Evaluating an Aspect-Oriented Approach for Production-Testing Software

Jani Pesonen
Nokia Corporation
Tieteenkatu 1
Tampere, Finland
jani.p.pesonen@nokia.com

Mika Katara
Tampere University of Technology
Korkeakoulunkatu 1
Tampere, Finland
mika.katara@tut.fi

Tommi Mikkonen
Tampere University of Technology
Korkeakoulunkatu 1
Tampere, Finland
tommi.mikkonen@tut.fi

ABSTRACT

Aspect-orientation enables an approach where tangled code can be addressed in a modular fashion. However, the design of interworking between object-oriented baseline architecture and aspects attached on top of it is an issue, which has not been solved conclusively. For industrial-scale use, guidelines on what to implement with objects and what with aspects should be derived. In this paper, we introduce a way to reflect the use of aspect-orientation to production testing software of mobile systems. Such piece of infrastructure software is used to smoke test the proper functionality of a manufactured device. The selection of suitable implementation technique is based on variance of devices to be tested, with aspects used as means for increased flexibility.

Keywords

Production testing, variability, aspects

1. INTRODUCTION

Aspect-oriented approaches provide facilities for sophisticated dealing with tangled and cross-cutting issues in programs [2]. With aspects, it is possible to weave new operations into already existing systems, thus creating new behaviors. Moreover, it is possible to override methods, thus manipulating the behaviors that already existed.

With great power comes great responsibility, however. The use of aspect-oriented features should therefore be carefully designed to fit the overall system, and ad-hoc manipulation of behaviors should be avoided especially in industrial-scale systems. This calls for an option to foresee functionalities that will benefit the most from aspect-oriented techniques, and focus the use of aspects to those areas. Unfortunately case studies on the identification of properties that potentially result in tangled or scattered code in a certain problem domain have not been widely available. However, understanding the mapping between the problem domain and its solution domain, which includes both conventional objects as well as aspects, forms a key challenge for industrial-scale use.

In this paper, we address domain-specific identification of types of properties that lend themselves to aspect-oriented methodology. The domain we will use as an example is that of production testing of a family of mobile devices, where common and device specific features form different categories of requirements that can be used as the basis for partitioning between object-oriented and aspect-oriented techniques. The way we approach the problem is that the common parts are included in the object-oriented base implementation, and the more device-specific ones are then woven into that implementation as aspects.

The rest of this paper is structured as follows. Section 2 gives an overview of production testing of mobile devices. The section also introduces a production-testing framework for Symbian OS based mobile devices. Section 3 discusses how we relate the problem domain and its aspect-oriented solution domain in this particular case. Section 4 provides an evaluation of aspect-orientation in this setting, and Section 5 concludes the paper with some final remarks.

2. PRODUCTION TESTING

Production testing is a verification process utilized in the product assembly to measure production line correctness and efficiency. The purpose is to evaluate devices' assembly correctness by gathering information on the sources of faults and statistics on how many errors are generated with certain volumes. In other words, production testing is the process of validating that a piece of manufactured hardware functions correctly. It is not intended to be a test for the full functionality of the device or product line, but a test for correct composition of device's components. With volumes typical to modern mobile terminals, the production testing involves software support that must be increasingly sophisticated, versatile, cost-effective, and adapt to great variety of different products. In software the most successful way of managing such variance is to use product families [1].

2.1 Overview

Individual design of all software for all mobile device configurations results in an overkill for software development.

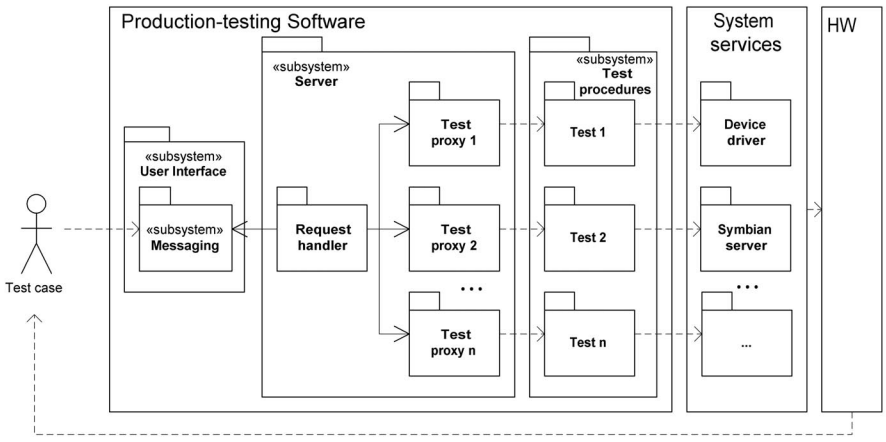

Figure 1: Production-testing framework.

Therefore, product families have been established to ease the development of mobile devices. In such families, implementations are derived by reusing already implemented components, and only product specific variance is handled with product specific additions or modifications. In this paper, we will focus on a product family where Symbian OS [3] is used as the common implementation framework. Symbian OS is an operating system for mobile devices, which includes context-switching kernel, servers that manage devices' resources, and rich middleware for developing applications on top of the operating system.

The structure of the production-testing framework in Symbian environment follows the lines of Figure 1 and consists of three subsystems: user interface, server and test procedures. Test procedure components (Test 1, Test 2, etc.) implement the actual test functionalities and together form the test procedures subsystem. These components form the system's core assets by producing functionality for basic test cases. Furthermore, adding specializations to these components produces different product variants hence dedicating them for certain specific hardware, functionality, or system needs. In other words, the lowest level of abstraction created for production testing purposes is composed of test procedure components that only depend on the operating system and used hardware. As a convenience mechanism for executing the test cases, we have implemented a testing server, which is responsible for invoking and managing the tests. This server subsystem implements the request-handling core and generic parts of the test cases, which are abstract test procedure manifestations as test proxies. Finally, a user interface is provided that can be used for executing the test cases. The user can be a human user or a robot that is able to recognize patterns on the user interface, for instance. The user interface subsystem implements the communication between the user and the production-testing system.

2.2 Variability management

From the viewpoint of production testing, the most important pieces of hardware are Camera, Bluetooth, Display and Keyboard. In addition, also more sophisticated pieces of equipment can be considered, like WLAN for instance. The test software on the target is then composed of components for testing the different hardware and device driver versions, which are specific to actual hardware. When composing the components, one must ensure that concerns related to a certain piece of hardware are taken into account in relevant software components as well. For instance, more advanced versions of the camera hardware and the associated driver, allow higher resolution than the basic ones, which needs to be taken into consideration while testing the particular configuration. Since the different versions can provide different functional and non-functional properties, the testing software must be adapted to the different configurations. For example, the execution of a test case can involve components for testing display version 1.2, Bluetooth version 2.1 and keyboard version 5.5. The particular display version may suggest using a higher resolution pictures as test data than previous versions, for instance. To further complicate matters, the composition of hardware is not fixed. All the hardware configurations consist of a keyboard and a color display. However, some configurations also include a camera or Bluetooth, or both. Then, when testing a Symbian OS device with a camera but without Bluetooth, for instance, Bluetooth test procedure components should be left out from the tester software.

To manage the variability inherent in the product line, the production testing software is assembled from components pertaining to different layers as follows. Ideally, the basic functionality associated with testing is implemented in the general testing components that only depend on the Symbian OS or certain simple, common test functionality of generic hardware. However, to test the compatibility of different hardware variants, more specialized test procedure components must be used. Moreover, to cover the particular hardware and driver versions, suitable components must be selected for accessing their special features. Thus, the test software is assembled from components, some of which provide more general and others more specific functionality and data for executing the tests.

3. APPLYING ASPECT-ORIENTED TECHNIQUES TO PRODUCTION TESTING

In the following we assess the possibilities of applying aspect-oriented techniques to production-testing by identifying the

most important advantages of the technique in this problem domain.

3.1 Identifying tangling

Strive for high adaptability and support for greater variability implies more complex implementations and a large amount of different product configurations. Attempts to group such varying issues and their implementations into optimized components or objects using conventional techniques make the code hard to understand and to maintain. This leads to heavily loaded configuration and large amounts of redundant or extra code, and complicates the build system. Thus, time and effort are lost in performing re-engineering tasks required to solve emerging problems. Hence, for industrial-scale systems, such as production-testing software, this kind of tangled code should be avoided in order to keep the implementation cost-effective, easily adaptable, maintainable, scalable, and traceable.

Code tangling is evident in test features with long historical background. The need for maintaining backwards compatibility causes the implementation to be unable to get rid of old features, whereas the system cannot be fully optimized for future needs due to the lack of foresight. After few generations the test procedure support has cluttered and complicated the original simple implementation with new sub-procedures and specializations. As an example consider testing a simple low-resolution camera with fairly small photo size versus a mega-pixel camera with an accessory flashlight. In this case the first generation of production-testing software had fairly simple testing tasks to perform, perhaps nothing else but a simple interface self-test. However, when the camera is changed the whole testing functionality is extended, not only the interface to the camera. In addition to new requirements regarding the testing functionality, also some tracing, monitoring or other extra tasks may have been added. While the test cases still remain the same, the test procedure becomes heavily tangled piece of code.

Another typical source of tangling code is any additional code that implements features not directly related to testing but still required for all or almost all common or specialized implementations. These are features such as debugging, monitoring or other statistical instrumentation, and specialized initializations. Although the original test procedure did not require any of these features, apart from specialized products and certainly should be excluded in software in use in mass production, they provide useful tools for software and hardware development, research, and manufacturing. Hence, they are typically instrumented into code using precompiler macros, templates, or other relatively primitive techniques.

In object-oriented variation techniques, such as inheritance and aggregation, the amount of required extra code for proper adaptability could be large. Although small inheritance trees and simple features require only a small amount of additional code, the amount expands rapidly when introducing test features targeted for not only one target but for a wide variety of different, specialized hardware variants. Redundant code required for maintaining such inheritance trees and objects is exhaustive after few generations and hardware variants. Hence, the conserved derived code segments should provide actual additional value to the implementation instead of gratuitous repetition. Furthermore, these overloaded implementations easily degrade performance. Hence, the variation mechanism should also promote light-weighted implementations, which require as little as possible extra instrumentation.

Intuitively, weaving the aspects into code only after preprocessing, or pre-compiling, does not add complexity to the original implementation. However, assigning the variation task to aspects does only move the problem into another place. While the inheritance trees are traceable, the aspects and their relationships, evolution and dependencies require special tools for this. Hence, the amount of variation implemented with certain aspects and grouping the implementations into manageable segments is the key asset in avoiding tangling with at least tolerable performance lost.

3.2 Partitioning to conventional and aspect-oriented implementation

The Symbian OS provides more abstract interfaces on upper and more specialized on lower layers. Hence, Symbian OS components and application layers provide generic services while the hardware dependent implementations focus on the variation and specializations. In order to manage this layered stucture in implementation a distinction between conventional and aspect-oriented implementation is required. Separating features and deciding which to implement as aspects and which using conventional techniques is, however, a difficult task. On the one hand, the amount of required extra implementation should be minimized. On the other hand, the benefits from introducing aspects to the system should be carefully analyzed while there are no guidelines or history data to support the decisions.

We propose a solution where aspects instrument product level specializations into the common assets and hence, provide linking time binding into the system. Furthermore, the common product specific and architecture and system level test functionalities comply with conventional object-oriented component hierarchy. However, certain commonalities, such as tracing and debugging support, should be instrumented as common core aspects and hence, optional for all implementations. Thus, we identify two groups of aspects: test specialization aspects and general-purpose core aspects. The specialization aspects embody product-level functionalities and are instrumented into the lowest, hardware related abstraction level. Secondly, the common general-purpose aspects provide product-level instrumentation of optional system level features.

In this solution we divided the implementation on the basis of generality: Special features were to be implemented using aspect-oriented techniques. These are all special, strictly product-specific features for different hardware variants clearly adding special dedicated implementations relevant to only certain targets and products. On the contrary, however, the more common the feature is to all products, it does not really matter whether it is implemented as part of the conventional implementation or as a common aspect. The latter case would benefit from smaller implementation effort but suffer from lack of maintainability. Hence, com-

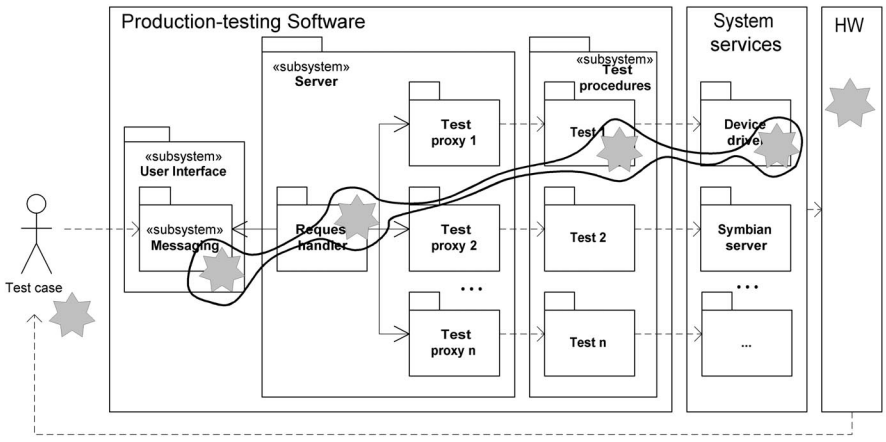

Figure 2: An aspect capturing specialization concern in production-testing framework.

mon aspects are proposed to include only auxiliary concerns and dismiss changes to core implementation structures and test procedures.

3.3 Camera example

We demonstrate the applicability of aspect-orientation in production-testing domain with a simple example of an imaginary camera specialization. In this example, an extraordinary and imaginary advanced mega-pixel camera with accessory flashlight replaces a basic VGA-resolution camera in a certain product in our product family. Since this unique hardware setup is certainly a one-shot solution, it is not appropriate to extend the framework of the product family. Evidently, changes in the camera hardware directly affect the camera device driver and in addition to that, require also changes to the production-line's test cases. New test procedure is needed for accessory flashlight and camera features and the old camera tests should be varied to take into account the increased resolution capabilities. Hence, enhanced camera hardware has an indirect effect on the production-testing software, which has to support these new test cases and algorithms by providing required testing procedures. Hence, camera related specialization concerns affect four different software components, which are all located on different levels of abstraction: the user interface, request handler, related test procedure component, and the device driver. Components requiring changes compared to the initial system illustrated in Figure 1 are illustrated in Figure 2 as grey stars.

From the figure it is apparent that the required specialization cuts across the whole infrastructure and is likely to be difficult to maintain using conventional techniques. In this case, that is how to comply with the extraordinary setup. In practice this could involve new initialization values, adaptation to new driver interface, and, for example, introduce new algorithms. With conventional techniques, such as object-orientation, this would entail inherited specialization class with certain functionalities enhanced, removed or added. Furthermore, a system level parameter for variation must have been created in order to cause related changes also in the server and the user interface level, which is likely to bind the implementation of each abstraction level together. Hence, a dependency is created not only between the hardware variants but also between the subsystem variants on each abstraction level. These modifications would be tolerable and manageable if parameterization is commonly used to select between variants. However, since this enhancement is certainly unique in nature, a conventional approach would stress the system adaptability in an unnecessary heavy manner.

However, the crosscutting nature of this specialization concern makes it an attractive choice for aspects that group the required implementation into a nice little feature to be included only in the specialized products. These aspects, which are illustrated in Figure 2 as a bold black outline, would then implement required changes to user interface, request handler, testing component, and device driver without intruding the implementation of the original system. Hence, the actual impact of the special hardware is negligible to the framework and the example thus demonstrates aspect-orientation as a sophisticated approach of incorporating excessive temporary fixes.

4. EVALUATION

In order to gather insight into the applicability of aspects to production-testing system, we assessed the technique against the most important qualities for such system. These include system's adaptability, variability, reliability and robustness, and performance. In addition, major concerns are the traceability and maintainability of the implementation.

Since the production-testing system is highly target-oriented and should adapt easily to a wide variety of different hardware environments, the system's adaptability and variability are the most important qualities. We consider that by carefully selecting the assets to implement as aspects could extend the system's adaptability with still moderate effort. A convincing distinction between the utilization of this technique and conventional ones is fairly dependent on the scope of covered concerns. While the technique is very attractive for low-level extensions, it seems to lack potential to provide foundation for multiform, large-scale implementations.

Including aspects in systems with lots of conventional imple-

mentations has drawbacks in maintenance and traceability. Designers can find it difficult to follow whether the implementation is in aspects or in the conventional part. As the objects and aspects have no clear common binding to features, following the implementation and execution flow becomes more complex and difficult to manage. Aspects can be considered as a good solution when the instrumented aspect code is small in nature. In other words, aspects are used to produce only common functionalities, for example tracing, and do not affect the internal operation of the functions. That is, aspects do not disturb the conventional development. However, these deficiencies may be caused by the immaturity of the technique and hence reflect designers' resistance for changes. Also the lack of good understanding of the aspect-oriented technology and proper instrumentation and development tools tend to create skeptic atmosphere. However, the noninvasive nature of aspect-oriented techniques makes it superior technique in incorporating tracing and debugging features.

Production-testing software should be as compact and effective as possible in order to guarantee highest possible production throughput. Hence the performance of the system is a critical issue also when considering aspects. Although the conventional implementation can be very performance effective, the aspects provide interesting means to ease the variation effort without major performance drawbacks.

5. DISCUSSION

In this paper, we have described an approach for assembling production-testing software from components that provide test functionality and data at various levels of generality. To implement this product line architecture, we have described a solution based on aspects. The solution depends on the capability of aspects to weave in new operations into already existing components, possibly overriding previous ones. Thus, the solution provides functionality that is specialized for the testing of the particular hardware configuration.

One practical consideration in mobile setting is the selection between static and dynamic weaving. While dynamic weaving adds flexibility, and would be in line with the solution of [4], static weaving has its advantages. The prime motivation for advocating static weaving is memory footprint, which forms an issue in mobile devices. Therefore, available tool support [5] is technically adequate for our purposes.

Unfortunately, tool support for weaving is not the only source of problems in our case. The tool chain of the Symbian development is built around GCC version 2.98, with some manufacturer specific extensions needed in mobile setting [6]. Our first attempts indicate that using tools enabling aspects in this setting is not straightforward but requires more work. While in principle we could circumvent the problem by using mobile Java and AspectJ [7] to study the approach, hiding the complexities of the implementation environment would not be in accordance to the spirit of the problem, where specialized hardware and tools are the important elements.

So far we have not tried out our approach in actual production testing, mainly due to the aforementioned problems. Thus, it remains as future work. We would also like to investigate more on the possibilities aspects could have in conjunction with product family architectures. Especially, the compositionality of aspects in the setting where platform-specific tools are needed is an open issue.

6. REFERENCES

[1] J. Bosch. *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. Addison–Wesley, 2000.

[2] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented Software Development*. Addison–Wesley, 2004.

[3] R. Harrison. *Symbian OS C++ for mobile phones*. John Wiley & Sons., 2003.

[4] J. Pesonen. Assessing production testing software adaptability to a product-line. In *Proceedings of the 11th Nordic Workshop on programming and software development tools and techniques (NWPER'2004)*, pages 237–250, Turku, Finland, August 2004. Turku Centre for Computer Science.

[5] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 2002.

[6] C. Thorpe. Symbian OS version 8.1 Product description. At http://www.symbian.com/ on the World Wide Web.

[7] AspectJ WWW site. At http://www.eclipse.org/aspectj/ on the World Wide Web.