Evaluating application generators for multi-platform mobile application development

Ville Pylkki

University of Tampere

School of Information Sciences

Computer Science

M.Sc. thesis

Supervisor: Zheying Zhang

August 2013

University of Tampere

School of Information Sciences

Ville Pylkki: Evaluating application generators for multi-platform mobile application development

M.Sc. thesis, 73 pages

August 2013

Abstract

The mobile application development scene is a difficult target for developers due to variation in development and distribution methods. Native development requires more effort, but has the best performance and a native look and feel for the user interface. Web-based solutions are easier to develop and distribute to several platforms, but suffer from the lack of access to device features. As a result, several tools have been created to bridge this gap.

Mobile application generators offer a way to access functionality that is native to a device, while the development is done using common web technologies. Choosing the most suitable generator could be difficult, due to issues in mobile development and differences between the tools. The issues in mobile application development include the lack of asset portability between mobile operating systems and multiple development technologies being used. Therefore, a set of metrics and criteria were proposed to evaluate mobile application generators, based on previous research and identified characteristics of these tools.

Information about the tools was hard to find. Nevertheless, many differences in provided functionality were noted, even within a single generator. Some mobile operating systems had better support, and differences in implementation could interfere with using the same assets for several target platforms. Three publicly available mobile application generators were evaluated using the proposed metrics and criteria. They turned out to be rather similar from the usage perspective, all capable of generating relatively simple multi-platform applications from user-created source code. Depending on whether faster multi-platform development or more efficient applications with a native look and feel are preferred, it is up to the developer to decide the development method.

Contents

1 Introduction . . . 1
  1.1 Separated platforms . . . 1
  1.2 Research motivation, goals and methods . . . 2
2 Mobile application development methods and issues with multiple platforms . . . 5
  2.1 Development tools and application distribution . . . 5
  2.2 Issues in mobile application development . . . 7
    2.2.1 Differences in hardware . . . 7
    2.2.2 Challenges for the developers . . . 8
    2.2.3 Possible solutions . . . 10
  2.3 Application types . . . 11
    2.3.1 Native applications . . . 11
    2.3.2 Web-based applications . . . 13
    2.3.3 Hybrid applications . . . 15
3 Software reuse . . . 17
  3.1 Why reuse? . . . 17
  3.2 Reuse methods . . . 18
    3.2.1 Components and reuse libraries . . . 18
    3.2.2 Software product lines . . . 20
  3.3 Reuse issues . . . 21
4 Application generators . . . 23
  4.1 Application generators in general . . . 23
  4.2 Mobile application generators . . . 27
5 Metrics and criteria for mobile application generators . . . 32
  5.1 Previous research . . . 32
    5.1.1 Software component evaluation . . . 33
    5.1.2 Component information . . . 39
    5.1.3 Evaluating mobile application types and approaches 40
  5.2 Suggested metrics and criteria . . . 41
    5.2.1 Entity, user group and characteristics . . . 41
    5.2.2 Measurable concepts and attributes . . . 44
    5.2.3 Metrics and criteria . . . 48
  5.3 Validation of the metrics and criteria . . . 52
6 Evaluation of existing generators . . . 54
  6.1 Appcelerator Titanium . . . 54

6.2 MoSync . . . 57
6.3 PhoneGap . . . 60
7 Conclusions . . . 63
7.1 Findings . . . 63
7.2 Future research and limitations . . . 67
References . . . 69

1 Introduction

Application stores are a common feature of modern smartphones. Since the launch of Apple's App Store[1] and Google Play[2] (previously known as Android Market) in 2008, the number of available applications has steadily risen. App Store has over 800 000 applications available, with a total of more than 40 billion downloads done by January 2013 [Apple, 2013]. Google Play also has more than 800 000 applications by June 2013[3], with a total of over 48 billion downloads in May 2013[4]. There are several other application stores for mobile devices, as different operating systems cannot run the same version of an application, or because the operating system providers want to increase their profits by controlling their own store.

1.1 Separated platforms

Different mobile operating systems have their own methods and rules for application development, distribution and download. The device-centric model is a common way for a user to access new applications [Hammershøj *et al.*, 2010]. In this model the device connects directly to the application store provided by the operating system developers, and a user can install new software with a few presses of a button. This kind of access to application stores has been made possible by the advancement of wireless data transfer technologies, which are available and affordable for the general public.

Some of the operating systems have limitations for application downloads, though. The top two operating systems at the moment, i.e. iOS by Apple and Android by Google [Gartner report, 2013], have very different approaches to distribution and development. For example, Apple limits the acquisition of new software to their own application store, while devices running the Android operating system can download applications from any source. In addition to stores provided by operating system creators, there are also several third-party stores. Development

[1] `https://itunes.apple.com/us/genre/ios/id36?mt=8`

[2] `https://play.google.com/store/apps`

[3] `http://www.rssphone.com/google-play-store-800000-apps-and-overtake-apple-appstore/`

[4] `http://www.androidauthority.com/google-io-android-activations-210036/`

for the iOS is also limited to Apple products, as the software development kit (SDK) is only provided for the company's own OS X. Applications must also be packaged on an Apple computer prior to submitting them. Android development tools are available for the Linux, OS X and Windows operating systems. Both approaches have their benefits and drawbacks.

Developing and maintaining mobile applications has become easier from one perspective, but more challenging from another. Operating system creators and device manufacturers provide extensive information on how to access the features of the device through application programming interfaces (APIs), while third-party frameworks and components make it easy to integrate additional features into the application. The devices themselves have evolved and have different advanced features, such as cameras, GPS receivers and motion sensors, and they are able to display 3D graphics on high-resolution displays.

Developing an application for one operating system does not mean it is ready to be distributed on the others. Different operating systems use different application packages for their native applications, which means that an application will not run on other systems. By a native application, we mean an application designed and implemented for a specific operating system. Application developers have to take the variation between device models and their operating systems into account and adapt their products accordingly. At the same time optimization should be taken into account to ensure that the application runs smoothly and does not drain the device battery more than necessary.

1.2 Research motivation, goals and methods

To address the problems caused by the large number of differences between device features and mobile operating systems, several mobile application generators have been developed by different organizations. The goal of these generators is to limit the development process to a selected programming language (or a set of languages) so that the created assets can be reused when developing an application for multiple target operating systems. In general, the created source code is then combined with necessary tool-specific additions and packaged in a container which is native to the target operating system. Some of them work by translating and packaging applications into native (or native-like) applications. Other, more simple generators, modify the user interface, usually a web page, to

fit the screen of a mobile device. Each type has its own specific capabilities and limitations.

While there is a large body of research on software reuse, including application generators (for example by Batory [2004], Biggerstaff [1998], and Smaragdakis and Batory [2000]), less research specifically on mobile application generators is available, the found research mostly being whitepapers from generator providers or informal tests by individuals. The goal of this research is to find out if the existing generators can address the problem caused by the splintered development methods and environments. To this end, some method of measuring mobile application generators in this regard is needed. Measuring reuse has been the target of some research (for example Her *et al.* [2007] and Washizaki *et al.* [2003]), but no research was found regarding application generator suitability in certain situations.

These issues are investigated with the following two research questions:

- Can the reuse of assets created using mobile application generators address the challenges of developing for multiple platforms?
- How to evaluate mobile application generator suitability for reusing assets in multi-platform development?

This study relies on previous research done on related topics. First, the main areas of the research, such as mobile application development, software reuse and application generators are discussed. Based on existing literature and suitable characteristics, we try to identify the problems with evaluating mobile application generators and suggest a set of metrics and criteria for such evaluation. The first research question will be discussed by identifying challenges in mobile development and analyzing existing generators. The second question will be discussed by discerning common features of mobile application generators, exploring previous research on reuse and combining these to form a set of criteria that can be used in evaluation of mobile application generators.

The research is structured in the following manner. After this introduction, the second chapter discusses the different methods of developing multi-platform mobile applications and the related issues. The third chapter explains the basics behind software reuse, which is further discussed with a focus on application generators in the fourth chapter. The fifth chapter explores previous research on reuse metrics and proposes a set of metrics and criteria, which are adapted

for evaluation of the selected mobile application generators in the sixth chapter. The final chapter summarizes the findings and suggests further research directions.

2 Mobile application development methods and issues with multiple platforms

The mobile application scene has gone through significant changes within a few years. Applications are now an important part of the mobile experience, and can even be the factor that makes users choose one device over another. With the introduction of smartphones, developers have been given more freedom in mobile application development. There are several target platforms, application formats, development languages and tools to choose from.

2.1 Development tools and application distribution

All the major mobile operating systems have their own application stores. Android has Google Play, iOS has App Store, BlackBerry has App World[1] and Windows Phone has Windows Phone Store[2]. In addition, there are third-party stores, such as the Amazon Appstore for Android[3].

None of the operating systems can run applications from another manufacturer's store, due to restrictions in application acquisition, application packaging methods, or several other limitations caused by differences in both software and hardware. This forces application developers to choose which platforms to support. The exceptions to this rule are BlackBerry's Android Runtime[4], which allows developers to repackage the software developed for a specific version of Android and publish it for BlackBerry devices. The upcoming Sailfish OS by Jolla will also support easy repackaging, since it can run most Android applications without any changes [SailfishOS wiki, 2013].

When releasing an application for multiple platforms, developers have to use multiple application publishing portals. Different portals have their benefits and drawbacks, which a developer has to weigh when choosing the ones to support. All portals have their own rules for accepting applications in their catalogs.

---

[1] `http://appworld.blackberry.com/webstore/?`

[2] `http://www.windowsphone.com/en-us/store`

[3] `http://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011`

[4] `http://developer.blackberry.com/android/`

When developing applications, the developers are offered a set of default tools for a specific operating system. For Android there is the Eclipse integrated development environment (IDE) with an Android Development Tools plugin[5], and Java is the default programming language. The Android SDK is also available separately, giving developers the chance to use another IDE. The Android SDK is available for all the common operating systems, such as Windows, Linux and OS X. For iOS development, the XCode IDE along with iOS development tools are available only for some versions of the OS X operating system[6]. This requires developers to use virtual machines or third-party development tools on other operating systems, which might not create native iOS applications. Tools not provided by Apple might also use programming languages other than Objective-C, which is the default language for applications running on Apple devices. Developing for Windows Phone is restricted in the same way. Tools are only provided for the Windows operating system, the IDE being Visual Studio and the programming language C#. BlackBerry takes a different approach, selecting the IDE based on the programming language the developer chooses. All smartphone operating systems are capable of running applications made with different programming languages, but one is often preferred and suggested as the default development language by the operating system provider. BlackBerry SDK is used with a suitable IDE, which is usually based on Eclipse. Java, C and C++ among a few other options are promoted on their web pages as the programming languages for BlackBerry application development.

It is generally agreed that developers have many options to choose from, when it comes to development languages [Hammershøj *et al.*, 2010, Gavalas & Economou, 2011]. Besides the languages promoted by the operating system providers, there are several third-party options as well. Java ME and Qt used to be common in mobile development before smartphones became popular. Both are still used on mobile devices, though, since they support portability [Java documentation, 2013, Qt documentation, 2013]. As with all third-party frameworks, they might not be able to access all the features of the platform they are running on, and they cannot be expected to run on all operating systems. Gavalas and Economou [2011] note that at least Java ME requires modified versions of an application to be created in order to run on different platforms, reducing the efficiency of reuse. They also

[5] `http://developer.android.com/tools/sdk/eclipse-adt.html`

[6] `https://developer.apple.com/downloads/index.action#` (requires login)

mention that ultimately the choice of language is dependent on the platform and the requirements set for the applications.

Holzer and Ondrus [2011] state in their article that developers are attracted by platforms with a large user base, while users are attracted to platforms supported by a large number of developers. This can be a problem for those platform developers who do not manage to attract either developers or users. It could be helped slightly by making high quality development tools, which might increase developers' interest towards the platform.

2.2 Issues in mobile application development

Mobile applications differ from desktop applications in some ways. There are limitations caused mainly by hardware, but application developers also have the opportunity to use functionality that is not normally present in desktop computers.

2.2.1 Differences in hardware

The most obvious differences between mobile and desktop application development exist due to the limitations set by portable devices. Battery life is always an issue and has been made more prominent by modern smartphones with more powerful processors, better displays, and more memory. Moore's Law, which states that the number of transistors on integrated circuits doubles every two years, making them more efficient, seems more or less true when it comes to the capabilities of mobile devices, with the exception of their batteries. Although there have been improvements in battery technology, these improvements have not been able to keep up with the advancements of other components. The improvements made are also often negated by the fact that device manufacturers choose to use the advancement in battery technologies to create smaller batteries in order to get thinner devices. Admittedly, the advances in component technologies have made mobile device components more energy-efficient, but not as much as to significantly reduce the energy consumption caused by applications with high processing requirements. Other special considerations include online connectivity and performance, which are dependent on the software and hardware installed in the device.

Due to the large number of different devices, there is also a large number of hardware configurations. Developers can utilize sensors and input methods available on the device. These physical aspects should be taken into account when choosing a target platform. Carefully selecting the target platforms helps the developer to make high quality applications on all of them [Abrahamsson *et al.*, 2004]. Therefore, testing should not be limited only to emulators, which are readily provided by operating system creators, but it should also happen on actual devices, preferably several different ones, to provide more accurate information [Wasserman, 2010]. Testing is identified as an important factor and is one of the main tasks in the Mobile-D method generated by Abrahamsson *et al.* [2004] to achieve deployment on multiple platforms. Prototypes resembling the final product as much as possible are suggested by de Sá *et al.* [2008] to help in designing applications for mobile platforms. They also say that tests should be performed in settings and situations where the applications are going to be used, instead of relying on virtual testing environments.

2.2.2 Challenges for the developers

Having several device and operating system providers increases competition and pushes them to innovate, and it gives application developers the chance to choose the platform with the features that they want and need. However, the different combinations of hardware and software are a source of difficulties for the developers [Mikkonen & Taivalsaari, 2011]. Studies by Hammershøj *et al.* [2010], Heitkötter *et al.* [2012], Holzer and Ondrus [2011], and König-Ries [2009] have all identified this problem. Developing for several platforms takes more resources, which smaller companies might not have. A survey conducted by Wasserman [2010] revealed that most of the mobile application development teams had only one or two members in them. On the other hand, the average application size was also relatively small. It is not enough for an application to run on a specific operating system, it also has to run on different versions of the system. This adds to the difficulties in optimizing performance, which, according to Wasserman [2010], is a problem. For example, out of all the Android operating system versions currently in use, there are three versions with a share of 25% or more each, as shown in Table 2.1. Nevertheless, optimization should be done on all targeted platforms, despite being expensive and resource consuming [Holzer & Ondrus, 2011], otherwise possible incompatibility problems might be encountered. Abra-

hamsson *et al.* [2004] and Heitkötter *et al.* [2012] also note that the hardware and software used on mobile devices are evolving at a fast pace, which makes it difficult for developers to keep up with all the changes when targeting multiple systems. The authors mention that platform-specific skills are needed to cope with these changes.

Version	Codename	API	Distribution
1.6	Donut	4	0.1%
2.1	Eclair	7	1.7%
2.2	Froyo	8	3.7%
2.3 - 2.3.2	Gingerbread	9	0.1%
2.3.3 - 2.3.7		10	38.4%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	27.5%
4.1.x	Jelly Bean	16	26.1%
4.2.x		17	2.3%

Table 2.1: Market shares of different versions of Android [Android report, 2013].

Online portals for applications also mean worldwide markets, which requires developers to internationalize and localize their applications, causing even more fragmentation, even when an application is released on a single operating system [Mikkonen & Taivalsaari, 2011]. Developers wanting to maximize the number of potential users for their application have no choice but to develop for several platforms. [Heitkötter *et al.*, 2012]

Wasserman [2010] notes that different operating systems have their own look and feel, which should be preserved in applications, if possible, in order not to disorient the users. He continues that interface components and input methods from the operating system should be used in an application as well, since style guides, recommendations and best practices are provided and based on the research and experience from the operating system providers. For example Android[7] and Apple[8] provide such documentation.

[7] http://developer.android.com/guide/topics/ui/index.html

[8] https://developer.apple.com/library/ios/navigation/#section=Topics&topic=User%20Experience

2.2.3 Possible solutions

To conclude, mobile application development issues originate from the fragmented operating systems and their different development and distribution methods. In addition to the operating system-specific application markets, some third-party markets also exist. However, some operating systems limit the sources where applications can be downloaded, and they set restrictions for using different development environments for native applications. From the users' perspective these limitations can simplify matters and increase general usability, but on the other hand they limit the possibilities for the developers. Although developing for a single operating system keeps the process simpler, those wanting a larger audience need to make their applications available for as many operating systems as possible. Developers have to take into account not only the differences between operating systems but also the different versions of them. The differences in hardware only add to this. While some features, such as the camera, are installed on nearly all mobile devices nowadays, Wi-Fi and near field communication (NFC) chips might be excluded, especially on budget devices. Developers also have to take into account different screen sizes, user interface components and interaction methods.

According to Hammershøj *et al.* [2010], Android was developed to answer the fragmented hardware problem by providing an open source operating system that any mobile device can run, making the development platform unified while being hardware-independent. The article also mentions that the vision was to create the system in such a way that applications could be run online in cloud services. This would make it easy for developers to create web-based applications, making the applications highly portable, since the hardware restrictions would not become as large an issue. Gavalas and Economou [2011] say that the software architecture of Android promotes reuse, which would help developers to create applications faster and cheaper with available libraries and components. The same authors mention, however, that an application may have difficulties in running on different versions of Android. This is problematic, since several versions are still in active use. Holzer and Ondrus [2011] mention that open source in general promotes reuse. The shorter time-to-market achieved through reuse can be important for application developers, as it can provide a competitive edge [Abrahamsson *et al.*, 2004]. Nevertheless, Android has managed to become the most popular mobile operating system, as is seen in Table 2.2, and is used on devices such as phones

and tablets, which come from a number of different manufacturers and in different price ranges.

Worldwide Smartphone Sales to End Users by Operating System in 1Q13 (Thousands of Units)

Operating System	1Q13 Units	1Q13 Market Share (%)	1Q12 Units	1Q12 Market Share (%)
Android	156,186.0	74.4	83,684.4	56.9
iOS	38,331.8	18.2	33,120.5	22.5
BlackBerry	6,218.6	3.0	9,939.3	6.8
Microsoft	5,989.2	2.9	2,722.5	1.9
Bada	1,370.8	0.7	3,843.7	2.6
Symbian	1,349.4	0.6	12,466.9	8.5
Others	600.3	0.3	1,242.9	0.8
Total	**210,046.1**	**100.0**	**147,020.2**	**100.0**

Table 2.2: Market shares of different mobile operating systems [Gartner report, 2013].

Wasserman [2010] points out that while most of the mobile applications are small and used for entertainment, business-critical applications are more complex and need good development practices to ensure quality. He also mentions that traditional development practices can often be used in mobile development, but there are some areas that need special attention. Gavalas and Economou [2011] state that a developer cannot easily write an application and publish it for several platforms. They suggest standardization and stricter enforcement of these standards to combat this issue.

2.3 Application types

One of the key choices a developer has to make is selecting the "form" of the application. Mobile applications can be roughly divided into three categories, i.e. native, web-based and hybrid. When deciding to create an application, the developer has to weigh the differences between them and choose the most appropriate one [Huy & vanThanh, 2012].

2.3.1 Native applications

Native applications are such applications that are installed and run on the device itself. The developer has better control over the distribution, and native appli-

cations are therefore easier to monetize, as they are usually distributed through controlled portals [Mikkonen & Taivalsaari, 2011]. From the developers' perspective, one of the main reasons to develop a native application is better performance [Charland & LeRoux, 2011]. Since the native API is used, there are no extra layers between the application and the operating system. The native API is often a product of years of development by domain experts with optimization knowledge. Especially applications requiring a substantial portion of processing power, such as games containing complex graphics, should be native applications. The native API also grants direct access to all features available for that operating system [Mikkonen & Taivalsaari, 2011]. These APIs are thoroughly documented by operating system providers. Charland and LeRoux [2011] say that a successful application takes into account all the given possibilities for interaction. Different methods of interaction are a part of the user experience. User experience is an important factor for application users [Charland & LeRoux, 2011]. Huy and vanThanh [2012] say that the best user experience is gained through native applications. With the native API, developers can create user interfaces matching the default behavior of the operating system [Heitkötter *et al.*, 2012, Mikkonen & Taivalsaari, 2011].

However, native applications are always developed for a specific operating system with the operating system-specific SDK [Heitkötter *et al.*, 2012, Huy & vanThanh, 2012, Mikkonen & Taivalsaari, 2011]. Since a native application uses an API specifically made for that operating system, the same application cannot be used on another system without modifications. Charland and LeRoux [2011] address the portability issues, and they note that native applications are expensive to develop and maintain on several platforms. Native user interfaces need to be customized for each operating system, as they are based on different APIs.

Different platforms require their own code base, and additional platforms to support mean more source code to maintain. Heitkötter *et al.* [2012], and Huy and vanThanh [2012] also note this, saying that native applications require more effort to create, especially when compared to web-based applications. This is probably due to the advanced programming skills that are needed to use the APIs effectively and correctly. Heitkötter *et al.* [2012] also mention that native development technologies often have good support given by the operating system provider. Successfully distributed native applications use product line management methods to control the development [Mikkonen & Taivalsaari, 2011]. Tools

supporting the development, such as error reporting and distribution tools, facilitate the development process for multiple operating systems.

Native applications could be used, for example, in sport activity monitoring. Such applications usually need access to a device's motion sensor and GPS receiver, while displaying route information on a map in real time. In addition to the data coming from the phone's own sensors, it is possible that the application needs to process data coming from a heart rate monitor. The combination of needed device features and processing power suggests that a native application should be used in this case.

2.3.2 Web-based applications

Web-based applications have been proposed as a solution to the fragmented platforms problem. Heitkötter *et al.* [2012] say that web-based applications are essentially web pages optimized for mobile devices and provide a good starting point for multi-platform development due to good browser support on all operating systems and the maturity of web standards. They also mention the ease of development due to common and easily learnable technologies, making a fast start possible. The authors note, however, that in case device-specific functionality or application store distribution is required, some other solution is needed. While native applications can be used to access information online, they still do their processing on the user's device [Huy & vanThanh, 2012]. According to Mikkonen and Taivalsaari [2011], native applications with online access are nowadays the most popular way of using online services. They also mention that using a native application requires downloading and installing files, and later possibly updating them. This requires actions from the user, decreasing the user experience slightly. Web-based applications, on the other hand, are deployed on servers and accessed with the device's web browser. Most—if not all—modern devices have a web browser with the capability to handle HTML 5 pages. For this reason, updates can often be handled on the server side, without any actions from the user [Mikkonen & Taivalsaari, 2011].

Common technologies for web-based applications include HTML 5, CSS and JavaScript. If the application does not require the use of advanced device features, these commonly used and straightforward technologies might be preferred over the ones used by native applications [Huy & vanThanh, 2012]. As the application

is running on a server, most of the application logic resides there, minimizing the need for native source code specific to one operating system [Wasserman, 2010]. This means less source code to maintain and availability for all devices.

Due to remote processing, it is suitable to implement content delivering applications, such as blogs, as web-based applications [Huy & vanThanh, 2012]. When complex information and continuous, real-time input and feedback are required, web-based applications might not be suitable. Although Charland and LeRoux [2011] mention that performance differences are noticeable only with processor-heavy applications, they also note that native applications are faster than those using JavaScript. However, the article mentions that since it is faster to write the program logic for multiple platforms with web technologies, this is a sacrifice the developers might be willing to make. Additionally, the authors note that performance is not the only issue, but web-based applications are also subject to slow connections and network problems. They cannot be distributed through applications stores, making control over the distribution difficult.

According to Charland and LeRoux [2011], web-based applications cannot access all the device features, since they have no access to the necessary APIs. The authors mention that the lack of API access affects the user experience, since the user interface elements native to that operating system cannot be accessed, but HTML and CSS have to be used instead. The article also mentions that the easily manageable source code is no consolation for the user, who expects native-like controls. Therefore, the authors suggests that when native user interface elements are not available, the developers might want to modify the usage to match the native controls. Although this requires taking into account the limitations of the used techniques, as well as more skills in user interface design, controls matching the native ones are entirely possible. Differences in interaction methods between applications made by different developers can cause fragmentation even within one operating system [Mikkonen & Taivalsaari, 2011]. Therefore, common practices and styles should be followed.

While bringing some advances to traditional web technologies, HTML 5 is still not fully capable of executing the same functionality as native applications. It is suggested, though, that eventually web technologies will catch up with native ones, when it comes to performance [Charland & LeRoux, 2011, Huy & vanThanh, 2012]. Huy and vanThanh [2012] list some functionality, also mentioned by Mikkonen and Taivalsaari [2011], which is added with HTML 5. This added

functionality includes audio and video playback, geolocation, offline mode, and a simple local data storage. Such advances help in bridging the gap between native and web-based applications. In addition to that, the World Wide Web Consortium also has a working group that is developing an API for client devices, which enables access to advanced device features [W3C documentation, 2013]. Currently web standards suffer from slow development times, though, making them lag behind native applications.

A web-based application could be the best solution when data is collected from different online sources and displayed to a user. For example, an application that goes through specified blogs, summarizes information from their most recent entries for easy previewing on a small display, and provides links to these entries, can be implemented as a web-based application. The data shown on the mobile device could be gathered and formatted remotely, and the links would take the user to the actual site of the blog. Since there is no need for real-time processing on the device, a web-based application is a working solution.

2.3.3 Hybrid applications

Development methods addressing the multi-platform development issues have been created. The main challenge is to allow developers to target different platforms while offering opportunities to utilize the advanced capabilities of smartphones [Heitkötter *et al.*, 2012]. Hybrid applications aim to provide such a solution. They are a mix of native and web-based applications, trying to offer the benefits of both. Usually hybrid applications are implemented with web technologies, such as JavaScript, wrapped inside a native container providing access to additional features, and run on the device. Another possibility is described by Huy and vanThanh [2012], where the work the application does is divided in three parts, a client, an application server, and a database. Every major operating system has its own API different from others, but they share the capability of having their browser engines used to display web application content on the screen, without any typical browser user interface elements [Charland & LeRoux, 2011]. This means that a hybrid application can use the deployment methods used by native and web-based applications.

However, hybrid applications have their weaknesses too. Additional layers between the operating system and the application reduce performance, making them

less suitable for processor-heavy tasks. Depending on the frameworks and tools used, the hybrid methodology can provide native-like user interfaces, but not in all cases. Using a framework also requires a developer to learn the concepts behind it. If the developer needs more functionality than what the current framework provides, they might have to switch to another one, which they have to learn and understand. Learning to use one framework can help in understanding others, but specific functions are very rarely directly transferable.

When an organization that has several different mobile devices in use wants an application that has to be portable, but also able to run on all devices and access some device features, while requiring some real-time processing, a hybrid application can be used. An example of such an application could be combining calendar and contacts data with a map.

As we can note, all platforms have their advantages and drawbacks. Table 2.3 summarizes the areas where a specific development method is strong when compared against the other methods, along with some other summarized information. It should be noted, however, that the comparison only applies to the popular development methods and especially hybrid applications can vary depending on the tools and frameworks used to create them.

	Native	Web-based	Hybrid
Multi-platform		X	
Performance	X		
Access to device features	X		X
Native UI	X		
Offline usage	X		X
Distribution	Application portal	Web access	Application portal
Development languages	Operating system-dependent	HTML 5, JavaScript	HTML 5, JavaScript

Table 2.3: A comparison between native, web-based and hybrid mobile applications.

3 Software reuse

Software reuse is thought of as a method to speed up the development process by recycling previously created assets or even whole products. Reusable assets are not limited to the source code, but can also include documentation or organizational practices. Modern programming languages provide support for reusing large portions of source code with classes, modules and frameworks. [Frakes & Kang, 2005, Jha & O'Brien, 2009, Mili *et al.*, 1995]

Although reuse often happens on the fly, for example by copying a snippet of source code to other functions, systematic reuse helps developers reap the most benefits [Hsieh & Tempero, 2006]. Therefore, if reuse is going to be used in an organization, it should be planned ahead. Several ways to reuse assets exist, some of which will be described in this chapter. Software frameworks, document templates, asset libraries and design patterns are commonly used in case recurring tasks are encountered.

3.1 Why reuse?

The main objective of reuse is to increase software quality and productivity, thus increasing the potential profit an organization can make. As software systems become more complex, starting from scratch every time can take too much resources to be feasible [Frakes & Kang, 2005, Jha & O'Brien, 2009]. Reuse eliminates unnecessary re-implementation and reduces the time-to-market. The benefits of reuse are often not realized immediately and setting up a reuse program can extend the time-to-market of the first projects in the reuse program [Biggerstaff, 1998]. This is due to the time and resources it takes to set up a reuse program.

Reuse addresses the issue of scale by taking development from component level to architecture level. Instead of defining functions at the component level, in theory reuse allows developers to work closer to the system architecture level by constructing systems out of ready-made components [Mili *et al.*, 1995]. Development becomes adapting and fitting suitable components together to build the system, not writing detailed application logic. Sztipanovits and Karsai [2002] agree that composition of this kind and composition-based design are tools for managing complexity, if the components have clearly defined, standardized interfaces and

connection methods. They say, however, that development on the architecture level works only with small systems with a low number of components, or with coarse-grained components with custom source code connecting them together. Building large systems is more difficult, since customizable components (either through parametrization of the components or selecting a specific implementation method) can cause brittleness in the developed product.

3.2 Reuse methods

When creating reusable assets, the asset developer has to keep in mind the high quality expected from them. The asset should also answer a common need and be understandable. For the asset to answer a common need, its functionality should be made more abstract, so that it is applicable in different situations. The asset can be made more understandable with appropriate documentation. These requirements are often difficult to achieve together, as increasing understandability can require implementing less abstraction, while increased abstraction can decrease understandability. Abstraction means that some domain-specific functionality has to be re-implemented before the asset is used, which makes the component less usable and possibly less understandable as well. Therefore, the asset developer needs to find balance between the two. [Mili *et al.*, 1995]

When the developer knows that the asset will be used by others, more effort should be made to ensure that the asset does what is promised. Otherwise the errors are reflected across all the systems using it. There is also the possibility that the asset users report fixes or improvements to the original asset provider, helping to further increase the quality of the asset. Studies have shown that defect density in reused assets can be as much as eight times lower when compared to assets developed using traditional methods [Biggerstaff, 1998]. There does not seem to be a statistically significant difference in this regard between unchanged assets and slightly modified assets [Biggerstaff, 1998, Frakes & Terry, 1996].

3.2.1 Components and reuse libraries

There are several approaches to reusing assets systematically. Biddle and Tempero [1998] note that when an asset is developed, the developer should decide whether or not to improve its reusability for future use. They also say that mod-

ifying an asset to be reusable rather than creating it as reusable from the start takes more resources. Therefore, knowledge of potential reuse should be taken into account even before the development starts. Washizaki *et al.* [2003] state that reusability is an important attribute when comparing reusable components. They define it as the degree at which a component can be reused, decreasing the amount of needed implementation effort. The authors also say that there is an issue with how users can detect the most reusable component from a pool of several similar ones, which is why reusability needs to be measured in order to use the components effectively.

Reusing existing assets is called the building blocks approach, and it is defined as a method where assets are made and reused, the reuse done with or without modification. A basic form of systematic reuse is a library consisting of assets. The library is used to store, search, represent and assess its contents [Frakes & Kang, 2005, Jha & O'Brien, 2009]. Although the concept of a reuse library is simple, creating and maintaining one is more difficult. First of all, the library has to provide some basic functionality, as the assets will only be reused if finding, analyzing and integrating them takes less resources than developing a similar component from scratch [Biddle & Tempero, 1998, Caldiera & Basili, 1991]. Mili *et al.* [1995] mention that the costs of initial development, including the assets in the library and the use of the component, are issues to consider, along with the expected frequency of use. They conclude that tool support for finding, analyzing and integrating assets is needed in order to make the process more efficient for the eventual users. The cost of these activities can be measured [Caldiera & Basili, 1991]. This makes it easier to evaluate the library and can help in finding more efficient methods of its usage.

Mili *et al.* [1995] state that reusing a component can be divided into black box and white box reuse. In black box reuse the asset is used without modifications or considerations to its inner workings, usually because the asset is in such a form that its functionality cannot be accessed. In white box reuse, the functionality can be accessed and the component is adapted to the new environment with the necessary modifications. White box reuse must be weighed against creating the component from scratch and is only cost-effective if the modifications are minor or planned.

Component reliability, which is the degree to which a component can be trusted to do what it is claimed to do, can be an issue. Therefore, components need thor-

ough documentation. However, some developers do not test integrated reusable components separately, but only as a part of a product [Jha & O'Brien, 2009]. Understanding the functionality of a component becomes crucial, since developers need to be aware of the possibilities and limitations set by the component. This is important in black box reuse, since the user cannot check the specifics from the source code [Mili *et al.*, 1995].

According to Bertoa and Vallecillo [2002], component-based software development (CBSD) and commercial off-the-shelf (COTS) components have enabled developers to move from application development to application assembly. As with other reuse methods, the goal is to reduce effort, in this case by minimizing the implementation work. Bertoa *et al.* [2003] also mention that CBSD helps in multi-platform development, since it enables using components that have already been tested and validated. The authors also note that a common rule in CBSD is that if more than 20% of the component has to be altered to fit its new environment, it is better to develop it from scratch. In multi-platform development, developers need to create or modify components specific to a platform.

3.2.2 Software product lines

Software product lines are another common method of reuse. They are used to create sets of products that are based on common assets. The amount of assets that can be reused in the product line depends on the similarity of the end products [Frakes & Kang, 2005, Jha & O'Brien, 2009]. According to Her *et al.* [2007], the success of a product line depends on the reusability of available core assets.

Experience in the product line is needed in order to identify functionality that can be shared between products [Jha & O'Brien, 2009]. Such functionality can be indirectly measured and identified by counting the number of times these functions are used with in the product line. Functions found this way are good candidates to be included in core assets, if it is assumed that often used components are also easily reusable [Caldiera & Basili, 1991]. Domain models are one way to store knowledge of the application domain and can help in identifying reuse opportunities [Mili *et al.*, 1995]. In order to preserve the quality of core assets, a continuous investment is needed, and the number of products in the product line should be kept at a manageable level [Jha & O'Brien, 2009].

3.3 Reuse issues

Although a potentially powerful technology, reuse has never been counted among the most interesting software topics. Aspects that hinder reuse include the lack of maturity of software development as a scientific or engineering discipline, the lack of training in software development and reuse, inadequate management structures and practices, and the lack of tools to support reuse [Mili *et al.*, 1995]. There is no single working strategy for reuse due to the differences in programming languages, development technologies, reuse technologies, the scale of components, breadth of domain applicability, feature variability, performance requirements, shelf-life of reuseable assets, the management structures and processes of a company, and so forth [Biggerstaff, 1998].

Object-oriented languages have been promoted as a solution for easier reuse. Mili *et al.* [1995] note, however, that object oriented languages merely enable reuse but do not guarantee it, and if used wrongly, can lead to bad class structures, awkward method names and unsafe inheritance, as well as unpredictable behavior. Therefore, good and proven practices need to be followed. The authors also say that well-documented assets give more information on opportunities for reuse, without the user having to look at the actual source code. The lack of reusable components also hampers reuse. Although the software market is large, reuse libraries are not common or available to everyone [Caldiera & Basili, 1991].

Jha and O'Brien [2009] have identified some disadvantages with product lines. The initial investment and maintenance costs are something organizations might hesitate to pay, as there is a risk of getting no return. Another risk is the dependencies that form between the domain and product engineering units. Reuse might also reduce innovation, if the usage of old assets is encouraged. There should be a balance between creating new, more efficient components when needed and reusing old ones, when they still satisfy the requirements. Caldiera and Basili [1991], and Mili *et al.* [1995] identified other problems, including the maintenance costs of the development team and the asset library, as well as the higher development costs of generalized assets due to their more complex nature.

Reusing assets does not in itself guarantee improvements. In order to be successful, reuse needs planning and resources before any benefits are realized. Biggerstaff [1998] notes that domain effect, which is the amount of domain-specific content in an asset, is the most important factor for reuse success. He also says

that the reuse technologies used, such as library infrastructure or generative methods, may be different between projects, but domain specific strategies increase the chances of success due to greater gains in reuse. He notes, however, that the technology does matter and is needed, but it does not ensure success, since it is not the most important factor in reuse strategies, with technologies often being interchangeable in terms of reuse effect. The author states that successful cases of reuse include domain-specific, necessarily large components and that there is no substitute for engineering domain content capturing the operational knowledge. This implies that abstracting components can start being counterproductive at some point, since it removes functionality meant for a specific domain, which has to be re-implemented later.

4 Application generators

Application generators are tools that can create desired assets according to specifications given to them. Besides reusing assets, application generators provide an approach to concept reuse. The reusable knowledge is captured in an application generator that can be programmed by domain experts using a domain-specific language that supports application generation. Generators are mostly used in well-defined domains. In theory, the output can be anything from an application to specifications for another generator.

4.1 Application generators in general

When a more automated reuse process is desired, application generators can be used. A well understood process can be automated [Smaragdakis & Batory, 2000]. Programming tasks that are too mundane for human developers to perform can be given to application generators, which will generate generic source code. An application generator is left to decide how to implement a problem that is described to it.

When building an application generator, the domain knowledge plays a key role, making application generators similar to core assets in product line engineering in this regard. Cleaveland [1988] divides the development of an application generator between a domain analyst, who specifies the requirements for the generator, and a domain designer, who implements the generator based on the requirements. A generator should decrease the amount of work a system designer has to do by generating parts of the developed system. Mili *et al.* [1995] say that an application is not always complete after a generator is used, which should be taken into account in the building process. Parts that are made by the generator, parts that are customized by hand, and interfaces between those two must be identified. Software assets can have parts that exist in all products and parts that vary, and these parts need to be identified and separated. Generating the common parts of software assets is the main purpose of a generator. These common parts need to be defined by the generator developer. The input method, meaning the form that the input has, also has to be defined.

Cleaveland [1988] proposes some general requirements for application generators. An application generator should be user-friendly, even so far that it is usable by a

nonprofessional programmer. Other factors specified by the author are reduction in software errors, support for fast prototyping, and the ability to easily implement standardized interfaces. Some of these requirements can be met with the used input languages, since they are easier to write when compared to traditional programming languages. Mili *et al.* [1995] add increased productivity to the requirements, meaning that generating an application should take significantly less time than creating an application with more traditional development methods. They also note that testing the output might be problematic, as expected and actual output need to be compared, but with generators the knowledge whether the expected output is correct also needs to be verified.

Application generators are not the solution for every problem, as they have some drawbacks. A generator can usually be used only in a very specific domain, which limits the number of possible uses. The difficulty of building one is also a significant issue, especially when taking into account the fact that usage possibilities are limited. Building a generator requires knowledge in both the application domain and language translators. [Cleaveland, 1988]

When using a generator, the specification for an application is usually given as a domain-specific language, which is often a language designed for a specific task, and is not a general-purpose language [Smaragdakis & Batory, 2000]. Domain-specific languages often have their own syntax and do not resemble conventional programming languages [Biggerstaff, 1998]. The generator takes the specification and generates an application in a desired implementation language [Mili *et al.*, 1995]. The problem with domain-specific languages is the cost of developing the semantic foundations they use along with tools that support them, which includes generators [Sztipanovits & Karsai, 2002].

According to Smaragdakis and Batory [2000], a typical application generator consists of three parts, as seen in Figure 4.1. A front-end is responsible for creating an internal (intermediate) representation (such as a flow graph or an abstract syntax tree) of the input, which is usually in a textual form. Therefore, a lexical analyzer and a parser are needed. The second part is a transformation engine, which in turn transforms the internal representation into an executable program, which is still in the form of an intermediate representation. The last part, the back-end, is responsible for generating a product as the final result, a high-level programming language version of the original input. Czarnecki [1999] says that the input needs to include both the properties of the platform as well as the

application, in order to make the physical properties of the platform computable and analyzable.

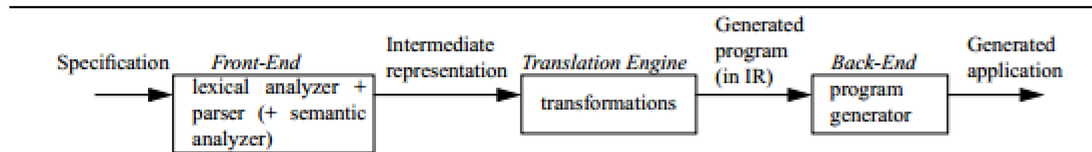

Figure 4.1: Different parts of an application generator [Smaragdakis & Batory, 2000].

Smaragdakis and Batory [2000] identify four variations points in the transformation processes in the existing generators. Generators can be stand-alone or general transformation systems. Stand-alone generators work as tools that work by themselves, while general transformation systems are a collection of transformations under a general system. General transformation systems are dependent on a complicated infrastructure, but can offer services to a broader domain than stand-alone generators. Transformations can be done as programmatic or pattern-based transformations. Programmatic transformations work by using separate programs for manipulating parts of the code. Pattern-based transformations use pattern languages to search for patterns in the code and replace them with suitable equivalents. Further, the transformation can be syntax- or flow-directed. Syntax-directed transformations make the transformations based on an internal intermediate representation. Flow-directed transformations use a control flow representation. Finally, the number of transformations applicable during the transformation process can vary between processes. Different transformation processes use a different number of transformations, depending on the methods that are used.

These transformation methods can be further divided into refinements or optimizations. Refinement implements concrete implementations according to an abstract specification, while optimization attempts to add efficiency while maintaining the abstraction level. The point of generators is refinement, where abstraction is transformed into a concrete implementation. Algorithm derivation is the most important refinement method. It takes declarative specifications and turns them into operational procedures that produce output according to specifications. An-

other method is data type refinement, which selects proper implementations for data types in the specification. [Smaragdakis & Batory, 2000]

In comparison to compilers, generators tend to offer more opportunities for optimization. Smaragdakis and Batory [2000] state that optimization transformations can be categorized in partial evaluation or incremental optimization. They describe partial evaluation as a method that takes a fragment of the source code and optimizes it under the assumption that its parameters satisfy certain conditions. In practice it means that abstract portions are specialized to fit the context in which they are used, if certain parameters are met. The authors describe incremental optimization as a method which performs complex optimization in increments. They continue that while code restructuring is usually a part of compilers, it can also be found in generators. Code restructuring works by removing dead code, unrolling loops and moving unrelated statements out of loops.

Biggerstaff [1998] says that optimization is needed especially for source code, since generators frequently create source code that a person would never write. He notes that optimization focusing on a specific part of source code can encounter problems with dependencies when two separately located but otherwise related parts of source code are handled. Therefore, global dependencies should also be taken into account in addition to local information when changes are made. The article mentions, however, that optimization methods which work on higher level domain notations can recognize optimization opportunities that are not present at the source code level. Some reasons why generators can produce more efficient output than human programmers are also listed. The fact that a generator can go through several different implementations faster than a human can is a major factor. Some optimization methods take more time from humans, which might not make manual implementation worth the time.

According to Biggerstaff [1998], the key factor in reuse success is domain knowledge. What makes a generator so effective is that it takes product line and other relevant information into account when input specification and transformation methods are decided [Frakes & Kang, 2005]. A component library might get too large when the applications that use it get more complex. In such cases generators are a better choice, as they can produce the components a component library would contain, but take less space [Smaragdakis & Batory, 2000]. Biggerstaff [1998] also notes that since the exact form of the output is not known during the transformation, it is not feasible to enter every possible output the generator can

produce into a library. Advocates of generators prefer them over libraries when a product is designed for reuse or the target domain shows systematic variability [Smaragdakis & Batory, 2000].

The size of the generated assets also makes a difference. Generation techniques are better at a subsystem level (tens of thousands of lines of source code), but after that composing systems from concrete components gives greater benefits. At a low scale, concrete components have several shortcomings, such as poor performance, difficulties in fitting with other components, missing functionality and the cost of customization. At a low level, generation-based reuse is more beneficial over concrete components. The best strategy seems to be generative reuse up to the subsystem level and composition with concrete components after that. [Biggerstaff, 1998]

The logic of an application is something that will probably not change between application versions for different operating systems. Therefore, an application generator could be a suitable tool for developing the logic of multi-platform mobile applications. An application can be ported for another mobile operating system, if the generator supports different types of output. The user interface and interaction methods are variation points between operating systems, which makes them parts that could need manual customization.

4.2 Mobile application generators

Several mobile application generators, such as RhoMobile Suite[1] and Sencha[2], are available online. Although several studies about application generators have been done, research focusing on mobile application generators is difficult to find. The research on mobile application generators is mostly limited to informal blog entries lacking a systematic approach. The methods mobile application generators employ often differ from the descriptions of generators used in the scientific literature. A well-designed generator could speed up development when multiple operating systems are targeted.

---

[1] `http://www.motorolasolutions.com/US-EN/Business+Product+and+Services/Software+and+Applications/RhoMobile+Suite`

[2] `http://www.sencha.com/`

Since the domain is clear (mobile applications), application generators are a well-grounded solution. Generated applications can have any kind of content, but some more specific domains, such as mobile games, have their own, more domain-specific tools and frameworks available. The application logic can be shared between operating systems, while the user interface and interaction methods would be modified to match the used operating system. In an ideal situation, a developer would be able to define the user interface components for the generator in a generic manner, which would then automatically be transformed into operating system-specific solutions. This appears to be something the generator developers are aiming towards, but due to the differences between user interface components, there are several exceptions which need to be taken into account during application development. For example, the button widget provided by Appcelerator Titanium has four states[3]. Android devices can set a different background color for each of the states, but on iOS this functionality is not available. In case an application developer wants to use such operating system-dependent functionality in their Android application, they will have to make a separate version of it for their iOS application. Optimally, the generator would recognize such cases and default to some basic functionality on operating systems that do not support specific features. The current method limits the developer to use common functionality, if the same source code is to be used for several operating systems.

One advantage of using mobile application generators is the possibility to use the same tools to create separate products for multiple operating systems. The tools themselves are often available for different computer operating systems, as opposed to the native tools from for example Apple and Microsoft, where the development tools are provided only for the company's own desktop operating systems. On the other hand, companies like Google and BlackBerry, which are not as strongly attached to the desktop operating system market, provide their mobile device development tools for several desktop operating systems.

Development with mobile application generators seems to be done with programming languages other than the languages native to mobile operating systems. Some mobile application generators offer an option to create user interfaces with common mark-up languages, such as HTML and XML, which means that at least some knowledge about mark-up languages is needed. Mark-up languages are in general easier to use than actual programming languages and they are easily

[3]`http://docs.appcelerator.com/titanium/latest/#!/api/Titanium.UI.Button`

scaled to different screen sizes, which in this case makes building user interfaces a more user-friendly task. The users often need to possess some programming knowledge in order to finish the application, though, as mark-up languages cannot contain rich interaction. This might change in the future with the progress of the HTML 5 standard. The use of domain-specific languages was not observed. Graphical user interface (GUI) builders, which generate source code for user interface components, are rare for mobile applications, despite being common in desktop application development.

HTML 5 is a popular mark-up language with application generators, due to the additional functionality it provides when making web-based applications. Although the functionality is not as rich as with native applications, it is sufficient for content-driven purposes. The World Wide Web Consortium is working on an API which would allow web-based applications to better access advanced device features, such as the calendar, camera and different sensors.

The output type produced by mobile application generators is one additional variable. The output can be a native, web-based or hybrid application. There is some discussion what can be considered a native mobile application, and the term "native" is used relatively loosely when talking about the output of mobile application generators. This can be credited to the lack of clarity with the definition of the term. Generator developers can also use the term more loosely and assume different meanings for it to promote their products. The output is usually a finished application, since any customization can be done before the generator is used. The choice of the application type should be made according to the end user and developer requirements.

In general, mobile application generators take input in the form of non-native source code, as is seen in Figure 4.2. Since the studied generators are domain-specific, the input contains domain-specific elements, such as user interface components and interaction techniques. If a native application is generated, the non-native input is created using a specific API, which the generator uses to map the functionality to a native equivalent. As mobile application generators often use web technologies as their input, web-based applications are usually created using standard web technologies, but an application run on a server is optimized for mobile devices. Hybrid applications are packaged inside a native container along with necessary API components. The native container interprets the application's functions so that they can be run on the device.

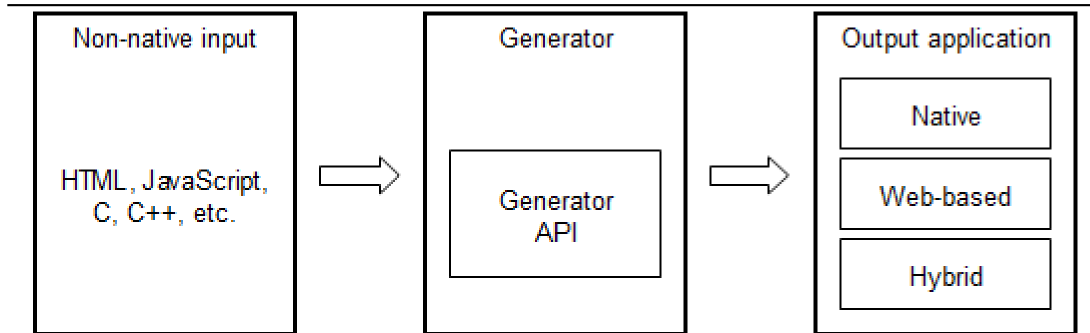

Figure 4.2: The simplified structure of a mobile application generator.

As there are several mobile application generators, each with its own working mechanism, describing the method with which they support reuse in a way that would cover all cases, is difficult. Generators creating native or hybrid applications enable reuse by providing a way to use one set of assets that is transformed into assets usable by different target operating systems. Thus, an application developer has to create only an abstract version of an application using tools and languages that are supported by a generator. The generator will then replace abstract implementations with target operating system-specific implementations. The application type also affects reuse. For example, web-based applications can use the same application logic, located on a server, across all target operating systems.

Some generators that modify web pages enable reuse in a similar way, but due to the generation process being more simple, the reuse effect is not as extensive. These generators simply transform existing HTML pages into another version, which is more suitable for mobile devices. Any functionality on the page remains the same, meaning that any functional logic can be used on both versions.

Examining some of the existing generators, a few characteristics were identified. The generators support different target operating systems, while being available for several desktop operating systems. The available functionality depends on the target operating system, the completeness of the generator and the application type. As stated earlier, native look and feel is important for the application users, so being able to provide that is an advantage. Therefore, some generators are advertised as such. The development languages also vary between generators, although HTML and JavaScript seem to be rather common. API documentation is structured, and videos and examples are often available. These help in learning

and understanding the tool and its usage. Together with the easily learnable and common implementation languages, this speeds up development times.

5 Metrics and criteria for mobile application generators

There are several tools available to help with mobile application development for multiple platforms. Although these tools share some features, there are always differences that might make one tool a better candidate for the users' needs. Some previous research exists that evaluates different aspects of reuse in software development, but research on application generator evaluation specifically is difficult to find.

Multi-platform development can be similar to the use of software product lines, making reuse a suitable approach. If a developer can create an application capable of running on different operating systems by using the same core parts in all of them, these core parts are considered to be reused and that decreases the overall effort. We first look into previous research and try to identify evaluation methods and attributes which would also be relevant when evaluating mobile application generators.

In this chapter, the terms entity, information need (or characteristic), measurable concept, attribute, metric, and criteria are used. This thesis uses the following definitions for them. *Entity* is considered to be the target product of the evaluation [Kitchenham *et al.*, 1995]. *Information need* (or *characteristic*) is a common feature in the product, something that is common to similar entities. *Measurable concepts* are sub-characteristics of the information need, further defining it. *Attributes* are the measurable aspects, with a value assigned according to the method described by a *metric* or *criterion*.

5.1 Previous research

Previous research on reuse evaluation is concentrated around software components, which is reflected in this section. Our approach is to adapt the findings from component evaluation research to mobile application generators.

5.1.1 Software component evaluation

Washizaki *et al.* [2003] suggest metrics that can be used to evaluate the reusability of software components. Their research focuses on black-box components, which rules out inspecting internal functionality. Black-box components are a common way to distribute commercial components, often leaving the control over the component functionality solely to its creator. In their paper, three tiers for representing different activity levels during development are defined as seen in Figure 5.1. These tiers are factors, criteria and metrics, and they represent the management level (as nonfunctional requirements), the application design level, and the product level respectively.

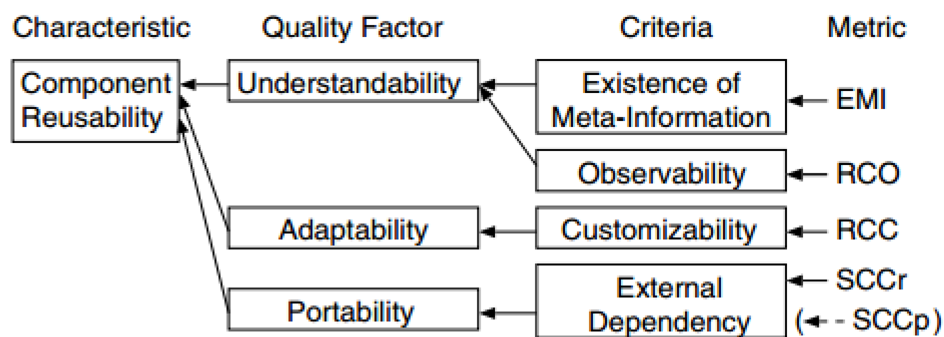

Figure 5.1: A model for component reusability [Washizaki *et al.*, 2003].

Three factors are selected by Washizaki *et al.* [2003] to represent component reusability by examining activities during black-box component reuse. These are understandability, adaptability and portability. According to the authors, these were chosen only to analyze the reusability of the components, and no other quality aspects were included. The authors mention that understandability is an important factor for developers, since the knowledge whether the component fulfills the requirements set for it is needed. Understandability is based on the effort needed to understand the concepts behind the component and how applicable the component is in a given situation. Adaptability is the effort needed to integrate the component into a new environment, which can be outside the context of the component's original and intended use. Portability is simply the effort required to use the component in a new environment.

Understandability is divided into existence of meta-information and observability criteria. Meta-information helps the users of the component to learn and understand the component's functionality. Therefore, it is important to know if such information is provided together with a component. Observability is evaluated, since black-box components cannot be thoroughly investigated by examining their source code. The only way to get information on what a black-box component contains is by observing the input parameters and the form of the output. Available read methods are mentioned as an important element in black-box observation. [Washizaki *et al.*, 2003]

The adaptability factor consists of the customizability criterion, which means the built-in customizing and configuring capabilities of the component. These capabilities are handled through write methods, which manipulate the internal features of the component. [Washizaki *et al.*, 2003]

Lastly, portability is defined by the external dependency criterion. It indicates how independent the component is from the system in which it was originally used. [Washizaki *et al.*, 2003]

Washizaki *et al.* [2003] assign a metric for each criterion. The metric for the existence of meta-information is called with the same, existence of meta-information (EMI). As their research focuses on JavaBeans components, the value of the metric is zero or one, depending on whether the component has a BeanInfo class delivered with it. The function of the BeanInfo class is to provide a developer with information about the component's usage. This metric is very specific and usable only with JavaBeans components, but it clearly states the need for documentation when using reusable components.

The rate of component observability (RCO) is proposed as a metric for observability. It is the percentage of readable properties available for the fields in a class. It is noted, though, that a high observability rating could make finding the relevant readable fields difficult due to the large number of properties available. Rate of component customizability (RCC) is the metric for customizability and works similarly as the rate of component observability, but read methods are replaced with write methods. A high rate of component customizability also has its drawbacks, since it can break the rules of encapsulation and give the user of the component more control than they should have. [Washizaki *et al.*, 2003]

External dependency is measured with metrics called the self-completeness of a component's return value (SCCr) and the self-completeness of a component's parameter (SCCp). They are used to analyze business methods, which are the methods not categorized as read or write methods. Self-completeness of a component's return value indicates the percentage of business methods without return values out of all business methods. Similarly, self-completeness of a component's parameter is the percentage of business methods without parameters. Methods without return values or parameters are said to be less likely to depend on external sources. The proposed approach is said to facilitate the development with reusable components by aiding in the evaluation process. [Washizaki *et al.*, 2003]

Bertoa and Vallecillo [2002] have researched quality attribute information suitable for evaluating COTS components. Bertoa *et al.* [2003] define COTS components as finished commercial components sold or licensed to other users. Bertoa and Vallecillo [2002] say that there are no generally agreed upon characteristics available for the purpose of evaluation. The authors mention that there is a lack of quality metrics (also mentioned by Bertoa *et al.* [2003]). Her *et al.* [2007] note that current quality models are too conceptual to be used in actual projects, due to abstract metrics only usable as subjective measures. Another issue identified by Bertoa and Vallecillo [2002] is that component vendors often leave such quality information out of their documentation, making it difficult for potential users to evaluate the components. Any existing standards are difficult to apply as well, since they are too general and cannot be directly applied to specific domains. In order to address these issues, the aim of the paper is to propose a set of attributes that vendors could use in their product descriptions to help developers and designers in assessing the components they are interested in.

The proposal made by Bertoa and Vallecillo [2002] is based on the ISO/IEC 9126 [ISO, 1991] quality model. According to Her *et al.* [2007], the standard was designed to assess complete applications, making it less suitable for the evaluation of components. Bertoa *et al.* [2003] define a quality model as a set of characteristics and their relations that provide a basis for quality requirements and evaluation. They also define characteristics as a set of properties which describe the quality of the product and can be used in evaluation. The characteristics proposed for COTS components by Bertoa and Vallecillo [2002] include functionality, reliability, usability, efficiency and maintainability. Sub-characteristics are divided into runtime and lifecycle categories. Runtime sub-characteristics are de-

scribed as performance-related. Lifecycle sub-characteristics are present during different steps of the development process (for example maintainability). The authors also list measurable attributes for each sub-characteristic. The list will not be completely covered here, and only the attributes relevant to this thesis will be explained. The relevant attributes are shown in Table 5.1. All of the selected sub-characteristics come from the lifecycle category.

Characteristics	**Sub-characteristics**	**Attribute**
Functionality	Suitability	Coverage
		Service implementation coverage
Reliability	Maturity	Evolvability
Usability	Learnability	Time to use
		Time to configure
	Understandability	User documentation
		Training

Table 5.1: Attributes relevant to this thesis based on the work of Bertoa and Vallecillo [2002].

The most interesting attributes from the study regarding reuse are coverage, service implementation coverage and user documentation. These attributes have a darker background in Table 5.1. Coverage and service implementation coverage are attributes that help to determine if a component contains the needed or expected functionality, while user documentation provides information on how easy it will be to comprehend the possibilities and limitations of a component.

Coverage is an attribute measuring the suitability sub-characteristic, giving the percentage of functionality that is provided by a component, when compared against the functionality required from the component. Another interesting attribute for suitability is the service implementation coverage, which compares the amount of functionality implemented by a component against the functionality specified by a standard (or some other specification) applied to the component. The next attribute falls under the understandability sub-characteristic, which is related to learnability. It is said that in order to learn how to use a component, it has to be understood first. User documentation measures the quality of the available user documentation, not the amount. [Bertoa & Vallecillo, 2002]

While not directly connected to reusing components, the following attributes are considered relevant to reuse. The maturity of a component is measured with evolvability, which is the number of the marketed versions of the component. The learnability sub-characteristic has two interesting attributes, which are time to use and time to configure. The first one is the average time for a user to learn how to use the component, while the second one measures the average time to understand the configuration parameters and actually configuring the component. Training indicates if courses on the use of the component are available, and it falls under the understandability sub-characteristic.

In another study by Bertoa and Vallecillo [2004], the authors focus on finding usability metrics that would help users in selecting software components. They note more issues with the existing methods, such as the fact that one attribute is often assigned to a single characteristic. According to the authors, this is not the case, since an attribute may affect several characteristics. For example, the size of the component may affect both maintainability and learnability. They also say that a metric needs to be connected to an attribute or a characteristic which it is measuring. This is something most proposals do not accomplish. The different characteristics and attributes are clearly defined in the study by creating a deep categorization structure from an entity to metrics. It is stated that the goal of the measurement process is to satisfy the information need by identifying necessary entities and their attributes. Attributes are the measurable aspects of the entity. Attributes are external when they are dependent on the environment in which they operate, or internal when there are no environmental dependencies. Attributes are measured with metrics which have a set scale and a unit. External attributes are suitable for black-box evaluation, although internal attributes can still provide indirect information about external characteristics [Bertoa & Vallecillo, 2002].

Bertoa and Vallecillo [2004] claim that the ISO/IEC 9126 standard metrics are ill-defined, as metrics are assigned to sub-characteristics without any reference or justification. Nevertheless, the authors choose the standard as a base for their usability metrics. They also say that since usability depends on the intended use as well as the user, the intended use and user need to be specified. The authors also try to achieve a balance between information required to get the metrics and information provided by component developers, as information about a component is usually difficult to find. Over thirty metrics are assigned to the quality of documentation and complexity of the design concepts. Some issues

with these metrics are also identified, such as the subjectivity of some of them, and possible ambiguities in the terminology used. However, the authors plan on improving the research in the future.

A study by Her *et al.* [2007] aims to find a method for evaluating the core assets in software product line engineering. In order to achieve this, they first identify the common characteristics of core assets. Based on these common characteristics, the authors define a set of quality attributes used in core asset evaluation. Two of the attributes they assign to the characteristics are based on the ISO/IEC 9126 standard, five others are created by the authors based on the common characteristics they identified in product lines. The characteristics, attributes and metrics relevant to this study are the only ones explained here, and are shown in Table 5.2.

Characteristics	Attributes	Metrics
Common functionality	Functional commonality	Functional coverage
	Applicability	Cumulative applicability
Embedding components	Component replaceability	Component compliance
Instantiation	Understandability	Overall understandability

Table 5.2: Attributes relevant to this thesis based on the work of Her *et al.* [2007].

The first attribute they define is functional commonality, which is used to evaluate the commonality of an asset's functionality in the applications of a product line. Another attribute, applicability, measures the possibility of applying a core asset to other product line members. Component replaceability is an attribute for measuring the possibility of swapping one component for another without having to make changes in the software architecture. Finally, understandability measures the ease of learning and understanding the use of a component. [Her *et al.*, 2007]

Interesting metrics regarding reuse include functional commonality and overall understandability. They are marked with a darker background in Table 5.2. Functional commonality is relevant to reuse because commonality can indicate opportunities for reuse. Overall understandability helps users of a component

to use it correctly and efficiently. Functional commonality is measured with the functional coverage metric. This metric uses the number of products in a product line and the number of applications using a functional feature to get a percentage value. Understandability is measured with overall understandability, which is defined as the number of comprehensible elements divided by the total number of elements. In this case the elements are items that help in understanding the component, such as manuals, specifications and guides. When understandability is low, it might slow down the use of the component. [Her *et al.*, 2007]

The following metrics are less relevant to reuse, but they provide additional information that can be used when evaluating components. The metric for applicability, called cumulative applicability, is more complicated, as it uses other metrics as a composition to determine the final value. The metrics for the composition are functional commonality, non-functional commonality (used to measure non-functional requirement commonality), and coverage of variability (used to measure variability richness). These three metrics are optionally weighted with values according to a formula or by an assessor familiar with the characteristics of the project in which the components are used. Component replaceability can be measured with component compliance, which is the number of replaceable components in an asset divided by the total number of components in the asset. A high number of replaceable assets makes it easier to find components to replace. [Her *et al.*, 2007]

5.1.2 Component information

A study by Bertoa *et al.* [2003] concentrates on CBSD, which is assembling components into working software. Components are defined as a collection of functions with interfaces for interaction with other components. The study is aimed at comparing the quality information given by the component providers against the information required to analyze the components. Several issues were identified. A major issue was the difficulty of obtaining information about the provided products, since there was no standard place for it in the documentation. This rules out the possibility of gathering the information automatically. The information that was found was in some cases already processed and ready to be used, while in the rest of the cases the information needed processing in order to acquire the desired attributes and their values. The authors also note that only a small percentage of all the products provided values for their metrics. The

research also discovered that data on self-tests and test suites were often lacking. This requires the user to retest the component, thus losing some of the benefits of CBSD. The authors concluded that the provided quality information should be well organized to enable automatic comparison.

5.1.3 Evaluating mobile application types and approaches

The study done by Heitkötter *et al.* [2012] is different from the others, since it does not propose metrics for reusable components, but compares the different application types currently used in mobile application development. The authors do not propose direct metrics, but rather subjectively evaluated criteria. The criteria for the comparison were created according to discussions with domain experts, along with the help of the experience gained making prototype applications. The generated fourteen criteria were divided between infrastructure and development perspectives, where infrastructure criteria cover the lifecycle of the application, including its usage, operation and functionality. Development criteria cover the development process, including testing and development tools.

Of the infrastructure criteria, license and cost is defined first. This criterion evaluates the distribution license, whether or not commercial software can be created with that license, and possible costs regarding the use of the application type. Supported target operating systems are also evaluated, which is done by considering the number and importance of the ones that the method supports. The evaluation also takes into account the equality with which the target operating systems are treated. Next, access to advanced device-specific features is evaluated by comparing the features available through an API. Basic functionality is covered by most frameworks, which is why access to advanced features, such as NFC and accelerometer, is focused on. The look and feel of the created applications is also evaluated, as it is an important factor for users expecting behavior found in native applications. [Heitkötter *et al.*, 2012]

With these criteria, Heitkötter *et al.* [2012] evaluated native and web-based applications, along with hybrid applications created with PhoneGap and Appcelerator Titanium. The evaluation was done together with experienced developers, while also being based on the research done by the authors. A grade from very good to very poor was assigned, as well as a verbal explanation of the matters affecting the grade. All development methods were found to have their strengths and weak-

nesses, although cross-platform approaches can be the most efficient even when only a single mobile operating system is targeted. According to the authors, the efficiency of cross-platform approaches comes from the used development techologies, which are common and easily learned, requiring application developers to learn less platform- and framework-specific skills.

5.2 Suggested metrics and criteria

Most evaluation proposals, as well as the ISO/IEC standards 9126 [ISO, 1991] and 25010 [ISO, 2011], categorize their metrics in some way. The research done by Bertoa and Vallecillo [2004] says that most of the software quality evaluation methods they have studied do not pay enough attention to the quality characteristics that they are assessing or the attributes that are measured. The authors suggest a collection of metrics with a deep category structure. This can be useful for developers wanting to study a specific aspect of a product, since they can pick the suitable attribute or concept at the necessary level and apply the metrics that fall under that category.

Therefore, an approach to identify attributes and characteristics for mobile application generators is assumed to be a suitable one, when new evaluation metrics are defined. The metrics and criteria are meant to be used in a black-box manner, since information about the internal functionality of some of the generators is difficult to find. Although some generators are open-source, we feel that going through the source code of a generator can be too time-consuming and requires extensive programming knowledge. The aim is to create a relatively lightweight set of metrics and criteria that can be used to assess the generators based on user needs. Most of the information needed for the evaluation of a generator should come from the documentation and include as little of actual generator use as possible. Configuring potential generators for testing purposes can take a significant amount of resources, especially when they might have to be configured to work with a number of target operating system.

5.2.1 Entity, user group and characteristics

As a base for the quality metrics we use the structure which Bertoa and Vallecillo [2004] introduced in their study. The structure will not be completely identical,

though, since it is consider to be too heavy due to its large number of indirect metrics assigned to attributes by using an indicator category. The number of category layers is too large for the smaller number of criteria and metrics defined here. However, the target entity, its characteristics (or information need, as they are called by the authors), measurable concepts, assessable attributes, and their criteria or metrics will be defined based on the discussion of issues in mobile application development, current mobile application generators, and prior research on software and reusability metrics. Table 5.3 shows the outline of the attributes considered important when developing for multiple mobile platforms using mobile application generators with asset reuse as an objective.

Entity	Information need	Measurable concept	Attribute
Mobile application generator for multi-platform development	Functional suitability	Functional completeness	Functional coverage
			Total functional coverage
		Functional portability	Supported target operating systems
			Output application types
			Operating system specificity of assets
	Usability	Learnability	Documentation and guides
			Development languages

Table 5.3: Attributes assigned to mobile applications generators.

According to the ISO/IEC 9126 standard, a quality model needs to have a target user group. Here the suggested metrics and criteria are meant to be used by the developers of multi-platform mobile applications. The intended use is to evaluate mobile application generators for their suitability for multi-platform development for mobile devices. An entity is an object which can be observed in the real world and is the target of the evaluation [Kitchenham *et al.*, 1995]. In this case, the entity is a mobile application generator.

After defining what, for whom, and for what purpose the metrics are created, we can select the characteristics that are relevant. To this end, we use the characteristics described in the ISO/IEC 25010 standard. The ISO/IEC 25010 standard,

which succeeded ISO/IEC 9126, defines a product quality model which is applicable to software products. The characteristics of interest from this standard are *functional suitability*, *usability*, *reliability*, and *portability*. Functional suitability and usability are the most relevant to identify reuse opportunities and reuse effectiveness.

Functional suitability will be used to assess the degree to which the tool provides the functionality that is required by the developer. Evaluating a set of generators from the suitability perspective allows a developer to estimate which generators offer the best coverage of device functionality access and how much adaptation is needed for each target operating system. Functional suitability also helps to evaluate the extent at which the developer can reuse created source code assets, when targeting several systems. If the input needs only minimal modifications in order to be used by the generator to create applications for several target systems, input assets can be reused.

Usability is the effectiveness and efficiency of the generator's use. In order to enable reuse, a generator needs to be understandable and users need to know how to use it. Documentation has a large role in these matters. The actual use of the generator needs to be efficient or possible reuse benefits are lost and native applications could be developed instead.

Reliability is used to assess if a generator performs in an expected manner, as well as the state of the generator's development. Since one of the benefits of reuse is increased quality, properly tested and mature generators are more likely to produce applications that work as expected. Errors in the generation process can cause systematic errors in the generated applications, which makes generator quality an important factor.

Portability determines how restricted the use of the generator is and how well it supports different operating systems (desktop operating systems running the generator). We feel that developers need to know what the external limitations (not directly caused by the generator itself) for using a generator are. Application developers are probably interested in using the generator on their preferred operating system, as well as having control over the licensing of the generated application.

When mobile application generators are used for multi-platform development, the user has to note that a generator might provide more functionality for some

target operating systems. Therefore, functional suitability was chosen to evaluate how well a generator supports the functionality that a developer requires in their applications. Functional suitability will also help in determining what kind of output types the generator offers, and if that output can match the user interaction methods of native applications. Usability will help in evaluating how effectively the generator can be used and how long it takes to learn how to use it. Like with all tools, the user needs to understand their usage, making usability a factor also for mobile application generators. If the tool is not effective, its users could as well develop native applications separately for each platform. Reliability is thought to be an important factor, as mobile application generators are a relatively new phenomenon. Especially organizations might want tools that they can rely on and use in professional development. For example, tools in their beta stages can change dramatically between release versions, which can cause problems. Portability is evaluated, since some tools are not available on all desktop operating systems or they have licenses that are not suitable for the user, making such generators useless, even though the other criteria are met. Thus it is considered a factor when choosing a tool.

5.2.2 Measurable concepts and attributes

As Bertoa and Vallecillo [2004] suggest, measurable concepts are defined next. These are based on the sub-characteristics defined in the ISO/IEC 25010 standard, but with the definitions modified to fit the evaluation of application generators. Functional suitability is further divided into *functional completeness* and *functional portability.* The purpose of functional completeness is to examine the functions provided by the generator in order to find out how comprehensive the provided functionality is when compared to functionality required by the generator user or the native functionality of the target operating system. Functional portability examines how well the generated applications can be ported to different mobile operating systems, as well as the types of applications the generator can produce. Usability is divided into *learnability*, *accessibility* and *operability.* Learnability is used to assess the used technologies, documentation, guides, and other material created by the generator developer to help users in understanding what they can do with the generator and how their tasks can be completed. Accessibility is used to determine what are the skills required from the user. Operability examines the ease of use of the generator and what kind of support is

provided. Reliability is measured with *maturity*, which tells how evolved the generator is. Portability is evaluated with *installability*, which examines how easy the generator is to acquire, install and configure before use.

The goal of setting these attributes is to evaluate the potential for reuse as well as the general usefulness of a generator. Especially functional completeness and functional portability are characteristics that help in identifying how well the created assets can be reused to create applications for several target operating systems. For example, the supported target operating systems attribute tells how widely the assets can be applied. A generator that supports four target operating systems wanted by an application developer is more more efficient than using two separate generators that together support the same four target operating systems. In the first case the developer only has to create input for one generator, while in the second case two different inputs are needed.

Learnability is considered important for reuse because it evaluates the ease of comprehending what the generator does and how tasks can be completed with using the generator. If the functionality of the generator is not clearly understood, it is possible that the output contains errors or is inefficient. The rest of the characteristics are not as directly connected to reuse, but help to determine if the generator is appropriate for a user.

The next step is to divide the concepts into attributes, which can be measured or evaluated. We start with functional completeness and assign the attributes *functional coverage* and *total functional coverage* to it. Functional coverage is the portion of the generator functionality that the user requires. This attribute is based on the coverage attribute suggested by Bertoa and Vallecillo [2002]. Due to the differences in support mobile application generators provide for different mobile operating systems, functional coverage works best if a value is assigned separately to each target operating system. Total functional coverage is based on service implementation coverage, also suggested by Bertoa and Vallecillo [2002]. It has similarities with functional coverage defined by Her *et al.* [2007]. Total functional coverage is defined similarly to functional coverage, except that it compares the functionality a generator provides to the functionality provided by the native API of a mobile operating system. The total functional coverage can require immense effort to determine and it would be best if the evaluation was done by the generator provider, as they are the most familiar with the functionality of the generator. This attribute also works best if used separately for different

mobile operating systems. To clarify, functional coverage is targeted more towards application developers, while total functional coverage should be used by generator providers.

Functional portability is divided into *supported target operating systems*, *capability for native user interfaces*, *output application types* and *operating system specificity of assets*. In multi-platform development, the number of supported target operating systems is a key issue. Having a generator supporting several target systems makes multi-platform development easier, since the same tools can be used during the development process, while there is a possibility that some created assets can be reused. The supported target operating systems attribute is similar to supported platforms described by Heitkötter *et al.* [2012] and adaptability in the ISO 25010 standard. Capability for native user interfaces is measured, since Charland and LeRoux [2011] state that the lack of native user interface components worsens the user experience. For that reason the capability of the generator to create native user interfaces is considered important. This attribute is similar to the look and feel criterion suggested by Heitkötter *et al.* [2012]. Output application types is an attribute used to determine if the generated application type (native, web-based or hybrid) suits the needs of the user. A specific type might be preferred by the application developer, although other types would not necessarily limit the functionality of the application directly. For example, a simple web-based application could contain the required functionality, but the developer might not want to maintain a server to run it. The attribute is not based on any previous research, but we recognize it as a characteristic of mobile applications and application generators. Lastly, the operating system specificity of assets evaluates if some of the functionality provided by the generator is tied to some specific target operating system. If not, those assets could be easily recycled for other operating systems. Reusability in the ISO 25010 standard and applicability suggested by Her *et al.* [2007] are similar to this attribute. These attributes were chosen, as reusability is considered important in multi-platform development. Washizaki *et al.* [2003] note that reusability needs to be measured efficiently.

Learnability has attributes for *documentation and guides*, *training*, and *development languages*. Documentation and guides evaluates the number and quality of documents that are provided to understand and learn the usage of the generator. Development with a tool can be enhanced with good documentation. Especially API documentation is important, since it is the main source of infor-

mation on what can be done and how. This attribute is chosen based on the user documentation attribute by Bertoa and Vallecillo [2002], and the existence of meta-information attribute by Washizaki *et al.* [2003]. Training is an additional method of learning to use the tool, and for example organizations can be interested if the generator providers offer formal training. For this reason, there is a dedicated attribute set for this aspect of learnability. It is based on the training attribute set by Bertoa and Vallecillo [2002]. Development languages are a factor for application developers, since they might be familiar with some of them. It is placed in the learnability category, as we feel it primarily affects the familiarity of development, whereas functional portability is more about the output of the generator. No mention of similar attributes was found in the previous research, but we identified it as a characteristic of mobile application generators and selected it to be used here.

The accessibility concept has just one attribute, *programming skills required*. This attribute is used to determine whether the generator can be used without prior programming skills. Generators that can be used without programming knowledge could take input specifications in the form of HTML or the output from a user interface builder. The created applications could be simple in their functionality but enough for the purposes of some users. The attribute is based on the accessibility in the ISO/IEC 25010 standard and the suggestion by Cleaveland [1988], which states that generators should not require programming skills from their users.

General usability, *customer support* and *community availability* are the attributes for operability. General usability is used to assess how easy it is to use the generator and its related tools. This is important for efficient use of the tool, and is similar to the general usability characteristic in ISO/IEC 9126 and 25101 standards. Customer support can help the user in case of problems with using the tool, which again is important for organizations wanting formal solutions. For others, community availability is a more important attribute. Sometimes the fastest solutions come from other users, who can help users of the generator to understand issues outside of the official documentation. The last two attributes are not noted in any previous research, but such features were noted to have been made available by several generator providers.

Versions is the only attribute for maturity. It can be used to determine whether the generator has been in steady development for long enough, so that it does not

have major flaws due to a short time of public usage and feedback. Experimental tools might provide advanced functionality, but can also be a risk if the final product is not what was expected. Again, organizations looking for professional development tools are probably more interested in tools that have been in development for some time, as they are less likely to have critical programming errors present. The same attribute has been noted by Bertoa and Vallecillo [2002] as evolvability.

The installability concept is important in cases where an organization uses a number of different operating systems and hardware for development. It is divided into *setup time*, *license* and *supported operating systems* attributes. Setup time tells the user about the ease of installing and configuring the generator prior to its use. Some tools might be difficult to set up, making them deficient for smaller development projects. This attribute is modified as the combination of time to use and time to configure attributes described by Bertoa and Vallecillo [2002]. The type of the license the generator is distributed under might pose limits to its usage. Therefore, it is set as a separate attribute. For example, low budget projects might prefer open source solutions, while those with more resources can purchase commercial products with better support options. The same attribute was suggested by Heitkötter *et al.* [2012]. Finally, since the generators can be used on different operating systems, the information saying which ones it can be installed on is important. As mentioned earlier, since some native development tools are limited to one specific operating system, mobile application generators could help here, if they can be used to develop cross-platform applications for those systems. Previous research does not mention this attribute, but due to the varied operating systems used by organizations and the possibility to bypass operating system restrictions, it was selected.

5.2.3 Metrics and criteria

In the final part of defining the evaluation method, a metric or criterion is assigned to each of the attributes. These criteria are based on previous research, if such research has been available. Table 5.4 summarizes the metrics. The suggested criteria and metrics are meant to be lightweight. Calculating values such as number of documentation pages and their images, as suggested by Bertoa and Vallecillo [2004], are considered to be too heavy on the user, especially if metrics with complicated formulas are used. Less strict criteria might require more

expertise to be assessed correctly and can be more subjective. However, the assessor can take into account the specific needs and requirements of the project, similarly to the criteria suggested by Heitkötter *et al.* [2012]. If suitable generators are identified, they can be selected for a closer inspection with more precise evaluation methods.

Attribute	Metric or criterion
Functional coverage	$\frac{\textit{needed functionality} \cap \textit{provided functionality}}{\textit{needed functionality}} \times 100\%$
Total functional coverage	$\frac{\textit{provided functionality}}{\textit{available functionality}} \times 100\%$
Supported target operating systems	List of supported target mobile operating systems.
Output application types	List of produced output application types from the set of native, web-based and hybrid.
Operating system specificity of assets	Grade from one to five, one being very operating system-specific and five being operating system-independent.
Documentation and guides	Grade from one to five, one being lacking and unstructured information, five being good quality and structured.
Development languages	List of accepted input languages for the generator.

Table 5.4: Attributes with their metrics and criteria.

The metric for functional coverage is essentially the same as the metric for coverage proposed by Bertoa and Vallecillo [2002]. Instead of interfaces, functionality is used and the metric is defined as the intersection of functionality needed by the developer and provided by the generator, divided by the needed functionality. The specific formula can be seen in Table 5.4. This metric is rather simple to use when the requirements for an application are known. If exploring available generators in general, the metric for total functional coverage is defined as provided functionality divided by all functionality available for that operating system. The formula is displayed in Table 5.4. The metric is also similar to the one suggested by Bertoa and Vallecillo [2002] for service implementation coverage. Although this metric is simple to calculate, the fact that generator developers do not list the provided functionality in the same manner as mobile operating system providers do makes it difficult to gather the necessary information. Therefore, in an optimal situation, this metric is calculated by a generator provider, as mentioned earlier.

The criterion for supported target operating systems is simply a list of mobile operating systems. This information is usually easily available in the generator documentation. A list is used instead of the number of supported operating systems, since it is not considered to be much more work to determine and list them. Heitkötter *et al.* [2012] do not define their criterion in the same way, as they give a grade for a development method's suitability for multi-platform development. The definition by the same authors also affected the next criterion, the capability for native user interface components, although they evaluate it with a grade, and in this thesis a simple yes/no choice is used. The information required to get the result should be relatively easily found in the documentation for the generator, as well as by examining example applications. The criterion for output application types is also presented as a list. It might be difficult to find the information from the generator documentation, since the difference between a native and hybrid application can be hard to detect, and the documentation can use ambiguous terminology in this matter. For this purpose it could be easier to examine the output of the generator, if possible. Operating system specificity of assets is evaluated with a grade from one to five (although this type of grading can be changed to suit the user, for example from one to three could be just as good), one meaning assets are tightly connected to a specific operating system and five meaning that the assets are operating system-independent. The main factor in specificity is the generator's API and the way it provides its functionality and customization. The metric can be evaluated through documentation. The applicability metric by Her *et al.* [2007] is considered unfitting in this context, since it uses a combination of other metrics, which are not used here, to get the final value.

The evaluation of documentation and guides is done with a grade from one to five, one standing for lacking and unstructured information and five meaning good quality and well-structured information. The metric proposed by Washizaki *et al.* [2003] for meta-information only checks if meta-information exists, but the metric for user documentation by Bertoa and Vallecillo [2002] is essentially the same as the one we use here. Training is evaluated as either yes or no, depending whether the generator provider offers formal training on how to use the generator. The metric is the same as the one used by Bertoa and Vallecillo [2002] for training. The criterion for development languages is a list of languages that can be used as input for the generator.

Required programming skills is another criterion with a yes or no value. This information should be easily deductible from the development languages and the possible API documentation available. However, the generator can provide an application builder, which does not require programming skills, as an optional way of making applications. Therefore, this criterion is evaluated separately from the used development languages.

General usability is measured by assigning a grade from one to five, one being almost or completely unusable and five standing for good usability enabling efficient use of the generator and the related tools. The same measurement is used by Bertoa and Vallecillo [2002], called effort for operating in their article. The evaluation can be done based on the fact that the generator tools can be based on existing software. Otherwise, the evaluation requires using the generator. Customer support is evaluated to a yes or a no, depending on its existence. Generator providers often promote such possibility with different subscriptions of their services, making this information easy to find. The same applies to community availability, which is assessed in the same way.

The version metric is measured with the number of different versions that have been available publicly, since the official release. The ease of finding this information depends on the documentation provided by the generator developers. Open source tools might also have such information available in their repositories. The method of measurement is the same as the one used by Bertoa and Vallecillo [2002].

Setup time is measured with the time it takes to install and configure the tools for development. The necessary information can be acquired by completing the setup and configuring processes. Setup time is measured in the same manner as the metrics for time to use and time to configure by Bertoa and Vallecillo [2002]. The time is dependent on the hardware, software and internet connection speeds, which means that the measurements of different tools should be performed on the same machine. The metric is meant to give some idea of the time it takes to make the generator usable, and probably will not be the most important aspect of the evaluation. Minor differences could be caused by a number of factors. The criterion gives some idea on how much effort it will take to install and configure the generator in an organization with several workstations. Heitkötter *et al.* [2012] evaluate license type as a grade, based on its impact on development and distribution, but in this study the used license type is simply listed. The

evaluation of the license type's impact can be done later or by other stakeholders. The license type is usually well documented by the developers. Finally, supported operating systems are evaluated as a list. This information can probably be found before starting the installation process needed for the setup time metric, from the source where the tool is acquired.

5.3 Validation of the metrics and criteria

In order to justify using the proposed metrics and criteria, they have to be validated. Research on validating software metrics can be found, but information on how to validate criteria is more difficult to find. For this thesis, the validation will be done as Kitchenham *et al.* [1995] suggested, although it is meant for metrics alone. We will apply their method to the criteria as well as possible. In addition, evaluation will be done through practically applying the proposed metrics and criteria to existing mobile application generators in the next chapter.

First, Kitchenham *et al.* [1995] argue that for the measurement to be valid, the set attributes have to be exhibited by the entity. This is to be done to all attributes without regard to whether they are directly or indirectly measured. In this case all the attributes are considered to be present, if not in the generator itself, then in the additional services the generator developers provide, since they are also part of the generator assessment. No indirect measures are used.

The next step is unit validity, which aims to determine if the unit used in measurement of an attribute is appropriate. Some of the attributes are measured with a grade, which is thought to be appropriate in those cases, since the attributes reflect the opinion of the person performing the evaluation, making a calculable value impossible. Other criteria have units based on previous research and are defined through the characteristics and measurable concepts, making them proper units for the attributes.

Instrument validity requires the measurement acquisition method to be well-founded. In this case, the acquisition relies on the user to find the required information and interpret it correctly. Most of the attributes are evaluated as criteria instead of metrics, as there are some problems in defining clear metrics for them. For example, the easiest way to get the information whether the target operating system is supported by the tool is to check if that operating system

exists on a list of the supported systems. A metric could possibly be invented, but whether that is practical is another matter.

Finally, a measurement protocol needs to be defined. In this case, the tool needs to have a public, generally available version. The generator also needs to have some documentation, although the quality of it is part of the evaluation. Basic information, such as supported operating systems and development languages, needs to be available, or they cannot be evaluated. It should be noted, though, that even if the developer provides, for example, training, but the users cannot find information about it, training could as well be thought not to exist. Generator providers that get profit from extra services such as training would suffer from such hidden information.

6 Evaluation of existing generators

In order to further evaluate the proposed metrics and criteria, they will be applied to existing mobile application generators. Three generators were chosen. Freeware and open source solutions were preferred, since it was expected that more information would be publicly available on such tools. The chosen generators have also been in development for some time now, as we wanted to be sure that they are mature enough for evaluation. The number of generators was limited to three to keep the amount of work at a reasonable level, but more generators should be used in the future to get a better idea whether the metrics and criteria work as intended. As the used solutions are open source, anyone can acquire the necessary information from the source code, if needed.

For the sake of functional coverage, we chose five requirements for functionality and two target operating systems. This functionality includes NFC, Bluetooth, camera, GPS and access to contacts. We aimed to demonstrate how generators support features that are advanced or commonly used (or both), and we feel that the chosen functionality consists of such features. As the target operating systems, Android and BlackBerry OS were chosen. Android was chosen due to its significant market share, while BlackBerry OS was chosen to represent less popular operating systems that are still being developed. The functional coverage will be calculated separately for each operating system.

6.1 Appcelerator Titanium

Appcelerator Titanium[1] is promoted as an open source solution for mobile development. It was originally released in 2008, with the most recent update published in May 2013. The generator has its own SDK, which is used for application development, meaning that the users have to familiarize themselves with it prior to using it. The tool also comes with its own IDE, called Studio.

There is no clear and up-to-date information available on how the generator actually works. According to an old entry in the development blog[2], the generator works by packaging the user-created JavaScript source code as a binary file with

---

[1] `http://www.appcelerator.com/`

[2] `http://developer.appcelerator.com/blog/2010/12/titanium-guides-project-js-environment.html`

the Titanium API, which is interpreted at runtime by a JavaScript engine. The used engine depends on the target operating system. Included with the engine is a namespace that contains the necessary API information. Table 6.1 shows the evaluation of the Appcelerator Titanium tools. Version 3.1.1 was used for the evaluation.

Attribute	**Value**	**Notes**
Functional coverage (Android/ BlackBerry OS)	80%/0%	Bluetooth is available for Android as a paid module and is not taken into account here.
Total functional coverage	-	
Supported target operating systems	Android, BlackBerry, iOS, Tizen	BlackBerry SDK is currently in a beta stage.
Output application types	Native, web-based, hybrid	
Operating system specificity of assets	3	Especially user interface components have functionality specific to an operating system, but these are generally well documented.
Documentation and guides	5	Structured API documentation, guides, videos and examples.
Development languages	JavaScript, XML, CSS	XML and CSS used for Alloy.

Table 6.1: Evaluation of the Appcelerator Titanium tools.

The primary reuse mechanism of Appcelerator Titanium is creating multi-platform applications from a single code base. An architecture solution called Alloy is offered. Alloy provides a model-view-controller architecture, where views are created with XML and CSS-based TSS files. This framework for developing applications allows reusing the application architecture and lets a developer concentrate on the actual requirements of the application without having to pay as much attention to the application information flow.

For Android, Appcelerator Titanium provides functionality for NFC, camera, GPS and contacts. Bluetooth can be used with modules developed by third parties and are available on the marketplace. They are not free, though. The value for provided functionality is therefore four. Thus, the value for functional coverage is 80%. There was no support for BlackBerry OS with the selected

functionality, since the BlackBerry OS support is not yet complete. As mentioned earlier, total functional coverage is an attribute that is best delivered by the generator developer and will not be calculated here.

Currently, Appcelerator Titanium supports four mobile operating systems. These are Android, BlackBerry OS, iOS and Tizen. However, the BlackBerry OS functionality is in a beta phase, with some features still missing and under development. Native user interface components and native application types are mentioned in the documentation, but there is some argument whether interpreted languages are truly native. Native applications are said to require compilation. If not native, applications produced with Appcelerator Titanium are hybrid, but it also supports web-based applications. However, some of the provided functionality can be operating system-specific. User interface components are different in all of the target operating systems, which requires generator providers to offer functions that are able to create components for all target operating systems. Otherwise, application developers need to take these differences into account. The previously mentioned button example applies here.

Learning to use the generator is supported by a good amount of documentation, guides, examples and videos. Especially the API documentation and guides are well-structured and seem up-to-date. There are many examples and videos, although the videos have not been updated to reflect the changes made recently. Training is also available, but only as classroom courses in selected cities, and it is organized by the generator providers or authorized partners. More flexible solutions, such as streamed lectures, might be available but could not be found.

Using the generator requires programming skills. The user interface is built with a markup language, which connects to program logic made with JavaScript. Optionally, JavaScript can be used to create the user interfaces, as well. As a more advanced solution, Alloy is offered. The general usability of the generator and the related tools is good, especially for those who have used the Eclipse IDE, since the Titanium Studio is based on it. Installing the generator and the additional tools is largely automated. Updates for the generator are also automatically detected and installed with a few clicks. Testing an application with the Android emulator was also easy. Support is available with a paid subscription during working hours or even 24/7 from the support team, depending on the type of the subscription. A free subscription user has to rely on the community forums and a wiki.

At this moment, Appcelerator Titanium was found to have had thirteen releases. A release was considered to be a version with numbering X.Y, while more detailed version numbers were regarded as fixes. The information was gathered from the repository of the generator. The Titanium SDK, Titanium CLI and Alloy are distributed under the Apache Public License v2, while Titanium Studio is proprietary software. If a user wants to submit, for example, a fix for one of the open source products, they cannot submit the source code directly to the repository, but have to request the Appcelerator developers to retrieve and apply the changed source code. The generator is available for all the major operating systems, meaning Linux, OS X and Windows support. Installing and setting up the development environment for Android applications took about 25 minutes, most of the time spent downloading updates. The whole process was easily managed and required little input from the user.

6.2 MoSync

MoSync[3] is another open source tool and has been in development at least since the year 2010, when version 2.0 was released. The current version is 3.3 and it was released in May 2013. The generator comes with its own IDE.

Again, there is no recent documentation on how the generator works, but there is a blog entry from 2010[4] describing the transformation process in detail. According to the description, the transformation process starts from compiling the user-made source code with a modified GCC compiler into MoSync's own code, called MoSync IL. This code is combined with the compiled MoSync standard libraries using a pipetool. The pipetool produces either Java code or a binary version of the MoSync IL code. The Java source code is compiled with Java ME or Android runtime, depending on the target device. Binary MoSync IL sources are bundled with a runtime depending on the target device. Table 6.2 shows the evaluation of the tool, where version 3.3 was used.

---

[3] `http://www.mosync.com/`

[4] `http://www.mosync.com/blog/2010/01/under-hood`

Attribute	Value	Notes
Functional coverage (Android/ BlackBerry OS)	100%/40%	
Total functional coverage	-	
Supported target operating systems	Android, BlackBerry, iOS, Java ME, Moblin, Symbian, Windows Mobile, Windows Phone	Android and iOS have the best functionality coverage, other operating systems have limitations.
Output application types	Native, web-based, hybrid	Web-based application are developed with MoSync Reload.
Operating system specificity of assets	4	User interface components have operating system-specific features, but not on a large scale.
Documentation and guides	3	API documentation is structured, but the tool lacks examples and guides.
Development languages	C, C++, JavaScript, HTML, CSS	

Table 6.2: Evaluation of the MoSync tools.

MoSync provides support for reuse similarly as Appcelerator Titanium. A single code base can be used to create multi-platform applications. There are, however, several more target operating systems available making the potential for reuse greater. The API enables reuse of the application architecture, but not to the same extent as for example modern web frameworks.

With the same requirements for functional coverage as with Appcelerator Titanium, functional coverage for Android was found to be 100%. BlackBerry OS support was lacking, since only Bluetooth and access to contacts were supported. Finding this information was relatively easy, since MoSync has a feature/platform support table, listing which classes and functions are available for different operating systems. Total functional coverage was not measured.

MoSync supports a wide range of mobile operating systems. These are Android, BlackBerry OS, iOS, Java ME, Moblin, Symbian, Windows Mobile and Windows Phone. It should be noted that Moblin, Symbian and Windows Mobile are not developed anymore, making them less meaningful targets for application developers. Support for the newest versions of BlackBerry OS and Windows Phone is lacking. The documentation states that native user interfaces are available, but

this is largely dependent on the target operating system. Android seems to have the best support in this regard. The created applications can be native or hybrid when MoSync SDK is used, while MoSync Reload is meant for web-based application development. MoSync Reload can also be used together with the MoSync SDK Wormhole JavaScript API, which enables the usage of native functionality. Some operating system specificity can be found in the API, but this seems to be limited to a small percentage of the components. This could also be due to the fact that some functionality has very limited availability on different operating systems.

The API documentation is well-structured, but some functionality lacks examples and the descriptions can be short. Function-specific examples could help in learning how the function should be applied in practice. There are some guides, example applications and videos, though. Classroom training is available for a fee, while customer support is available with paid subscriptions. In addition, support from the community is available on a forum. The generator supports C, C++, JavaScript, HTML and CSS as possible input languages. An application developer can use C, C++, or JavaScript along with HTML, or a combination of those in their applications. This is possible with the Wormhole functions, which work as a bridge between JavaScript and C/C++ source code. HTML and CSS are used for creating user interfaces. Therefore, programming skills are necessary to make applications with any functionality beyond basic navigation. The IDE that comes with the SDK is based on Eclipse, making it easy to learn for those who have used it before. The installation and setup processes were simple, since most of the tasks were automated.

Seven releases were identified, when the same criteria for a release as with Appcelerator Titanium were used. This information was acquired from the repository, although judging from the version numbers, the generator has been in development for several more versions. The time to get the generator in working condition for Android development takes around 10 minutes. The MoSync SDK is available for OS X and Windows, while MoSync Reload is available also for Linux. MoSync SDK and MoSync Reload can be used with the GPL2 license, if the resulting applications are shared as open source. Otherwise a proprietary license is required.

6.3 PhoneGap

PhoneGap[5] is a distribution of the Apache Cordova project. The development started in 2008 and has reached version 2.9 in June 2013. This will also be the version used for the evaluation. In 2011, the PhoneGap source code was submitted to the Apache Software Foundation (ASF) for incubation, which is a way for projects from outside Apache to become full ASF projects. The developers believe that the web is the best solution for multi-platform development, and are aiming to make it a better development platform by providing support in areas where it is lacking.

A blog post from May 2012[6] explains how PhoneGap works. The user interface is created inside an instance of the device's browser view. This view does not contain any typical browser interface components, just the content area. The provided PhoneGap API allows a developer to access the native functionality of the device. The created source code is packaged as a binary application archive, which is native to the target operating system. Table 6.3 summarizes the information found on PhoneGap.

[5] http://phonegap.com/

[6] http://phonegap.com/2012/05/02/phonegap-explained-visually/

Attribute	Value	Notes
Functional coverage (Android/ BlackBerry OS)	80%/80%	NFC functionality as a third-party plugin.
Total functional coverage	-	
Supported target operating systems	Android, Bada, BlackBerry, iOS, Symbian, WebOS, Windows Phone	
Output application types	Web-based, hybrid	
Operating system specificity of assets	4	Operating systems may have their specialties, but these are documented well.
Documentation and guides	3	API is well documented and has examples, but only a small amount of guides and no videos or example applications.
Development languages	JavaScript, HTML, CSS	

Table 6.3: Evaluation of the PhoneGap tools.

Similarly to the previous mobile application generators, PhoneGap also enables reuse by creating multi-platform applications from the input source code along with the architecture. A client-server architecture is common, according to the previously mentioned blog post, when data-driven applications are made. The client-server architecture allows a developer to reuse the application logic and the data handling on the server and only the client applications potentially have to be modified for each target operating system.

Out of the same functionality inspected before, Bluetooth was not implemented in PhoneGap. All other functionality was available for both Android and BlackBerry OS. There were some third-party plugins for Bluetooth, but they had not been updated for the newer versions of the tool. NFC functionality was also lacking in the official API, but an up-to-date version was available as a third-party plugin. A table with operating system-specific compatibility helped in finding the necessary information.

The generator supports Android, Bada, BlackBerry OS, iOS, Symbian, WebOS and Windows Phone. Some functionality is missing for some of the operating systems, but in general the functionality provided by the generator can be used on

all capable target operating systems. There is some specificity in the component attributes, which may cause issues if used. These are well documented in the appropriate API entries, though. Due to the fact that the views are rendered in an instance of the device's browser view, native user interface components are not available. The output applications are not native for the same reason.

The API documentation is well-structured and there are examples to help in understanding the provided functionality. There are not that many guides, while video tutorials and example applications are not provided at all. There are guides made by users, but these guides are often old and contain deprecated information. The structured API and its examples are valued higher, making the grade three. Training is available for all subscribers, and a free of charge subscription exists. Subscribers have access to recordings of old training sessions and are able to participate in future online training sessions, which makes up for the lack of guides and videos. The development languages are JavaScript, HTML and CSS. This means that programming skills are required from the application developer. Customer support is available, with the amount of support methods and times dependent on the type of the subscription. There is a content-wise small wiki available, while a forum is available only for subscribers. The tool does not come with its own IDE implementation but uses Eclipse instead.

Information gathered from the repository suggests that there have been 20 major versions of the tool. Distribution is done under the Apache License version 2.0. While Linux, OS X and Windows are supported, some tools are not available for all operating systems. For example, BlackBerry OS development is not possible with Linux. The setup process requires some advanced system knowledge. Command line tools need to be used for the generator-specific tools, while Eclipse and Android Development Tools are downloaded separately. Unfortunately, after several tries, the tools could not be successfully installed on a PC running Windows. Although the command line tools seemed to install the generator without errors, the generator failed to respond to any usage attempts. Therefore the setup time could not be accurately measured, but it can be estimated to take around 20 minutes, when the completed tasks are compared to the tasks in the provided setup guide.

7 Conclusions

The goal of this research was to identify the problems behind application development for multiple mobile operating systems and see if mobile application generators could help with these issues. A set of metrics and criteria was created to evaluate existing tools in this regard.

The differences between mobile application types have to be weighed by application developers to find the best solution. For example, games requiring more processing power should be implemented with more efficient native technologies, while content-driven applications benefit from the easily maintained web-based methods. In order to solve this problem, reuse technologies in the form of application generators have been created. These tools allow developers to output different types of applications, and even native-like applications can be developed simultaneously for several target operating systems using the same source code as a base for each application. Application developers should identify their needs as early as possible to be able to choose the best tool to support the development process.

To evaluate these tools some metrics and criteria have to be used. However, there is not much research available on the matter, since previous studies have not focused on mobile application generator evaluation. Previous research from other areas of reuse along with characteristics identified in mobile development and application generators were used as a base for a set of metrics and criteria to evaluate existing generators. These metrics and criteria were then applied to existing generators to see how well they address the multi-platform development issues.

7.1 Findings

As previous research has already noted, the mobile application scene is splintered due to the large number of development, distribution and execution methods. While native applications are preferred by users due to their consistent look and feel, developers are looking for easier solutions to cover as many platforms as possible. Native applications are not necessarily the best solution for multi-platform development due to the large number of technologies that would have to be learned and used.

Finding information on some of the generators proved to be difficult. This was noted also by Bertoa *et al.* [2003] when examining software components. Although such information is important for a generator user, the availability of clear and up-to-date documentation about the transformation process and the exact nature of the output is limited. This information could help an application developer to select the right generator.

The differences in the functionality offered by a single generator between different target operating systems were surprising. The supported functionality should be clearly presented for each mobile operating system in order to have this information available to the developers. MoSync provides a helpful feature in their IDE that lets users to select functionality they prefer to have or even require in an application, and the tool shows which operating systems support the selected functionality. MoSync and PhoneGap also have good documentation in this regard, as they provide tables clearly showing the differences in the offered functionality between the target operating systems. MoSync goes somewhat further with their table, since it separates the classes and functions available for each target operating system.

To analyze these differences between tools, a set of metrics and criteria was suggested for evaluating functional suitability, usability, reliability and portability. When analyzing the generators with these metrics and criteria, it was found that the generators are the most suitable when applications using only a few device features are developed, as there is no guarantee that advanced functionality is supported. Even though good support for features might exist today, new features added on a device tomorrow might not be immediately available through these third-party generators. The learning curve for the evaluated generators is thought to be more moderate than learning the native development methods for several platforms. It is up to the application developer to decide whether faster development or more efficient applications with native look and feel are desired. One more benefit the generators provide is that applications can be developed on operating systems that are not usually supported. For testing and packaging, however, native tools might be required.

Because of the faster development methods, mobile application generators could be used for fast prototyping. It is said, though, that although modern tools simplify the creation of mobile applications, they are too focused on single developers getting their implementation done swiftly [Wasserman, 2010]. Heitkötter *et*

al. [2012] state that the available tools are mature enough to be used instead of native solutions. The authors continue that due to the lower development skill requirement, these solutions might be preferred even instead of native applications, although advanced functionality might still require a native application.

Mobile application generators can address some of the issues in mobile application development. In case an application does not require advanced functionality, generators can make the development process shorter. Some generators also have the capability to produce different application types, which makes it easy to use the same tool for different applications. The need for learning several development languages is also largely eliminated.

As for the suggested metrics and criteria, we feel that they are a good starting point for generator evaluation. They provide a fast way to eliminate tools that are not suitable as application generators in the light of the users requirements. However, some of the criteria are subjective, making the evaluation specific to one developer's needs. For example, the value for general usability can vary between persons doing the evaluation.

We feel that the functional suitability is a good way of evaluating whether the generator provides enough features for an application developer, as well as defining if assets using those features are portable to other target operating systems. Reusing developed assets to generate multi-platform applications lowers the total development time when compared to application development with native methods. When selecting a generator, functional suitability is an important matter because without proper functionality support an application developer cannot create applications with required features.

Usability can help in assessing how much effort will be needed to understand the usage of the generator. Application developers need to understand how the generator works, if they wish to get all benefits. For example, if it is unclear how assets should be adapted to different target operating systems, the final product might not have the expected quality. Application developers are likely to be able to adapt to new technologies, but if familiar technologies can be used, developed products are finished faster and are less likely to contain errors.

It is not enough that the input is error-free, but the generator also needs to function as expected. One of the goals of reuse is increasing quality by decreasing the number of defects in the final product. This cannot be achieved if the gener-

ator has an error that creates systematic defects in the applications it produces. Therefore, an application developer needs to know that the product is tested and fixes are applied when necessary. When selecting a generator, those that are updated regularly could be preferred, as they can be considered to contain less errors.

Portability helps selecting generators that run on devices used in development. The generator is useless if an application developer cannot refine the input into an application. Licensing factors are also considered, as application developers might want to choose the license of an application themselves, instead of using one forced by the generator providers. Usually a commercial license is required for using an open source generator to create closed source applications.

The use of the suggested metrics and criteria was relatively simple in the cases where the generator providers offered clear documentation. For example, the previously mentioned functionality tables offered by MoSync and PhoneGap made evaluating functional coverage easy. Usable development languages were also clearly stated by all the analyzed generators. On the other hand, the number of versions that have been available was more difficult to determine and required examining the repositories of the generators. The value for this metric could be impossible to determine for closed source generators, if the generator providers remove information about past versions from their web pages.

Some of the issues with reuse can be reduced with mobile application generators. Generators themselves are tools that support reuse and enable using the same assets to develop an application for multiple platforms. When compared to software product lines, application generators can decrease the needed investments by reducing the work an application development team would normally do. Mobile application generators include target operating system-specific modifications in the input, while system-specific features are interpreted and implemented by the generator. This reduces—if not eliminates—the need for application development teams. Additionally, since mobile application generators often use common programming languages or web technologies as input, and the input already contains domain-specific fragments, the generalized input can be less complex when compared to more general reusable assets or core assets used in software product lines. As Biggerstaff [1998] notes, domain specificity is also a key component in reuse success.

7.2 Future research and limitations

Some of the metrics and criteria could be altered to measure the attributes in a different way. For example, maturity could be determined as the difference in time between a generator's original release date and the date of the most recent update. The date of the most recent update could also be used as a separate metric for maturity. This would tell for how long the generator has been in development, and if the generator has been recently updated. The number of versions used in the analysis does not tell whether the tool has had any recent updates, which could indicate whether the generator can support the most recent changes in a target operating system's functionality.

More precise metrics and criteria can also be created. Development languages could be divided between user interface- and application logic development languages. This change would make it easier to understand if a development language can be used to create a complete application or not. For example, HTML 5 can be used to create web-based applications, but if a generator creates just native applications, it is more likely that HTML 5 is used to define only the user interface. General usability can also be analyzed more thoroughly with the use of existing usability heuristics. It should be noted, though, that adding more metrics and criteria while refining the existing ones to be more specific can make them more cumbersome and time-consuming to use.

Cleaveland [1988] suggested that application generators should be usable without programming knowledge. All of the evaluated generators required programming skills for the application logic. There are some generators advertised as not requiring any programming. For example Magmito[1] is one such generator. The applications are created with a simple GUI builder, but they are content-based and cannot access any device features. In general, generators that do not use programming languages as input seem to generate applications which have limited functionality but can be less prone to errors, while generators using programming languages as input provide better access to device features, but the applications can be more prone to errors due the potential mistakes present already in the user-written input.

[1] `http://www.magmito.com/`

The metrics and criteria described here were applied only to existing generators that were very similar in functionality. To get better results, evaluation should be done on more generators, especially on those with a different approach, such as Magmito or AppsGeyser[2]. They are advertised as being usable even without programming skills, making them very different from the ones evaluated here. They were not evaluated here due to not being open source and having minimal documentation. The HTML 5 standard could also have a significant impact on mobile development. Furthermore, the proposed metrics and criteria were largely based on previous research on reuse metrics. More relevant information could be acquired if the metrics were developed by someone that is experienced in mobile application development as well as in using application generators.

Generators could also be investigated based on how they actually work. As was discovered, up-to-date information about the internal functionality is difficult to find, but at least the open source solutions could be examined more thoroughly. Another way to compare the generators would be to create a similar application with different tools and compare the quality of the output. However, the output should be tested on several physical devices, since emulation might not show accurate results.

While the goal was to create a set of metrics and criteria that would be easy to use, the measurement process could also be modified to accommodate more exact measurements. For example, as Bertoa and Vallecillo [2004] say, attributes could be divided to affect more than one measurable concept. For example, community availability could also measure learnability. More objective measurements could be also used, as subjective measurement is difficult to automate [Bertoa & Vallecillo, 2004]. Automation could be difficult, though, as relevant information cannot be found in any standard places.

Bertoa *et al.* [2003] mention that component developers should provide quality information about their products, although it may be unlikely to happen. Many metrics and criteria have been suggested by different authors, but it is unlikely to get component developers to agree on a specific set, since they are likely to emphasize the ones that show their own products in a favorable way [Bertoa & Vallecillo, 2002]. Since different tools are competing for users, we feel that these statements also apply to generators.

[2] `http://www.appsgeyser.com/`

References

[Abrahamsson *et al.*, 2004] Pekka Abrahamsson, Antti Hanhineva, Hanna Hulkko, Tuomas Ihme, Juho Jäälinoja, Mikko Korkala, Juha Koskela, Pekka Kyllönen, & Outi Salo. Mobile-D: an agile approach for mobile application development. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 174–175, New York, NY, USA, 2004. ACM.

[Android report, 2013] Android report. Dashboards, May 2013. Available as *http://developer.android.com/about/dashboards/index.html*. Checked 30.5.2013.

[Apple, 2013] Apple. Apple updates iOS to 6.1, January 2013. Available as *http://www.apple.com/pr/library/2013/01/28Apple-Updates-iOS-to-6-1.html*. Checked 16.7.2013.

[Batory, 2004] Don Batory. The road to utopia: A future for generative programming. In Christian Lengauer, Don Batory, Charles Consel, & Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2004.

[Bertoa & Vallecillo, 2002] Manuel F. Bertoa & Antonio Vallecillo. Quality attributes for COTS components. *I+D Computación*, 1(2):128–144, November 2002.

[Bertoa & Vallecillo, 2004] Manuel F. Bertoa & Antonio Vallecillo. Usability metrics for software components. *Proceedings of the 8th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'04)*, 2004.

[Bertoa *et al.*, 2003] Manuel F. Bertoa, José M. Troya, & Antonio Vallecillo. A survey on the quality information provided by software component vendors. In *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2003)*, pages 25–30, July 2003.

[Biddle & Tempero, 1998] Robert L. Biddle & Ewan D. Tempero. Towards tool support for reuse. In *Proceedings of the 1998 International Conference on Software Engineering: Education & Practice*, pages 126–133, January 1998.

[Biggerstaff, 1998] Ted J. Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5:169–226, 1998.

[Caldiera & Basili, 1991] Gianluigi Caldiera & Victor R. Basili. Identifying and qualifying reusable software components. *Computer*, 24(2):61–70, February 1991.

[Charland & LeRoux, 2011] Andre Charland & Brian LeRoux. Mobile application development: Web vs. native. *Queue*, 9(4):20–28, April 2011.

[Cleaveland, 1988] J. Craig Cleaveland. Building application generators. *Software, IEEE*, 5(4):25–33, July 1988.

[Czarnecki & Eisenecker, 1999] Krzysztof Czarnecki & Ulrich W. Eisenecker. Components and generative programming. In Oscar Nierstrasz & Michel Lemoine, editors, *Software Engineering – ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 2–19. Springer Berlin Heidelberg, 1999.

[de Sá *et al.*, 2008] Marco de Sá, Luís Carriço, Luís Duarte, & Tiago Reis. A framework for mobile evaluation. In *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '08, pages 2673–2678, New York, NY, USA, 2008. ACM.

[Frakes & Kang, 2005] William B. Frakes & Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, July 2005.

[Frakes & Terry, 1996] William Frakes & Carol Terry. Software reuse: metrics and models. *ACM Comput. Surv.*, 28(2):415–435, June 1996.

[Gartner report, 2013] Gartner report. Gartner says Asia/Pacific led worldwide mobile phone sales to growth in first quarter of 2013, May 2013. Available as *http://www.gartner.com/newsroom/id/2482816*. Checked 1.6.2013.

[Gavalas & Economou, 2011] Damianos Gavalas & Daphne Economou. Development platforms for mobile applications: Status and trends. *Software, IEEE*, 28(1):77–86, January–February 2011.

[Hammershøj *et al.*, 2010] Allan Hammershøj, Antonio Sapuppo, & Reza Tadayoni. Challenges for mobile application development. In *2010 14th International Conference on Intelligence in Next Generation Networks (ICIN)*, pages 1–8, October 2010.

[Heitkötter *et al.*, 2012] Henning Heitkötter, Sebastian Hanschke, & Tim A. Majchrzak. Comparing cross-platform development approaches for mobile applications. In *8th International Conference on Web Information Systems and Technologies*, pages 299–311, 2012. Unpublished.

[Her *et al.*, 2007] Jin Sun Her, Ji Hyeok Kim, Sang Hun Oh, Sung Yul Rhew, & Soo Dong Kim. A framework for evaluating reusability of core asset in product line engineering. *Information and Software Technology*, 49(7):740–760, 2007.

[Holzer & Ondrus, 2011] Adrian Holzer & Jan Ondrus. Mobile application market: A developer's perspective. *Telematics and Informatics*, 28(1):22–31, 2011.

[Hsieh & Tempero, 2006] Min-Sheng Hsieh & Ewan Tempero. Supporting software reuse by the individual programmer. In *Proceedings of the 29th Australasian Computer Science Conference*, volume 48 of *ACSC '06*, pages 25–33, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.

[Huy & vanThanh, 2012] Ngu Phuc Huy & Do vanThanh. Evaluation of mobile app paradigms. In *Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia*, MoMM '12, pages 25–30, New York, NY, USA, 2012. ACM.

[ISO, 1991] ISO. *ISO/IEC 9126:1991 Software enginnering - Product quality*, December 1991.

[ISO, 2011] ISO. *ISO/IEC 25010:2011 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*, March 2011.

[Java documentation, 2013] Java documentation. About Java ME, 2013. Available as *http://www.oracle.com/technetwork/java/javame/about-java-me-395899.html*. Checked 31.5.2013.

[Jha & O'Brien, 2009] Meena Jha & Liam O'Brien. Identifying issues and concerns in software reuse in software product lines. In Stephen H. Edwards & Gregory Kulczycki, editors, *Formal Foundations of Reuse and Domain Engineering*, volume 5791 of *Lecture Notes in Computer Science*, pages 181–190. Springer Berlin Heidelberg, 2009.

[Kitchenham *et al.*, 1995] Barbara Kitchenham, Shari Lawrence Pfleeger, & Norman Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995.

[König-Ries, 2009] Birgitta König-Ries. Challenges in mobile application development. *Information Technology*, 51(2):69–71, February 2009.

[Mikkonen & Taivalsaari, 2011] Tommi Mikkonen & Antero Taivalsaari. Apps vs. open web: the battle of the decade. In *Proceedings of the 2nd Workshop on Software Engineering for Mobile Application Development*, 2011.

[Mili *et al.*, 1995] Hafedh Mili, Fatma Mili, & Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[Qt documentation, 2013] Qt documentation. Supported platforms, 2013. Available as *https://qt-project.org/doc/qt-5.0/qtdoc/platform-details.html*. Checked 31.5.2013.

[SailfishOS wiki, 2013] SailfishOS wiki. Wiki - QA, 2013. Available as *https://sailfishos.org/wiki/QA*. Checked 30.5.2013.

[Smaragdakis & Batory, 2000] Yannis Smaragdakis & Don Batory. Application generators. *Encyclopedia of Electrical and Electronics Engineering*, 2000. J.G. Webster (ed.), John Wiley and Sons.

[Sztipanovits & Karsai, 2002] Janos Sztipanovits & Gabor Karsai. Generative programming for embedded systems. In Don Batory, Charles Consel, & Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 32–49. Springer Berlin Heidelberg, 2002.

[W3C documentation, 2013] W3C documentation. Device APIs working group, 2013. Available as *http://www.w3.org/2009/dap/*. Checked 3.6.2013.

[Washizaki *et al.*, 2003] Hironori Washizaki, Hirokazu Yamamoto, & Yoshiaki Fukazawa. A metrics suite for measuring reusability of software components. In *Proceedings of the Ninth International Software Metrics Symposium, 2003*, pages 211–223, 2003.

[Wasserman, 2010] Anthony I. Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP Workshop on Future*

of Software Engineering Research, FoSER '10, pages 397–400, New York, NY, USA, 2010. ACM.