Production-Testing of Embedded Systems with Aspects

Jani Pesonen[1], Mika Katara[2], and Tommi Mikkonen[2]

[1] Nokia Corporation
Technology Platforms
P.O.Box 88, FI-33721 Tampere, FINLAND
jani.p.pesonen@nokia.com
[2] Tampere University of Technology
Institute of Software Systems
P.O.Box 553, FI-33101 Tampere, FINLAND
Tel. +358 3 3115 {5512, 5511}, Fax +358 3 3115 2913
mika.katara@tut.fi tommi.mikkonen@tut.fi

Abstract. A test harness plays an important role in the development of any embedded system. Although the harness can be excluded from final products, its architecture should support maintenance and reuse, especially in the context of testing product families. Aspect-orientation is a new technique for software architecture that should enable scattered and tangled code to be addressed in a modular fashion, thus facilitating maintenance and reuse. 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 embedded systems. Such piece of a test harness 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. Towards the end of the paper, we also present the results of our experiments in the Symbian OS context that show some obstacles in the current tool support that should be addressed before further case studies can be conducted.

1 Introduction

A reoccurring problem in embedded systems development is the maintenance and reuse of the test harness, i.e., software that is used to test the system. Although code implementing the tests is not included in the final products, there is a great need to reuse and maintain it especially in the context where the system-under-test is a part of an expanding product family. However, as the primary effort goes into the actual product development, quality issues concerning the test harness are often overlooked. This easily results in scattered and tangled code that gets

more and more complex each time the harness is adapted to test a new member of the product family.

Aspect-oriented approaches provide facilities for sophisticated dealing with tangled and crosscutting issues in programs [1]. Considering the software architecture, aspects should help in modular treatment of such code, thus facilitating maintenance and reuse, among other things. 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 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, especially in testing 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, based on our previous work [2], we analyze the use of aspect-oriented methodology in production testing of a family of embedded systems. In this domain, common and device-specific features correspond to different categories of requirements that can be used as the basis for partitioning between object-oriented and aspect-oriented implementations. 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. As an example platform, we use Symbian OS [3, 4] based mobile phones.

The rest of this paper is structured as follows. Section 2 gives an overview of production testing and introduces a production-testing framework for the Symbian platform. Section 3 discusses how we apply aspect-orientation to the framework. Section 4 provides an evaluation of our solution including the results of our practical experiments. 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 the correctness of devices' assembly 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 high volumes, the production testing involves test

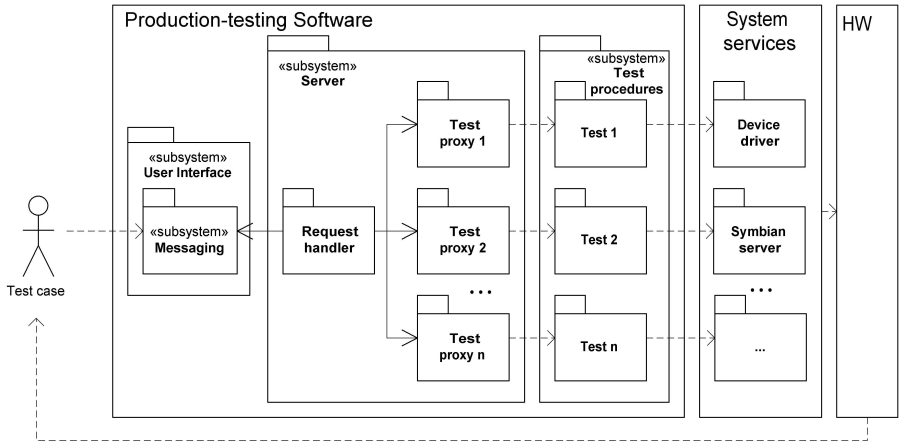

Fig. 1. Production-testing framework

harness software 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 [5, 6].

2.1 Overview of Symbian Production Testing Framework

Individual design of all software for a variety of embedded system configurations results in an overkill for software development. Therefore, product families have been established to ease their development. 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 be using a running example where a product family based on Symbian OS [3, 4] 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. According to the Symbian home page, there are almost 40 million installations of the operating system and over 50 different phone models running the system currently on the market from seven licensees worldwide [4].

The structure of a 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 a certain specific hardware, functionality, or system needs. In other words, the lowest level of abstraction created for production test harness is composed of test procedure

components that only depend on the operating system and underlying hardware. As a convenience mechanism for executing the test cases, a testing server has been implemented, 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 graphical 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, in the case of Symbian, the most important pieces of hardware are Camera, Bluetooth, Display and Keyboard. In addition, also other, less conventional pieces of equipment in mobile phones can be considered, like WLAN for instance. The test harness on the target is then composed of components for testing the different hardware and device driver versions, which are specific to the 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 harness 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. Moreover, possible manufacturer-specific issues need to be taken into consideration. The particular display version may suggest using a higher resolution pictures as test data than previous versions, for instance. To complicate matters further, 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 harness.

To manage the variability inherent in the product line, the test harness is assembled from components pertaining to different layers as follows. Ideally, the basic functionality associated with testing is implemented in the general test procedure components that only depend on the operating system 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 harness 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 traditional object-oriented designs, there are usually some *concerns*, i.e. conceptual matters of interest that cut across the class structure. The implementation of such concerns is *scattered* in many classes and *tangled* with other parts inside the classes [7]. Scattering and tangling may suggest emergence of problems concerning traceability, comprehensibility, maintainability, low reuse, high impacts of changes and reduced concurrency in development [8].

Aspect-oriented programming provides means for modularizing crosscutting concerns. The code that would normally be scattered and tangled is written into its own modules, usually called aspects. There are basically two kinds of aspect-oriented languages, asymmetric and symmetric. The former ones assume an object-oriented base implementation to which the aspects are woven either statically or at run time. The latter treat the whole system as a composition of aspects. 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 Scattering and Tangling

Strive for high adaptability and support for greater variability implies implementations that are more complex and a large number 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 scattered and tangled code should be avoided in order to keep the implementation cost-effective, easily adaptable, maintainable, scalable, and traceable.

Code scattering and tangling is evident in test features with long historical background and several occasional specializations to support. 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 a relatively small photo size versus a megapixel camera with an accessory flashlight. In this case, the first generation of production-testing software had 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 or other extra tasks may have been added. This kind of expansion of

hardware support is in general fully supported by the product-line, but several one-shot products rapidly overload the product-line with unnecessary functionalities. While the test cases remain the same, the test procedures subsystem becomes heavily scattered and tangled piece of code.

Another typical source of scattered and tangled 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 precompiling, 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, such as interdependencies, require special tools for this. Hence, the amount of variation implemented with certain aspects and grouping the implementations into manageable segments forms the key asset in avoiding scattering and tangling with at least tolerable performance lost.

3.2 Partitioning to Conventional and Aspect-Oriented Implementation

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 structure in implementation a distinction between conventional and aspect-oriented implementation is required. However, separating features and deciding which to implement as aspects and which using conventional techniques is 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. Ideally, aspects should be bound into the system during linking. The common product-specific, as well as 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 be 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. These aspects provide a managed, cost-effective and controlled tool for adding support for any extraordinary product setup. Secondly, the common general-purpose aspects provide product-level instrumentation of optional system level features.

In this solution, we divided the implementation based on generality; special features were to be implemented using aspect-oriented techniques. These are all strictly product-specific features for different hardware variants clearly adding 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 core aspect. The latter case could benefit from smaller implementation effort but suffer from lack of maintainability, as it would not be part of the product-line. Hence, common core aspects are proposed to include only auxiliary concerns and dismiss changes to core implementation structures and test procedures.

Our solution thus restricts the use of aspects to implementations that are either left out from final products or dedicated to only one specific product. In other words, conventional techniques are utilized to extend the product-line assets, while the aspect-orientation is used in tasks and implementations supporting product peculiarities as well as research and development tasks. Hence, the product-line would remain in proper shape despite the support for aggressive and fancy products with tight development schedules.

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 of a product family. Since this unique hardware setup is certainly a one-time 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 changes to the test cases. New test procedure is needed for accessory flashlight and camera features and the old camera tests should be

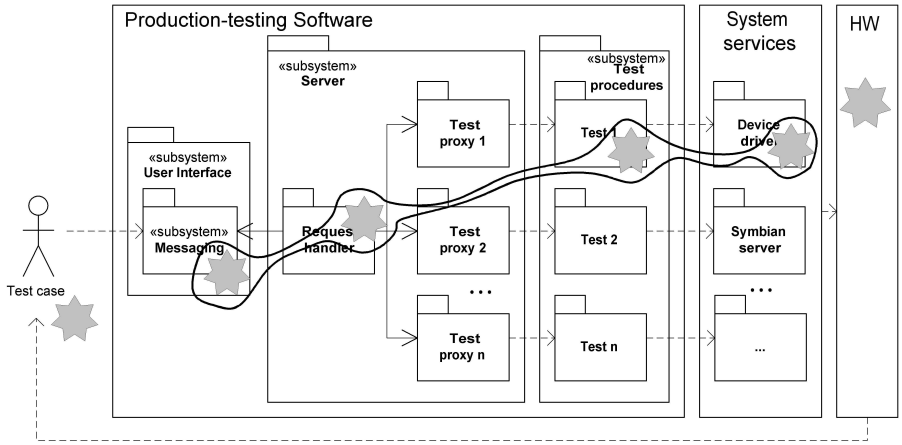

Fig. 2. An aspect capturing specialization concern in production-testing framework

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 it will be difficult to comply with the extraordinary setup without producing excessive scattered or tangled code inside the product-line assets. In other words, maintaining the generality of the system core assets using exclusively conventional techniques is likely to be difficult when the system faces such requirements. In practice, the new set-up 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.

The crosscutting nature of this specialization concern makes it an attractive choice for aspects that group the required implementation into a nice little fea-

ture 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, test procedure 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 traceability and maintainability of the implementation as well as performance. In the following, we describe the experiments we carried out and provide an evaluation.

4.1 Practical Experiments

To investigate further the possibilities of aspect-oriented techniques in this context, we executed a set of practical experiments on an industrial production-testing system. Our first experiment was to infiltrate simple tracing support into an existing testing component and then, according to the results, actually provide an aspect-oriented solution to an example case described in Section 3.3. In the specialization experiment, a one-time system variant was branched out from the product-line by introducing camera related specialization aspects. These aspects introduced all the required modifications to the existing system in order to provide testing support in this special case. The benefit from the aspect approach was, as anticipated, the grouping of specialization concerns.

The prominent aspect implementation in C++ environment is AspectC++ [9, 10] supporting the asymmetric view of aspect-orientation. The tool does a source-to-source compilation adding templates and namespaces, among other things, to the original C++ source. When using the AspectC++ compiler (version 0.9.3) we encountered some tool related issues that we consider somewhat inconvenient but solvable: Firstly, the compiler did not support Symbian OS peculiarities in full in its current form, and hence the tool required a lot of parameterization effort prior to being able to instrument code correctly. Another problem related to the AspectC++ compiler itself was its performance, since instrumenting even a small amount of code took a considerable amount of time on a regular PC workstation. This could have been acceptable, however, if the output would have been the final binary or even pre-compiled code.

The pre-processing of source code prior to normal C++ compilation added an extra step to the tool chain. This raised an issue on whether the aspects should be woven directly into the object files or the final binaries. Based on feedback and experiments, our conclusion is that industrial systems prefer cases where

the source code is not available, but instead the aspects should be woven into already existing systems in binary format.

Other practical difficulties were related to the missing synergy between the used aspect compiler and C++ tool chain. Since the tool chain of the Symbian development is built around GCC [11] version 2.98, with some manufacturer-specific extensions needed in mobile setting [12], the AspectC++ compiler sometimes produced code that was not accepted by the GCC compiler. It was not evident whether this was caused by the GCC version or the nature of the Symbian style C++ code. Switching to other systems with different compilers not directly supported by AspectC++, for example the ARM RVCT (version 2.1) in Symbian OS version 9.1 [13], revealed that this was a recurring problem. Furthermore, the reason why the instrumented code was unacceptable varied. These tool problems were finally solved by modifying the original source and correcting the resulting instrumented code by hand. This added another extra step that admittedly could be automated into the instrumentation process.

After the technical difficulties were solved as described above, our experiments with a simple tracing aspect revealed that the current AspectC++ implementation produced heavy overhead. A single tracing aspect woven into a dynamic link library (DLL) consisting of 22 functions with total size of 8kB doubled the size of the final DLL binary to 16kB. Unfortunately, this 100% increase in size is unacceptable in our case. Although weaving the tracing aspect into the whole component and each function was an overweighed operation, it revealed difficulties associated with the aspect implementation itself. This is a major drawback when considering testing of embedded systems, which are often delicate regarding memory footprint issues.

To summarize the above, a current feature of aspect-orientation in this context seems to be the tight relationship between the tools. This forces the organization to maintain a functional tool chain until a proven alternative is provided, which can become a burden for product families with a wide variety of different hardware platforms.

Moreover, the structure and architecture of the base system has a strong effect on how straightforward the aspect weaving is. Based on these experiments, we tend to believe that the highly adaptive framework actually enables such techniques as aspect-orientation to be adopted. However, this might not be the case with differently structured systems. The product-line architecture makes the adoption at least a bit more straightforward.

4.2 Evaluation Results

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 somewhat dependent on the scope of covered concerns. Based on our experiments, 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 implementations has drawbacks in maintenance and traceability. Designers can find it difficult to follow whether the implementation is in aspects or in the conventional code. As the objects and aspects have no clear common binding to higher-level operations, following the implementation and execution flow becomes more complex and difficult to manage without proper tool support. 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. In addition, the lack of good understanding of the aspect-oriented technology and proper instrumentation and development tools tend to create skeptic atmosphere. Nevertheless, the noninvasive nature of aspects makes them a useful technique when incorporating one-time specializations.

Production-testing software should be as compact and effective as possible in order to guarantee highest possible production throughput. Thus, the performance of the system is a critical issue also when considering aspects. Although the conventional implementation can be very performance effective, the aspects could provide interesting means to ease the variation effort without major performance drawbacks. This, however, requires a careful definition and strictly limited expression of the woven aspect.

5 Conclusions

In this paper, we have described an approach for assembling production-testing software that is composed of components that provide test functionality and data at various levels of generality. To better support 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, especially in the embedded systems setting, is the selection between static and dynamic weaving. While dynamic weaving adds flexibility, and would be in line with the solution of [14], static weaving has its advantages. The prime motivation for static weaving is memory footprint, which forms an issue in embedded systems. Therefore, available tool support [9, 10] is technically adequate in this respect.

Unfortunately, we encountered some problems in practical experiments. Our first attempts indicate that using tools enabling aspects in the case of Symbian OS is not straightforward but requires more work. In principle we could circumvent the problem by using mobile Java and a more mature aspect language implementation such as AspectJ [15] to study the approach. However, hiding the

complexities of the implementation environment would not be in accordance to the embedded systems development, where specialized hardware and tools are important elements. Furthermore, even a superficial analysis reveals that the problems related to size would be hardened, due to the size restrictions of MIDP Java [16].

Due to the reported problems, the results from our practical experiments in actual production testing are somewhat limited. It seems that the tool chain could be fixed to better support the target architecture, but more changes that are considerable would be needed to reduce the memory footprint. Defining the correct use cases and applications that could still benefit from aspect-orientation remains as future work. In addition, we would also like to investigate more on the possibilities aspects could have in the embedded systems domain. For instance, the compositionality of symmetrical aspects in the setting where platform-specific tools are needed is an open issue. Moreover, the effects of aspects to the configuration management need further research.

It should be possible to generalize our results further. Another experiment was actually inspired by the specialization experiment. Because of the noninvasive nature of aspects, the system and integration testing of the production test harness itself could benefit from aspects. However, this remains as a piece of future work.

References

1. Filman, R.E., Elrad, T., Clarke, S., Akşit, M.: Aspect-Oriented Software Development. Addison–Wesley (2004)
2. Pesonen, J., Katara, M., Mikkonen, T.: Evaluating an aspect-oriented approach for production-testing software. In: Proceedings of the Fourth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2005) in conjunction with AOSD'05,, Chicago, USA, College of Computer and Information Science, Northeastern University, Boston, Massachusetts, USA (2005)
3. Harrison, R.: Symbian OS C++ for Mobile Phones. John Wiley & Sons. (2003)
4. Symbian Ltd.: Symbian Operating System homepage. (At URL http://www.symbian.com/)
5. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison–Wesley (2000)
6. Clements, P., Northrop, L.: Software Product Lines : Practices and Patterns. Addison–Wesley (2001)
7. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.M.: N degrees of separation: Multi-dimensional separation of concerns. In Garlan, D., ed.: Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, USA, ACM Press (1999) 107–119
8. Clarke, S., Harrison, W., Ossher, H., Tarr, P.: Subject-oriented design: towards improved alignment of requirements, design, and code. ACM SIGPLAN Notices **34** (1999) 325–339
9. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: 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 (2002)

10. AspectC++: AspectC++ homepage. (At URL http://www.aspectc.org/)
11. Free Software Foundation: GNU Compiler Collection homepage. (At URL http://gcc.gnu.org/)
12. Thorpe, C.: Symbian OS version 8.1 Product description. (At URL http://www.symbian.com/)
13. Siezen, S.: Symbian OS version 9.1 Product description. (At URL http://www.symbian.com/)
14. Pesonen, J.: 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), Turku, Finland, Turku Centre for Computer Science (2004) 237–250
15. AspectJ: AspectJ WWW site. (At URL http://www.eclipse.org/aspectj/)
16. Riggs, R., Taivalsaari, A., VandenBrink, M.: Programming Wireless Devices with the Java 2 Platform, Micro Edition. Addison-Wesley (2001)