The Journal of Systems and Software 123 (2017) 1–32

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A product-line model-driven engineering approach for generating feature-based mobile applications

Muhammad Usman*, Muhammad Zohaib Iqbal, Muhammad Uzair Khan

Software Quality Engineering and Testing (QUEST) Laboratory, National University of Computer and Emerging Sciences, Islamabad, Pakistan

ARTICLE INFO

Article history:
Received 30 June 2016
Revised 31 August 2016
Accepted 28 September 2016
Available online 29 September 2016

Keywords:
Mobile applications
Software product-line engineering
Feature model

ABSTRACT

A significant challenge faced by the mobile application industry is developing and maintaining multiple native variants of mobile applications to support different mobile operating systems, devices and varying application functional requirements. The current industrial practice is to develop and maintain these variants separately. Any potential change has to be applied across variants manually, which is neither efficient nor scalable. We consider the problem of supporting multiple platforms as a 'software product-line engineering' problem. The paper proposes a novel application of product-line model-driven engineering to mobile application development and addresses the key challenges of feature-based native mobile application variants for multiple platforms. Specifically, we deal with three types of variations in mobile applications: variation due to operation systems and their versions, software and hardware capabilities of mobile devices, and functionalities offered by the mobile application. We develop a tool MOPPET that automates the proposed approach. Finally, the results of applying the approach on two industrial case studies show that the proposed approach is applicable to industrial mobile applications and have potential to significantly reduce the development effort and time.

1. Introduction

Mobile application development has recently emerged as one of the most focused areas in the software industry (Dehlinger and Dixon, October, 2011). Exponential growth of mobile users, extensive use of mobile devices, and the variety of mobile platforms has resulted in significant increase of mobile application development industry. According to recent statistics, there are around 4 million registered mobile applications available for various platforms (Statista 2015) with approximately 40 thousand mobile applications being added per month (Android Apps Submitted Per Month 2015) making the mobile application industry a multi-billion dollar industry (Mobile Application Revenue Generation 2013).

With the increase in the variety of mobile operating systems and device features, a typical challenge faced by mobile application industry is the requirement to support different mobile platforms. The term mobile platform refers to both the software platform (i.e., operating system and software features such as contact and message) and the hardware platform (i.e., the mobile device hardware such as bluetooth and camera). As an example, consider an application being developed may need to support various Android operating systems' versions (Google 2015) (such as, *Ice-cream Sandwich, Kitkat, Lollipop, Marshmallow*), their forks (Android fork such as *Cyanogen*), and various iOS versions (Apple 2013) (such as, *iOS 7, iOS 8, iOS X*). The same application may also need to support the mobile device hardware variations, i.e., various mobile devices support specific hardware features that are not available in other devices. Some mobile devices provide support for LTE/4 G Networks, Bluetooth, GPS, External storage, or Accelerometer and other devices do not support these.

Another requirement for mobile applications is to support multiple functional requirement variations, i.e., different functionality for different clients. For example in a banking application, some clients may require support of scheduled wireless backups, while other clients only require manually triggered wired backup support. Similarly, some clients are interested in payment via traditional credit card option only whereas others may want support for more novel options, such as, bitcoins or Near Field Communications payment systems, such as Apply pay (Apply Pay 2016) and Android pay (Android Pay 2016) in an e-store application.

A common mobile development industry practice to address the above mentioned problems is to develop separate native variants of the same application for each mobile platform (Dehlinger and Dixon, October, 2011; Joorabchi et al., October 2013; Top 100 Apps Availability for iOS 2016). This is a very resource heavy task and the complexity of maintaining multiple variants increases exponentially as a large number of application variants need to

* Corresponding author:.
E-mail addresses: m.usman@questlab.pk (M. Usman), zohaib.iqbal@nu.edu.pk (M.Z. Iqbal), uzair.khan@nu.edu.pk (M.U. Khan).

http://dx.doi.org/10.1016/j.jss.2016.09.049
0164-1212/

be developed and maintained over time (Joorabchi et al., October 2013). Any potential change in the application requirements has to be applied to all the different variants of the mobile application manually. Similarly, any new functionality needs to be added in all the variants separately. The graphical user interface (GUI) for these applications is developed specifically for particular mobile platforms. The GUI is typically developed using drag-and-drop tools available for various operating systems and is tailored specifically for optimal performance on each set of devices (Google 2016; Microsoft, 2013; Apple 2016) that are then used to generate code corresponding to the GUI. Similarly, user-interface modeling languages, such as, IFML (OMG 2016), UML*i* (Da Silva and Paton, 2000), or model-based user-interface modeling techniques (Cimino and Marcelloni, 2012; Botturi et al., 2013; Sabraoui et al., 2012) can be used for developing GUI. The development of business logic of native variants requires redundant effort and the approach of developing these separately is neither scalable nor feasible. The problem of maintaining multiple variants has also been highlighted by a number of software engineers as one of the key challenges in mobile application development (Dehlinger and Dixon, October, 2011; Joorabchi et al., October 2013; Wasserman, 2010).

An alternative to developing multiple native variants of a mobile application is to either develop a web application or a cross-platform hybrid application by using web-scripting languages (Raj and Tolety, 2012). Web applications execute in a web browser, whereas hybrid applications execute inside native containers of mobile devices (Raj and Tolety, 2012). Web and hybrid mobile applications are generally considered low in performance and cannot access mobile device hardware directly (Charland and Leroux, 2011). Web applications also require internet connectivity that cannot be ensured at all times. Native applications, on the other hand, are considered to be more stable and secure, better in terms of performance, allow direct access to device hardware and also have better look and feel. For example, Facebook application was first launched as a hybrid application but due to the performance issues and hardware non-accessibility, later on it was developed as a native application for multiple platforms (Facebook Hybrid App to Native App 2015). Some cross-platforms tools exist for development of web and hybrid mobile application (Ohrt and Turau, 2012). The mobile applications developed using native application development tools (Google, 2016; Microsoft, 2013, Apple 2016) have superior look and feel as compared to the applications developed from the cross-platform development tools. Similarly, the debugging support offered by the cross-platform tools is inferior to the support provided by the native application development tools (Ohrt and Turau, 2012; Dalmasso et al., 2013; Heitkötter et al., 2012). Therefore, developing native applications is often the preferred choice if not the only choice.

Software product-line engineering (SPLE) is a well-accepted approach of developing a set of products that share a common set of features (Pohl et al., 2005). The concept of SPLE has been adopted from the broader product-line engineering that has been adopted and applied in different domains to handle large number of product variations, for example, in the automobile industry (Thiel and Hein, 2002) and embedded systems (Polzer et al., 2009). SPLE uses feature models that consist of a set of features and their variations that are required by the family of products being developed (Lee et al., 2002). The various products in the family (also referred to as product variants) in a product-line are derived using the feature model (Webber and Gomaa, 2004). The problems of supporting mobile application variants that are highlighted above can be positioned as an SPLE problem and the existing SPLE concepts can be applied in this domain.

A widely accepted methodology for developing software systems that has successfully been applied in other domains is Model-Driven Software Engineering (MDSE) (Gomaa, 2008; Larman, 2004; Iqbal et al., 2015). In MDSE, models are considered as the key software development artifact (Brambilla et al., 2012) and Unified Modeling Language (UML) (OMG 2013a) is commonly used for developing software systems. MDSE has a high potential for use in developing mobile applications because it allows platform independent modeling, which can later on be transformed to multiple mobile platforms.

In our work, we address the challenges faced by the mobile application industry (highlighted earlier) by proposing a 'product-line model-driven engineering approach' to support automated generation of mobile application variants of multiple platforms. We refer to these variants as feature-based variants and our approach supports three variations required by mobile applications industry: (i) variations in application due to mobile operating systems and their versions, (ii) variations due to software and hardware capabilities of the mobile devices, and (iii) variations based on the functionalities offered by the mobile application.

The problem of supporting multiple mobile application variants (hardware, software, and functional) is well-acknowledged in literature and is a recurring problem in industry. This is also true for our industrial partner, Invotyx (Invotyx 2014), which is developing native mobile applications for various mobile platforms with different features and the developers are currently maintaining a number of variants for each application, as per the prevalent industrial practice. Invotyx is interested in an efficient approach for developing and maintaining the various variants of the applications being developed at the company. Our proposed approach provides an automated solution and is applied on two case studies provided by our industrial partner, *Scramble* and *Instalapse*.

As part of the solution, for SPLE we provide a generic mobile application product-line feature model (FM_G) that captures the mobile domain specific concepts (for example, Android v5.1, iOS X, bluetooth, WIFI, contact, and message). The FM_G can be used by the application designer to develop an application specific feature model (FM_A). The FM_A contains the operating system-related features, software-related features, hardware-related features, and application-specific functional requirement-related features and combine these as a feature model specific for the application product-line under development. The application specific feature model (FM_A) can then be used to generate a mobile application product-line modeling profile specific for the application product-line. The modeling profile allows the application designer to model mobile domain specific concepts during mobile application modeling. Considering mobile applications typically have short time to market and require quick delivery and deployment (Joorabchi et al., October 2013), we select a minimal but sufficient subset of UML that includes UML use-case diagram for requirements gathering, class diagram for structural modeling, and state machine diagram for behavioral modeling of the mobile application. To support the proposed approach, we develop an application generation tool MOPPET to automate our product-line model-driven engineering approach for mobile applications. The proposed approach is applied on two industrial case studies developed by our industrial partner.

The rest of the paper is organized as follows: Section 2 highlights the industrial context and describes the case studies. Section 3 overviews the related work. Section 4 presents the details of the proposed approach briefly. Section 5 provides details about the proposed product-line engineering approach while Section 6 discusses the proposed model-driven engineering approach. Section 7 describes the feature-based mobile application variants generation while Section 8 presents the developed MOPPET tool to implements the proposed approach. Section 9 validates the proposed approach through industrial case studies and also highlights the limitations, threats to validity, and open questions of the proposed approach. Finally, Section 10 concludes the paper.

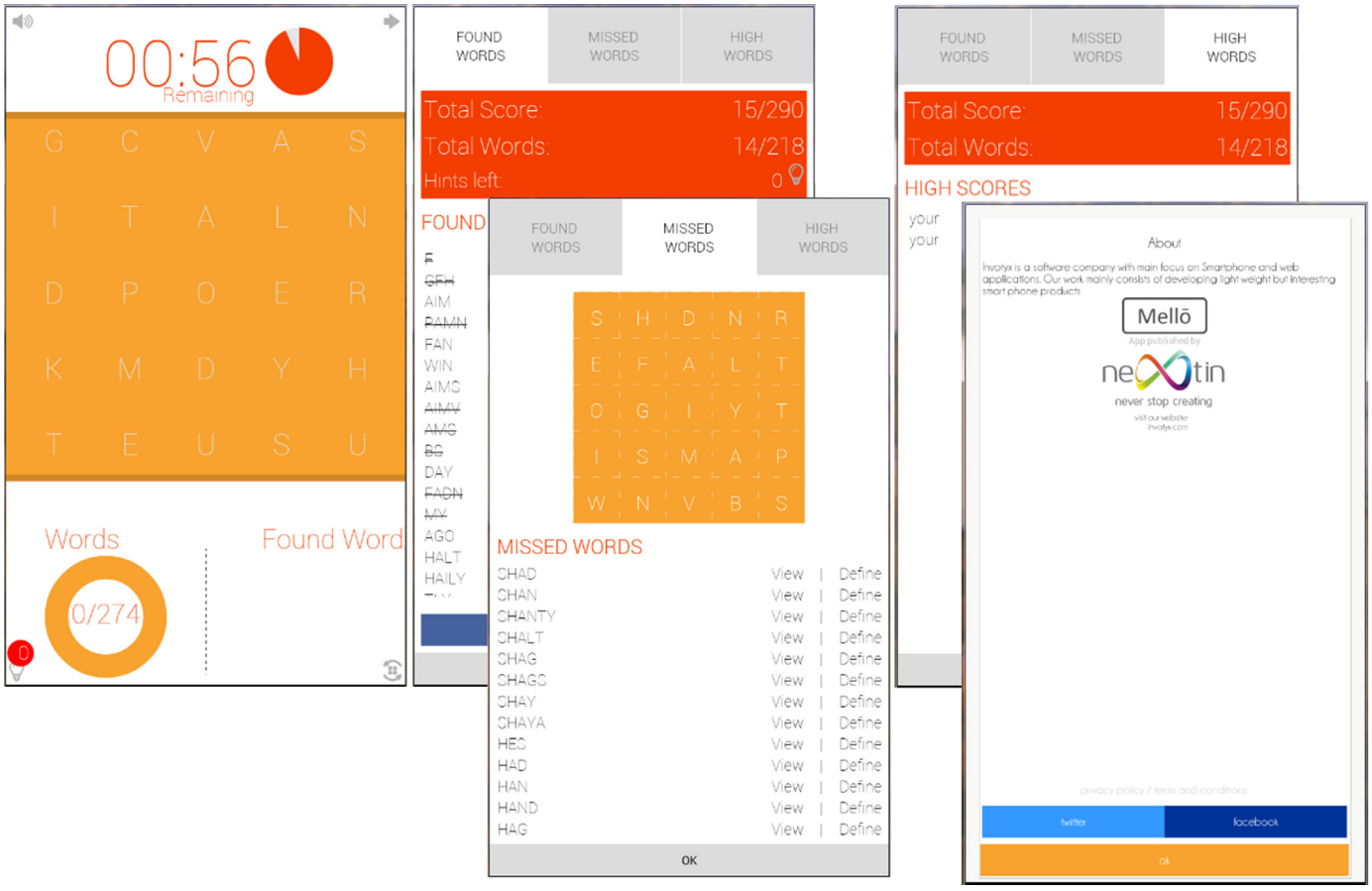


Fig. 1. Scramble case study application screens.

2. Industrial context and description of case studies

This section describes the practical motivation of the proposed approach by introducing the industrial partner and the problems the company is facing due to the requirement of supporting multiple mobile platforms and their versions as well as functionality-driven variants. We also introduce the 'Scramble' (Scramble 2014) and 'Instalapse' (Instalapse 2014) case studies by Invotyx. The Scramble case study is being used as a running example throughout the paper and 'Instalapse' case study is discussed later in the paper to evaluate the proposed approach.

Our industrial partner, Invotyx is a mobile application development organization, which is developing a number of native mobile applications ranging from games to general-purpose utility applications for multiple platforms. Invotyx developers are currently maintaining a number of native versions for each application. Some of their well-known native applications are *scramble, flow dots, slideagram, instalapse*, and *true or false*.

Scramble is a board-based English words' learning mobile application. It is currently available for *Android* platform. The board consists of a 5×5 squares containing a set of 25 characters (one at each position). The words are scrambled, so a particular word can be found in an irregular pattern. The player can create words by moving in 8 different directions, i.e., up, down, left, right, up-right, up-left, down-right, and down-left. Fig. 1 shows various application screens of the Scramble case study. In Scramble, the score is calculated by number of correct words created by the player. The application also shows a list of all possible words in the given board. It also has the facility to share results with the friends on *Twitter* or *Facebook*.

Instalapse is another mobile application by Invotyx, currently available for *Android* platform. It is a hyper-lapse application that helps in making time-lapse videos. The time-lapse videos are created either by using mobile camera or by selecting already existing videos. It also helps in applying the desired time-lapse speed and audio to the video. Moreover, Instalapse application facilitates to share the created videos with the friends on *Twitter, Facebook*, and *Instagram*.

Invotyx intends to develop the mobile application for other mobile platforms including *Windows Phone* platform. The company currently builds separate variants of the same application to support the different required platforms. For example, Scramble and Instalapse are developed for *Android* platform. The company now want to support the application for other platforms, starting with *Windows Phone* platform. Apart from the need to support multiple platforms, the company also requires functionality-oriented variants of the mobile applications. For example, Fig. 1 shows the Scramble application screen with social media sharing support, i.e., application offers the facility of sharing results with friends on social media, such as, Facebook or Twitter. Invotyx plans to launch different variants of Scramble application each containing one or more of the new features: support of persistence with database or file, lightweight phone storage feature or SD card storage, 4×4 or 5×5 playing board size application feature, and social media sharing functionality through Facebook or Twitter. Maintaining multiple native variants of the same application independently is a tiresome task and requires redundant effort by the development and maintenance teams. The work presented in this paper is conducted in collaboration with Invotyx. We had multiple sessions with the developers to understand the needs of the development team and

formed a solution that is applicable to mobile application development in general. The solution was then provided to Invotyx, which then applied MOPPET to generate the feature-based application variants.

The identified problem of supporting multiple native variants of an application is not specific to our industrial partner, rather it is a widely acknowledged problem (Joorabchi et al., October 2013). This problem is also not specific to any particular operating system platform or hardware device, but in general, a native application written for one platform cannot execute on other platforms. Also, whenever there is a change in mobile application, all the developed variants of the mobile application for different platforms have to be updated. In summary, a well-known mobile application development organization, Invotyx wants to develop multiple variants of both *Scramble* and *Instalapse* applications that should support different mobile operating systems, hardware platforms as well as functional variants through efficient developing strategies to reduce the development and maintenance efforts. The work presented in this paper provides a product-line model-driven engineering approach for generating feature-based variants of mobile applications to solve the real challenges faced by mobile application industry.

3. Related work

Mobile Application Product-line Engineering is an active research area with a number of approaches discussing the application of well-defined product-line engineering practices on mobile applications. Similarly, active research is going on in model-driven engineering for mobile applications. In our previous work (Usman et al., 2014), we had developed a model-based mobile application generation approach for multiple operating systems. This paper builds upon the previous work and handles functionality-driven variations, hardware and software variations along with operating system variations.

Following, we discuss the related work on product-line engineering and model-driven engineering of mobile applications.

3.1. Mobile application product-line approaches

Myllymäki et al. (2002) provides a four-layers architecture for creating application product-line for Symbian platform. The focus is to apply SPL at the architecture level, to highlight the variation points from very basic level of the application. The approach uses layer architecture style and introduces four different layers (i.e., resource platform, architecture platform, product platform, and product) to develop a mobile application. The paper defines variation points at the architecture level. The approach is specific to Symbian platform and only supports variants corresponding to the application functional requirements.

Zhang et al. (2003) use product-line engineering to reengineer existing Java desktop application for Java ME mobile devices. The authors use the existing city guide application to build a product-line and further include the missing features for the city guide application. The variation points are defined to generate city guide applications for PC, Pocket PC, and smart device. As the existing application is developed for Java, so Java supported devices are targeted in the SPL generated applications only. The approach is specific to Java ME and handles application functional variations only. Nascimento et al. (2008a, 2008b) apply product-line engineering to design mobile game for Java ME. The paper distributes the SPL process for mobile game intro three main parts as component modeling, component implementation, and component testing. The paper used three mobile games (i.e., Monga, American Dad – Roger's Scape, and Zaak), analyze the games, extract their features, define the feature variations, and then configure the variation points to regenerate the games using SPL. The generated mobile games are then compared to the existing mobile games. The results conclude that more or less in all the cases SPL prove better than the traditional mobile games development process. The presented approach handles the functional variation and is specific to the generation of mobile game variants for Java ME only. The approaches (Zhang et al., 2003; Nascimento et al., 2008a, 2008b) supports a single mobile platform (i.e., Java ME), uses reengineering for extracting application features that lacks the inclusion of software and hardware features, and incorporates application functional features variation only.

Albassam and Gomaa (2013) presents an application of software product-line in video game domain. Feature dependency model is presented to define video games features and their dependencies. Mobile applications are generated for only Windows desktop and Windows Phone platforms. The approach is specific to generating hardware specific variants of video games for Windows platforms.

Muthig et al. (2004) applies different product-line engineering approaches for the development Java ME platform mobile devices for Go Phone corporation. The company uses product-line engineering to generate nine products with variations of software features. The paper (Muthig et al., 2004) does not propose a product-line engineering approach for mobile application and is more of a Java ME platform generator for mobile devices offered by Go Phone corporation requirements. It focuses on a single platform and lacks the hardware and functional features variability.

Vaupel et al. (2014) presents a model-driven approach to generate mobile application variants. The paper uses app models (i.e., data, UI, and process) for application modeling and provider models (i.e., process instance models) for app variations. The approach is designed specifically for data-intensive application and generates privilege-based variants. Though the approach might be useful for supporting role-based variants of data-centric mobile application but it only supports variants for functional requirements. Quinton et al. (2011) presents an approach to generate mobile application variants from multiple feature models. The approach uses two feature models, one is for application product-line and the other is for mobile device. The tool AppliDE is developed to automate the application generation process. The approach focuses on the generation of user interface for application variants and also does not generate behavioral application code. The approaches (Vaupel et al., 2014; Quinton et al., 2011) does not generate native application for multiple platforms (e.g. Android v4.4, Android v5, Windows Phone v8, iOS X, etc), lacks the modeling of platform specific features (e.g. bluetooth, wifi, camera, gps, etc), and only incorporates functional feature variations for variants generation.

Dagef et al. (2016) extends MD^2 DSL (Heitkötter et al., 2013) for product-line integration to generate Android and iOS mobile applications. The processes in a business scenario are presented as features of product-line. These processes are modeled as workflow elements of MD^2 DSL and are combined together to form an application product-line. The MD^2 DSL generator is used to generate apps for multiple mobile platforms. The work presented in the paper is specific for the data-driven business apps and only supports the functional variations and does not handle variations for hardware and software features.

3.2. Mobile application model-driven approaches

Kraemer propose an approach to develop Android application using UML activity diagram and state machine (Kraemer, 2011). The approach models Android specific concepts in UML diagrams. They use an Arctis SDK for creating and validating UML diagrams, later on state machine are transformed to Android application. The approach is specific to Android platform and generates a single application for functional requirements only. UML class and sequence

diagrams are used by Parada and Brisolara to develop Android application (Parada and d. Brisolara, 2012). Android specific concepts are modeled as UML classes. The authors use their developed GenCode tool to generate the Android application. The approach generates mobile application for Android platform only. The approaches (Kraemer, 2011; Parada and d. Brisolara, 2012) focus on the modeling and generation of mobile application specific to Android platform and also lacks the feature-based variability.

Ko et al. propose an approach to develop Android applications using UML metamodel (Ko et al., 2012). UML profile is defined for Android specific concepts. Model-view-controller pattern is used to implement the communication between interface and hardware. UML models are used to generate Android application. The approach supports Android platform and only generates code structure for a mobile application. Benouda et al. propose an MDA based approach by using UML class diagram to generate an Android application (Benouda et al., 2016). QVT is used transforming UML class diagram (PIM) to Android model (PSM) at metamodel level and then Acceleo is used for the code generation from Android model (PSM) through an MVC pattern implementation. The approach supports Android platform only and also generate structural code for a mobile application. Min et al. present an approach to develop Windows Phone 7 applications using UML metamodel (Min et al., 2011). UML profile is used to model Windows Phone 7 platform specific concepts. MVC pattern is used for user interface and hardware communication. UML models are transformed to Windows Phone 7 platform specific application. The approach does not include behavioral details and only generates code structure for a Windows Phone mobile application. The approaches (Ko et al., 2012; Benouda et al., 2016; Min et al., 2011) supports a single mobile platform, provides the application modeling specific to the underlying platform, lacks the behavioral details in the generated application code, and also unable to incorporate feature-based variability.

Son et al. propose code generation rules to generate Java code for Android platform from UML sequence diagram (Son et al., 2013). The approach focuses on six different kinds of interactions among object in sequence diagram. The MOF is used to present the metamodel for sequence diagram. The transformation rules are defined using Acceleo that transforms the sequence diagram modeling constructs to Java. The approach provides code generation rules for Android platform only and does not even generate an application.

Cimino and Marcelloni define an approach to design user interface for mobile applications (Cimino and Marcelloni, 2012). The user interface is modeled using UML as a platform independent model. This model is transformed to mobile platform specific models. The approach uses UML class and sequence diagrams for modeling. The approach focuses on the generation of user interface for the mobile application. Botturi et al. demonstrate an approach to develop user interface for mobile applications (Botturi et al., 2013). Finite-state machines are used to model user interface. FSMs are presented as XML document which are rendered on the actual screen of the mobile device. The approach only generates user interface for the mobile application. Sabraoui et al. generate user interface for Android platform using UML object diagram (Sabraoui et al., 2012). Object diagram is transformed to XMI that is rendered on Android device. The technique does not define events behavior for user interface. The approach is specific to Android platform and generates user interface only. The approaches (Cimino and Marcelloni, 2012; Botturi et al., 2013; Sabraoui et al., 2012) focus on the modeling and generation of user interface (that is mostly specific to the mobile device and is developed usually using drag and drop tool for specific mobile platform) while the presented work in this paper models and generates the business logic which is the core part of the application.

Heitkötter et al. (2013) propose MD^2 (a model-driven domain specific language) to generate native mobile applications for Android and iOS platforms specific to data-driven business apps. The MD^2 DSL is a textual language that is based on MVC pattern; where *model* represents the business entities, *view* defines the user interface while *controller* links the model and view of the application. The focus of the paper is on the native look and feel of the business app. The native code generation for Android and iOS platforms is discussed in detail. The work is extended by Heitkötter et al. (2015) to include algorithmic details such as *conditional actions, boolean expressions, relational expressions, arithmetic expressions*, and *recursive action calls*. The MVC pattern is also upgraded to incorporate the support for the device-dependent layout for better native look and feel of the business app. Another extension of MD^2 DSL is proposed by Majchrzak et al. (2015) that focuses on the inclusion of iterators. The paper also discusses the quality attributes like scalability, performance, long-term support, user-friendliness, user feedback, extension, and competition in the perspective of the MD^2 DSL. Ernsting and Majchrzak (2016) presents a reference architecture for MD^2 DSL. The paper discusses the reference architecture the MD^2 DSL using MVC pattern (i.e., model, view and controller). The MD^2 DSL and its extensions (Heitkötter et al., 2013; Heitkötter et al., 2015; Majchrzak et al., 2015) are limited to data-oriented mobile applications only. Although the native applications are generated for Android and iOS platforms but the integration of the mobile platform's native libraries in the generated application is not specified.

There are numerous tools and frameworks available to develop mobile applications and mobile games for multiple platforms (Ohrt and Turau, 2012), such as, Sencha (Sencha 2014), PhoneGap (Phone Gap 2015), Appcelerator Titanium (Appcelerator 2013), Cocos2d (Cocos2d 2016), Unity3D (Unity3D 2016), Corona (Corona 2016), Qt (Qt 2016), and Xamarin (Xamarin 2016). The applications generated using these tools and frameworks are less efficient due to restricted access to hardware resources (Oehlman and Blanc, 2011) and overhead of running inside a container (Charland and Leroux, 2011; Ohrt and Turau, 2012). These tools do not support the requirements of native application generation – a common industry requirement. More importantly, these tools do not support the feature-based application variants generation.

3.3. Analysis

To summarize, the product-line and model-driven engineering of mobile applications is an active research area. However, none of the existing works support the three kinds of variations, i.e., (i) variations in application due to mobile operating systems and their versions, (ii) variations to software and hardware capabilities of the mobile devices, and (iii) variations based on the functionalities offered by the mobile application. The existing approaches are either limited in their scope, e.g., they only target a particular kind of variability (Nascimento et al., 2008a, 2008b; Vaupel et al., 2014; Dagef et al., 2016) or they are specific to a certain platform (Ko et al., 2012; Benouda et al., 2016; Min et al., 2011). Some of the works, such as, (Dagef et al., 2016; Heitkötter et al., 2013; Heitkötter et al., 2015; Majchrzak et al., 2015) provide support for modeling mobile applications and generating native variants for multiple platforms however, they do not support modeling and variants generation for mobile domain specific hardware and software features.

Our approach is independent of mobile platforms unlike the works presented in (Myllymäki et al., 2002; Zhang et al., 2003; Albassam and Gomaa, 2013) and combines software product-line engineering with model-driven engineering to gain the benefits of both that is addressed by only one approach in the existing literature (Dagef et al., 2016). Software product-line engineer-

ing allows us to capture application information in feature models while model-driven engineering allows us to model the application product-line independently of the underlying platform, i.e., the operating system and hardware device. We use automated application generation to derive native variants for multiple platforms. Our approach supports not only device and operating system driven variants but also the feature-based variants due to differing requirements. Based on our interaction with industry practitioners, there is a real need to automated strategy to support feature-based variant generation for mobile applications. In short, the proposed product-line model-driven engineering approach generates business logic in native application variants that support all three types of variations required by the mobile application industry.

4. Proposed approach

This section presents a brief overview of the proposed 'Product-line Model-driven Engineering approach' to generate feature-based native mobile application variants for multiple platforms. Fig. 2 highlights the proposed approach. There are two roles of the participants in the proposed approach, i.e., (i) *Framework Provider* and (ii) *Application Designer* as shown in Fig. 2. The *Framework Provider* role is defined for the developers of the proposed approach while the *Application Designer* role is assigned to the appliers of the proposed approach for the development of the mobile application. *Framework Provider* provide the guidelines to the *Application Designer* to model a product-line for the under development mobile application using the proposed approach. In the presented approach, the application designer starts with the feature modeling of the mobile application and then moves towards the business logic modeling of the application product-line as shown in Fig. 2. The feature modeling is achieved through the software product-line engineering and the model-driven engineering is used for the business logic modeling.

From Fig. 2, the proposed product-line engineering approach for mobile applications consists of the generic feature model, the application specific feature model, and the configuration of the application feature model. The generic feature model (FM_G) is provided by the framework provider that contains the mobile domain specific concepts (such as, Android v5.1, contacts, message, bluetooth, and camera). The application designer uses the FM_G to develop application specific feature model (FM_A) based on the desired features required by the specific application. Based on the requirement of the application variant(s) to be generated, the designer configures the FM_A by selecting the features during application variants generation.

As shown in Fig. 2, the proposed model-driven engineering approach for the modeling of mobile application product-line consists of the UML modeling profile and the UML models. The modeling profile is generated by the framework provider using the developed FM_A and used for the modeling of application product-line concepts in the UML models. From the UML models, the use-case, class, and state machine are selected for mobile application modeling as shown in Fig. 2. To capture the mobile application requirements, the designer models the UML use-case diagram in collaboration with the mobile application screens. The structural details of the mobile application are modeled through UML class diagram. The framework provider generates the UML class diagram for those features in FM_A that are derived from the FM_G. The designer uses the generated class diagram for further modeling of mobile application and also applies the application product-line modeling profile. The UML state machines are modeled for the behavioral details of the mobile application under development.

The application specific feature model and its configuration, and the developed UML models are used to generate application variants. The Mobile Application Product-line Generator (MOPPET) tool automates the application variants generation, shown in Fig. 2. The tool supports generating feature-based variants for multiple mobile platforms. The proposed product-line model-driven engineering approach is applied on two industrial case studies from our industrial partner. The MOPPET tool is used to generate application variants for both the case studies. Results show that the proposed approach is successful in generating multiple variants according to the desired features for *Android* and *Windows Phone* platforms. Development of these variants manually would have required a significant redundant effort. Although a considerable initial learning curve is involved in developing the feature model and UML diagrams, the approach is beneficial in reducing the time and effort required for developing and maintaining subsequent application variants.

5. Software product-line engineering approach for mobile applications

As the framework provider (Fig. 2), a Generic Feature Model (FM_G) is provided that contains a set of typical mobile domain features, that includes the mobile operating system (e.g., Android v4.4, Windows Phone v7.8), software features (e.g., Notes, Contacts) and hardware features (e.g., Bluetooth, GPS). The FM_G is used by application designer to develop an Application specific product-line Feature Model (FM_A). The FM_A contains the application specific hardware, software, operating systems, and functional requirement-related features. For application variant(s) generation, the FM_A is configured (i.e., the selection of features) as per requirements of the application user. In the following sections, the details about FM_G and FM_A and how these feature models can be used for product-line engineering of mobile applications are presented.

5.1. Generic feature model (FM_G)

The generic feature model (FM_G) presents the feature and their variations for the mobile application domain specific concepts. The FM_G is developed based on our analysis of the mobile applications domain. For this purpose, a large number of mobile applications are analyzed belonging to different application stores (Google Play 2014; Microsoft 2015; Apple App Store 2015). The various hardware and software related features are identified. Note that these features do not include functional requirement-related features, but are restricted to the various hardware and software components of mobile platform that are used in the mobile applications development industry. For the identified features, the various variations, relationships, and constraints are defined. To develop FM_G, the feature modeling guidelines by Lee et al. (2002) are used. The excerpt of FM_G developed after domain analysis is shown in Fig. 3. For the visual representation of feature model, there are different tools available, such as, FeatureIDE (Thüm et al., 2014), pure:: variants (pure.systems 2015), and SPLOT (SPLOT - Software Product Line Online Tools 2015). The FeatureIDE tool (an Eclipse plug-in) is used for the visual representation of the developed FM_A (Fig. 3). The FeatureIDE tool is an academic tool for feature modeling that facilitates the definition of features, their variations, relationships and constraints.

Following, the features, relationships, and constraints in the FM_G are described. The guidelines on extending the FM_G are also presented, in cases where a particular feature is missing and needs to be included at FM_G level.

5.1.1. Features in FM_G

A feature in a feature model represents a unique characteristic of a system (Kang et al., 1990). The FM_G contains mobile domain operating system, software and hardware features. The *root*

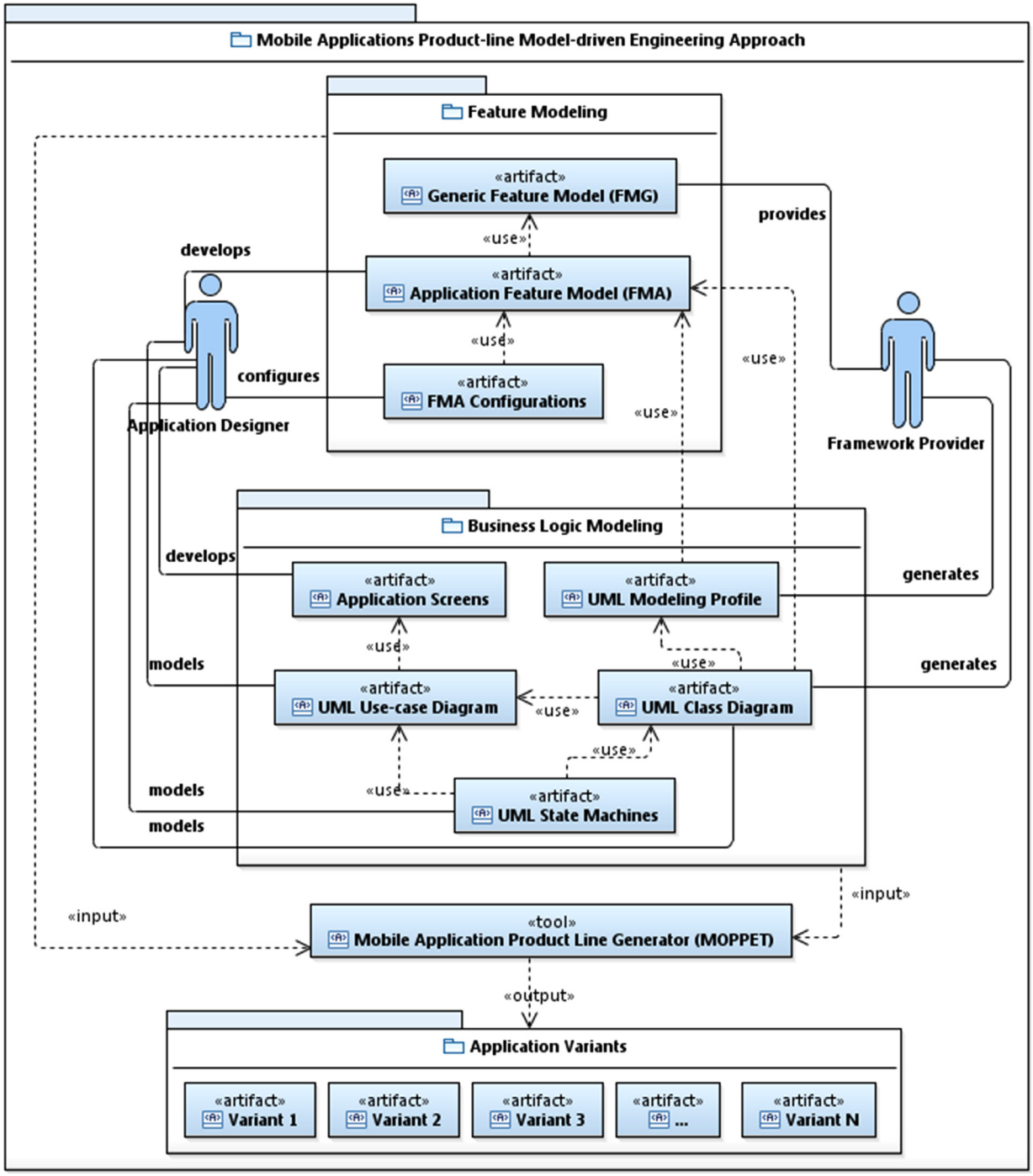


Fig. 2. An overview of product-line model-driven engineering approach.

element of the FM_G represents the mobile application for which the application specific feature model is being developed (see Fig. 3). The *operating system* element represents the existing mobile operating systems and their versions (e.g., *Android v5.1, Windows Phone v8.1, IOS v9*). The software features are defined under the *software* element (e.g., *Contacts, Messages, Notes*) while hardware features are listed under *hardware* element and deal with mobile device specific features (e.g., *Bluetooth, Wifi, GPS*). Based on the nature of software features, the *Software* features are categorized into three categories, i.e., *Core Utilities, Media*, and *Application Utilities* as shown in Fig. 3. The *Core Utilities* consists of features consist of the basic utilities that are required by a large number of mobile applications, for example *ApplicationActivity* and *NotificationHandler*. The *Media* category comprises of features that represent media related concepts, for example, *picture, audio*, and *video*. The third category is *Application Utilities* that contain features that are provided by most of the operating systems and are widely used in the mobile applications, for example, *Timer* and *Calendar*. The hardware features in the FM_G refers to the hardware that may vary in mobile devices (such as, all mobile devices do not have Bluetooth). The *Hardware* features in the mobile devices are *Storage, Bluetooth, GPS*, and *Camera*.

Variation points are identified to model variability in FM_G. A variation point defines the possible variations of the features in a

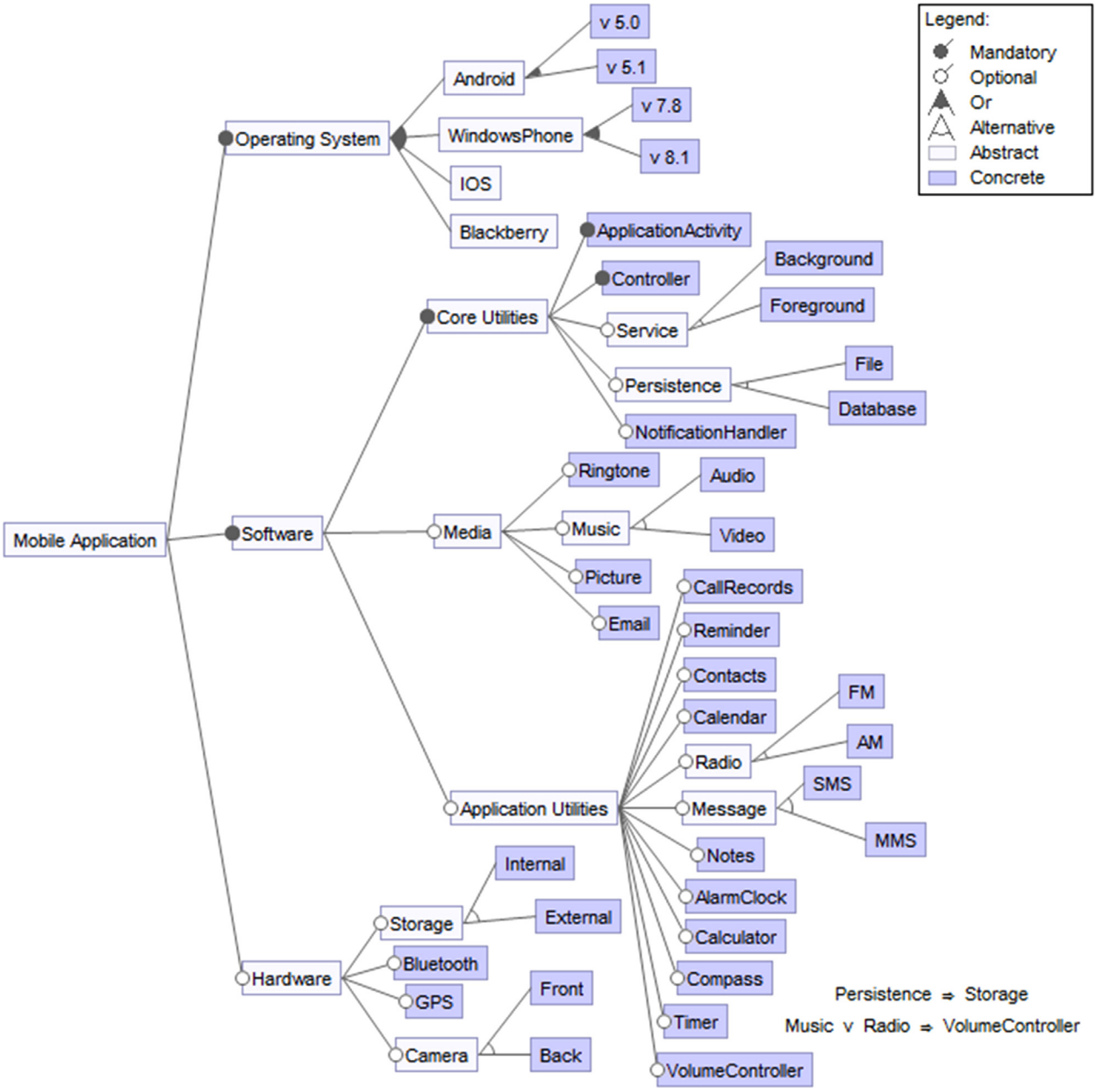


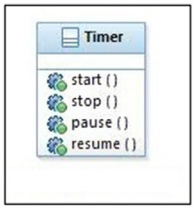
Fig. 3. Extract of generic mobile application product-line feature model (FM_G).

feature model. Based on the domain analysis, the potential variations of the software and hardware features of FM_G are identified. For example, in FM_G the *Persistence* feature is a variation point that has two variation features, i.e., *File* and *Database*. The FM_G serves as a basis for development of application specific feature model (FM_A).

In the proposed product-line model-driven application development approach, the FM_G forms the basis of mobile application development. The various features modeled in FM_G are later converted to design interfaces and then to code as required by the application under development. The framework provider has provided implementation of the features in various languages compatible to the various mobile operating systems available. For example, for the software feature *Timer*, the implementation APIs for *Android* and *Windows Phone* platforms are provided. The design interface corresponding to the *Timer* feature is shown in Fig. 4. The design interfaces of various features and their corresponding implementation APIs are maintained through an XML document. Listing 1 shows an XML extract for *Timer* feature that contains the design interface (*Operations* in Listing 1) and its mapping to the corresponding implementation APIs for various supported platforms (*Platforms* in Listing 1). For example, *TimerImpl.java* class provides the implementation of *Timer* feature for Android platform. The same feature is implemented using *TimerImpl.cs* for Windows Phone platform.

5.1.2. Relationships in FM_G

As discussed earlier, FM_G was constructed after detailed domain analysis of mobile applications. For the features that are included in FM_G, the relationships between them are defined as part of the feature model. In feature model, there are typically three types of parent-child relationships: *AND, OR*, and *ALT* (Lee et al., 2002). For FM_G, all these three relationships are applied. The *AND* relationship facilitates to mark child features as *Mandatory* or *Optional*. The




```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Features>
    <Feature name="Timer">
        <Operations>
            <Operation name="start" returnType="void">
                <Parameters />
            </Operation>
            <Operation name="stop" returnType="void">
                <Parameters />
            </Operation>
            <Operation name="pause" returnType="void">
                <Parameters />
            </Operation>
            <Operation name="resume" returnType="void">
                <Parameters />
            </Operation>
        </Operations>
        <Platforms>
            <OperatingSystem name="Android" version="v 5.0">
                <SourceClass name="TimerImpl" path="D:\TimerImpl.java" />
            </OperatingSystem>
            <OperatingSystem name="Android" version="v 5.1">
            <OperatingSystem name="WindowsPhone" version="v 7.8">
                <SourceClass name="TimerImpl" path="D:\TimerImpl.cs" />
            </OperatingSystem>
            <OperatingSystem name="WindowsPhone" version="v 8.1">
        </Platforms>
    </Feature>
</Features>
```

Fig. 4. *Timer* feature design interface. Listing 1. Timer XML mappings.

mandatory features are compulsory and must be present while optional features are not compulsory in all the configurations of the feature model. The *OR* relationship ensures that one or more child features should be selected during every configuration of the feature model. The *ALT* relationship restricts the selection to only one child feature in each configuration of the feature model.

In FM_G (Fig. 3), there is an *AND* parent-child relationship between the *root* feature and its children *Operating System, Software*, and *Hardware*. The *Software* and *Operating System* features are marked as mandatory because these are required by all mobile applications. The *Hardware* feature is considered as optional because some applications may only be using the software utilities and not the hardware-related features. In FM_G, an *OR* parent-child relationship is created between the *Operating System* feature and its children (i.e., *Android, Windows Phone, IOS,* and *Blackberry*) since a mobile application can be developed for one or more of the operating systems and their versions. In FM_G, there is an *ALT* parent-child relationship between six features and their children (i.e., *Service, Persistence, Music, Message, Storage*, and *Camera*). From the children of these features the application designer may select only one feature. For example, for a mobile application can either be *foreground* or *background* but can never be both simultaneously which is represented by the relationship between *Service* feature and its children. Similarly, *Persistence* can either be *file-based* or *database-based*.

During the modeling of application feature model (FM_A), the application designer cannot change the relationships of *Operating System, Software, ApplicationActivity,* and *Controller* features. However, the designer has the leverage to change the relationship of other features based on the requirements of the mobile application product-line under development.

5.1.3. Constraints in FM_G

The feature model also allows applying restrictions on the features that do not have direct relationship with each other. These restrictions are applied through constraints. The constraints highlight the important dependencies between various features of the FM_G. During FM_G configuration, the constraints enforce to include or exclude a feature(s) based on the selection of other feature(s).

There are five different operators (i.e., OR, AND, NOT, IMPLIES, IFF) to apply constraints in feature model (Lee et al., 2002). The *IMPLIES* constraint allows to include a feature on the basis of another feature. For example, 'featureA *implies* featureB' that means *featureB* is required by *featureA* and selected automatically on the selection of *featureA* during feature model configuration. The *AND* constraint allows to include more than one features on the basis of another feature. For example, 'featureA *implies* featureB *and* feature' that enforces *featureB* as well as *featureC* to be included automatically on the selection of *featureA*. The *OR* operator allows to include both or either of the two features on the basis of another feature. For example, 'featureA *implies* featureB *or* featureC' that facilitates both (i.e., *featureB* and *featureC*) or either *featureB* or *featureC* is included on the selection of *featureA*. The decision to include *featureB* or *featureC* is taken by the application designer. The *IFF* operator allows including feature based on a condition, if the condition is true then feature is included automatically. The operators (i.e., OR, AND, NOT, IMPLIES) are used to specify the condition. For example, 'featureA *iff* featureB *and* (featureC *or* featureD)'that means if *featureB* and either of *featureC* or *featureD* or both are selected then *featureA* is included. The NOT operator allows to exclude a feature on the basis of another features. For example, 'featureA *implies not* featureB' that refers *featureB* cannot be selected if *featureA* is selected.

In FM_G (Fig. 3), the *IMPLIES* constraint is, 'Persistence *implies* Storage' that refers to the inclusion of *Storage* feature on the selection of *Persistence* feature. The *OR* constraint in FM_G is, 'Music *or* Radio *implies* VolumeController' that enforces the inclusion of *VolumeController* feature on the selection of either *Music* feature or *Ringtone* feature.

5.1.4. Extending the FM_G

The FM_G contains the typical features that are required by mobile applications. However, there is a possibility that with the passage of time new features need to be added in the FM_G. Since FM_G forms the basis of the proposed product-line model-driven application development approach, a mechanism is provided for extending the FM_G. As discussed in Section 5.1.1, the software and hardware features of FM_G have a corresponding design interface and implementation APIs for the various platforms provided by the framework provider. In case where new software or hardware features are added in FM_G, the application designer also provides corresponding platform specific implementation APIs and a design in-

Table 1
Scramble product-line FM_A features description and variations.

Feature	CT/AT	MD/OP	OR/ALT	Variation possible configurations	Description
Application activity	CT	MD	–	–	It defines an active object in the mobile application.
Controller	CT	MD	–	–	It handles the communication between user interface and business logic.
Persistence	AT	MD	ALT	File, Database	It defines the data storage mechanism for the mobile application.
File	CT	OP	–	–	It stores data in the file.
Database	CT	OP	–	–	It stores data in the database.
Timer	CT	MD	–	–	It allows including timer in the mobile application.
Sound	CT	MD	–	–	It allows including sound in the mobile application.
Storage	AT	MD	ALT	Internal, External	It defines the storage mechanism in the mobile application.
Internal	CT	OP	–	–	It provides support for internal phone memory storage in the mobile application.
External	CT	OP	–	–	It provides support for external phone memory storage in the mobile application.
Board size	AT	MD	ALT	FourByFour, FiveByFive	It defines the board size for the scramble board.
FourByFour	CT	OP	–	–	It presents the 4×4 squares board size.
FiveByFive	CT	OP	–	–	It presents the 5×5 squares board size.
History	CT	MD	–	–	It defines the player history.
Color theme	CT	MD	–	–	It defines the color theme for the board.
Hint	CT	MD	–	–	It presents the hints to the player.
Score	CT	MD	–	–	It represents the player points.
Player	CT	MD	–	–	It represents the playing person.
Game	CT	MD	–	–	It represents the playing game.
Feedback	CT	OP	–	–	It defines the feedback for the application developers.
Social media sharing	AT	OP	OR	Facebook, Twitter	It defines the social media sharing of the scramble application.
Facebook	CT	OP	–	–	It allows sharing the data on the facebook.
Twitter	CT	OP	–	–	It allows sharing the data on the twitter.

*CT: Concrete, AT: Abstract, MD: Mandatory, OP: Optional, ALT: Alternative

terface specifying the various functions of the APIs (as shown in Fig. 4 for *Timer* feature example). The design interface is used during the mobile application modeling. The designer also adds variations, relationships, and any dependency constraints due to the modifications in FM_G.

The application designer may also add operating system features corresponding to any new release of mobile operating system versions or their forks. For each new operating system feature, the application designer should provide the implementation APIs for the software and hardware features. Note that these APIs are not specific to any application therefore the effort is a one-time overhead.

The developed MOPPET tool (shown in Fig. 2) implements a component for providing design interface and implementation APIs for the added feature. The mappings of the design interface and implementation APIs are stored in an XML document, as shown in Listing 1.

5.2. Developing application product line feature model (FM_A)

The application designer uses the FM_G as a basis to develop the application specific product-line feature model (FM_A). An FM_A is created for each mobile application product-line under development. The application designer simply selects a configuration from FM_G. As discussed earlier (in Section 5.1), FM_G provides a set of features that are generic to mobile application domain. The purpose of FM_A is to model the features that are specific to the mobile application product-line under development. Selecting a configuration of FM_G to obtain an FM_A allows the application designer to restrict (from the generic feature set) the subset of features available for the application product-line under development. The designer may also restrict the potential operating systems, hardware and software features as per the requirements of the application product-line. The designer adds the functional requirement-related features in the FM_A, which is then used to generate application product-line specific UML modeling profile. Later on, the application FM_A is configured to generate the desired application variants for multiple platforms. The developed FM_A for the *Scramble* case study is shown in Fig. 5. Table 1 provides the detailed description of all the features, their variation points and variations for Scramble product-line FM_A.

5.2.1. Selecting features from FM_G

A configuration of FM_G is selected to develop an FM_A for the product-line being created. The FM_G contains *Operating System* features, *Software* features, and *Hardware* features. For the selection of features from FM_G, the application designer reviews the requirements of the mobile application product-line under development focusing on the mobile operating systems, their versions and the required software and hardware features. If a required operating system, software, or hardware feature is not present in the FM_G, the designer follows the guidelines presented in Section 5.1 to add a new feature in FM_G.

For the Scramble case study, the developed FM_A is shown in Fig. 5. From the operating system features, the designer selected *Android* and *Windows Phone* with their variations *v5.1* and *v8.1* respectively from FM_G. Among the software features, the designer selected *Persistence*, and *Timer* features with their variations from FM_G while *ApplicationAcitivty* and *Controller* features are selected automatically to be included in FM_A as these are mandatory in FM_G. A new feature *Sound* feature is added as *Software* feature in FM_G (through the guidelines presented in Section 5.1) and then selected to be included in FM_A. Among the hardware features, the *Storage* feature is selected. As the application required variants with both internal and external storage, the designer included *Storage* along with its variations.

5.2.2. Adding functional features in FM_A

After the selection of required features from FM_G, the basic structure of FM_A is obtained that contains the required operating system, software, and hardware features. The next step is adding the functional requirement-related features in the FM_A. To add these features, the designer reviews the functional requirements of the application product-line being developed and follows the typical product-line modeling strategies (Pohl et al., 2005; Lee et al., 2002) to identify the functional requirement-related features and add them in the FM_A for the application product-line under development. These requirement-related features are important and

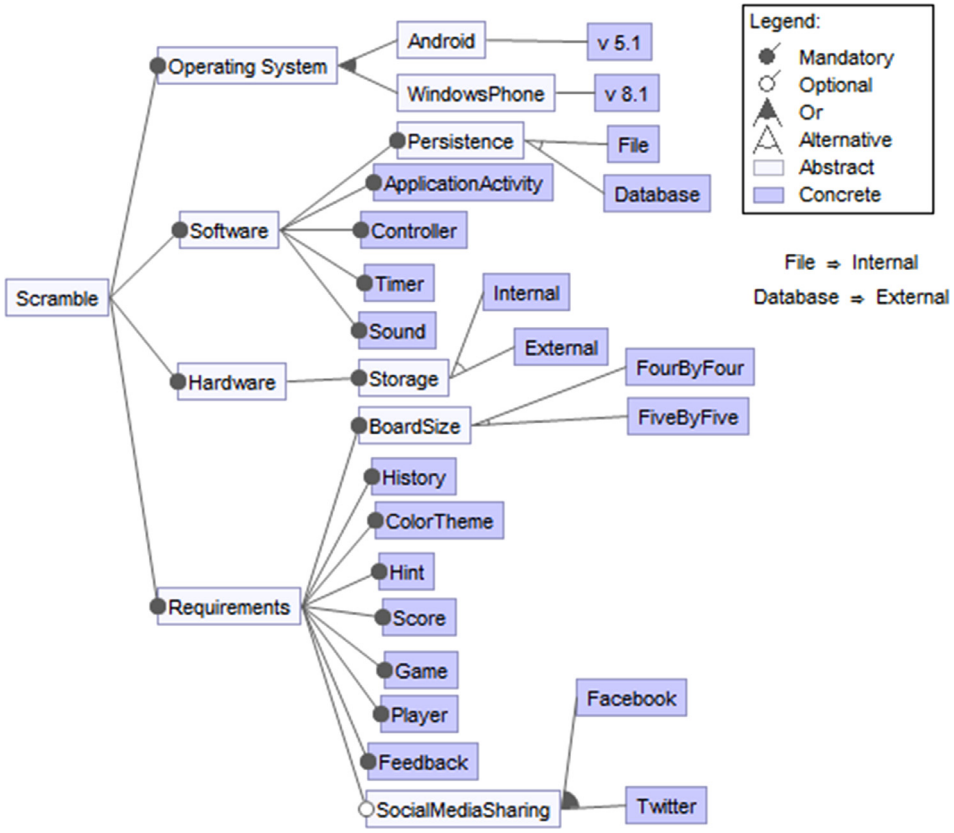


Fig. 5. Scramble mobile application product-line feature model (FM_A).

form the basis of the mobile application being developed. Note that the features in FM_A correspond to the entire product-line of the under development mobile application and not just one product.

For the Scramble case study, the application designer included the features that are specific to the application business logic. For example, there is a functional requirement that the scramble game board should be of 4×4 size or 5×5 size. From this functional requirement, the designer identifies *BoardSize* as a functional characteristic. The designer also identifies the possible variations for the identified functional characteristics. For example, for the *BoardSize*, the designer identifies two variations, i.e., *FourByFour* and *FiveByFive*. In this way, the designer identifies all the functional characteristics of the under development mobile application product-line and adds the identified functional characteristics with their variations under the *Requirements* element in the FM_A. For the Scramble product-line FM_A (Fig. 5), the other requirement features include *History, ColorTheme, Hint, Score, Game, Player, Feedback*, and *SocialMediaSharing*. The *Facebook* and *Twitter* features are the variations for *SocialMediaSharing* feature.

5.2.3. Marking features as optional or mandatory in FM_A

For the application product-line FM_A, the designer marks the software, hardware, and requirement-related features as mandatory or optional. Marking the feature as mandatory or optional is based on the requirements of the application product-line under development as it varies from application to application.

The software and hardware features that are obtained from FM_G may be mandatory or optional. If a feature is mandatory in FM_G it implies that it must be included in all applications (e.g., *ApplicationActivity*). Such features cannot be made optional while developing FM_A. The features that were optional in FM_G imply that they may or may not be present in a mobile application (e.g., *Timer*). The application designer while developing FM_A can modify these features to be mandatory if the application product-line being developed requires these features to be present in all the variants of the mobile application product-line.

For the Scramble case study, there are a number of features that are marked mandatory. For example the *Persistence* feature is optional in FM_G but marked as mandatory in FM_A because of the requirements of the Scramble application, the persistence of player's score is mandatory. In the Scramble product-line FM_A (Fig. 5), the mandatory features are *ApplicationActivity, Controller, Persistence, Timer, Sound, Storage, BoardSize, History, ColorTheme, Hint, Score, Player*, and *Game* while the optional features are *Feedback* and *SocialMediaSharing*.

5.2.4. Defining feature relationships in FM_A

Another important concept to model in a feature model is relationship between parent features and their child features. There are typically three types of relationships between features: OR, AND, and ALT. The relationship within the operating system, software and hardware features is already defined in the FM_G by the framework provider. For FM_A, the designer provides the relationships within the requirement-related features. For the Scramble product-line FM_A (Fig. 5), there is an *AND* parent-child relationship between the *Requirements* and its child features (i.e., *BoardSize, History, ColorTheme, Hint, Score, Player, Game, Feedback*, and *SocialMediaSharing*). For the *BoardSize* and its child features (i.e., *FourByFour* and *FiveByFive*), an *ALT* parent-child relationship is defined by the designer. An *OR* parent-child relationship exists between the *SocialMediaSharing* and its child features (i.e., *Facebook* and *Twitter*).

5.2.5. Defining feature constraints in FM_A

In the FM_A, the application designer defines the constraints to apply restrictions on the features. The constraints depend on the

requirements of the application product-line under development. Some of the constraints on the software and hardware features that are applicable to all mobile applications in general are already provided in the FM_G by the framework provider. The designer should not weaken the constraints, but can modify the existing constraints to further strengthen these. The application designer includes the constraints applicable on the requirement-related features in the application product-line FM_A. These feature constraints are transformed into Object Constraint Language (OCL) (O.M. G. (OMG) 2016) constraints during the generation of UML modeling profile from the FM_A. The OCL is a standard language for specifying constraints on the UML. For the mobile application product-line modeling, all the OCL constraints must be satisfied. The designer uses the operators (i.e., OR, AND, NOT, IMPLIES, IFF) for constraints modeling (Batory, 2005) in application product-line FM_A. These operators allow the designer to apply constraints on the features in FM_A. The constraints play a vital role during the FM_A configuration for application variants generation.

For the Scramble case study, there is a requirement to store database on mobile external storage. In FM_G, there is an *IMPLIES* constraint, i.e., '*Persistence implies Storage*' . During FM_A development, the designer has further strengthened the constraint by defining a constraint, i.e., '*Database implies External*' as shown in Fig. 5. During FM_A configuration, whenever a *Database* feature is selected, the *External* feature will be selected automatically.

The constraints in a feature model are defined to either include or exclude the feature(s) on the basis of another feature(s). The mandatory features are compulsory in all configurations of the feature model, so the constraints that include the mandatory features only are not required in feature model. The designer should provide those constraints that apply to both mandatory and optional features or constraints that apply to only optional features in feature model. For the Scramble case study, the requirement is to store the score. In Scramble product-line FM_A (Fig. 5), the constraint '*Score implies Persistence*' should be included but as both the features are mandatory, so the constraint is not required.

6. Modeling the mobile application product-line using model-driven engineering

Once the mobile application product-line feature model (FM_A) is developed, the next step for the application designer is to model the business logic of the mobile application using UML modeling profile and notations. The FM_A is used to generate a UML modeling profile corresponding to the application product-line. This profile contains the concepts of the domain and product-line that the application designer can use during modeling. For the modeling notations, a subset of UML is identified that meets the needs of mobile application development. The subset includes UML use-case diagram, UML class diagram and UML state machines. During the application variants generation, the UML models are used to generate code for the selected features in FM_A configuration. Section 6.1 presents the details of generating application product-line modeling profile. Section 6.2 presents the details for mobile application product-line modeling.

6.1. Mobile application product-line modeling profile

The FM_A consists of mobile operating system, software, hardware, and requirement-related features for the mobile application under development The FM_A is used to generate a corresponding UML modeling profile for the product-line. Once a profile is generated, it allows the application designer to model mobile applications using the mobile domain specific concepts and application product-line concepts. For the generation of UML profile from the feature model, the various features of the FM_A are converted to stereotypes of the profile. The stereotypes are generated for both mandatory and optional features. Since each of the features may have its structural and behavioral properties, we extend the stereotype classes from the UML meta-class *Class*. For the features that have a parent-child relationship, there can be three possible types, AND, OR, and ALT. For all these three types, while generating the profile, a stereotype is only generated for the parent feature. The child features are handled through enumeration with each of the child feature as a separate enumeration literal. The generated enumeration is then associated with the generated stereotype, such that during application of the profile, the exact child features can be selected as a stereotype attribute.

Fig. 6a shows a feature model for AND, OR, and ALT parent-child relationships. The *ANDParent* feature and its child features represents an AND parent-child relationship and vice versa for the OR and ALT parent-child relationships. Fig. 6b presents the generated UML profile for the feature model (Fig. 6a). The UML profile is generated using the proposed profile generation strategy. In Fig. 6b, the *ANDParent* stereotype is generated for the *ANDParent* feature. For the child features of *ANDParent*, the *ANDParentType* enumeration is generated. The *andparenttype* attribute is associated with the *ANDParent* stereotype. Fig. 6c shows the application of the generated UML profile (Fig. 6b) on UML classes. The *ANDClass* UML class is stereotyped with the *ANDParent* stereotype. The values *FeatureB* and *FeatureA* are selected from generated *ANDParentType* enumeration.

Note that the difference of the relationship types (OR, ALT, AND) is handled by assigning different Multiplicity values to the relationship between the generated stereotype (corresponding to the parent feature) and the generated enumeration (corresponding to the child features). For the OR relationship, the value of Multiplicity element is: *1..**, to ensure the modeling of at least one child feature and at most all the child features. The value of the Multiplicity element for the ALT relationship is: *1*, to ensure that only one child feature is selected. For the AND relationship, the value of *Multiplicity* element is: *, to facilitate the modeling of at most all the child features. The AND relationship allows to mark child features as mandatory or optional. To ensure that mandatory child features are selected during modeling, additional OCL constraints are generated for each of the mandatory features. No such constraints are required for features corresponding to optional AND parent-child relationship. The template for the generation of the OCL constraint for the mandatory feature is shown in Listing 2.

For both the OR and ALT parent-child relationships, the lower bound of the *Multiplicity* element are 1. This restricts to specify the *Default Value* for the associated attributes with OR and ALT parent-child relationships. In the proposed profile generation strategy, the *Default Value* is generated for all the relationship types (OR, ALT, AND). In Fig. 6b, the value of the Multiplicity element is different for the AND, OR, and ALT parent-child relationships. For example, for *orparenttype* attribute, the value of the Multiplicity element is: *1..**. The default value for the *orparenttype* attribute is *FeatureH* enumeration literal that is selected from *ORParentType* enumeration.

There are a number of possible constraints in FM_A that an application designer may model. There are five types of constraint operators in FM_A (Lee et al., 2002; Batory, 2005): AND, OR, IMPLIES, NOT, and IFF (as discussed in Section 5.2) that are used to model possible constraints. These constraints are also translated to OCL and are applied on the generated profile. The templates to translate the feature model constraints to OCL are shown in Table 2.

The operating system feature of application product-line FM_A is not included in the proposed profile generation strategy as the application product-line modeling (using the generated UML profile) is not specific to any mobile operating system. The operating sys-

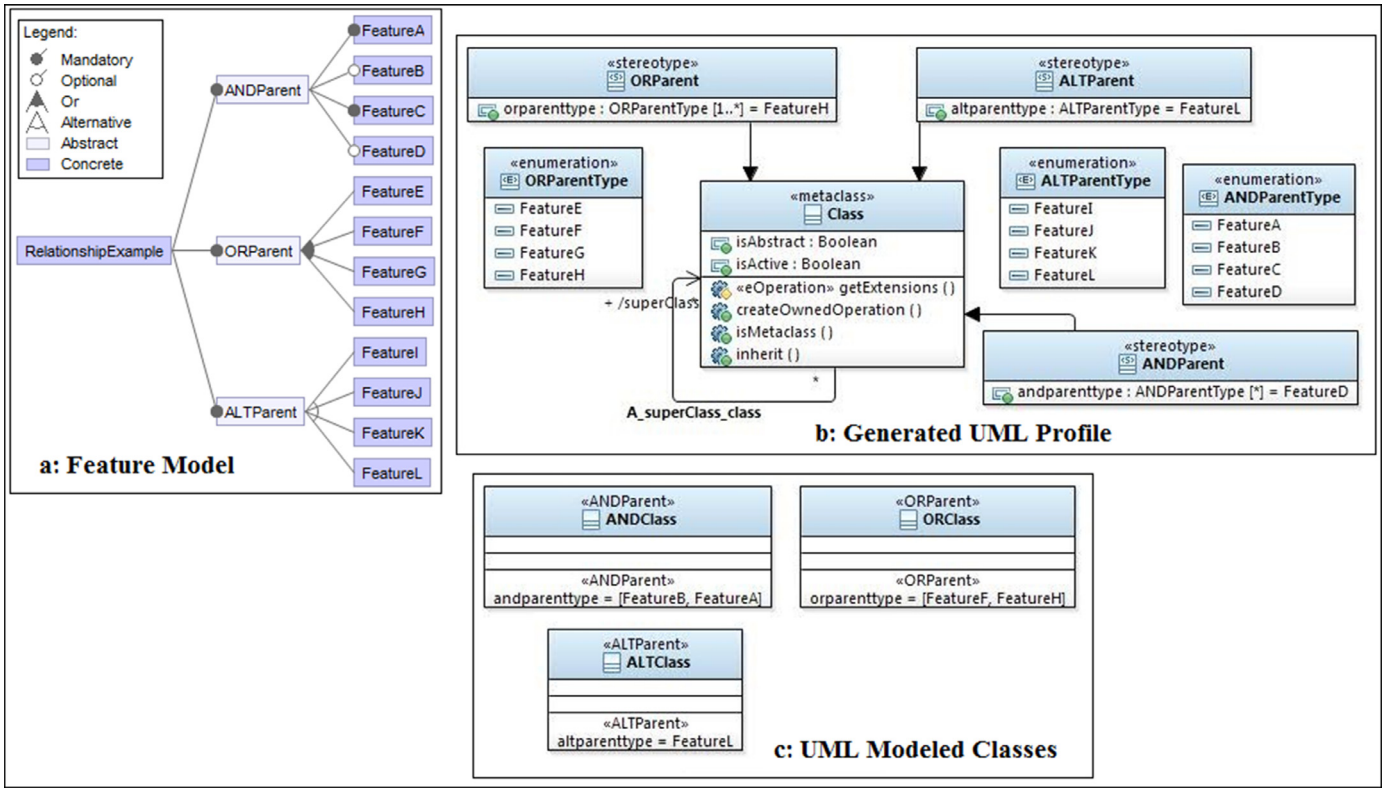


Fig. 6. Feature model relationships example (from feature modeling to modeled classes).

```
1  self.base_Class.owner.allOwnedElements()
2      -> exists(  el:Element | el.oclIsKindOf(uml::Class)
3                  and
4                      not(el.getAppliedStereotype('mandatoryfeature').oclIsUndefined())
5              )
```

Listing 2. OCL constraint for mandatory feature.

tem feature is used during the configuration of FM_A to specify the mobile platforms for the generation of application variants.

For the Scramble case study, the generated profile is shown in Fig. 7 that corresponds to the Scramble product-line FM_A (Fig. 5). From Fig. 7, the stereotypes are generated for software features (e.g., *Persistence, ApplicationActivty,* and *Controller),* hardware feature (e.g., *Storage),* and requirement-related features (e.g., *BoardSize, History,* and *Feedback).* In Fig. 7, three features (i.e., *Persistence, Storage*, and *BoardSize*) contain an ALT parent-child relationship with their child features while *SocialMediaSharing* feature holds an OR parent-child relationship with its child features. The *Persistence* feature has two child features, i.e., *File* and *Database*. The stereotype is generated for the *Persistence* feature only as shown in Fig. 7. For the child features, an enumeration of *PersistenceType* is generated. A *persistencetype* attribute is included in the *Persistence* stereotype. For the ALT parent-child relationship, the value of *Multiplicity* element for *persistencetype* attribute is: 1. From *PersistenceType*, the *File* enumeration literal is selected as a *Default Value* for *persistencetype* attribute. During the application product-line modeling, the designer can modify the value of the *persistencetype* attribute as required. The OCL constraints are generated for all the mandatory features and the constraints in FM_A. Fig. 7 presents an example of the generated OCL constraint for the mandatory *BoardSize* feature and *'File implies Internal'* constraint in FM_A. Table 3 presents all the generated OCL constraints from the Scramble product-line FM_A (Fig. 5).

6.2. Mobile application product-line modeling

UML is the most widely used modeling language and UML and its profiles have been successfully applied for modeling of applications in various domains (Gomaa, 2008; Larman, 2004). Because of high market competition between various vendors, mobile applications are typically developed in short cycles to achieve quick delivery (Joorabchi et al., October 2013). For the proposed mobile applications modeling, a subset of UML is selected that meets the industry needs of application modeling. The selected subset consists of three UML models: (i) use-case diagram, (ii) class diagram, and (iii) state machine. Following, the modeling of mobile applications with these UML models and generated UML profile is discussed.

6.2.1. Use-case modeling of mobile application product-line

The application designer starts with the modeling of use-cases that are widely accepted as a useful way of gathering and recording requirements (Bittner, 2002; Kulak and Guiney, 2012). The typical use-case modeling (Bittner, 2002) (i.e., UML use-case diagram and use-case description) with concrete writing style is used for mobile application requirements modeling. The concrete writing style includes the details of user interfaces during use-case description (Bittner, 2002; Kulak and Guiney, 2012). For this reason, the designer is restricted to develop application screens (i.e., user interfaces) before modeling use-cases. Note that the mobile appli-

Table 2
OCL Constraint template for feature model constraint operators.

Operator	Feature Model Constraint	Template for OCL
AND	featureA *implies* featureB *and* featureC …	**self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureA').oclIsUndefined())) **implies** **self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureB').oclIsUndefined())) **and** **self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureC').oclIsUndefined())) ...
OR	featureA *implies* featureB *and* featureC …	**self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureA').oclIsUndefined())) **implies** **self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureB').oclIsUndefined())) **or** **self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureC').oclIsUndefined())) ...
IMPLIES	featureA *implies* featureB	**self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureA').oclIsUndefined())) **implies** **self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureB').oclIsUndefined()))
IFF	featureA *iff* featureB *and* (featureC *or* featureD)	**self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureA').oclIsUndefined())) **implies** **self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureB').oclIsUndefined())) **and** (**self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureC').oclIsUndefined())) **or** **self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureC').oclIsUndefined())))
Not	featureA *implies* *not* featureB	**self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureA').oclIsUndefined())) **implies** **not** **self**.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) **and not**(el.getAppliedStereotype('featureB').oclIsUndefined()))

* featureA, featureB, featureC, and featureD are replaced with the actual features in FM_A.

cation variants are not directly generated from the use-cases modeled and this information is included for the purpose of completeness.

Developing Application Screens. Application screens (the interface displayed to the mobile application user) are helpful in gathering requirements for mobile application and can also serve as application prototype. For mobile applications, the application screens are mostly developed using the mobile applications graphical user interface native modeling tools (e.g., Android Studio (Google 2016), Microsoft Visual Studio (Microsoft, 2013)) for specific mobile platforms. These tools facilitate to generate native code for the application screens automatically. The designer uses these tools to develop application screens and automatically generate the code for the screens (Google 2016; Microsoft, 2013).

The proposed product-line model-driven engineering approach focuses primarily on automated generation of the native business logic code for diverse mobile platforms. For this reason, the proposed model-driven engineering approach separates the business logic layer from the user interface layer. For the purpose of separating the two layers, the controller pattern (Larman, 2004; Gamma et al., 1994) is used. The class for the controller pattern acts as a bridge between the business logic and user interface. The application designer refers the controller class in the code for the application screens.

For the Scramble case study, the application screens were developed using Android Developer Tools (ADT) for Eclipse. The screens for the Scramble case study include *Play-screen, Result-screen, History-screen*, and *Sharing-screen* and are already presented in Fig. 1 (see Section 2 for details).

Designing Use-cases. The application designer follow the typical use-case modeling (Bittner, 2002) to identify use-cases for the mobile application product-line under development. The designer further uses the developed application screens to identify the missing use-cases. For the use-cases, the designer develops a UML use-case diagram. The mobile application runs in a highly interruptible environment, such as, a call or a message or a mobile user can interrupt the mobile application during execution. For this reason, the typical use-case description (Bittner, 2002) is extended to include two new sections, i.e., *onInterrupt* and *onResume*. The *onInterrupt* section contains the details whenever a 'call' or 'message' or 'mobile user' interrupts during the mobile application execution. The *onResume* section contains the details when a mobile application

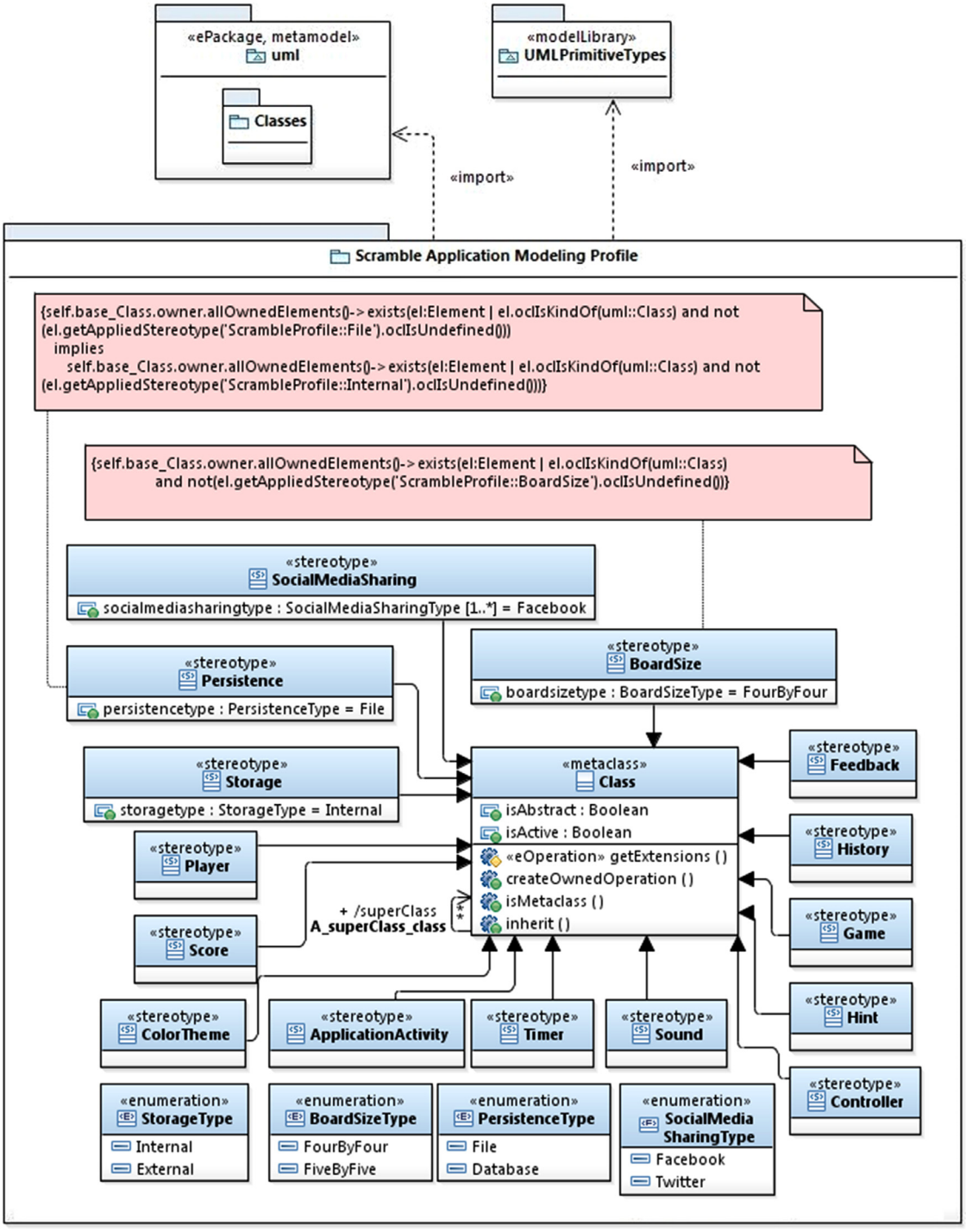


Fig. 7. Generated modeling profile for scramble mobile application product-line.

is resumed after an interrupt. The designer uses the extended use-case description to write use-cases in a typical concrete writing style (Bittner, 2002; Kulak and Guiney, 2012). The designer refers the developed application screens for the mobile application in the use-case description.

The use-cases for the Scramble case study include *Play, Show Hints, View History, Show Score, Take Feedback, Share Score, Time Settings*, and *Color Theme Settings*. Fig. 8 shows the extended use-case description for the 'Play' use-case in a concrete writing style. The 'Play' use-case description presents the 'Main Success Scenario' using the Scramble application screens (shown in Fig. 1) and the newly introduced sections specific for mobile applications.

6.2.2. Structural modeling of mobile application product-line

After the use-case modeling, the next step for the application designer is to model the structure of the mobile application product-line under development. UML class diagram is used to capture structural details of the mobile application. The application designer also applies the generated modeling profile during the modeling of UML class diagram for the under development application product-line.

Table 3
Generated constraints for scramble application modeling profile.

Feature / Constraint	Generated OCL Constraint
Persistence	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Persistence').oclIsUndefined())
ApplicationActivity	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::ApplicationActivity').oclIsUndefined())
Controller	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Controller').oclIsUndefined())
Timer	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Timer').oclIsUndefined())
Sound	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Sound').oclIsUndefined())
Storage	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Storage').oclIsUndefined())
BoardSize	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::BoardSize').oclIsUndefined())
History	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::History').oclIsUndefined())
ColorTheme	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::ColorTheme').oclIsUndefined())
Hint	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Hint').oclIsUndefined())
Score	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Score').oclIsUndefined())
Player	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Player').oclIsUndefined())
Game	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Game').oclIsUndefined())
Persistence	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::File').oclIsUndefined())) implies self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Internal').oclIsUndefined()))
Persistence	self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::Database').oclIsUndefined())) implies self.base_Class.owner.allOwnedElements()->exists(el:Element | el.oclIsKindOf(uml::Class) and not(el.getAppliedStereotype('ScrambleProfile::External').oclIsUndefined()))

Use Case Name: Play

Scope: Board
Level: user-goal
Primary Actor: Player
Stakeholders and Interests:
- Player: Wants to create words on the board.

Precondition: Player is using the scramble application.
Success Guarantee: Game result is generated.
Main Success Scenario:
1. Player initiates "Play" operation by clicking 'Play' button on Main screen.
2. Player creates words on the board, in the allocated time by joining different characters on Play screen.
3. As the time finishes, player views the Result screen.

Extensions:
*a. At any time, application fails.
1. Player restarts the application and start from the Main screen because there is no recovery mechanism supported in the application.

2a. Application displays "hints" to the player on Play screen.
1. Player views the hints on Play screen and creates words on the board with the help of hints.

3a. Application automatically "save" the result as history and overwrites the previous result history.
1. Player views the results and saves it as history on Results screen.

onInterrupt:
1. Application pauses the game timer on Play screen.
2. Application saves the current state of the game as position of the board, hints, and words created from Play screen.

onResume:
1. Application restarts the timer from where it was paused on Play screen.
2. Application retrieves the save state of the board, hints, and created words.
3. Application designs the board as it was before using the retrieved state on Play screen.

Fig. 8. Play use-case description in concrete writing style with extended sections.

Classes for Software and Hardware features. The designer does not need to model the classes for software and hardware features during UML class diagram modeling. As discussed earlier in Section 5.1, the design interface for each of the software and hardware features in FM_A are already provided and stored in an XML document. For example, there is a *Software* feature *Audio* in application product-line FM_A, the design interface *AudioImpl* for the *Audio* feature was already provided. The *AudioImpl* contains the functionalities offered by the *Audio* feature. For the design interfaces of software and hardware features in FM_A, a UML class diagram is generated. The designer further uses the generated UML class diagram to model the classes for requirement-related features in FM_A during the class diagram modeling. A strategy is used to generate UML class diagram from FM_A. Following, the details of the UML class diagram generation strategy are discussed.

For the generation of UML class diagram, the design interfaces (e.g., in Fig. 4) are extracted for each of the software and hardware features in FM_A from an XML document. A UML meta-class *Class* is used to model each of the design interfaces. The methods in the design interface are included as UML meta-class *Operation*. Using the above *Audio* example, a *Class* named *AudioImpl* is generated in the class diagram. The *getDuration* method in *AudioImpl* design interface is generated as an *Operation*.

The constraints in application product-line FM_A are used to model the relationships between the classes in the generated UML class diagram. In FM_A, five types of operators (i.e., IMPLIES, AND, OR, NOT, AND IFF) are used to define constraints on the features. The relationships are generated for AND, IMPLIES, and OR constraints only as these refer to the dependency in which a feature(s) require other feature(s). For example, *featureA* implies *featureB*, it means *featureA* is dependent on *featureB* or *featureA* requires *featureB*. The association relationship is generated for IMPLIES constraint as IMPLIES operator refers to a strong dependency between features. For example, *featureA* implies *featureB*. The association of 1 to 1 cardinality is generated between the *featureAImpl* and *featureBImpl* classes, i.e., the instance of *featureBImpl* is created in the *featureAImpl* class and vice versa. For both AND and OR constraints, the dependency relationship is generated. For example, *featureA* implies *featureB* and *featureC*. The dependency relationship is generated from *featureA* to *featureB* and *featureA* to *featureC*. The NOT and IFF constraints in FM_A are not used during the generation of relationships, as the NOT constraint excludes a feature and the IFF constraint refers to the conditional inclusion of the feature. Note that the generated class diagram represents the complete application product-line and not a specific application variant, none of the features are excluded on the basis of NOT and IFF constraints.

The generated UML modeling profile is applied on the generated class diagram. As discussed in Section 6.1, the UML profile is generated from FM_A, the stereotypes in the UML profile coincides with the features in FM_A. The stereotypes for the software and hardware features are applied on the generated classes for the software and hardware features. For this purpose, the name of the feature in FM_A is matched with the name of stereotype in UML profile. On successful match, the generated class for the matched feature is stereotyped with the matched stereotype. In *Audio* and *AudioImpl* example, the *Audio* stereotype is generated in UML profile for the *Audio* feature in FM_A. As the *Audio* name coincides for both the feature and stereotype, the generated *AudioImpl* class is stereotyped with *Audio* stereotype in the generated class diagram.

For the Scramble product-line FM_A (Fig. 5), the design interfaces for *Persistence, Sound, Timer*, and *Storage* features were already provided as part of the framework. Fig. 9 shows the generated UML class diagram for the software and hardware features in the Scramble FM_A.

Class for Controller feature. As discussed earlier (in Section 6.2.1), the proposed approach generates code for the business logic layer and provides a controller class that acts as a bridge between the business logic and user interface. As part of the class diagram generation strategy, a *BLController* class is generated corresponding to the *Controller* feature of the application FM_A. A *Controller* stereotype in the generated UML profile is applied on the *BLController* class to specify the controller for the business logic layer. During the mobile application modeling, the designer does not have to model the structural or behavioral details for *BLController* class. Once the application variant is generated from FM_A configuration, the designer provides the implementation of the *BLController* class using the controller pattern (Larman, 2004; Gamma et al., 1994). Fig. 9 shows the *BLController* class for the Scramble case study.

Identification of Requirement-related Classes. The application designer identifies the functional requirement classes for the mobile application product-line under development. For this purpose, the designer follows the typical object-oriented designing methods (Booch, 2006; Rumbaugh et al., 1991). The designer reviews the requirements of the application, the requirement-related features in FM_A for the application, and the use-cases (i.e., user interfaces and use-cases description) of the application to identify the missing classes in the application product-line. The application designer uses the generated class diagram for the software and hardware features in Scramble product-line FM_A to further model the requirement-related classes. For the Scramble product-line FM_A, *History* is a potential candidate to become a class. In 'Play' use-case for Scramble application, *Board* is a potential candidate to become a class. Fig. 10 shows the UML class diagram for the Scramble case study. The designer follows the object-oriented designing methods (Booch, 2006; Rumbaugh et al., 1991) to model the relationships between the classes in the UML class diagram.

The designer applies the requirement-related stereotypes along with the *ApplicationActivity* stereotype. The requirement-related stereotypes represent the features and their potential variations for the functional requirements of the under development mobile application. The designer reviews the modeled classes and applies the stereotypes on these classes. For the Scramble case study, *Scramble* is an active class that continuously interacts with the player. The *Scramble* class is stereotyped with *ApplicationActivity* stereotype. In Scramble case study requirements, *Board* is a class that has a board size, so *Board* is stereotyped with *BoardSize* stereotype. On applying the *BoardSize* stereotype, the attribute *boardsizetype* is automatically included in the *Board* class. The designer sets the value of the *boardsizetype* as *FourByFour* or *FiveByFive* using the *BoardSizeType* enumeration. The application designer applies the stereotypes on the modeled requirement-related classes in the UML class diagram.

6.2.3. Behavioral modeling of mobile application product-line

After the creation of the class diagram, the next step is the behavioral modeling of the under development mobile application product-line. The UML state machine is used to capture the behavioral details.

Behavior for the Generated Software and Hardware Classes. As discussed earlier in Section 5.1, the implementation APIs for each of the software and hardware features in the FM_A is already provided. The generated classes for these features (see details in Section 6.2.2) contain operations that are referred by the designer during the behavioral modeling of the requirement-related classes.

Behavior for the Requirement-related Classes. The designer models the behavior for all the requirement-related classes in the UML class diagram (Fig. 10). For this purpose, the designer develops the UML state machine for each of the requirement-related classes. The designer follows the typical UML state machine modeling (Mellor et al., 2002) to identify the states, event, and guards for each of the UML state machines. Further, the designer specifies the actions using Java programming language. The actions refer to the steps that

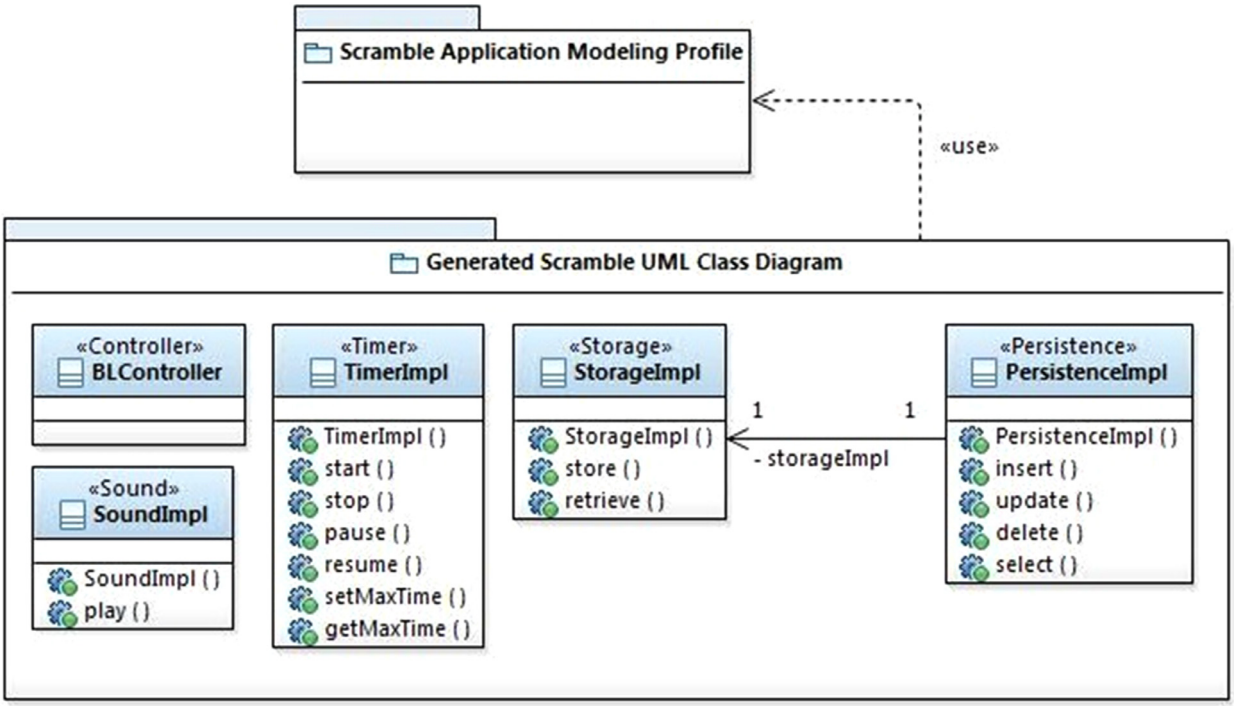


Fig. 9. Generated scramble class diagram for software and hardware features.

should be performed whenever an event is fired in the transitions of the state machine. Java is used as it is widely used programming language.

For the Scramble case study, Fig. 11 shows the UML state machine for *Board* class. For the *Board* class state machine (Fig. 11), in the transition from *Initialized* state to *NewWordCreated* state contains an action in addition to the event. It means when *createWord* event is called, it performs the actions associated with it. The designer may also refer the class in the action. For example, in Fig. 11, in the transition to *Initialized* state, the designer creates an instance of the *Score* class. In the transition from *NewWordCreated* state to *Validating* state, the designer uses an instance (i.e., *score*) of the *Score* class to call the *incrementCorrectScoreCounter* method.

In the proposed model-driven engineering approach, the designer does not need to consider the target application development platform while modeling. The developed models are platform independent. The mobile application generator automatically generates the target application variants according to the selected platforms, transforming the platform independent model into mobile platform specific code.

7. Mobile application variants generation

Once the designer has completed the development of application feature model (FM_A) and UML modeling (i.e., the development of use-cases, class, and state machines) for the under development mobile application product-line, next is the application variants generation. For the required application variants, the designer configures FM_A as a part of the proposed product-line engineering approach for mobile applications (presented in Section 5). The FM_A along with its configuration, the generated UML profile, and the modeled UML models are used to generate application variants. Following, the details of the FM_A configuration, linking FM_A with the modeled UML class diagram, and generation of application variants are presented.

7.1. Application product-line feature model (FM_A) configuration

The application FM_A is developed for the under development mobile application product-line (see Section 5.2 for details). The configuration of application product-line FM_A is the selection of the features and their variations for the generation of required mobile application variant(s). The FM_A contains operating system, software, hardware, and requirement-related features. For the selection of features from FM_A, the application designer reviews the requirements of the mobile application variant(s) to be generated. The designer selects the potential operating system, software, hardware, and requirement-related features. The number of selected operating system features reflects the mobile platforms while the selected software, hardware, and requirement-related features define the application variant(s). A number of possible configurations of FM_A can be developed as per the requirements of the application variants to be generated. Fig. 12 shows some of the possible configurations of the FM_A (Fig. 5) for the Scramble case study. The FeatureIDE (Thüm et al., 2014) tool is used for building configurations from FM_A. The tool presents the application product-line FM_A as a simple tree-like structure to the designer, from which features can easily be selected.

As discussed in Section 5.2, the FM_A contains all three types of parent-child relationships: *AND, OR*, and *ALT*. The mandatory features in the AND parent-child relationship are compulsory in all the configurations of FM_A (e.g. *Persistence*), these features are selected automatically. The designer selects the optional features as per application variant requirement (e.g. *Feedback*). In AND parent-child relationship, in case the parent feature is optional, the designer has to select the parent feature for the selection of its child features. For *OR* parent-child relationship, the designer has the facility to select at least one child feature but more child features can be selected during every configuration of the FM_A. For the OR parent-child relationship, the selection of the child feature is compulsory in which the parent feature is mandatory or selected. During each FM_A configuration for the ALT parent-child relationship, the designer is restricted to select exactly one child feature in case

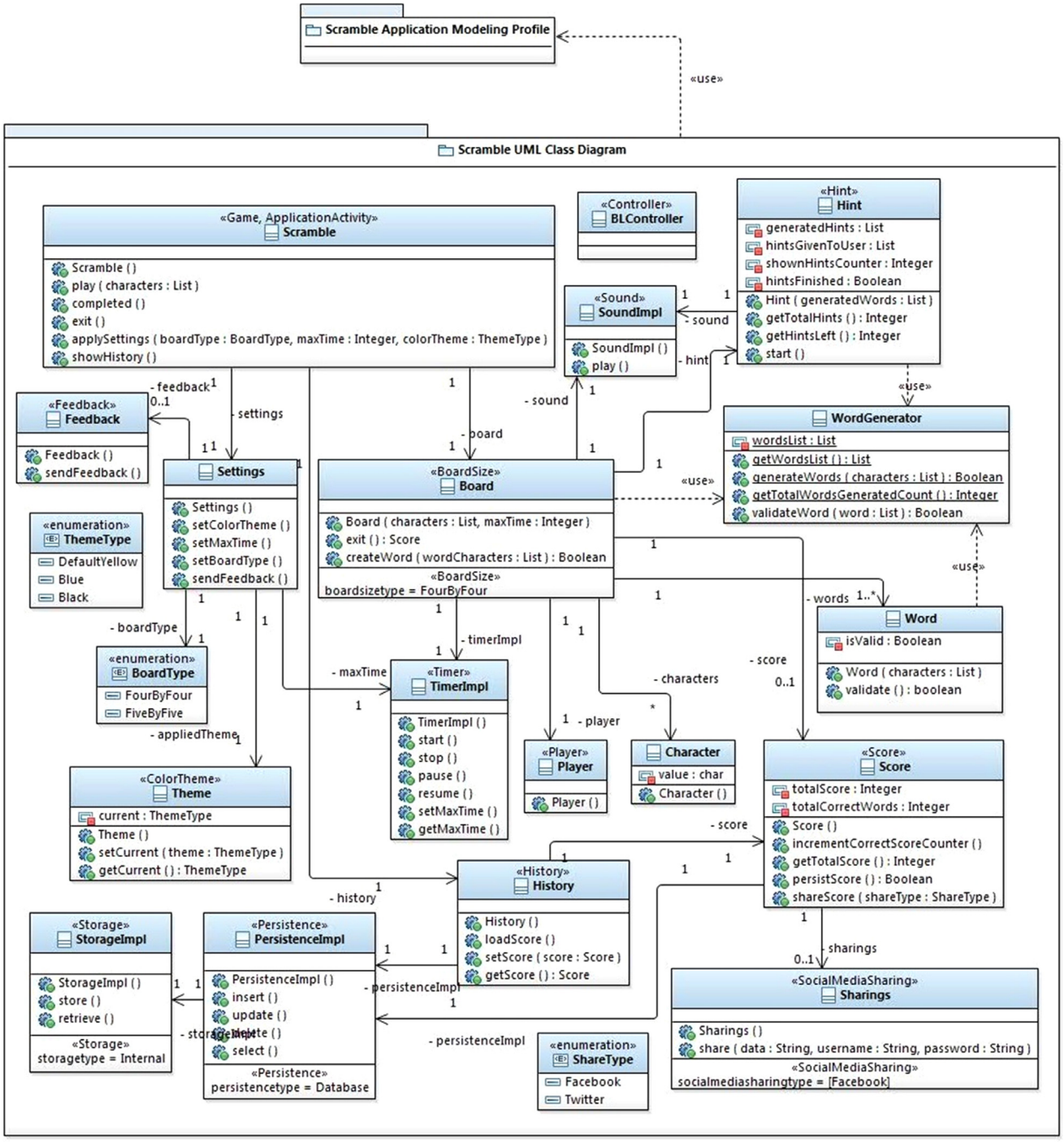


Fig. 10. Scramble mobile application product-line UML class diagram.

the parent feature is mandatory or selected. The FeatureIDE tool facilitates the designer during FM_A configuration by selecting the mandatory features automatically and also restricts the designer for deselection of these features. If none of the child features is selected in the OR and ALT parent-child relationship, the tool shows a validation error to the designer for the selection of at least one child feature for OR parent-child relationship and exactly one child feature for ALT parent-child relationship. Moreover on the selection of one of the child features in ALT parent-child relationship, the tool disables the remaining child features for selection.

In FM_A, the operating system feature is mandatory and contains an OR parent-child relationship. For this reason, the designer has to select at least one child feature of operating system feature but more child features can be selected. The operating system feature alone cannot specify a platform (e.g. *Android*), the version of the operating system is also important (e.g. *Android v 4.4, WindowsPhone v 8.1*). The designer selects the version of the operating system feature to specify the platform. For example, the designer selects *Android (v 4.4), WindowsPhone (v 7.8)*, and *IOS (v 7)* in FM_A configuration. These results in three operating system features being selected and application variants are generated for these platforms. For the required application variant, the designer selects the required software, hardware, and requirement-related features during FM_A configurations.

During FM_A configuration, the constraints on the features in FM_A are imposed automatically. The application designer uses the operators (i.e., IMPLIES, AND, OR, NOT, IFF) to define constraints on the features. For example, the constraint '*featureA* implies *fea-*

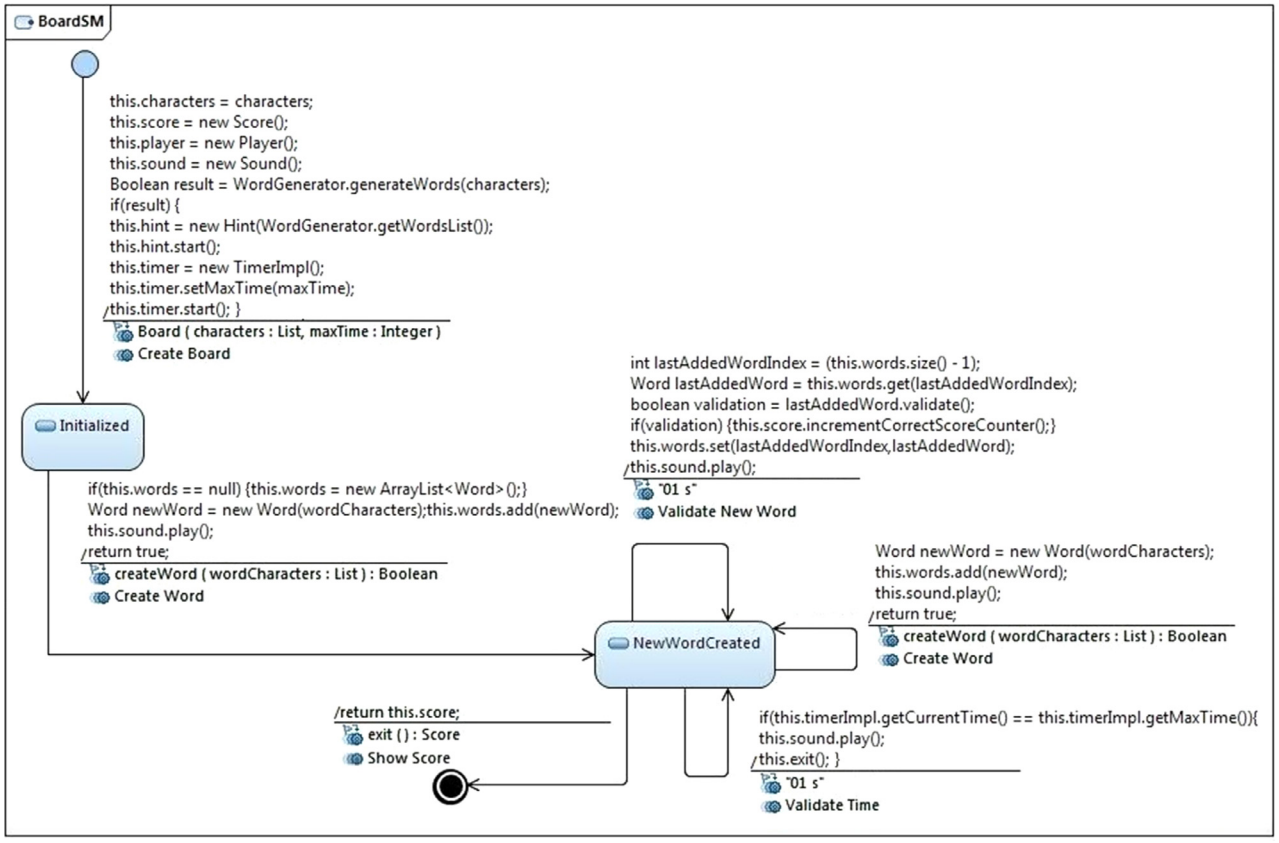


Fig. 11. Board UML state machine.

tureB' is imposed as: on the selection of *featureA, featureB* is selected automatically and vice versa, on the selection of *featureB, featureA* is selected automatically. Likewise IMPLIES, the AND, OR, IFF, and NOT operators are also imposed during FM_A configuration automatically (see constraint operators details in Section 5.1.3 and Section 5.2.5). The FeatureIDE tool imposes the constraints automatically during FM_A configuration.

Note that in the proposed product-line model-driven approach for mobile applications, the application variants are generated for the selected features in FM_A configuration only. No variants are generated on the basis of optional or unselected features in FM_A configuration. The designer has to select the features carefully that are required to be generated in the desired application variant(s) during FM_A configuration.

For the Scramble case study, Fig. 12 shows four configurations (out of many possible) as example developed from Scramble product-line FM_A (Fig. 5) using FeatureIDE tool. From Fig. 12, the FM_A configurations include (from LEFT to RIGHT): (a) Database persistence with Facebook social media sharing, (b) File persistence with Facebook and Twitter social media sharing, (c) Database persistence with Feedback, and (d) File persistence with Twitter social media sharing. In Fig. 12, the filled boxes represent the mandatory features in FM_A configurations that are either defined directly by the designer or included as a result of imposed constraints of an already selected feature. The ticked features represent features that the application designer requires to include in a desired application variant(s).

For the Fig. 12a FM_A configuration, the mandatory features (i.e., *Operaeting System, ApplicationActivity, Controller, Persistence, Timer, Sound, BoardSize, History, ColorTheme, Hint, Score, Player*, and *Game*) are selected automatically based on the AND parent-child relationship. There is no optional feature that is selected by the designer for the Fig. 12a FM_A configuration. For the *OR* parent-child relationship between the *Operating System* feature and its child features, the designer selects both the child features, i.e., *Android* and *WindowsPhone*. Moreover *Facebook* feature is selected by the application designer as a part of the OR parent-child relationship for the *SocialMediaSharing* feature. For the Fig. 12a FM_A configuration, the designer selects *FiveByFive* feature. For the ALT parent-child relationship for the *BoardSize* feature, the *FourByFour* feature is disabled for selection automatically on the selection of *FiveByFive* feature. The designer selects the *Database* feature that has the ALT parent-child relationship with the *File* feature for the *Persistence* feature. The selection of *Database* feature not only disables the *File* feature but also selects the *External* feature automatically. The *External* feature is selected as of the applied constraint in FM_A (Fig. 5), i.e., '*Database* **implies** *External*'.

Similarly in Fig. 12b FM_A configuration, the designer selects the *File* feature instead of the *Database* as a child of the *Persistence* feature. The selection of *File* feature automatically imposes the selection of the *Internal* feature based on constraint in the FM_A, i.e., 'File *imples* Internal'. As the OR parent-child relationship allows the selection of more than one child features, therefore the designer selects both the *Twitter* and *Facebook* features as children of *SocialMediaSharing* feature. For the Fig. 12c FM_A configuration, the optional feature, i.e., *Feedback* is selected by the designer. Moreover for the ALT parent-child relationship of *BoardSize* feature, the designer selects *FourByFour* instead of *FiveByFive* feature. For the *SocialMediaSharing* feature, the designer has not selected any child feature. The designer selects the *WindowsPhone* feature for the Fig. 12d FM_A configuration. Finally, the *Twitter* feature is selected as a child of *SocialMediaSharing* feature.

The four FM_A configurations in Fig. 12 are used to generate Scramble application variants based on the selected features in

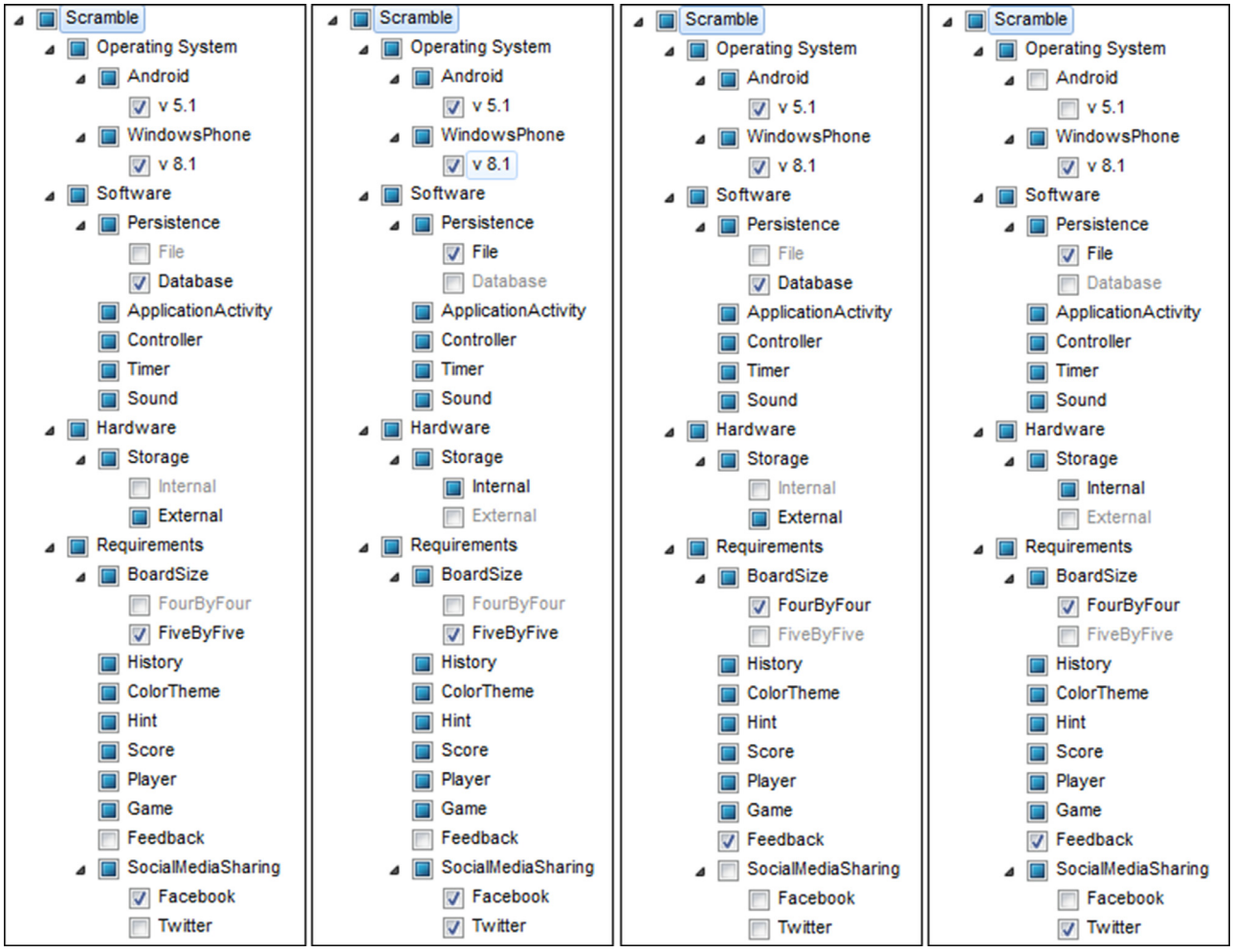


Fig. 12. Scramble product-line feature model FM_A configurations (a: database & facebook, b: file & sharing, c: database & feedback, d: file & twitter).

each of the FM_A configuration. For example, for the FM_A configuration in Fig. 12a, the Scramble application variants are generated for the selected *Android (v 5.1)* and *WindowsPhone (v 8.1)* platforms. Each of the platform variants includes *ApplicationActivity, Controller, Database, Timer, Sound, External, FiveByFive, History, ColorTheme, Hint, Score, Player, Game*, and *Facebook* features.

7.2. Linking application feature model (FM_A) with the modeled UML class diagram

In this section, the details of how the application product-line FM_A is linked with the modeled UML class diagram are discussed. For this purpose, the features in the FM_A are linked with the classes in the UML class diagram for the mobile application product-line under development. As a consequence, the code is generated only for those classes which are linked with the features selected during the FM_A configuration.

To link the features with the classes, the UML modeling profile plays a vital role. The modeling profile is generated from the application product-line FM_A (see details in Section 6.1), so the stereotypes in the profile coincide with the features in the FM_A. Further, the generated profile is applied on the modeled classes in the UML class diagram (see Section 6.2.2 for details). An algorithm is provided to implement the linking of the features in the FM_A with the stereotyped classes in the class diagram. According to the algorithm, for each of the classes, the stereotypes are extracted. For each of the stereotype, a match is performed between the name of the stereotype and the name of the feature in the FM_A. On a successful match, the link for the modeled UML class is created in the feature. For example, *Bluetooth* is a feature in the application FM_A. The stereotype named *Bluetooth* is generated in the UML profile. In class diagram, a *Bluetooth* stereotype is applied on the *BluetoothImpl* class. For this example, the algorithm extracts the *Bluetooth* stereotype applied on the *BluetoothImpl* class. The *Bluetooth* feature is searched for the *Bluetooth* stereotype. On successful search, the *Bluetooth* feature is linked with the *BluetoothImpl* class.

For the requirement-related features in the FM_A that have a parent-child relationship, a possibility is that the designer may apply the generated stereotype for the parent feature on more than one class in the UML class diagram. In this case, a feature is linked with more than one class. Another possibility is that the designer may apply more than one stereotype in the generated UML modeling profile on a class. For this case, more than one feature is linked to a class. To handle these cases, a list of classes is used in the feature to link more than one class in the UML class diagram.

The linking algorithm is presented in Listing 3. The input to the algorithm is the application product-line feature model (FM_A) and the classes in the UML class diagram. The output of the algorithm is an updated FM_A linked with the UML classes. The algorithm starts with traversing the list of classes as shown in line 2. A list of applied stereotypes is extracted for each class. Every stereotype is matched in the FM_A as shown in line 9. When the match is successful then a link is created by adding a class attribute in the feature as shown in line 10 of Listing 3.

```
Algorithm     LinkFeaturesWithClasses(F, C)
Input         F: root feature of the tree data structure to traverse feature model, C: an array of classes
Declare       I_C: contains an instance of class, S: an array of stereotypes, I_S: contains an instance of stereotype,
              N_S: presents name of a stereotype, CD_F: an array of children features, I_CD: contains an instance of
              feature, N_CD: presents name of a feature
1.  begin
2.    for each I_C in the C
3.      Initialize S with the applied stereotypes on I_C
4.      for each I_S in the S
5.        Initialize N_S with the name of stereotype from I_S
6.        Initialize CD_F with the children of F
7.          for each I_CD in the CD_F
8.            Initialize N_CD with the name of feature from I_CD
9.            if N_CD == N_S          then
10.             Add I_C to the class list in I_CD
11.             break the for each loop
12.           end if
13.       end foreach
14.     end foreach
15.   end foreach
16. end
```

Listing 3. Algorithm to link FMA features with UML classes.

For the Scramble case study, the generated Scramble UML modeling profile (Fig. 7) is used to link the features in the Scramble product-line FM_A (Fig. 5) with the classes in the Scramble UML class diagram (Fig. 10). For example, the *Controller* feature is linked with *BLController* class as of *Controller* stereotype is applied on the *BLController* class. Similarly, the *Timer* feature is linked with the *TimerImpl* class, the *Storage* feature is linked with the *StorageImpl* class, and the *Player* feature is linked with *Player* class. In Scramble class diagram (Fig. 10), the *Scramble* class is stereotyped with two stereotypes, i.e., *Game* and *ApplicationActivity*. For this reason, the *Game* and *ApplicationActivity* features are linked to a *Scramble* class.

7.3. Variant generation

The configuration of FM_A, the linked application FM_A, and the modeled UML models are used to generate the desired mobile application variant(s) for multiple platforms. The FM_A and its configuration are used to compute not only the mobile platforms and application variants but also the features to be included in each of the application variants.

The FM_A consists of operating system, software, hardware, and requirement-related features (see Section 5.2 for details). In FM_A configuration, the selected operating system features are used to extract the mobile platforms while the selected software, hardware, and requirement-related features define the application variants. All mandatory features are selected along with any selected optional features. It is possible that the designer selects more than one child features in OR parent-child relationship. For this case, all the selected child features are included in the application variant. For each of the mobile platform, the included features in the application variant that are already linked with the UML classes are used to generate the application variant.

The features in the application variant and the modeled UML models are used to generate variants for the under development mobile application product-line. From the modeled UML models, the UML class diagram and the UML state machines are used. UML class diagram is used to generate structural code while the behavioral code is extracted from UML state machines. Code generation assumes that the classes linked with the selected features are modeled completely using the UML class diagram and state machines.

The UML class diagram contains classes that are linked with the software, hardware, and requirement-related features in the application FM_A or the classes that are in a relationship with the linked classes. For the classes that are linked with the selected software and hardware features are not candidate for the code generation. As these classes were already generated as part of the proposed approach (see details in Section 6.2.2) and the implementation for these classes was already provided (see details in Section 5.1.1). The code is generated for the classes that are linked with the selected requirement-related features. The code is also generated for those classes that are in a relationship with these classes but not linked with any unselected feature. For the generation of code from classes, the code generation rules by Usman and Nadeem (2009) are used. The UML class diagram modeling elements that are included in the code generation rules (Usman and Nadeem, 2009) are: *Class, Stereotyped Class, Interface, Stereotyped Interface, Enumeration, Attribute, Stereotype Attribute, Operation*, and *Relationships* (i.e., *Inheritance, Association, Composition* and *Aggregation*). The code generation rules do not include the rule for the *Package* element in the UML class diagram modeling. As per the code generation rules, for each modeled class in the UML class diagram, a class in the programming language is generated. For each property in the modeled class, a field in the associated class is generated. For each association relationship between the modeled classes, a field is generated in the associated class or classes. The generation of field in the association relationship is guided by the navigation property. The inheritance relationship is implemented as an extension from the base class. The code generation rules generate getter and setter for all the fields in the classes automatically. To include the class methods and their implementations, a UML state machine is used.

For the generation of code from state machine, the well-known state pattern (Gamma et al., 1994) is implemented. The state pattern implementation by Iqbal et al. (2015) is used to generate code from the state machine. Based on the implementation, a parent state class is created for the state machine. For every state, a class is generated. The generated classes from states are inherited from the parent state class. Fig. 13 shows the implementation of state pattern for the *Board* state machine (presented in Fig. 11). For each state, the event on the outgoing transition is included as a method in the generated class for the state. For example, from Fig. 11, an outgoing transition from state *NewWordCreated* to state *Validating*

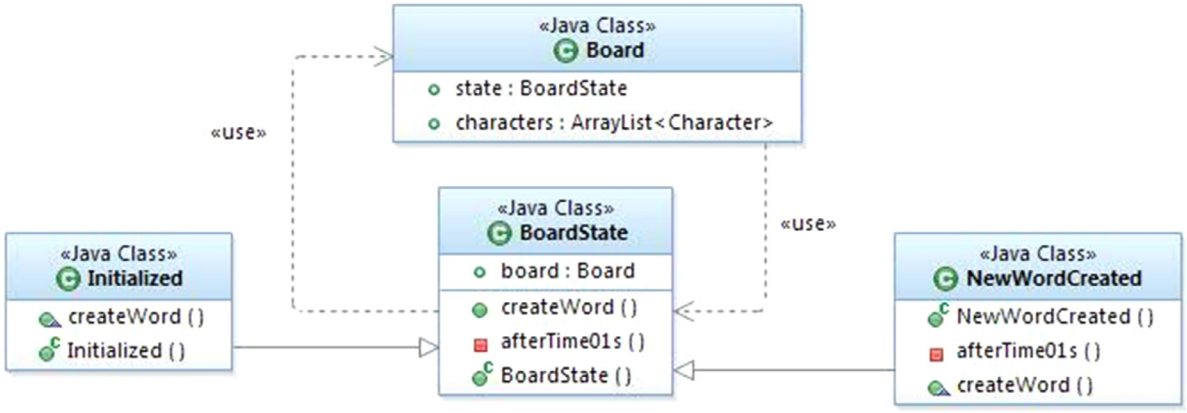


Fig. 13. State pattern implementation for board state machine.

contains an event *validate*. For this case, *validate* method is included in class *NewWordCreated* as shown in Fig. 13. The actions on the outgoing transition are included as method's body in the generated class for the state and guard in the outgoing transition of the state transforms to an if-else block in the method.

As discussed in Section 6.2.2, the controller pattern is used to bridge the user interface layer and the generated business logic layer. For this purpose, the *BLController* class is generated in all the application variants. The designer implements the typical controller pattern (Larman, 2004; Gamma et al., 1994) in the *BLController* class.

The *ApplicationActivity* stereotype is applied on the active class in the application. For the active class in the *Android* platform, the requirement is the implementation of methods (i.e., onCreate, onPause, onResume, onStart, onStop, and onDestroy) and is not compulsory for other platforms. During the generation of code for the classes that are stereotyped with *ApplicationActivity* stereotype, these methods are generated for the *Android* platform application variant only. The designer will implement these methods.

Note that the generation of class for the parent-child relationships in FM_A is dependent on the selection of the child feature(s) during FM_A configuration. For the ALT parent-child relationship, exactly one child feature is selected, so the class is generated for the selected child feature only. For the AND and OR parent-child relationships more than one child features can be selected. In this case, all the classes for the selected child features are generated as each of the child features refer to a class.

For the Scramble case study, some of the possible FM_A configurations are shown in Fig. 12. For the FM_A configurations in Fig. 12a, the Scramble application variants are generated for the selected *Android (v 5.1)* and *WindowsPhone (v 8.1)* platforms. Each of the platform variants includes *ApplicationActivity, Controller, Database, Timer, Sound, External, FiveByFive, History, ColorTheme, Hint, Score, Player, Game*, and *Facebook* features.

In case, a feature in FM_A is not selected during configuration, the code corresponding to the feature will not be generated. This will lead to a compilation error in the generated application variant in the code where the feature is required. To avoid this situation, the designers have to provide the constraints in FM_A carefully to links the dependent features or to select the dependent features during FM_A configurations.

8. Automation

The **MOPPET** (**MO**bile a**P**plication **P**roduct-lin**E** genera**T**or) tool is used to automate the mobile application product-line generation. The MOPPET tool not only implements the mobile application variants generation for multiple platforms (see variant generation details in Section 7) but also integrates the components that are used during the product-line generation of mobile applications. Fig. 14 shows the integration of components with the MOPPET. The integrated components include *FeatureDetails, ProfileGenerator*, and *ClassDiagramGenerator*. The *FeatureDetails* component is used for the storage of design interface and implementation APIs of the software and hardware features in the FM_C (see details in Section 5.1.4). The *ProfileGenerator* component implements a strategy to generate UML modeling profile for the software, hardware, and requirement-related features in the FM_A (see details in 6.1). The *ClassDiagramGenerator* component implements a UML class diagram generation strategy for the software, and hardware features in the FM_A (see details in 6.2.2).

For the generation of mobile application variants for multiple platforms, the MOPPET tool takes the application FM_A, the FM_A configuration, the generated UML modeling profile, and the UML models (i.e., class and state machines) for the under development mobile application product-line. The current implementation of the MOPPET tool supports only *Android* and *Windows Phone* mobile platforms, but can easily be extended to support other platforms. The MOPPET tool is developed using *Eclipse*. The architecture of the MOPPET tool is presented in Fig. 15. The developed tool consists of three major components, i.e., (i) *Model Reader*, (ii) *Model Linker*, and (iii) *Application Generator*. In model reader, the input models are read. In model linker, the application feature model is linked with designed class diagram. In application generator, the product-line feature model and its configuration along with the UML diagrams are used to generate feature-based mobile application variants. The detailed description of these components is presented as follow.

The *Model Reader* is the first component in the MOPPET tool. It reads the developed models. The inputs are: (i) Application feature model (FM_A), (ii) FM_A configuration, (iii) UML modeling profile, and (iv) UML models. As none of the inputs are dependent on each other, therefore the model reader reads all the inputs in parallel. The application product-line FM_A is developed in the FeatureIDE tool that uses an XML document (.xml file) for a feature model and a configuration file (.config) to store feature model configuration. The model reader reads the .xml file to a 'feature model data structure'. Fig. 16 presents a 'feature model data structure' that consists of the features, their variations, relationships, and constraints. The model reader reads the .config file that contains one feature per line. The FeatureIDE tool creates a configuration file for selected features only. In 'feature model data structure' (Fig. 16), the *Feature* class contains an attribute *selected* of *boolean* type to specify the selection of the feature in the configuration. The default value for the *selected* attribute is *false*. On reading a feature from .config file, the model reader sets the *selected* attribute for the specific fea-

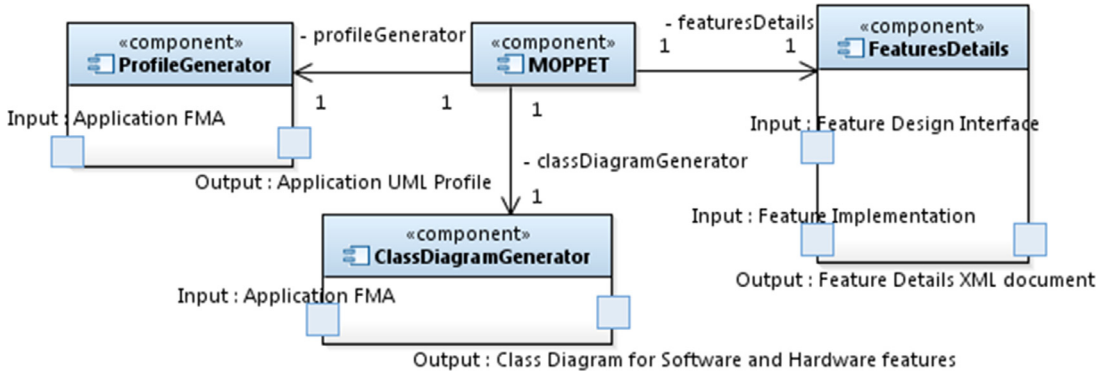


Fig. 14. MOPPET tool with integrated components.

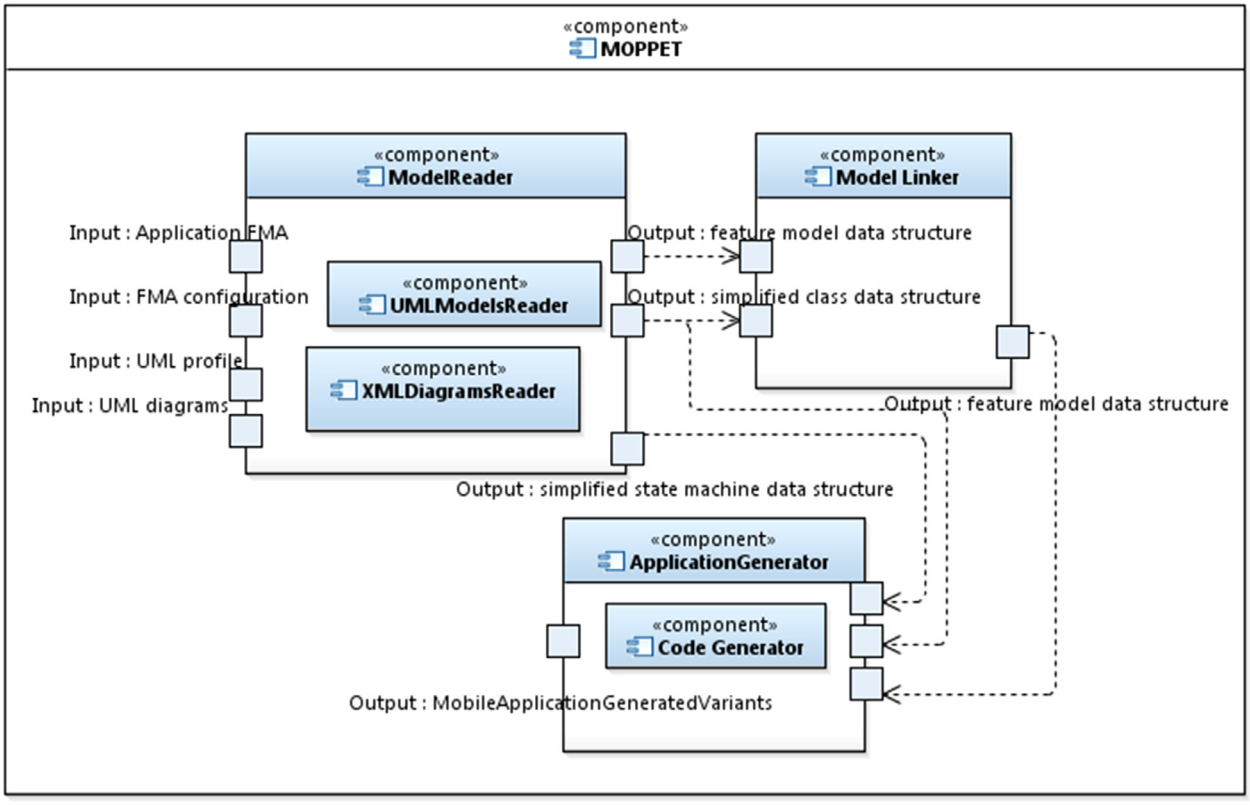


Fig. 15. Mobile application product line generator (MOPPET) tool architecture.

ture to *true*. For example, on reading *ApplicationActivity* feature, the reader set the *selected* attribute of the *ApplicationActivity* feature to *true*. The UML profile is generated in the standard EMF format for UML (.uml file) that is widely accepted format for the UML models. The model reads the .uml file for the UML profile. The *Rational Software Architect* UML modeling tool (IBM 2015) is used for the designing of UML models that uses the .uml file for the UML models. The model reader reads the .uml file for the UML models.

The *Model Linker* is the second component in the MOPPET tool as shown in Fig. 15. It links the feature model with the class diagram (see details in Section 7.2). The model linker implements the linking algorithm that is used to link features in feature model with classes (algorithm presented in Listing 3). The 'feature model data structure' contains the features in feature model which links the classes in the class diagram.

The *Application Generator*, shown in Fig. 15, generates the required application variants for the mobile platforms. The inputs to this component are the linked feature model data structure and UML class diagram and state machine along with the UML profile. First, the application generator identifies the mobile platforms and application variants. For this purpose, the 'feature model data structure' is used. The application generator uses the *Feature* class and *FeatureAttributes* class to identify the platforms and application variants (see details in Section 7.3). For each of the application variant, the features are used to generate code from the 'class' and 'state machine'. For the generation of code, the application generator implements the mechanism discussed in Section 7.3. The code is generated in the specific programming language for the mobile platform (e.g., Java is used for *Android* platform, C# is used for *Windows Phone* platform, and Objective-C is used for *iOS* platform). The 'class' is used to generate structural code while behavioral code is generated from the 'state machine'. As the actions in the state machine are written in Java, so these are used as it is for the Android platform application variant. For the other platforms, the actions are transformed according to the platform specific programming language.

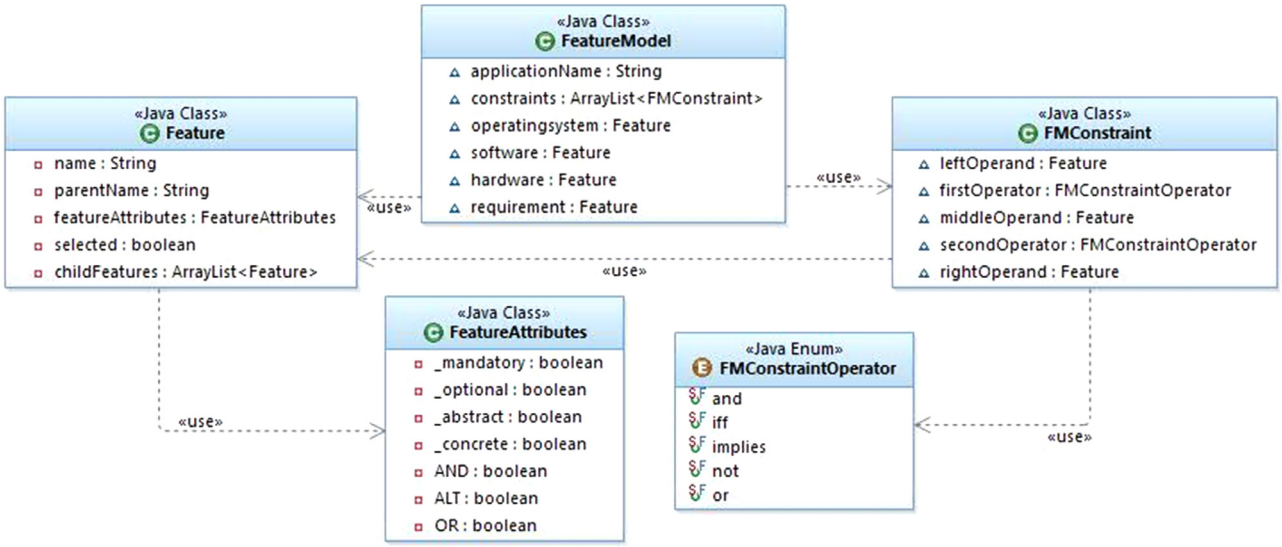


Fig. 16. Feature model data structure.

9. Validation of the proposed approach

In this section, the practicality of the proposed product-line model-driven engineering approach is evaluated by applying it on two case studies, i.e., *Scramble* and *Instalapse*. Following, the details of these case studies are discussed.

9.1. Case studies

The *Scramble* case study is used as a running example throughout the paper and *Instalapse* is discussed in this section.

To apply our product-line model-driven engineering approach on Instalapse case study, the designer first model the mobile application product-line feature model. The designer uses the proposed generic feature model (FM_G) as a guideline to develop a feature model for the Instalapse mobile application product-line (FM_A). Fig. 17 presents the Instalapse product-line feature model (FM_A). The designer selects the configuration of FM_G by selecting the features. The dependencies, conflicts and variations for the selected features are automatically included in the Instalapse product-line FM_A. The designer extends FM_G to include 'MediaPlayer' and 'Gallery' software features that are required by the Instalapse application. The designer defines the functional requirement-related features, their variations, dependencies, and constraints in the Instalapse product-line FM_A. The Instalapse application consists of features like data storage support through file, picture or video capture through front or back camera, lightweight phone storage feature or SD card storage, and social media sharing through *Facebook*, *Twitter*, and *Instagram*.

In model-driven approach, the mobile application is modeled using the generated product-line UML modeling profile and UML models. The Instalapse product-line UML modeling profile is generated using the Instalapse product-line FM_A (see discussion in Section 6.1). Once the application product-line UML modeling profile is generated, the designer starts with the requirement specification through UML use-case diagram. The designer develops use-case diagram by our approach presented in Section 6.2.1 for the Instalapse case study. Next step is to model UML class diagram. The UML class diagram is generated for the software and hardware features in Instalapse product-line FM_A that include *CameraImpl, StorageImpl, PersistenceImpl, BLController, MusicImpl, MediaPlayerImpl*, and *VolumeControllerImpl*. The designer uses the generated UML class diagram to include requirement-related classes (see discussed in Section 6.2.2). The class diagram consists of a number of classes that represent the auto-generated classes as well as those modeled by the user. After structural modeling, the next step is to model the behavior using UML state machine. The designer models the state machines for the requirement-related classes only. The behavior for software and hardware features in Instalapse product-line FM_A is already provided. Fig. 18 shows the UML state machine for *Record* class. The designer also specifies the actions in Java programming language to add behavioral details in the state machine.

The configuration of the Instalapse product-line feature model (FM_A) drives the generation of variants. Fig. 19 presents the configuration for Instalapse product-line FM_A. The mandatory features in the Instalapse FM_A are selected automatically that include *ApplicationActivity, Controller, Video, MediaPlayer, Gallery, VolumeController, Camera, Timelapse, Record*, and *Duration*. The designer selects the desired features as per requirement of the Instalapse application variant that include *Android* (*v 5.0* and *v 5.1*), *WindowsPhone* (*v 7.8* and *v 8.1*), *Persistence* (*File*), *Camera* (*Back*), and *SocialMediaSharing* (*Facebook* and *Twitter*). The *Storage* (*External*) is selected automatically on the selection of *Persistence* (*File*) feature because of the constraint in Instalapse FM_A.

The developed MOPPET tool is used to generate the Instalapse feature-based variants for multiple platforms. The inputs to the tool are the Instalapse product-line FM_A, Instalapse FM_A configuration, and Instalapse UML models (class diagram and state machines) with generated profile. Using the Instalapse FM_A configurations (Fig. 19), the tool generates mobile application variant for four platforms, i.e., two for the Android platform and two for the Windows Phone platform. The application variant includes features that are: *ApplicationActivity, Controller, File, Video, MediaPlayer, Gallery, VolumeController, External, Back, Timelapse, Record, Duration, Facebook*, and *Twitter*.

For our product-line model-driven engineering approach in both the case studies (*Scramble* and *Instalapse*), the mobile application generic feature model (FM_G) contains most of the features and provides a good starting point. The application designer develops their application product-line feature models (FM_A) using the generic feature model as guideline, adding any missing feature and the functional features manually. Each application product-line FM_A only contains features that are necessary for that particular application and its variants. For example, Scramble application does not require *Camera* feature that is a core feature in Instalapse

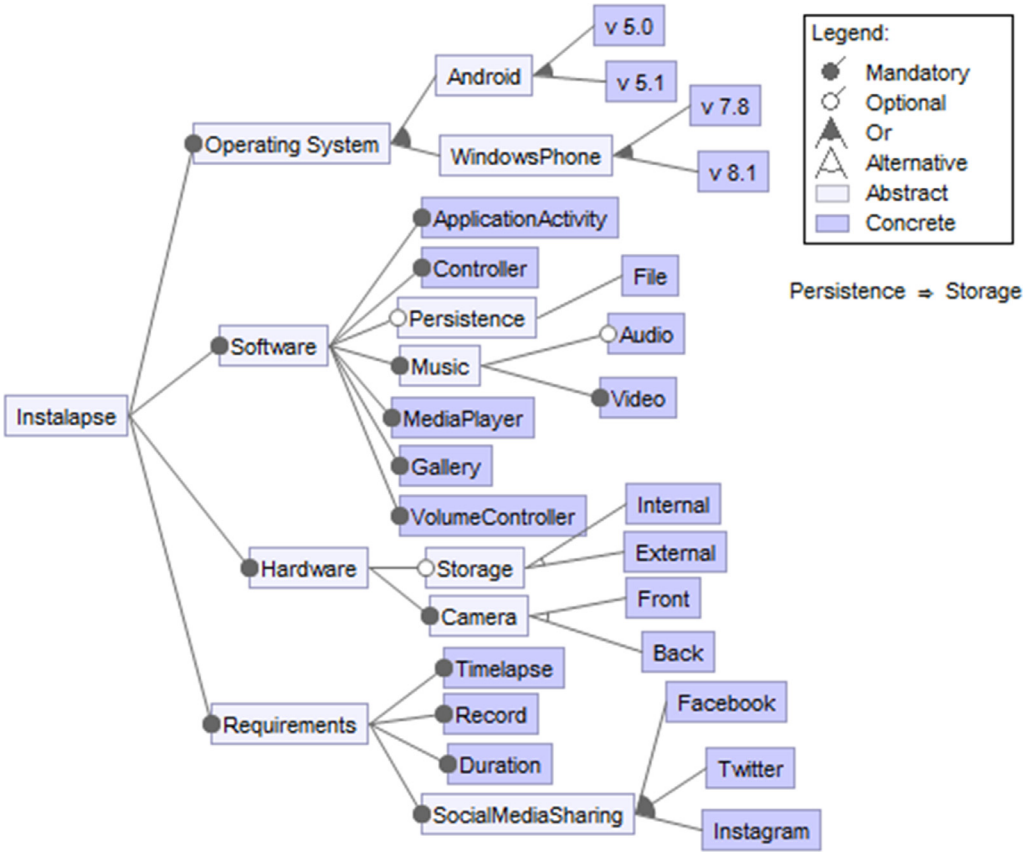


Fig. 17. Instalapse mobile application product-line feature model (FM_A).

Table 4
Mobile application product-line feature model statistics.

Application	Total	Mandatory	Optional	Abstract	Concrete	Composite
Scramble	27	14	2	6	21	6
Instalapse	29	11	4	7	22	7

Table 5
Mobile application product-line generated UML profile statistics.

Application	Total features	Required to apply	Parent–child relationship	OCL Constraints
Scramble	15	13	4	15
Instalapse	13	10	5	11

application, so it is included in Instalapse product-line FM_A and excluded from the Scramble product-line FM_A. The FM_A configurations for both the applications depend on requirements for variant to be generated and available hardware features in target mobile device.

9.2. Evaluation

The results covering generated application variants for both the case studies through the proposed product-line model-driven engineering approach are presented in the following.

Table 4 shows the statistics for the application product-line feature models developed for the *Scramble* and *Instalapse* case studies. For Scramble product-line FM_A (Fig. 5), there are twenty-seven features, out of which fourteen are mandatory features, while two features are optional. The feature model contained a total of twenty-one concrete features. Over six different composite features are present in the feature model. For Instalapse product-line FM_A (Fig. 17), there are twenty-nine features, out of which eleven are mandatory features, while two features are optional. For both the case studies, the operating system, software and hardware features are derived from the generic feature model (FM_G) (see details in Section 5.1). For the operating system feature, the *Android* and *Windows Phone* platforms are selected from FM_G but the versions for the platforms vary in both the case studies. For the Scramble case study, *Android (v5.1)* and *Windows Phone (v8.1)* are selected while *Android (v5.0* and *v5.1)* and *Windows Phone (v7.8* and *v8.1)* are selected for the Instalapse case study. Similarly, nine out of ten software and hardware features in Scramble product-line FM_A are already available in FM_G and the designer extends FM_G to add the *Sound* feature for the Scramble case study. The Instalapse product-line FM_A contains sixteen software and hardware features, the FM_G contains fourteen features while the designer adds the *MediaPlayer* and *Gallery* features in the FM_G for the Instalapse case study. This reflects that for both the case studies, FM_G plays a vital role in the development of the application FM_A. The requirement-related features are specific to the mobile application functional requirements, and these vary from application to application, such as, there are thirteen requirement-related features in Scramble product-line FM_A while seven requirement-related features in Instalapse product-line FM_A. This is a rare case in which both the case studies have a common requirement-related feature, i.e., *SocialMediaSharing* but its variations are different.

Table 5 shows the statistics of the generated UML modeling profiles for both *Scramble* and *Instalapse* product-lines. For the Scramble case study, a total of fifteen stereotypes are generated in the modeling profile. This includes four stereotypes that be-

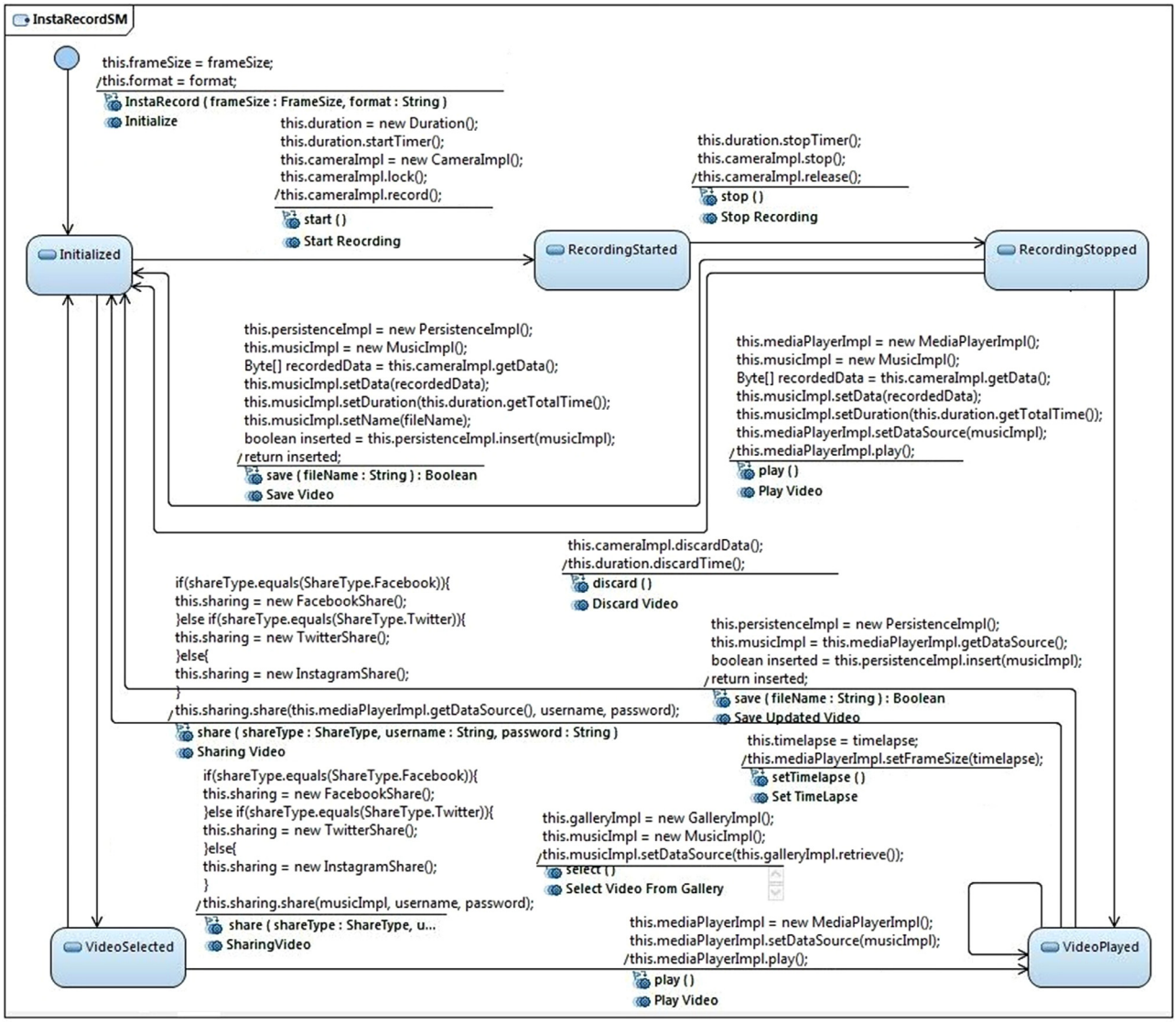


Fig. 18. Record UML state machine.

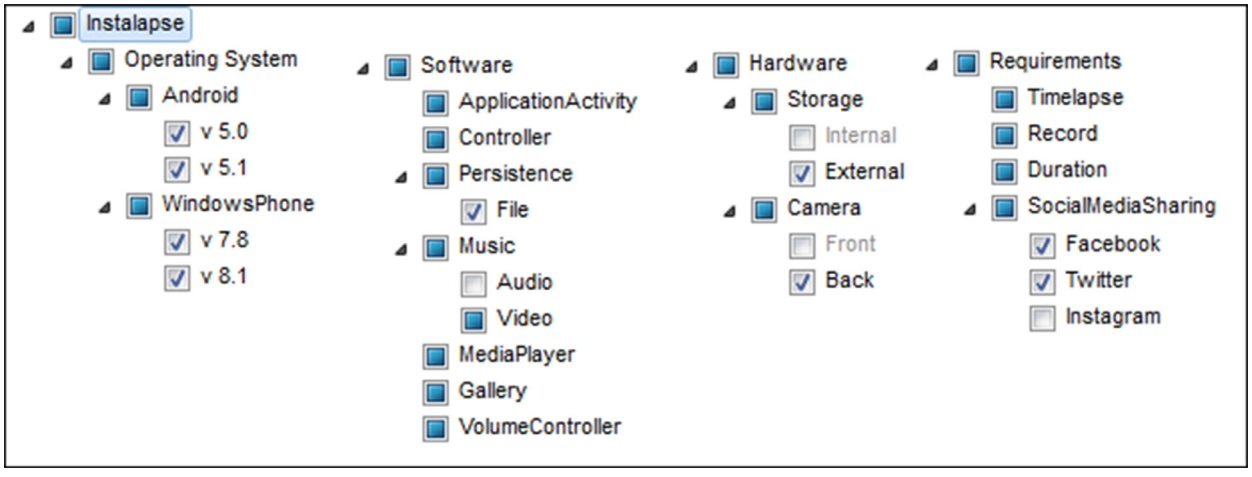


Fig. 19. Instalapse product-line feature model (FM_A) configuration.

Table 6
Mobile application product-line UML class and state machine modeling statistics.

Application	Classes	State machine statistics			Class diagram statistics			
		States	Triggers	Lines of actions	Attributes	Methods	Associations	Applied stereotypes
Scramble	Board	2	6	31	0	3	7	1
	Settings	1	5	11	0	5	4	0
	Hint	2	6	24	4	4	1	1
	History	2	4	6	0	4	2	1
	Score	4	6	17	2	5	2	1
	Scramble	3	6	11	0	6	3	2
Instalapse	Duration	2	6	18	3	5	1	1
	Record	5	12	55	1	10	9	1
	Instalapse	3	5	6	0	4	1	1
	Timelapse	1	3	4	1	3	1	1

long to the features in a parent-child relationship for the Scramble product-line FM_A. A total of thirteen stereotypes are generated in the modeling profile for the Instalapse case study. The five stereotypes are for the features in the parent-child relationship. In Scramble product-line UML profile, thirteen stereotypes are required to apply during the modeling of Scramble application while ten stereotypes should be applied during the Instalapse application modeling. There are fifteen OCL constraints generated from the Scramble product-line FM_A while eleven OCL constraints are generated from the Instalapse product-line FM_A. For the Scramble case study, seven stereotypes are applied automatically on the classes for the software and hardware features while for the Instalapse case study, eight stereotypes are applied on the classes for the software and hardware features. Nearly half of the stereotypes in UML modeling profile are applied automatically for both the case studies. During the Scramble application product-line modeling, the designer was required to apply seven stereotypes that belong to the features that are mandatory in the FM_A. For the Instalapse case study, four mandatory stereotypes were required to be applied by the designer.

Table 6 shows the statistics of the modeled requirement-related classes developed for both *Scramble* and *Instalapse* case studies. For the Scramble application product-line, the developed class diagram comprises of twenty-two different classes, with fifteen stereotypes. This also includes five generated classes from the Scramble product-line FM_A (see details in Section 6.2.2). There are sixteen different classes with thirteen stereotypes in the developed class diagram for the Instalapse application product-line. Eight classes are generated from the Instalapse product-line FM_A. For both the case studies, the designer implements the *BLController* class through the controller pattern (see discussion on *Controller* class in Section 6.2.2) to integrate the generated business logic layer with already developed application screens (see details in Section 6.2.1). The largest state machine modeled for Scramble case study is for the *Board* class that contains 2 states and 6 transitions while for *Instalapse* mobile application, the largest state machine modeled is for the *Record* class that contains 5 states and 12 transitions. For both the case studies, the modeled state machines contain *simple* states, mostly *call* event is used but there are situations where *time* and *change* events are also used.

Table 7 shows the statistics corresponding to generated application variants for *Scramble* and *Instalapse* case studies. For the Scramble application, a total of sixty-two classes are generated by the MOPPET tool for both *Android* and *Windows Phone* variants. This is a good indicator for the amount of coding effort for variant development that is saved by the MOPPET tool. For Instalapse application, a total of forty classes are generated for both *Android* and *Windows Phone* variants. For Scramble application, the total generated lines of code for Android platform are around 1800 while for Windows Phone platform around 1700 lines of code are generated. For Instalapse application, the total generated lines of code for Android and Windows Phone platforms are around 1300 and 1150 respectively.

Table 7
Mobile application product-line generated code statistics.

Applications	Operating system	No. of generated classes	Lines of generated code
Scramble	Android	62	1815
	Windows phone	62	1699
Instalapse	Android	40	1278
	Windows phone	40	1145

The results show the approach was successful in generating multiple application variants as per the requirement of the product-line. Development of these variants manually would have required a significant redundant effort. This was also the point of view of the development team at Invotyx.

To summarize, the work was conducted in collaboration with our industry partner. Invotyx was facing the problem of developing and maintaining a large number of cross-platform versions and approached the researchers for solving the problem. These challenges are not unique to Invotyx but are rather common to the mobile application industry that faces similar challenges when supporting multiple platforms, devices, and feature-based variants. Therefore the presented case studies in the paper are representatives of the challenges faced by the industry that the approach is designed to solve. While developing the approach, the researchers had multiple sessions with the development team to understand their requirements. The solution was developed keeping the requirements of the general mobile application development. The solution was then provided to Invotyx, which then utilized MOPPET to generate the *Scramble* and *Instalapse* feature-based application variants. One observation from the case studies was that there is a significant initial learning curve involved. While software engineers are somewhat familiar with UML, they are generally not well-versed in using state machines, or applying product-line engineering concepts, such as, feature models. To overcome this issue, training sessions were held for the developers. Once the development team became familiar with the modeling approach used, a significant reduction in time and effort of generating and maintaining subsequent product-line variants was observed.

9.3. Limitations

One potential limitation of the presented approach is that we focus on generating “business logic” code only and do not generate the user interface. As per the discussion with the industrial partner, there are many techniques and tools available to develop native graphical user interface for mobile applications. One of the

most prevailing technique for developing graphical user interface is drag-and-drop utility tools designed especially for the targeted mobile platforms, such as, Android Studio (Google 2016) for Android platform, Microsoft Visual Studio (Microsoft, 2013) for Windows Phone platform, and XCode (Apple 2016) for iOS platform. The native graphical user interface is tightly coupled with the mobile platform's SDK (i.e., the operating system and its libraries) and the hardware of the mobile device. Though managing multiple native graphical user interfaces is an overhead, therefore the scope of this work is limited to business logic generation. This limits the application of the proposed approach on extensive graphical applications, e.g., games.

Another limitation is that our approach requires understanding of feature model and UML models. Developing such models is not a trivial task. It can be argued that for applications that require large number of variants and need to be supported over time, effort of modeling is less than effort of manually maintaining different code versions. The development of application feature model FM_A (derived from generic feature model FM_G) and UML models are a one-time effort that pays off throughout the lifetime of the application. This however, requires further empirical investigation. In previous studies, model-driven development has been shown to be efficient in terms of cost and resources (Kapteijns et al., 2009; Heijstek and Chaudron, 2009).

9.4. Threats to validity

In addition to the above-mentioned limitations of our approach, threats to validity of our results are discussed in the following.

The proposed approach was successfully able to solve the problem of our industrial partner, i.e., the generation of feature-based mobile application variants for multiple platforms. One potential threat is the general applicability of the approach to other case studies. To reduce this threat, we selected two mobile applications from different domains: *Scramble* being a board game and *Instalapse* being a mobile-utility application. This allows us to demonstrate our approach to two vastly different applications belonging to different domains.

Another potential threat regarding the results of the case study is the impact of correctness of the generated code. To reduce this threat, we evaluated the variants on the available set of test cases of two applications to verify the business scenarios offered by the applications.

9.5. Open questions and future work

Our approach does not address generating the mobile application GUI. The native mobile application GUI's are tightly coupled with the underlying device and operating systems. Additionally, there is a significant aesthetics part involved in development of application GUI that is not easily automatable.

Possible research directions include investigating ways of automating generation of application GUI. A number of proposed user interface modeling languages, such as, IFML (OMG 2016), UML*i* (Da Silva and Paton, 2000), or model-based user interface modeling techniques (Cimino and Marcelloni, 2012; Botturi et al., 2013; Sabraoui et al., 2012) for mobile application user interface modeling and code generation may be leveraged. This will also allow us to generate the *Controller* class automatically for the interaction between the user interface layer and business logic layer; something our current approach lacks.

Another consideration is the use of an action language (for example, ALF (OMG 2013b)) for defining algorithmic details during UML state machine transition modeling rather than Java programming language. An action language is a high-level declarative language, independent of programming language syntax. It is easier to define algorithmic details in action language and later on transform them to the relevant platform specific programming language. However, this could potentially introduce another learning barrier to application development. The application designers will need to learn action language as well, in addition to UML.

Another possible direction is further evaluation of the approach on larger and more complex mobile applications.

10. Conclusion

Supporting variability is a key challenge for mobile application industry. Application variants are developed to support hardware devices, software platforms, and variations in functional requirements. The current industrial practice is to develop multiple native variants of an application separately, which is a very challenging task and poses significant overheads in terms of effort and cost. Any potential change or bug fix has to be applied across all variants manually.

In this paper, we address the problem of supporting variability by providing a 'product-line model-driven engineering approach' for automated generation of feature-based application variants for multiple mobile platforms. Our approach allows the developers to generate native mobile application variants covering three types of variations: variation due to operation systems and their versions, variations due to software and hardware capabilities of mobile devices, and variations based on the functional requirements of a mobile application. We provide a generic mobile application feature model (FM_G) as guideline for the identification of mobile domain features. Using the provided FM_G, the designer builds application specific product-line feature model (FM_A) that contains the required features, their dependencies, conflicts and variations. We use the application product-line feature model (FM_A) to generate mobile application product-line UML modeling profile. The product-line modeling profile is used to model application specific concepts in UML use-case, class, and state machine diagrams. The FM_A along with UML diagrams is configured to produce feature-based variants of a given mobile application. The MOPPET tool is developed to automate the proposed approach.

The approach is successfully applied to generate native variants for two industrial case studies: Scramble and Instalapse being developed by our industrial partner. The variants are successfully generated for Android and Windows Phone platforms and are also different in terms of their functionality. The results of applying the approach on real case studies show that automated generation of native mobile application variants is promising, has a potential to significantly reduce the development effort and cost, and can replace manual practice of developing and maintaining native variants. Despite the initial learning curve, there are significant long-term benefits to using our proposed automated approach.

References

Albassam, E., Gomaa, H., 2013. Applying software product lines to multiplatform video games. In: Proc. Proceedings of the 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change, pp. 1–7.

Android Apps Submitted Per Month (September 2015). Available: http://www.appbrain.com/stats/number-of-android-apps.

Android Pay (June 2016). Available: https://www.android.com/pay/.

Appcelerator, Titanium (2013). Available: http://www.appcelerator.com/titanium/.

Apple App Store 2015. Available: http://www.apple.com/osx/apps/app-store.html.

Apple, iOS platform (November 2013). Available: http://www.apple.com/ios/.

Apple, XCode (March 2016). Available: https://developer.apple.com/xcode/.

Apply Pay (2016). Available: http://www.apple.com/apple-pay/.

Batory, D., 2005. Feature models. Grammars, and Propositional Formulas. Springer.

Benouda, H., Azizi, M., Esbai, R., Moussaoui, M., 2016. MDA approach to automate code generation for mobile applications. Mobile and Wireless Technologies. Springer, pp. 241–250 2016.

Bittner, K., 2002. Use Case Modeling. Addison-Wesley Longman Publishing Co., Inc.

Booch, G., 2006. Object Oriented Analysis & Design with Application. Pearson Education India.

Botturi, G., Ebeid, E., Fummi, F., Quaglia, D., 2013. Model-driven design for the development of multi-platform smartphone applications. In: Proc. Specification & Design Languages (FDL), pp. 1–8. Forum on, 2013.
Brambilla, M., Cabot, J., Wimmer, M., 2012. Model-driven software engineering in practice. Synth. Lectures Softw. Eng. 1, 1–182.
Charland, A., Leroux, B., 2011. Mobile application development: web vs. Native. Commun. ACM 54, 49–53.
Cimino, M.G., Marcelloni, F., 2012. An efficient model-based methodology for developing device-independent mobile applications. J. Syst. Architect. 58, 286–304.
Cocos2d (2016). Available: http://cocos2d.org/.
Corona (2016). Available: http://www.coronalabs.com/.
Da Silva, P.P., Paton, N.W., 2000. UMLi: the unified modeling language for interactive applications. In: Proc. International Conference on the Unified Modeling Language, pp. 117–132.
Dagef, J.C., Reischmann, T., Majchrzak, T.A., Ernsting, J., 2016. Generating app product lines in a model-driven cross-platform development approach. In: Proc. 2016 49th Hawaii International Conference on System Sciences (HICSS), pp. 5803–5812.
Dalmasso, I., Datta, S.K., Bonnet, C., Nikaein, N., 2013. Survey, comparison and evaluation of cross platform mobile application development tools. In: Proc. Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International, pp. 323–328.
Dehlinger, J., Dixon, J., 2011. Mobile application software engineering: challenges and research directions. In: Proc. Workshop on Mobile Software Engineering.
Ernsting, J., Majchrzak, T.A., 2016. Towards a reference architecture for model-driven business apps. In: Proc. 2016 49th Hawaii International Conference on System Sciences (HICSS), pp. 5731–5740.
Facebook Hybrid App to Native App (May 2015). Available: http://venturebeat.com/2012/09/11/facebooks-zuckerberg-the-biggest-mistake-weve-made-as-a-company-is-betting-on-html5-over-native/.
Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: elements of reusable object-oriented Software: pearson education, 1994.
Gomaa, H., 2008. Model-based software design of real-time embedded systems. Int. J. Softw. Eng. 1, 19–41.
Google Play 2014. Available: https://play.google.com/store?hl=en.
Google, Android platform (May 2015). Available: http://www.android.com/.
Google, Android studio (2016). Available: https://developer.android.com/studio/index.html.
Heijstek, W., Chaudron, M.R., 2009. Empirical investigations of model Size, complexity and effort in a large Scale, distributed model driven development process. In: Proc. 2009 35th Euromicro Conference on Software Engineering and Advanced Applications, pp. 113–120.
Heitkötter, H., Hanschke, S., Majchrzak, T.A., 2012. Evaluating Cross-platform development approaches for mobile applications. In: Proc. International Conference on Web Information Systems and Technologies, pp. 120–138.
Heitkötter, H., Kuchen, H., Majchrzak, T.A., 2015. Extending a model-driven Cross–platform development approach for business apps. Sci Comput. Program 97, 31–36.
Heitkötter, H., Majchrzak, T.A., Kuchen, H., 2013. Cross-platform Model-driven development of mobile applications with MD 2. In: Proc. Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 526–533.
IBM, Rational software architect (RSA) v8.0.4 (2015). Available: https://www.ibm.com/developerworks/downloads/r/architect/.
Instalapse (October 2014). Available: https://www.amazon.com/AppMetrik-InstaLAPSE-Make-timelapse-go/dp/B0193U41ZM/.
Invotyx (2014). Available: http://invotyx.com/.
Iqbal, M.Z., Arcuri, A., Briand, L., 2015. Environment modeling and simulation for automated testing of soft real-time embedded software. Softw. Syst. Model. 14, 483.
Joorabchi, M.E., Mesbah, A., Kruchten, P., 2013. Real challenges in mobile app development. In: Proc. 7th International Symposium on Empirical Software Engineering and Measurement (ESEM). Baltimore, Maryland, USA.
Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., and Peterson, A.S., "Feature-oriented domain analysis (FODA) feasibility study," DTIC Document 1990.
Kapteijns, T., Jansen, S., Brinkkemper, S., Houët, H., and Barendse, R., "A comparative case study of model driven development vs traditional development: the tortoise or the hare," From code centric to model centric software engineering: Practices, Implications and ROI, vol. 22, 2009,
Ko, M., Seo, Y.-J., Min, B.-K., Kuk, S., Kim, H.S., 2012. Extending UML Meta-model for android application. In: Proc. Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on, pp. 669–674.
Kraemer, F., 2011. Engineering android applications based on UML activities. In: Whittle, J., Clark, T., Kühne, T. (Eds.), Model Driven Engineering Languages and Systems, 6981. Springer Berlin Heidelberg, pp. 183–197.
Kulak, D., Guiney, E., 2012. Use cases: Requirements in Context. Addison-Wesley.
Larman, C., 2004. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd ed. Prentice Hall 3rd Edition.
Lee, K., Kang, K.C., Lee, J., 2002. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. Software Reuse: Methods, Techniques, and Tools. Springer, pp. 62–77.
Majchrzak, T.A., Ernsting, J., Kuchen, H., 2015. Achieving business practicability of model-driven cross-platform apps. Open J. Inf. Syst. (OJIS) 2, 4–15.
Mellor, S.J., Balcer, M., Foreword By-Jacoboson, I., 2002. Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley Longman Publishing Co., Inc.
Microsoft, Windows Phone Apps Store 2015. Available: http://www.windowsphone.com/en-us/store.
Microsoft, Visual Studio Express 2013 for Windows 2013. Available: https://www.visualstudio.com/en-US/products/visual-studio-express-vs.
Min, B.-K., Ko, M., Seo, Y., Kuk, S., Kim, H.S., 2011. A UML metamodel for smart device application modeling based on windows phone 7 platform. In: Proc. TENCON 2011-2011 IEEE Region 10 Conference, pp. 201–205.
Mobile Application Revenue Generation (December 2013). Available: https://www.abiresearch.com/press/tablets-will-generate-35-of-this-years-25-billion-.
Muthig, D., John, I., Anastasopoulos, M., Forster, T., Dörr, J., Schmid, K., 2004. GoPhone - a Software product line in the mobile phone domain. IESE-Report No 25, 1–104.
Myllymäki, T., Koskimies, K., Mikkonen, T., 2002. On the structure of a software product line for mobile software. In: Software Infrastructures for Component-Based Applications on Consumer Devices (in conjunction with EDOC 2002). Lausanne, Switzerland, pp. 85–91.
Nascimento, L., Almeida, E., Meira, S., 2008b. Core Assets Development in Software Product Lines: Towards a Practical Approach for the Mobile Game Domain. Federal University of Pernambuco, Recife, Pernambuco, Brazil M. *Sc* dissertation.
Nascimento, L.M., de Almeida, E.S., de Lemos Meira, S.R., 2008a. A case study in software product lines - The Case of the mobile game domain. In: Proc. Software Engineering and Advanced Applications, 2008. SEAA'08. 34th Euromicro Conference, pp. 43–50.
O.M.G. (OMG), Object Constraint Language (OCL) v2.4 (2016). Available: http://www.omg.org/spec/OCL/.
Oehlman, D., Blanc, S., 2011. Pro Android Web Apps - Develop for Android using HTML5. CSS3, & Java script: Apress.
Ohrt, J., Turau, V., 2012. Cross-platform development tools for smartphone applications. Computer (Long Beach Calif) 45, 0072–0079.
OMG, Action language for fundamental UML (ALF) v1.0.1 (December 2013). Available: http://www.omg.org/spec/ALF/1.0.1/PDF/.
OMG, Unified Modeling language (UML), v2.4.1 (November 2013). Available: http://www.omg.org/spec/UML/2.4.1/.
OMG, The interaction flow modeling language (IFML) (August 2016). Available: http://www.ifml.org/http://www.omg.org/spec/IFML/.
Parada, A.G., d. Brisolara, L.B., 2012. A model driven approach for android applications development. In: Proc. Computing System Engineering (SBESC), 2012 Brazilian Symposium on, pp. 192–197.
Phone Gap (2015). Available: http://phonegap.com/.
Pohl, K., Böckle, G., Van Der Linden, F., 2005. Software Product Line Engineering: Foundations. Principles, and Techniques: Springer.
Polzer, A., Kowalewski, S., Botterweck, G., 2009. Applying software product line techniques in model-based embedded systems engineering. In: Proc. Model-Based Methodologies for Pervasive and Embedded Software, 2009. MOMPES'09. ICSE Workshop on, pp. 2–10.
Pure.systems, pure: variants (2015). Available: https://www.pure-systems.com/pure_variants.49.0.html.
Qt (2016). Available: http://www.qt.io/.
Quinton, C., Mosser, S., Parra, C., Duchien, L., 2011. Using multiple feature models to design applications for mobile phones. In: Proc. Proceedings of the 15th International Software Product Line Conference, 2, p. 23.
Raj, R., Tolety, S.B., 2012. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In: Proc. India Conference (INDICON), 2012 Annual IEEE, pp. 625–629.
Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.E., 1991. Object-Oriented Modeling and Design vol. 199. Prentice-hall Englewood Cliffs, NJ.
Sabraoui, A., Koutbi, M.E., Khriss, I., 2012. GUI code generation for android applications using a MDA approach. In: Proc. Complex Systems (ICCS), 2012 International Conference on, pp. 1–6.
Scramble (April 2014). Available: http://www.amazon.com/Scramble-Kindle-Tablet-HD-HDx/dp/B00GHKO98Q/ref=sr_1_3?s=mobile-apps&ie=UTF8&qid=1397135064&sr=1-3.
Sencha (July 2014). Available: https://www.sencha.com/.
Son, H.S., Kim, W.Y., Kim, R.Y.C., 2013. MOF based code generation method for android platform. Int. J. Softw. Eng. Appl. 7, 12.
SPLOT - Software Product Line Online Tools (2015). Available: www.splot-research.org.
Statista, Number of Apps Available In Leading App Stores (May 2015). Available: http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.
Thiel, S., Hein, A., 2002. Modelling and using product line variability in automotive systems. Software, IEEE 19, 66–72.
Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014. FeatureIDE: an extensible framework for Feature-oriented software development. Sci Comput. Program 79, 70–85.
Top 100 Apps Availability for iOS, Android, Windows Phone & Windows 8 2016. Available: http://www.infragistics.com/community/blogs/nick-landry/archive/2013/08/06/top-100-apps-availability-on-ios-android-windows-phone-amp-windows-8.aspx
Unity3D (2016). Available: http://unity3d.com/.
Usman, M., Iqbal, M.Z., Khan, M.U., 2014. A model-driven approach to generate mobile applications for multiple platforms. In: Proc. 21st Asia Pacific Software Enigneering Conference (APSEC). Jeju, Korea, pp. 111–118.
Usman, M., Nadeem, A., 2009. Automatic generation of java code from UML diagrams using UJECTOR. Int. J.Softw. Eng. Appl. 3, 21–37.

Vaupel, S., Taentzer, G., Harries, J.P., Stroh, R., Gerlach, R., Guckert, M., 2014. Model–Driven development of mobile applications allowing role-driven variants. Model-Driven Engineering Languages and Systems. Springer, pp. 1–17.
Wasserman, T., 2010. Software engineering issues for mobile application development. In: presented at the FSE/SDP workshop on Future of software engineering research, FoSER 2010.
Webber, D.L., Gomaa, H., 2004. Modeling variability in software product lines with the variation point model. Sci. Comput. Program 53, 305–331.
Xamarin (2016). Available: https://www.xamarin.com/.
Zhang, W., Jarzabek, S., Loughran, N., Rashid, A., 2003. Reengineering a PC-based system into the mobile device product line. In: Proc. Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of, pp. 149–160.

Muhammad Usman is currently doing his PhD in Computer Science from National University of Computer and Emerging Sciences (Fast-NU), Islamabad, Pakistan. He received his Master's degree in systems and software engineering from Mohammad Ali Jinnah University (M.A.J.U), Islamabad campus, Pakistan in 2009. He is also a research fellow at Software Quality Engineering and Testing (QUEST) Laboratory, Pakistan. His research interests include mobile software engineering, model-driven engineering, product-line engineering, and software testing.

Muhammad Zohaib Iqbal is currently an Associate Professor at the Department of Computer Science, National University of Computer & Emerging Sciences (Fast-NU), Islamabad, Pakistan and a research fellow at the Software Verification & Validation Lab, Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. He is also the lead scientist at Software Quality Engineering and Testing (QUEST) Laboratory and President of Pakistan Software Testing Board. He received his PhD degree in software engineering from University of Oslo, Norway in 2012. Before joining Fast-NU, he was a research fellow at Simula Research Laboratory, Norway. His research interests include model-driven engineering, mobile software engineering, software testing, and empirical software engineering. He has been involved in research projects in these areas since 2004.

Muhammad Uzair Khan is currently an Assistant Professor at the Department of Computer Science, National University of Computer & Emerging Sciences (Fast-NU), Islamabad, Pakistan. He is heading the Software Quality Engineering and Testing (QUEST) Laboratory, and is a founding member of Pakistan Software Testing Board. He completed his PhD research work at INRIA, France and received his PhD degree in computer science from University of Nice, France in 2011. His research interests include model-driven engineering, empirical software engineering, aspect-oriented software engineering, model refactoring and software testing. He is also the current Secretary/Treasurer of IEEE Islamabad Section.