environment that comes as an extension to the popular NetBeans Java IDE). It uses a concept called *project configurations*, where a single application can have multiple project

configurations, one for each different version we want to derive.

The DERIVE-MULT approach can be further subdivided into the three approaches *SELECTIVE*, *META*, and *GENERATE*.

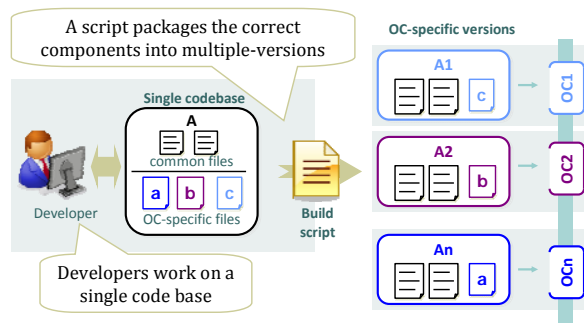


Figure 4. The *SELECTIVE* approach

The *SELECTIVE* approach (Figure 4) localizes variations into interchangeable components (e.g., classes, files, etc.) and uses a build script (or a linker) to create one version for each OC, picking out only the components required for that particular OC. This approach is frequently used when including images of different resolution to fit different screen sizes. An example of this approach can be seen in the J2ME Polish tool [6]. For instance, we can put an image file in the *resources/ScreenSize.240+x320+* folder, and J2ME Polish will include this image for devices with a screen size of at least 240x320 pixels.

The *META* approach uses meta-programming (and similar code manipulation techniques) to *specify* how to derive OC-specific versions of the application. There are two ways of achieving this: the *EMBED* approach and the *INJECT* approach.

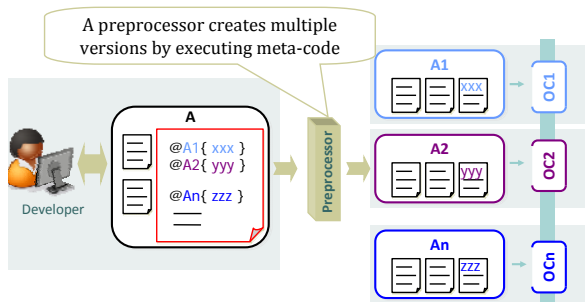


Figure 5. The *EMBED* approach

The *EMBED* approach embeds OC-specific variations in the source files using meta-programming directives/tags. A preprocessor derives multiple versions by processing these directives/tags. An example of this approach can be seen in NetBeans Mobility pack, which uses a concept called *preprocessor blocks* to specify OC-specific code segments. The example preprocessor block given in Figure 6 (adapted from [8]) is used to derive two different versions of the application, one for devices having 128x128 screens, and one for devices having 176x182 screens.

```






```

Figure 6. A NetBeans Mobility Pack preprocessor block

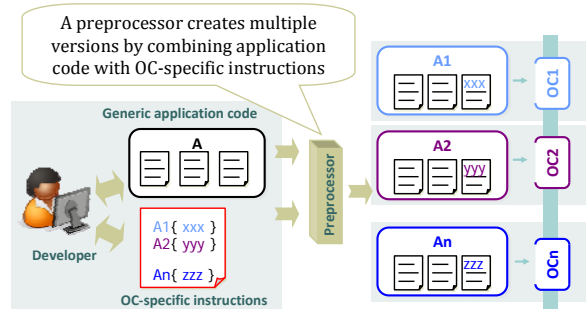


Figure 7. The *INJECT* approach

The *INJECT* approach requires the developer to write the OC-specific instructions separated from the application code. For example, Tira Jump [9] (a tool for developing mobile applications) uses aspect oriented programming techniques to achieve such an effect. It lets developers write the application code against a reference OC and derives OC-specific versions by “weaving” OC-specific variations into it.

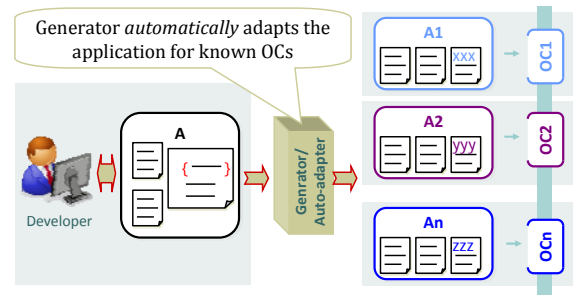


Figure 8. The *GENERATE* approach

The *GENERATE* approach automatically generates multiple versions using an intelligent generator that knows how to adapt a generic application to suit a specific OC. Instead of merely following instructions supplied by the programmer (as in the *META* approach), the generator uses its inbuilt knowledge in the generation process, requiring less manual coding. The feasibility of such fully automatic generation is rather limited, and we expect such generators to be limited to a narrow mobile application domain or a narrow range of OCs. For example, alcheMo tool [1] promises to automatically generate BREW format applications from JavaME applications.

4.3 The *SINGLE-ADAPT* approach

The *SINGLE-ADAPT* approach builds a single version of the application that can work on multiple OCs. This approach can be further sub-divided into two: *FITS-ALL* and *ALL-IN-ONE*.

The FITS-ALL approach develops a one-size-fits-all application that sidesteps all variations between OCs. There are two ways to accomplish this, called *AIM-LOW* and *ABSTRACTION-LAYER*.

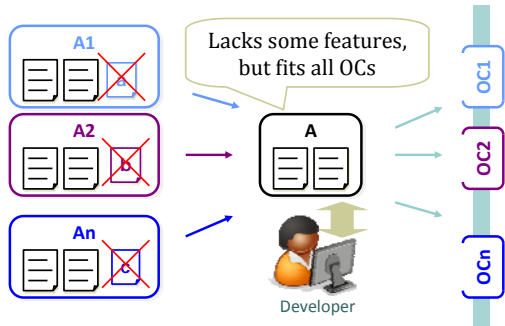


Figure 9. The AIM-LOW approach

The AIM-LOW approach (Figure 9) uses only what is common to all targeted OCs. For example, the UI will be designed to fit the smallest screen size of the targeted device range. This approach is sometimes referred to as the “lowest common denominator” approach.

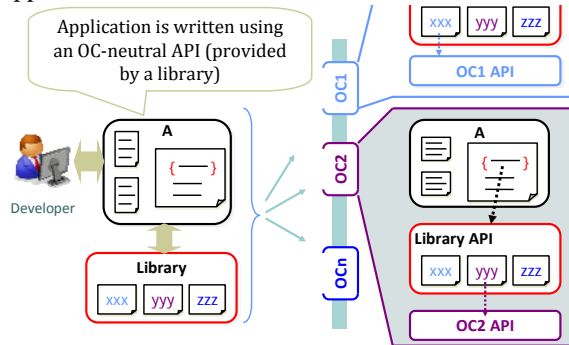


Figure 10. The ABSTRACTION-LAYER approach

The ABSTRACTION-LAYER approach (Figure 10), hides variations in the OCs behind an abstraction layer. This abstraction layer is usually a library (third-party or built in-house), and the application will be developed using the API of the library. Both the library and the application will be deployed on the mobile device, and it is the responsibility of the library to execute generic method calls from the application in an OC-specific manner. TWUIK [10] (a UI library for mobile applications) is one example tool that uses the *ABSTRACTION-LAYER* approach to write a single UIs that can adapt for multiple OCs.

The ALL-IN-ONE approach makes the software adapt at run-time to a given OC, using either the *SELF-ADAPT* approach or the *DEVICE-ADAPT* approach.

The SELF-ADAPT approach (Figure 11) makes the application programmatically discover information about the OC and adapt itself to the OC at run-time.

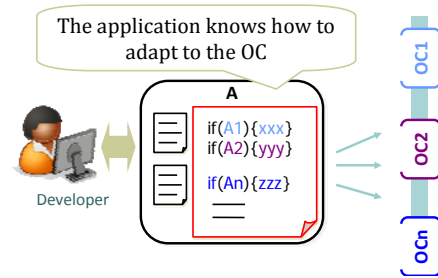


Figure 11. The SELF-ADAPT approach

```
Canvas c = new Canvas();
w = c.getWidth (); h = c.getHeight ();
if(w==128 && h==128)
    ballWidth=10;
else if(w==176 && h==182)
    ballWidth=16;
```

Figure 12. Example of SELF-ADPT

In Figure 12 we see an example code snippet written in *SELF-ADAPT* fashion. This single piece of code will work for both screen sizes 128x128 and 176x182. The difference between this and the *EMBED* example in Figure 6 is that *EMBED* will include either `ballWidth=10;` or `ballWidth=16;` (but not both) in each OC-specific version, while *SELF-ADAPT* will include all code in Figure 12, resulting in a bigger application.

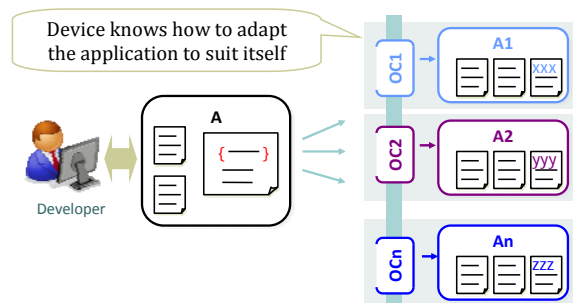


Figure 13. The DEVICE-ADAPT approach

The DEVICE-ADAPT approach (Figure 13) requires the application to be written in an abstract way, and the device decides how to adapt it to the prevailing OC, at run-time. This approach is commonly used when dealing with fragmentation in the UI part of an application, often with unsatisfactory results. In Figure 14 we see how the same calculator application appears differently on two different phone emulators, after it has been adapted by the device.

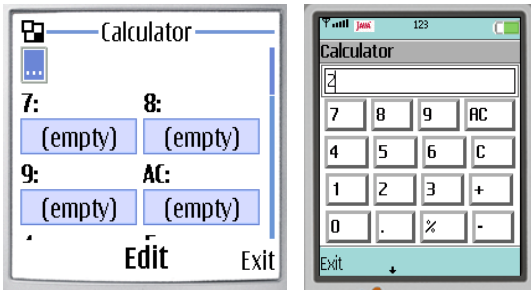


Figure 14. An example result from DEVICE-ADAPT

5. Related work

Fragmentation is one of the most talked about topics among practitioners (e.g., [3][4]). In academic research, fragmentation in the Mobile-Web has received frequent attention (e.g., [5]). Another related area is adaptable user interfaces. For example Gojas et al [2] describes a GENERATE type technique used to automatically generate UIs to fit different screens. The use of meta-programming to generate product lines is a well known technique, which could be adapted to de-fragment mobile applications. For example, Zhang and Jarzabek [11] shows how to use the XVCL meta-programming language (XVCL uses a combination of EMBED and INJECT approaches) to de-fragment a mobile game product line.

6. Conclusions and future work

In this paper we analyzed the fragmentation problem faced by mobile developers today. We defined the terms “operating context”, a concept central to the way we define fragmentation. We also explained our opinion of what it means to “de-fragment” an application, and contrasted it with eliminating diversity. As the major contribution of the paper, we presented a taxonomy of de-fragmentation techniques currently used in the industry, and used existing industry tools to illustrate each technique. Our future plans include a comprehensive evaluation of the techniques included in the taxonomy, to discover their strengths/weaknesses, to find synergies among them, and look for more effective alternatives. We shall continue to refine this taxonomy, based on interactions with the practitioners and our own experimentation.

7. Acknowledgements

Input from the following persons helped towards refining the material in this paper: Bhojan Anand and Nguyen Thi Tuyet Nhung (National Uni. of Singapore), Chris Abbott (DetectRight), Himath Dissanayake (OrangeHRM Inc), Kutila Gunasekera (Monash University), Jason Delpont (Paxmodept), Luca Passani (WURLF) Mihai Fonoage (Florida Atlantic Uni.), Reto Senn (Bitforge), Ruchith Gunaratne (hSenid Software Intl), Tom Hume (FuturePlatforms).

8. References

- [1] alChemo home <http://www.innaworks.com/alchemo>
- [2] Gajos, K, Christianson, D., Hoffmann, R., Shaked, T., Henning, K., Long, J. J., and Weld, D.S., “Fast And Robust Interface Generation for Ubiquitous Applications,”. Proceedings of the Seventh International Conference on Ubiquitous Computing (UBICOMP'05), 2005
- [3] JavaME: De-fragmentation Technical Overview and Design Guidelines Index, available at http://developers.sun.com/mobility/reference/techart/design_guidelines/overview.html
- [4] Lau, A., “ Fragmentation effect,” <http://www.javaworld.com/javaworld/jw-05-2004/jw-0524-fragment.html>
- [5] Liang, A., Guo, S., and Li, C., “Dynamic Mobile Content Adaptation Abstracting in Device Independent Web Engineering,” Global Telecommunications Conference, 2006. (GLOBECOM '06), pp. 1 - 4
- [6] J2ME Polish homepage <http://www.j2mepolish.org>
- [7] JavaME mobile Service Architecture, <http://java.sun.com/javame/technology/msa/>
- [8] Resolving JavaME Device Fragmentation Issues Using NetBeans 6.0 Mobility <http://www.netbeans.org/kb/60/mobility/javame-devicefragmentation.html>
- [9] Tira Jump home page <http://www.tirawireless.com>
- [10] TWUIK homepage <http://www.tricastmedia.com/twuiik/>
- [11] Zhang, W. and Jarzabek, S. “Reuse without Compromising Performance: Experience from RPG Software Product Line for Mobile Devices,” *9th Int. Software Product Line Conference, SPLC'05*, September 2005, Rennes, France, pp. 57-69