Towards Architectural Styles for Android App Software Product Lines

Tobias Dürschmid, Matthias Trapp and Jürgen Döllner
Department of Digital Engineering
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
tobias.duerschmid@student.hpi.de, matthias.trapp@hpi.de, juergen.doellner@hpi.de

***Abstract*—Software product line development for Android apps is difficult due to an inflexible design of the Android framework. However, since mobile applications become more and more complex, increased code reuse and thus reduced time-to-market play an important role, which can be improved by software product lines. We propose five architectural styles for developing software product lines of Android apps: (1) activity extensions, (2) activity connectors, (3) dynamic preference entries, (4) decoupled definition of domain-specific behavior via configuration files, (5) feature model using Android resources. We demonstrate the benefits in an early case study using an image processing product line which enables more than 90% of code reuse.**

***Keywords*-Software Product Line; Android; Reuse; Image Processing**

I. Introduction

While mobile applications (apps) grow in popularity and complexity [1], [2], [3], the question of how to apply established software engineering principles, in particular software product lines (SPLs), to Android app development has not yet been discussed substantially [1].

SPL engineering in general, i.e., the managed process of creating product families, has been widely adopted in professional software development [4]. Practical benefits include large-scale productivity gains, mass customization, improved time-to-market, higher customer satisfaction, and higher development morale [5]. Therefore, SPL engineering can be considered as one of the major accelerators in the software development process.

Almost half of the professional mobile developers consider Android their primary platform and 80% of the mobile applications professionally target the Android platform [2]. The source code of Android apps is inherently complex [1]. Apps with higher complex code are rated higher by users, because of more features offered to the users [6], [3]. To manage variations of a feature set and tailor each concrete app to a different target group, Android developers demand mobile adaptions of SPLs [7], [8]. Furthermore, SPL engineering would encourage developers of mobile applications to focus on common requirements, design, and resources across different hardware [7]. Therefore, all benefits resulting from large-scale reuse consequently apply to the multi-billion dollar market of Android apps.

Current Android apps substantially apply software reuse concepts [9], [10]. However, on the current state of our knowledge, this reuse is not supported by architectural styles for app SPLs. This is a challenge, because the Android development framework lacks support for creating flexible apps with a large variety of variation points. Reasons include the following: (1) Android does not directly support a general reuse model, since Android fragments restrict the reuse to features that have a user interface (UI); (2) preference entries for settings offered to end-users are static for each project. Therefore, it is difficult to have a different set of preferences for each concrete app; (3) aspect-oriented programming for Android lacks tool support. Therefore, implementing a feature model for the SPL is difficult.

The main contributions of this paper are: (A) a proposal of five architectural styles [11], [12] which simplify large-grained reuse by targeting the challenges mentioned above. They can be used by developers to easily create Android app SPLs; (B) a case study of their utilisation in an image processing app SPL demonstrating a large amount of reuse. With respect to this, we discuss three concrete apps produced by the case study SPL: (1) an app for watercolor painting; (2) an app for pencil hatching; (3) an app for rapid prototyping of image stylization effects used for teaching image processing classes.

II. Background

This section briefly explains concepts of SPLs and Android development that are used in the remainder of the paper.

Software Product Lines: "A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific need of a particular market segment or mission and that are developed from a common set of core assets in prescribed way" [5]. SPL development requires a clear separation of generic code and product-specific code [13]. Therefore, an SPL comprises two main parts: (1) the *core assets* containing an architecture shared by the products in the SPL, common software components for systematic reuse, design documents, test cases, requirements specifications and domain models; (2) the *concrete products* containing the configurations and extensions for specific instantiations of the SPL architecture [5]. SPLs enable large-grained reuse [4], therefore modularity principles such as separation of concerns play an important role in SPL architectures [14]. According

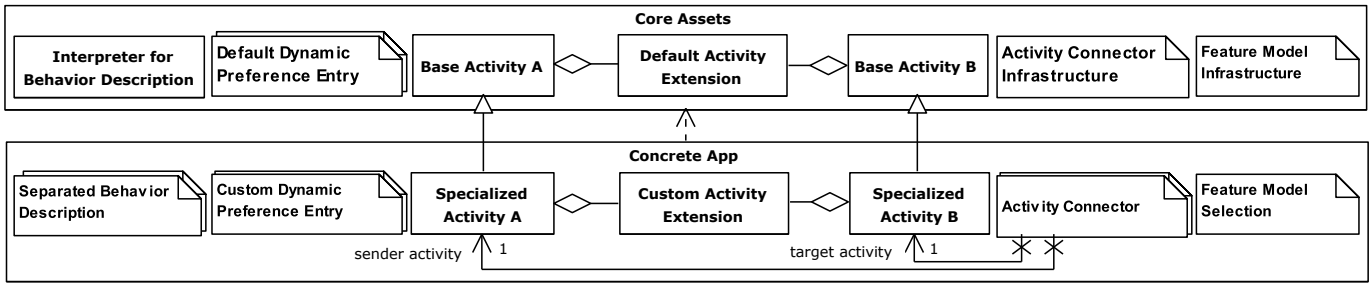

Fig. 1: Overview of the proposed architectural styles: (1) activity extensions, (2) activity connectors, (3) dynamic preference entries, (4) separated behavior description, and (5) feature model using Android resources. The infrastructure for the architectural styles is implemented inside the core assets. Concrete apps contain only the configuration and extensions of the variation points.

to Svahnberg and Bosch [13], architecture-based support for variability in SPLs are:

- V1: *Inheritance* is used when a feature needs to be implemented differently for a product or a product should extend a type defined in the core assets.
- V2: *Extensions and extension points* are used when parts can be extended with additional behavior. Design Patterns [15] are a common way to build extensions points into an SPL [13].
- V3: *Parametrization* is used when parameters can be referenced inside the core asset code and instantiated with a concrete value in the concrete products. A parameter has an initial value which can be adjusted by the concrete products.
- V4: *Configuration* is used to connect modules and components to each other. Configuration is the process of assembling a product by selecting and connecting modules and components out of the core assets.
- V5: *Generation* is used when a higher level language can be applied to define concrete component properties.

Android Development: The Android framework contains special concepts for app development: An *activity* is an Android window that is shown to the user and should be focused on one task a user can perform. Activities should be loosely coupled and should not contain business logic in order to separate the user interface from the domain code. An *intent* is an object wrapping an operation to be performed to start a new activity or service. It can contain data (e.g., an image) that is passed to the target activity or service.

III. Architectural Styles

An overview of the proposed architectural styles is shown in Fig. 1 and discussed in the remainder of this section. To simplify development of concrete apps, we propose to define the core assets of the SPL in one or more Android library modules used by the concrete apps, which are implemented using Android application modules. The proposed solutions delay the feature configuration to the start up time, because the configuration of Product Flavors, the Android mechanism for compile-time configuration, is not suited for SPLs. The reason is, that feature configuration with Product Flavors takes place in the Gradle file of the module that defines the feature, but should take place in the concrete app that uses the features to modularize the product configuration.

A. Activity Extensions

Problem: Android does not directly support a general reuse model, since Android fragments must have a UI. To reuse common functionality, that does not have its own UI, a reuse model is required that is tailored to activities.

Solution: An *activity extension* is an object that is responsible for a single feature offered to the end-user and can be associated to different activities. Usually, it is a variation of the Observer design pattern [15] receiving multiple events to respond to. Examples of transition events are the creation of the options menu in order to add menu items, destruction of the activity, or events specific to the domain of the SPL. Furthermore, the parent activity can request its extensions to perform actions such as triggering the visibility of their respective UI elements. This supports seamless integration of the extensions in the life cycle of the parent activity.

Consequences: This approach provides internal white box reuse [16] of the activity extensions. It enables extension variability (V2) for Android app SPLs. Furthermore, it improves separation of concerns, because each activity extension contains one single feature which is loosely coupled from the corresponding activities. However, they can introduce higher code complexity, because the resulting behavior can depend on the order of adding activity extensions.

Example: The SPL of the case study offers a button for choosing presets for the parameter configuration of the image processing effects. This feature is modularized in an activity extension to add it to multiple activities. Furthermore, the functionality for on-screen painting, the menu entry for opening the settings dialog, for printing the current image, and for social media communication are activity extensions.

B. Activity Connectors

Problem: To provide a custom work flow for concrete apps, the SPL should support refinements of the intents connecting activities. This means that the SPL should enable the substitution of base activities with specialized activities or calls to external services (e.g., the Android gallery, the camera or other apps).

Solution: An *activity connector* is a transition in the activity state machine starting in a *sender activity*, triggered by a *transition event*, and resulting in a *target activity*. Activity connectors are implemented using a dictionary that maps the pair of sender activity and transition event to an intent that starts a target activity. By overriding existing transitions or adding new ones, concrete apps can manipulate the behavior defined in the core assets.

Consequences: This concept enables configuration variability (V4) of Android app SPLs. It decouples the activity classes, because no activity class directly depends on any other activity. The centralization of all activity transitions results in improved separation of concerns and enhanced readability of the activity code. However, the code can become more complex because of additional infrastructure of the activity state machine.

Example: Our case study SPL uses activity connectors for all transitions between activities in three levels of abstraction of variation points: (1) a hook method returning the concrete activity to start for each conceptional activity, that can be overridden by concrete apps in order to exchange a complete activity without changing the transition logic; (2) a hook method for each standard transition of the SPL getting context information as input and returning an intent as output; and (3) providing the ability to add new transitions to the state machine.

C. Dynamic Preference Entries

Problem: In Android, user-visible settings are statically defined in an XML (eXtensible Markup Language) file which does not provide a visibility flag. Therefore, all apps in the SPL would have the same preference entries. In order to provide variability of preference entries, an Android app SPL should support custom preference entries for each concrete app.

Solution: We propose to define *preference record* objects each representing one end-user visible preference entry. Each consists of an action to be performed when the value changes, a default value, a preference fragment for presentation, and a wrapped Android preference object. This is similar to the *Active Record* [17] design pattern but using Android shared preferences instead of databases. Concrete apps can register custom preference records using a Singleton [15] maintaining a list of preference records. Each entry in the list is shown in the preference activity and loaded from shared preferences during start-up.

Consequences: Dynamic preferences facilitate the production of apps for different target groups. This approach enables extension variability (V2) by defining which preference entries should be shown for each concrete app. However, this can slightly impact the start-up time, if a large number of preference entries are defined in the SPL.

Example: The SPL of the case study offers preference entries for image processing experts, for developers, and for novice use, which can be chosen during the implementation of a concrete app.

D. Separated Behavior Description

Problem: Many apps share a common domain or business process while each app should be tailored to the needs of a different target group.

Solution: We propose to inject the behavior and the content of the concrete apps by separately defining it using a high-level configuration language. An interpreter reading the files generates the domain objects and injects them into the SPL.

Consequences: This concept enables generation variability (V5) for Android app SPLs and provides a loose coupling of the core assets and the concrete product. Therefore, it facilitates an SPL with large variability and many possible products by simplifying the construction of new apps. However, the implementation and maintenance of the XML interpreter is additional effort. If the SPLs is used to generate a large amount of apps, this efforts pays off due to fast implementation using an high-level XML representation. However, backward compatibility for older XML files after the evolution of the XML schema needs to be considered in SPL engineering.

Example: Our case study SPL uses an XML effect file to define the image processing pipeline [18] for an effect (e.g. watercolor, pencil hatching) with user-definable parameters that are generated from the effect file and automatically presented in the UI. Furthermore, XML files are used to generate an OnBoarding activity for each app.

E. Feature Model using Android Resources

Problem: SPLs usually provide a feature model defining which features are contained in each concrete product. However, product flavors, the static configuration mechanisms offered by Android, do not support modularization of the feature selection and feature configuration.

Solution: To provide a feature model for Android apps, we propose to define a configuration file in the Android resources of the core asset module. This resource file contains variation points and default values for each of them. These values can be referenced inside the core assets. To override values in a concrete app, developers can redefine them in the configuration resource file within concrete apps modules. Therefore, Android uses the configuration of the concrete app, if defined, and the default values defined in the library otherwise.

Consequences: This approach provides internal black box reuse [16] and enables parametrization variability (V3) of the concrete apps. It supports coarse-grained activation of features [19].

Example: Our case study SPL uses the configuration XML file to activate/deactivate activity extensions for the corresponding activities. Furthermore, the definition which preferences entries are shown in each concrete app is implemented using Boolean flags within the configuration file.

IV. Preliminary Evaluation

We evaluate the presented approach by applying it to the case study SPL, which currently comprises three image

processing apps derived from the framework described in [20]. Large-grained reuse is one of the core advantages of SPLs, therefore we evaluated the case study by measuring the amount of reuse [16]. Furthermore, we describe how the architectural styles support variability in order to demonstrate that they offer enough variation points for practical development.

Amount of Reuse: At total, this SPL comprises 61,723 lines of Java code (excluding comments, generated code, and XML files) in 524 classes. The reason, why this project is much larger than usual Android apps, which consist of 5,600 lines of code (LOC) on average [1], is that currently GPU-supported image processing is a niche in the Android market. 511 classes (60,482 LOC) are part of the SPL core assets. Hence, the internal reuse factor for the Java code in the whole SPL is 98%. XML files are not considered and can not be compared with Java code because of differences in structure and complexity. The domain-specific XML files for defining the effects (e.g., watercolor effect & pencil hatching effect) comprise almost 1,000 lines for complex effects. However, even if the XML files for both effects are nominally added to the overall LOC, the amount of reuse would still remain at 95%. Furthermore, this case study contains only one SPL and therefore not necessarily provides information on the application of the architectural styles to other domains.

Variability: The domain-specific XML files containing effect specifications support state-of-the-art image stylization effects such as pencil hatching, watercolor, oil painting, cartoon, bloom, and others [20]. Hence, the resulting apps can vary in domain-functionality such as the visual effect, its parameters and presets by changing the effect file. Some of our concrete apps are tailored to novice users by providing a simple and discoverable UI using a painting metaphore while the prototype app is tailored to image processing experts by offering many preferences and more advanced functionality for rapid prototyping of new image processing effects strongly varying in the presented UI. The activity extensions enabling the users to share the results across several social platforms can be deactivated for user studies to focus on the main functionality. The extension for image deformation that is used for artistic purpose (e.g., cartoon looks) can be deactivated for other apps.

V. Related Work

This section briefly distinguishes the proposed architectural styles from similar research.

SPLs for Mobile Devices

Previous SPLs have been developed for mobile platforms, e.g., role playing games [21], mobile browsers [22], and data collection apps [23]. These approaches are mainly focused on case studies only, but do not discuss abstract concepts of wider applicability. However, not much research has been performed on how to tailor SPL architectures to mobile platforms. Several papers on SPLs for mobile devices focus on the heterogeneity of mobile devices [24], [25], [26], [27]. They provide variability across different hardware. In addition to this, our architectural styles provide variability of domain features by dealing with the lack of SPL support by frameworks for mobiles platforms and helping developers in creating an extensible architecture tailored to Android.

Architectural Styles and Design Patterns

Since the introduction of design patterns [28], [15] and architectural styles [11], [12], a variety of domain-specific patterns have been written down, e.g., in the field of concurrency [29], distributed systems [30], and resource management [31]. However, to the current state of our knowledge, no design patterns or architectural styles have been introduced or discussed for mobile SPLs.

VI. Discussion

Modularity: The architectural styles provide separation of concerns (e.g., by keeping together the code for preferences, for one feature modularized in an activity extension or the domain-specific use case encoded in an XML file).

Feature configuration during run time: Our SPL architectures dynamic preference entries and feature model configurations are loaded during run time which increases the size of the executable, because Android resources are evaluated during run time. Hence, all features and alternatives of the core assets library are compiled into the executable Android package (APK). This increases the download time of the apps.

Software Complexity: Similar to other SPL architectures, the proposed styles can lead to a more complex architecture than single concrete apps would have. The flexibility that is essential for a configurable architecture often results in more abstract code which can raise challenges in dealing with temporal dependencies.

VII. Conclusion and Future Work

We have proposed five architectural styles for creating SPLs with the Android framework and contributed to the question how to adopt SPLs for mobile applications [7]. Our concept used all five kinds of variability postulated by Svahnberg and Bosch [13]: *inheritance* via sub-classing conceptional activities, *extensions* via activity extensions and dynamic preference entries, *parametrization* via the feature model using Android resources, *configuration* via activity connectors and *generation* via parsing domain-specific XML files. Our case study shows that established software principles for SPLs can be adjusted to mobile application development. This is one next step towards the general question how to tailor software engineering principles to mobile applications.

To avoid feature configuration during run time, further research can be performed with respect to compile time configuration of concrete apps according to feature-oriented programming [32], [33] for Android apps.

VIII. Acknowledgments

This work was partially founded by the German Federal Ministry of Education and Research (BMBF) in the InnoProfile Transfer group ”4DnDVis”.

REFERENCES

[1] R. Minelli and M. Lanza, "Software analytics for mobile applications–insights & lessons learned," in *European Conference on Software Maintenance and Reengineering (CS '13)*. IEEE, 2013, pp. 144–153.

[2] B. Ray, M. Wilcox, and C. Woskoglou, "Developer Economics - State of the Developer Nation Q3 2016," Vision Mobile, London, Tech. Rep., July 2015.

[3] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? a case study on free android applications," in *International Conference on Software Maintenance and Evolution (ICSME '15)*. IEEE, 2015, pp. 301–310.

[4] L. M. Northrop, "SEI's software product line tenets," *IEEE software*, vol. 19, no. 4, p. 32, 2002.

[5] P. Clements and L. Northrop, *Software product lines*. Addison-Wesley, 2002.

[6] L. Guerrouj and O. Baysal, "Investigating the android apps' success: An empirical study," in *International Conference on Program Comprehension (ICPC '16)*. IEEE, 2016, pp. 1–4.

[7] J. Dehlinger and J. Dixon, "Mobile application software engineering: Challenges and research directions," in *Workshop on Mobile Software Engineering (MobiCASE '11)*, vol. 2, 2011, pp. 2–2.

[8] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan, "Understanding reuse in the android market," in *International Conference on Program Comprehension (ICPC '12)*. IEEE, 2012, pp. 113–122.

[9] L. Li, J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Mining families of android applications for extractive SPL adoption," in *International Systems and Software Product Line Conference (SP '16)*. ACM, 2016, pp. 271–275.

[10] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large-scale empirical study on software reuse in mobile apps," *IEEE software*, vol. 31, no. 2, pp. 78–86, 2014.

[11] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[12] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice Hall Englewood Cliffs, 1996.

[13] M. Svahnberg and J. Bosch, "Issues concerning variability in software product lines," in *International Workshop on Software Architectures for Product Families (IW-SAPF '00)*. Springer, 2000, pp. 146–157.

[14] J. Van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*, 2001, pp. 45–54.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] W. Frakes and C. Terry, "Software reuse: metrics and models," *ACM Computing Surveys (CSUR '96)*, vol. 28, no. 2, pp. 415–435, 1996.

[17] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley, 2002.

[18] R. Vergne and P. Barla, "Designing gratin, a gpu-tailored node-based system," *Journal of Computer Graphics Techniques*, vol. 4, no. 4, p. 17, 2015.

[19] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *International Conference on Software Engineering (ICSE '08)*. ACM, 2008, pp. 311–320.

[20] A. Semmo, T. Dürschmid, M. Trapp, M. Klingbeil, J. Döllner, and S. Pasewaldt, "Interactive Image Filtering with Multiple Levels-of-control on Mobile Devices," in *SIGGRAPH ASIA Mobile Graphics and Interactive Applications (MGIA '16)*. ACM, 2016, pp. 2:1–2:8.

[21] W. Zhang and S. Jarzabek, "Reuse without compromising performance: industrial experience from rpg software product line for mobile devices," in *International Conference on Software Product Lines (SPLC '05)*. Springer, 2005, pp. 57–69.

[22] A. Jaaksi, "Developing mobile browsers in a product line," *IEEE software*, vol. 19, no. 4, p. 73, 2002.

[23] G. M. Waku, E. R. Bollis, C. M. Rubira, and R. d. S. Torres, "A robust software product line architecture for data collection in android platform," in *Brazilian Symposium on Components, Architectures and Reuse Software (SBCARS '15)*. IEEE, 2015, pp. 31–39.

[24] V. Alves *et al.*, "Identifying variations in mobile devices." *Journal of Object technology*, vol. 4, no. 3, pp. 47–52, 2004.

[25] D. Muthig, I. John, M. Anastasopoulos, T. Forster, J. Dörr, and K. Schmid, "Gophone-a software product line in the mobile phone domain," *IESE-Report No*, vol. 25, 2004.

[26] R. E. V. Rosa and V. F. Lucena, Jr., "Smart composition of reusable software components in mobile application product lines," in *International Workshop on Product Line Approaches in Software Engineering (PLEASE '11)*. ACM, 2011, pp. 45–49.

[27] J. White and D. C. Schmidt, "Model-driven product-line architectures for mobile devices," *International Federation of Automatic Control (IFAC '08)*, vol. 41, no. 2, pp. 9296–9301, 2008.

[28] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture, A system of patterns*. Wiley New York, 1996.

[29] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013, vol. 2.

[30] F. Buschmann, K. Henny, and D. C. Schmidt, *Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing*. John Wiley & Sons, 2007, vol. 4.

[31] M. Kircher and P. Jain, *Pattern-Oriented Software Architecture, Patterns for Resource Management*. John Wiley & Sons, 2013, vol. 3.

[32] S. Apel and C. Kästner, "An overview of feature-oriented software development." *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[33] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *European Conference on Object-Oriented Programming (ECOOP 97)*. Springer, 1997, pp. 419–443.