A Mobile Software Development Using Aspect Orientation Approach

Paniti Netinant, Rangsit University, Thailand

The Asian Conference on Society, Education & Technology 2016
Official Conference Proceedings

Abstract
Today mobile devices are widely used everywhere, and have rapidly changed. One of the major characteristics of the mobile software is to continuous need and demand for faster development. Separation of concerns in the mobile software development is essential for adaptability and extensibility. An adaptability is a capable to adapt with respect to the environment that will need to perform. An extensibility is a capable to extend with respect to the new features or requirements that will add in a mobile software. Most software deficiencies and deteriorations are caused by changes in software. Generally, these deviations cannot be avoided. Such changes are often the result of the mobile software evolution and changes in the underlying requirements of mobile software development to meet these evolving needs. Certain refinements can be applied to traditional object-oriented analysis and design techniques. However, refinements are very complicate. A simplicity of software development is considered an important characteristic of a good software development model. Mobile software engineering approaches are a midlife with many accomplishments already achieved, but with many significant works yet to do. An aspect orientation approach have shown to be an effective means of capturing, communicating, and combining software components. We believe that an aspect orientation approach can be applied to a mobile software development on several aspects. To demonstrate the simplicity and practical of the adaptable and extensible mobile software model, we propose a separation of concerns in a mobile software development using an aspect orientation approach.

Keywords: Aspect Orientation, Adaptability, Extensibility, Framework, and System Software.

iafor
The International Academic Forum
www.iafor.org

Introduction

The recent expansion of smart devices has abundantly created a unique opportunity for researchers to use all their capabilities to provide new application software. Mobile application software development is rapidly changing the way we have commonly worked and interacted. Mobile software development has to comprehend how separation of concerns can be achieved and how individuals choose to properly develop mobile software to effectively utilize a separation of concerns. At present there are more than hundred thousand of application software available through the various stores, some of which are available for multilingual and multiple types of devices. Most of mobile application software divide between native and web applications (Ali, N., & Ramos, I. 2012; Wasserman, 2010). Native applications run entirely on the mobile device. Web applications consist of a remote server and a small device-based client executing and interacting user's commands through communication networks. There are several of comprehensive mobile application design and development available for the major mobile platforms. IPhone developers use Xcode package across all Apple products (Apple Developer Connection, 2015). Android developers uses the Android development tools (Android Developer site, 2015) or eclipse programming tools (Eclipse website, 2016). Windows phone developers use Microsoft Studio for mobile development (Windows Phone Developer site, 2016). These dominant development gears and structures greatly simplify the task of design and implementation of a mobile application software. However, they are based on object-oriented design and implementation. The intra-concern system properties are associations and necessities over confined state of processes or components of states. The inter-concern system properties are associations and necessities over dissimilar confined processes or components of states that describe the reliabilities and collaboration among a collection of supportive processes or components. Both processes and components of system properties are critical for a system development and verification. The intra-concern properties are relatively easier to express and carry on through a system development life cycle. One of the major characteristics of the system software is to continuous need and demand for faster adaptability and extensibility. Adaptable system software is system software that can be adapted with respect to the environment that will need to perform. Extensible system software is system software that can be extended with respect to the new features or requirements that will add in system software. Most software defects and deterioration are caused by changes in software (Fayad, M. & Altman, A., 2001). Generally, these changes cannot be avoided. Such changes are often the result of the system software evolution and changes in the underlying requirements of system software to meet these evolving needs. Certain refinements can be applied to traditional object-oriented analysis and design techniques. However, such refinements must not complicate. Simplicity is considered an important characteristic of a good model.

A mobile software consists of separating multiple concerns crosscutting many components of the system. A mobile software is notorious of many crosscutting concerns such as synchronization, scheduling, fault tolerance, logging, and etc. We refer to these crosscutting concerns as system properties. System properties are aspectual. By supporting separation of concerns in the system software, we can provide a number of benefits such as easy to comprehension, reusability, extensibility, and adaptability for system software. In both the design and implementation of system

software, the system designer has to consider how a number of system properties can be captured, and how a separation of concerns (Parnas, D., 1972) will be addressed. Functional decomposition has so far been used as well as achieved along two dimensions - based on the components and layering paradigm. In object-oriented programming, these dimensions are layers and components; included methods, objects and classes. Current programming languages and techniques have been supportive to functional and object-oriented decomposition. However, languages are specific domain. Furthermore; a mobile software design has also been aligned with traditional functional decomposition techniques. No functional decomposition technique has yet managed to address a complete separation of concerns. Object-oriented programming seems to work well only if the problem can be described with relatively simple interfaces among objects. Unfortunately, this is not the case when we move from sequential programming to concurrent and distributed programming. As distributed systems become larger, the interaction of their components is becoming more complex. This interaction may limit reuse, make it difficult to validate the design and correctness of system software, and thus force reengineering of these systems either to meet new requirements or to improve the system. Certain system properties of the mobile software do not localize well. They tend to crosscut groups of components or services (functions or methods) in the system. System properties tangle in components or services making the system difficult to adapt and extend. Changing needs to understand and correctly identify both system properties and core services of components. It is tightly couple design and implementation between components and system properties. In this paper we focus on adaptability and extensibility by proposing an adaptable and extensible model that is the basic of a framework for the system software development. This adaptable and extensible model, using the aspect-oriented techniques (Kiczales, G., Lamping, J., et al 1997; Lopes, C., Tekinerdogan, B., et al, 1998) provides a declarative way of developing, handling, and characterizing adaptable and extensible system software and represents a novel attempt to decompose and compose the system properties and components.

The Architecture

In this section we briefly describe the framework (Netinant 2006; Netinant, 2001). We have designed the framework in order to support the development and deployment of adaptable and extensible system software. Figure 1 shows the overall framework architecture, which is composed of the following components:

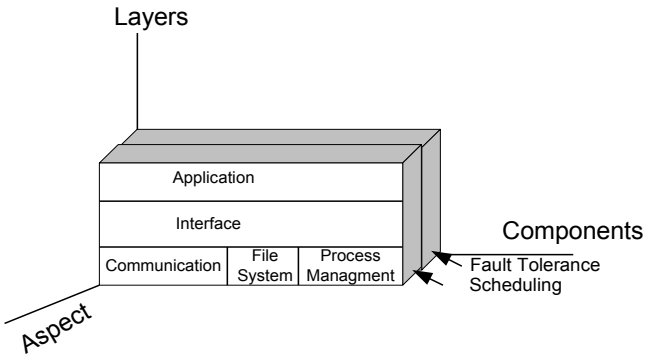

Fig. 1 The Framework Architecture

Our framework is based on aspect-orientation, which is a three-dimensional system design consisting of components, aspects, and layers. Components consist of the modules that provide the basic functionality of the system such as the file system, communication, and process management, etc. Aspects are crosscutting system properties, and they can be a fault tolerance, synchronization, and scheduling, naming, etc. Layers consist of the components and system properties. In general, lower layers deal with a far shorter time scale. The lower the layer, the closer it is to the hardware. The higher layer deals with interaction with the user.

By adding the aspect dimension to a two-dimensional model, system properties and functional components are separated from each other in every layer. It makes the system software design and implementation more modular, but makes it loosely coupled. Each layer has well-defined functionalities, system properties, and input-output interfaces with the two adjacent layers. Each layer can be designed, implemented, and tested independently. The upper layer can reuse the layer beneath without knowing how the lower aspects or components are implemented. The upper layer does not have to build own system property components from scratch. However, new aspectual property components can be added to a layer without interfering with system property components or functional components in the layer underneath. It gives the system software easier extensibility and adaptability. Adding new system property components, which are orthogonal, requires no changes in functional components or system property components in other layers. Modifying a system property component needs no changes in system property components in the other layers. With current growth and rapid change, in technology and the features of system software, this architecture allows both functional components and system property components to be added into the system software more easily. The three-dimensional model makes it possible to manage both system property components and functional components in each layer.

By isolating the different system property of each component, we can separate functional components, system properties, and layers from each other (components from each other, system properties from each other, layers from each other, functional components from system properties, functional components in each layer, and system property in each layer). It would thus be possible to abstract and compose them to produce the overall system. This would result in the clarification of interaction and increased understanding of system properties of each functional component in the system. A high level of abstraction is easier to understand. Further, the reusability achieved by the higher level can use the lower level of the implementation not only to promote extensibility and refinement, but also to reduce cost and time in system development. A change in the implementation at a lower level would not result in a change at the higher level if the interface level has not been changed. Thus the design can achieve stability, consistency, and separation of concerns as well. A system property may have multiple domains. Some system properties (scheduling, synchronization, naming, and fault tolerance, e.g.) are scattered among many components in the system with varying policies, different mechanisms, and possibly under different applications. To reduce the tangling of system properties in system software each system property can be considered and analyzed separately. For example, a system property of scheduling in file systems can be considered in different domains in each layer. This would separate policy from a system property of each layer. A system property interface would represent the general specifications needed to

provide the abstraction. Further, a policy can be added or modified in each layer for each specific domain. This approach can support reusability to achieve adaptability.

The Framework

One way of structuring system software is to decompose it into layers. Each layer is decomposed into its components. This decomposition of the system design both horizontally and vertically helps to deal with the complexity and reusability of system software. The layered architectural design decomposes a system into a set of horizontal layers where each layer provides an additional level of abstraction over the next lower layer and provides an interface for using the abstraction it represents to a higher-level layer. Every layer is decomposed into system components and system properties. System components and system properties are separated from each other. Changing either system components or system aspectual properties does not affect the other. The advantage of this decomposition is that system software tends to be easy to understand, adapt, extend, and maintain. Each layer can be understood, adapt, extend, and maintained individually without affecting other layers. However, it may be bad for performance and traceability because of using lower layer components.

The framework expresses a fundamental paradigm for structuring system software, a vertical composition of each layer where system components and system aspectual properties are composed into an abstraction of the layer. The framework structure can be described by the design pattern (Gamma, E., Helm, R., Johnson, R., & Vlissides, J., 1995). The framework uses a client-server model in which the server components (Functional Components and System Components) are composed by the Aspect Moderator and make their services available to clients. Clients access the server component services by sending requests to the Proxy component. The Proxy component intercepts a requesting message from clients and forwards the message to the Aspect Moderator component. The Aspect Moderator component locates and instantiates the composition rules defined by *pointcut(s)* – a collection of join points consists of join points between functional components and system property components. Figure 2 illustrates the modeling of the adaptable and extensible framework.

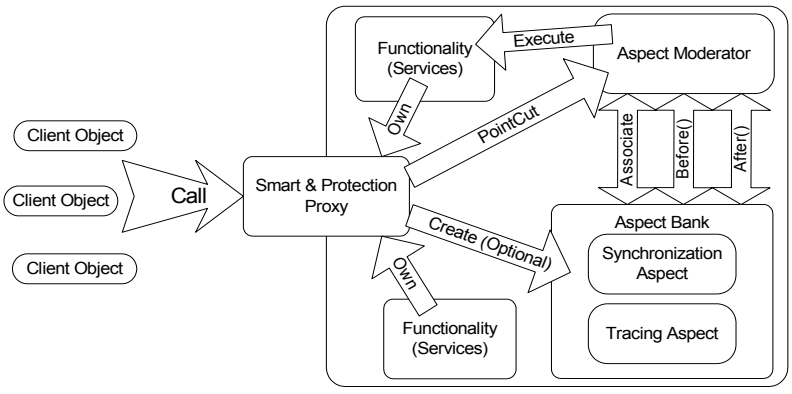

Fig. 2 The Model of the Adaptable and Extensible Framework

The framework supports both vertical and horizontal reusability. Reusable assets in the framework can be found in the vertical composition, where the upper neighbor layer can reuse a functional component or an aspectual property component from the lower

layer. There are two levels of reuse in the aspect-oriented framework: Inter-layer reuse: reuse of functional components or aspectual property components from the lower layer, such as using an aspectual component derived from an abstract aspect. Intra-layer reuse: reuse of functional components or aspectual property components from the same layer, such as using an aspectual component to solve another problem. The aspect-oriented framework provides a better way to reuse both design and implementation code. Both inter and intra-layer reuse can be divided into three levels of reuse in the aspect-oriented framework as follows:

Functional component: Reuse of functional component(s), such as reuse or redefinition of the functional component.
System property component: Reuse of an aspectual property component, such as reuse or redefinition of the aspectual property component.
Framework reuse: Reuse of a framework provides a set of classes that manifest as an abstract design and implementation for solutions to a set of related problem.

The aspect-oriented framework supports both vertical and horizontal compositions. Functional and aspectual property components in the framework can be composed vertically or horizontally. In vertical composition, the upper layer can use the lower functional or aspectual property components from the lower layer. In horizontal composition, functional and aspectual property components in the particular layer only use to be composed. The framework is based on system aspectual decomposition of crosscutting concerns in operating system design and implementation. The framework consists of two frameworks: the Base Layer and the Application Layer Framework. A system aspectual property is implemented in the SystemAspect class, while a component of the system is implemented as a Component class. The framework uses PointCut, Precondition, and Advice. The AspectModerator class, where the point cut is defined, combines both system aspectual properties and components together at runtime. Pointcuts are defined collections of join points, where system aspectual properties will be altered and executed in the program flow. Every aspectual property can identify and implement preconditions. A precondition is defined a set of conditions or requirements that must hold in order that an aspect may be executed. Advice is a defined collection of methods for each aspectual property that should be executed at join points. Advice can be either before or after advice. Before advice can be implemented as blocking or non-blocking. Before advice is executed when the join point is reached, before the component is executed, if the precondition holds. After advice is executed after the component at the join point is executed. Every aspectual property will define advice methods.

One important aspect of the framework is that it can separate functional components and system property components (system and application depending on the layer of a framework). A client object calls the services from the servers through a proxy object, rather than having services called directly from a client object. Then the framework creates necessary objects and calls the appropriate system properties to perform a specific service. In other words, the framework is like a mixer that combines and coordinates the crosscutting concerns of a specific service. The rules are used to combine and coordinate a functional component (a service of the system) defined by the pointcuts. Thus, for a particular system or application, it can adapt the generic functional components or system property components defined in the framework. The framework supports adaptability and extensibility in either of two ways:

Extensibility: Derive new components from the framework: They can be either functional components or system property components.
Adaptability: Instantiate and compose existing classes: They can be either functional components or system property components.

Implementation of the Framework

The framework consists of four components comprising the architecture of the framework. Each functional object (component) provides its services (methods) stripped of any aspectual properties (for example, no synchronization is included in Buffer objects).

A proxy object intercepts called methods and transfers the calls to the AspectModerator.
An AspectModerator object consists of the rules and strategies needed to bind aspects at runtime. Aspects are selected from the AspectBank. The AspectModerator orders the execution of aspects. The order of execution can be static or dynamic. Then, each precondition will be checked whether it is satisfied or not.
An AspectBank object consists of aspect objects that implement different policies of a variety of aspects.

This section presents the design and development of aspect-oriented framework. The model is presented to demonstrate horizontal composition of the framework. The system service must be implemented as a Component class. The system aspectual property (SystemAspect class) must be derived from the SystemAbstractAspect interface to implement the required behavior of a system aspectual property. A SystemAspectFactory consists of many system aspectual properties such as synchronization, tracing, logging, and reliability. The SystemAspectFactory, derived from the SystemAbstractAspectFactory interface, is known as an aspect bank. During runtime, each SystemAspectFactory will be associated with one SystemAspect. The AspectModerator class must be derived from the AspectModerator interface to implement the required behavior. The following points are important about the aspect-oriented framework:

A client object requests a service through a ProxyObject object of a framework.
A functional component is implemented as a Component class without any aspectual property.
A SystemAspectFactory object consists of various SystemAspect objects. A SystemAspect object is controlled by a SystemAspectFactory object.
Each system aspectual property must be implemented as a SystemAspect object.
Each crosscutting between Component object and a SystemAspect object must be defined in AspectModerator object as joinpoints in a Pointcut method.
A client requests a service by sending a message to a ProxyObject object. The ProxyObject object changes the request to a specific pointcut method, and forwards it to the AspectModerator object.

The Proxy class is responsible for intercepting and forwarding the message sent from Client object to request a service. The Proxy class must implement the behavior of intercepting a service request. A client object of an aspect-oriented framework must

request a service by calling the call() method. A call() method consists of at least two parameters: object name provided a service and a service requested to serve. The first parameter is of type string, and the second is type of string as well. The ProxyObject class will forward a request to the AspectModerator object by calling a PointCut() method. A PointCut() method must have the same number parameters and the same parameter type as the call() method.

The SystemAspectFactor class must be derived from the SystemAspectFactoryAbstract interface to implement the required behavior. The AspectModerator class is responsible for composing the functional components and the system aspectual property into a service request. The AspectModerator class acts like a coordinator between functional components and system aspectual properties, when and where system aspectual properties will be composed into a functional component. The composition of system aspectual properties and functional components must be guided and defined as PointCut() method. Each PointCut() method must have at least two parameters: component name and service name (methods of the component) that will be composed. The first parameter is of type string, and the second is type of string as well.

Conclusion

In this paper, we stressed the importance of the better separation of concerns within the context of an adaptable and extensible framework. We show how this technique could provide an alternative to system software design and implementation, and show how our approach can be achieved separation of crosscutting concerns of the system. Our work concentrates on the decomposition of system properties crosscutting functional components in the systems implementation of system software to separate the crosscutting concerns.

Our design framework provides an adaptable model that allows for open languages and our goal is to achieve a better design and architectures where new system property and components can be easily manageable and added without invasive changes or modifications. The framework approach is promising, as it seems to be able to address a large number of system software and system property components. The advantage of decomposing of functional components and system property in every layer is to promote reusability, adaptability, manageability, and extensibility of both components and system property in system software easier without interfering each other. In the future, the framework will be extended and demonstrated for distributed object environment.

References

Ali, N., & Ramos, I. (2012). Designing mobile aspect-oriented software architectures with ambient. USA: IGI Global. DOI: 10.4018/978-1-61520-655-1.ch029

Apple Developer Connection. (2016). https://developer.apple.com/resources Accessed on March 15, 2016.

Android Developer site. (2016). http://developer.android.com Accessed on March 15, 2016.

Eclipse website. (2016). http://www.eclipse.org Accessed on March 15, 2016.

Fayad, M. & Altman, A., (2001) An introduction to software stability, *Communications of ACM*, 44(9), 95-98.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J., (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Massachusetts: Addison-Wesley.

Kiczales, G., Lamping, J., Mendhekar, J.,Maeda, C., Lopes, C., Loingtier, J., & Irwin, J., (1997) Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors. Proceedings of the 11th European Conference on Object-Oriented Programming, *Lecture Notes in Computer Science number 1241*, Springer Verlag, Berlin, 220-242.

Lopes, C., Tekinerdogan, B., Meuter, W., & Kiczales, C., (1998) Aspect-Oriented Programming. In M. Aksit and S.Matsuoka, editors, *Proceedings of the 12th European Conference on Object-Oriented Programming ECOOP'98*, Springer Verlag.

Parnas, D., (1972) On the Criteria to be Used in Decomposing Systems into Modules, *Communications of ACM*, 15(12), 1053-1058.

Netinant, P., (2006) Extensibility Aspect-Oriented Framework to Build Agent-Based System Software, *Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE 2006)*, Los Angeles, California, USA.

Netinant, P., Elrad, T., & Fayad, M., (2001) A Layered Approach to Building Open Aspect-Oriented Systems, *Communications of ACM*, 44(10), 83-85.

Wasserman, I. A., (2010). Software engineering issues for mobile application development. *Proceedings of 2010 Future of Software Engineering Research*, Santa Fe, New Mexico, USA, pp. 397-400. DOI: 10.1145/1882362.1882443