Variability Handling for Mobile Banking Apps on iOS and Android

Jens Bæk Jørgensen†, Bjarne Knudsen†, Lennert Sloth†, Johan Rugager Vase†, Henrik Bærbak Christensen‡
Mjølner Informatics A/S (†), Computer Science, University of Aarhus (‡)
Aarhus, Denmark
{jbj,bkn,lsl,jrv}@mjolner.dk, hbc@cs.au.dk

***Abstract*—During the last four years, Mjølner Informatics has developed mobile banking apps for 11 Danish banks on the two major platforms iOS and Android, in total 22 apps. We make two contributions with this industrial practice paper. In the first place, we describe the development of these apps with emphasis on issues regarding variability handling and maximizing reuse. Secondly, we discuss a number of lessons learned from this process. This includes consideration of how ready we are to possible future challenges regarding variability and observation of significant differences between the two platforms.**

I. Introduction

Mjølner Informatics is a small Danish software company that develops custom-made software solutions for Danish and international customers. In this paper, we consider an ongoing development project in which Mjølner develops mobile banking apps for 11 cooperating Danish banks, one iOS app and one Android app for each bank.

The apps provide standard banking features such as viewing account balances and account entries, transferring money, paying bills and trading stocks. The apps are popular in Denmark; they are used every day by hundreds of thousands of users. The apps present a main menu giving access to 15-20 feature areas. Each app has around 100 different screens, and tens of thousands of hours have been used on the development.

The apps integrate with a backend system developed and hosted by a bank service provider through a REST-based API. The backend encapsulates the majority of the domain functionality, thus the apps serve as client/presentation layer, e.g., user login, security, communication with the backend and proper state transitions through the large set of screens.

In this paper, we describe the software architecture of the 22 apps on the two mobile platforms with emphasis on variability handling. Of the apps for the 11 banks on a given platform, ten differ only in graphical appearance and textual contents. They share a large, common codebase to produce a framework/product line architecture, amounting to 99.9% code reuse, and about 50% reuse of other assets, e.g., graphics and texts. The 11th bank is a larger bank and its app provides a larger feature set as well as a slightly different domain model, compared to the other ten banks. Still, we achieve about 96% code reuse from the common framework for this bank.

We discuss a number of lessons learned from the project, and, among other issues, analyze how well variability management across a large set of similar apps is supported on the two competing mobile platforms; we also make proposals for improvement. Readers interested in the project from a process perspective may consult Bruun et al. [1].

II. Commonality/Variability Analysis

The variability space of 22 variants, 11 banks over two platforms, may seem intimidating, but as outlined above, the primary difference is style and appearance of user interface, whereas core banking functionality is identical, the large bank being the one exception to this rule.

To sum up, the following five aspects vary: (1) Mobile platform: The two major platforms, iOS and Android must be supported; (2) Text values: Apps present themselves with textual differences such as bank names and terminology. As an example of the latter, one bank uses 'branch' whereas the rest uses 'department' as term for a local representation; (3) Graphics: The visuals presented differ at a basic level between the banks; (4) Styling: The visuals all share the same workflow and page contents, but with different styling for the different banks; (5) Functionality: Ten of the banks have fully identical functionality, while the 11th bank has special requirements.

Variability can be handled by many means both in source code, in processes and in tool chains. In the following, we use the model by Nadi [2] that uses the terms *code space, build space* and *configuration space* to classify techniques. *Code space* denotes techniques that use source code to handle variability, such as parameterized variability, or polymorphism. *Build space* includes techniques that handle variability through the build process, such as a-build-file-per-variant, conditional compilation, or overwriting or modifying configuration files during the build process, such as GNU Autoconf, or Ant token filters during copy. Finally, *configuration space* includes techniques where variability is configured at run-time through configuration files that define simple key-value pairs, such as INI files, or cascading style sheets (CSS).

III. Project and Software Architecture

We maintain a project structure that enables us to share as much as possible between the apps on a given platform. However, common and shared functionality were developed individually for each platform, i.e., there was no cross-platform development for reasons we will explain in Section IV.D. Thus, in the discussion below, we focus on the architecture of shared assets (source code, resource files) as well as variability configuration on each individual platform.

978-1-5090-2131-4/16 $31.00 © 2016 IEEE
DOI 10.1109/WICSA.2016.29

Below we use the term *generic apps* for the apps that only differ in visual and text appearance (the ten banks) and use the term *specialized app* to denote the app for the large bank with additional functionality.

A. Project Architecture

The iOS apps are developed in a single project in Apple's integrated development environment Xcode, with a separate target for each of the 11 apps. The Xcode project contains all assets for all variants, both common as well as those particular to a given variant. Assets include source code and resources in the form of text and graphics. Each of the 11 targets define the complete app by referencing asset files that belong to a particular variant.

The Android apps are developed in Eclipse, and are organized with a separate application module for each of the 11 apps. The application modules have very little content and mainly serve as the required entry point for the apps. The shared assets are implemented in a library module, which is referenced by each of the application modules.

B. Handling the Variability in Texts

On both platforms, shared texts to be presented in the apps are defined in a shared text resource file. During build, the resource file is converted to the appropriate format for the platform.

On iOS, all the texts that are common for all banks are specified in a key-value formatted file, the Localizable.strings file, but as Xcode does not support overriding of keys with new values for a given variant, there is no built-in support for text variations. Instead, we have two ways of handling text variations, depending on the type of variability. For one of the banks, there is one single word that should be replaced ("department"), which is handled by a pre-compile script that does a simple find-and-replace. Larger text variations, typically the "about box" that describes a bank, its mission and values etc., are handled via another property file that is unique for the particular variant and thus only referenced by that bank's target. Both are classified as build space since the proper resource file is included in the app during the build process.

Android has a resource system based on XML files that are read at build time to generate the contents of the final deployment unit. These XML resources can be strings, layouts, styles, etc. The resource system makes it easy to overwrite single resources in different modules, which are using the same library. This feature, similar to the overwriting mechanism of CSS, is used to handle the variability in texts. The library module defines all shared texts with a unique key and assigns a default value for each. The application module can override these values by including a string resource with key value pairs for those texts that are different for a specific bank. During the build, the XML resources are merged for the specific bank. This technique covers both of the approaches described above for the iOS platform. Thus, on Android, we handle variability in texts by build space.

C. Handling the Variability in Graphics

On iOS, all graphical content is contained in asset catalogs, which is a structured way to handle different image sizes etc. provided by Xcode. These asset catalogs can be referred to by targets in the same way as source files. A given app therefore has a shared asset catalog defining shared resources and an asset catalog with assets for that particular variant. Again, it is not possible to overwrite anything from the shared resource. Therefore, a special requirement to a graphical element in one bank forces all asset catalogs for the other variants to have a copy of that element.

On Android, default image files are placed in the shared library module. Files with identical names and sizes are placed in each application module. This makes it possible to overwrite the images that differ in a given variant during build time.

Thus, on both iOS and Android, we handle variability in graphics by build space.

D. Handling the Variability in Styling

On Android, the technique used to handle variability in styling is similar to the technique used to handle variability in texts. A default style document defines common styling in the shared library module, and in each application module, any style property that differs can be overwritten. During build, the style documents are merged into one document that is packaged into the app.

Unlike the Android platform, there is no "style" concept in iOS. All styling must be set up either in Xcode's Interface Builder or in code. With regard to varying styles, we therefore only have the options to copy the UI builder files for each app, or to manage all variability in styling in the code. To avoid maintaining multiple copies, we decided to do it in code, and chose to separate style handling from the code that controls the different screens (view controllers). This is done using an extension method, setStyle(String), for every view element; an extension method in general adds new functionality to an existing class, structure, enumeration or protocol type. The extension method for each view forwards the setStyle call to a common class that sets up styling of the view. The architecture makes it possible to subclass the "common class" and override the styling in a given app. This implementation makes it possible to configure which style to use on a given view element directly from the Interface Builder. Hence, the implementation of styling variability in iOS is done using code space.

On both platforms, the specialized bank app has a few cases of differences in the visual appearance that have not been implemented by the styling techniques described above. Instead, in these cases, the code for the specialized bank app is embedded in the shared code base using parametric variability, e.g.:

```
if specializedApp then show(image.blur())
else show(image);
```

This approach is mainly used in areas where the styling concepts does not provide sufficient possibilities for modifying the graphical appearance.

E. Handling the Variability in Functionality

Only the specialized app has additional functionality in addition to that provided by the shared assets. Particularly the menus and login screens behave differently and define a variant in the interaction and screens presented.

On iOS, this is handled by creating new implementations of these screens and adding these to the app target for the specialized bank. The screens that are unique to the specialized bank app are of course only referenced by the app target for the specialized app.

On Android, we have overwritten the complete layout files from the shared library module by placing a layout file with the same name in the application module for the specialized app. Differences in the functionality of the screens are also handled by subclassing the implementation from the shared library module.

On both platforms, a few minor differences are handled parametrically, through if-statements in a manner similar to handling small styling variations as described above.

Hence, for the specialized app, the variability in functionality is handled using a mix of techniques from both the build space and the code space for both platforms.

IV. Discussion of Lessons Learned

A. Summary and Elaboration of Variability Handling

The techniques we have applied to handle the different kinds of variability can be summarized as follows.

	Code	Build	Configuration
Texts		i A	
Graphics		i A	
Styling	i (A)	A	
Functionality	(i) (A)	i A	

Figure 1. Variability spaces (i for iOS, A for Android). The parenthesized letters express that the space is only used to a small extent on that platform.

Figure 1 shows that regarding Nadi's spaces [2], similar choices have been taken on the two platforms. The only difference is the use of code space variability management for iOS regarding styling. However, even though the same spaces are used for the two platforms, there are significant differences in the handling of variability on iOS and Android.

Making a note of the commonality first, build space techniques handle most variability. For texts and graphical variability (and styling variability for Android), this is done through referencing bank specific property files that are included in the app during build time. All common and default values are in the shared asset library and may be overridden by an individual banking app. This overriding is supported directly on Android, but not on iOS, where we apply different techniques to obtain the same effect. This includes custom-made code for handling styling as well as copying assets to all banks in order to make one bank have specialized graphics, texts or functionality.

Regarding functional variability and some non-trivial styling variability, code space is furthermore used, notably parameterized common code (if-statements in common code alter the flow based upon the value of certain variables) as well as inheritance based variability (classes in common code are sub classed and methods overridden in individual apps). Parameterization is used for simple variability, typically simple attribute value modifications, while inheritance is used for major functional variability.

On iOS, we use code space techniques for handling the styling due to the lack of a styling mechanism. Furthermore, build space is used for text variability due to the lack of a built-in overwriting of properties. The technique includes a pre-compilation step that modifies the property files before compilation.

The precompilation step was chosen very early in the project and from a maintainability point of view, having a number of different techniques for handling variability is problematic, and the use of different techniques across platforms also. Having a number of different techniques for essentially the same type of variability induce arbitrary choices and lower architectural integrity; do the same thing in the same way is a sensible principle. In addition, when developers are transferred from one platform to the other, the differences in techniques may cause some confusion.

The lack of techniques in configuration space is interesting and is a result of the underlying platforms' decision to favor static, build time techniques. Arguably, this choice leads to smaller deployment units, which is relevant in the mobile domain.

B. Improved Support for Future Variation Requirements

In an increasingly competitive world, the banks strive to stand out from the crowd, which leads to increased demands on having mobile banking apps that provide more customer value than the competitors do. Therefore, we anticipate a trend of supporting more and more variability across the currently supported 11 banking apps. The challenge here is both streamlining the current set of variability techniques as well as improving support for more demanding types of variability, notably functional variations.

Currently, the variability that is mostly concerned with graphics and text contents is adequately supported by build space techniques, but there is room for improving the current code base to avoid pre-compilation.

Regarding the code space techniques used, parametric variability (if-statements in common code) has the unfortunate property that every new variation introduced requires a renewed analysis and change in the common library code. This is problematic regarding stability, reliability and maintainability. It has not implied undesired consequences until now, but with expectations of larger variability, these issues are likely to become more severe in the future. An alternative approach that reduces this problem is to use compositional/delegation based techniques like design patterns, e.g., a strategy pattern is commonly used for algorithmic variability and dependency injection based architectures.

C. Improved Variability Support in the iOS Platform

Our analysis has shown that iOS offers weaker support for the requirements of the banking apps than Android. The iOS platform lacks two basic features that Android provides.

In the first place, there is no styling concept that can be used to separate the styling from the code and/or the interface builder. This lead to our custom development of a styling concept based on simple key-value maps with definition of styling. Secondly, the lack of overwriting support similar to CSS concerning property files leads to duplication of graphics and texts as well as custom development of code that handles overriding of shared key-value pairs.

Therefore, an obvious wish for a future update of iOS is to include direct support for styling and overwriting in the iOS libraries.

D. Cross-platform Versus Native Development

The development of the apps discussed in this paper started before viable cross-platform toolkits emerged. This alone explains that the apps are native; cross-platform development was not considered when the project started. Since then, there has been some evaluations and considerations.

Two approaches have been considered. In the first place, development teams working for the banks have extensively evaluated use of HTML5 for a new tablet app, but experiments established this as inferior to native development when comparing performance and user experience, which were considered the primary quality attributes.

Secondly, new tools that have emerged recently and that compile to native binary code and claim to have solved the issues of previous cross-platform tools have been considered. For example, under some circumstances, Xamarin has shown similar performance compared to native developed apps. However, the user interface must still be developed exclusively for each platform to obtain the quality that the banks and their customers expect. Therefore, it has still not been considered feasible to use Xamarin. It might have been, if we had more common/shared business logic located in the app, instead of in the backend.

In summary, the apps are now and in the near future developed natively. The main reasons for this is that the native apps represent a huge investment, they are of very good quality and it would be costly and risky to change to cross-platform development. Moreover, the apps are almost exclusively graphical and visual with emphasis on a delightful user experience. As the design guidelines differ for the two mobile platforms, cross-platform development would need to yield a "native" user experience, which could be challenging.

On the other hand, at least from a theoretical point of view, cross-platform development obviously brings a number of advantages, including lower development cost, and the considerations about native versus cross-platform are likely to continue in the future.

V. Related Work And Conclusions

The presented architecture is a software product line (SPL) [3] for mobile banking apps. SPL for the mobile domain has received some attention in the research community. Francese et al. [4] present a process that supports multi-platform mobile app development, which has a focus similar to ours. The approach is based upon developing state transition diagrams, HTML5 and using the PhoneGap framework. Their approach is interesting, but their choice of HTLM5 as UI is not applicable in our case due to the high usability demands. Morais et al. [5] present a systematic review of SPL for mobile middleware, and summarize and classify techniques to support product line development for the mobile domain. In comparison, our approach generally uses techniques embedded in the respective platform and, in the case of iOS, adds our own custom-built code for variability management. Zhang et al. [6] report from the research project PLIMM (Product Line Enabled Intelligent Mobile Middleware) that uses context modelling using ontologies and variability management using meta-ontologies. While flexible, their approach seems overly complex for our case.

In conclusion, we have described our efforts to handle variability across a large set of mobile banking apps on the two major mobile platforms, and applied the variability framework of Nadi to classify the techniques used to handle variability across the five major dimensions of required variability: platform, text contents, graphics, styling and functionality.

The lessons learned from the considered app development project and the analysis presented in this paper can be summarized as follows. Build space techniques are able to handle most of the required variability, mainly in styling and text contents. Functional variability may be better supported by employing alternative techniques. Android offers better support than iOS. Cross-platform development is still not seen as a viable option for the apps in consideration, whose primary quality attributes are user experience and performance.

References

[1] L. Bruun, M. B. Hansen, J. B. Iversen, J. B. Jørgensen, B. Knudsen, "Handling design-level requirements across distributed teams: developing a new feature for 12 Danish mobile banking apps", 22nd IEEE International Requirements Engineering Conference (RE14), Karlskrona, Sweden, 2014

[2] S. Nadi, "A study of variability spaces in open source software", Doctoral Symposium on the 35th International Conference on Software Engineering (ICSE13), San Francisco, California, 2013, IEEE

[3] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice 3rd edition.* Addison-Wesley, 2013

[4] R. Francese, M. Risi, G. Tortora, G. Scaniello, "Supporting the Development of Multi-Platform Mobile Applications". In *Web Systems Evolution (WSE), 2013 15th IEEE International Symposium*. IEEE.

[5] Y. Morais., T. Burity, G. Elias. (2009, April). "A systematic review of software product lines applied to mobile middleware." In *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference*. IEEE.

[6] W. Zhang, T. Kunz, and K.M. Hansen, "Product Line Enabled Intelligent Mobile Middleware", In *12th IEEE International Conference on Engineering Complex Computer Systems*, July 2007.