Tracing Variability from Implementation to Test Using Aspect-Oriented Programming

Peter Knauber, Johannes Schneider
Mannheim University of Applied Sciences
Windeckstraße 110, 68163 Mannheim, Germany
p.knauber@fh-mannheim.de, jschneid@gmx.de

Abstract. Software product lines have proven to be a very promising approach to software reuse. One of the key concepts in product lines is *variability* that enables the adaptation of common core assets to varying requirements. Variability corresponding to functionality can be managed using well-established techniques like conditional compilation, inheritance, or parameterization. To manage variable cross-cutting concerns like security, transaction management, or error handling, aspect-oriented programming (AOP) provides a mechanism that has recently gained a lot of attention: so-called *aspects* allow to develop and maintain code fragments in a modular way that are then woven into (e.g., object-oriented designed) program code using automated tool support.
This paper explores a way to combine two very popular tools in this context, AspectJ (being an AOP implementation for Java) and JUnit (being the most popular Java testing tool). It illustrates how the same aspects that encapsulate code for cross-cutting concerns in Java programs can also contain the code necessary to test these aspects. This allows to achieve traceability from the implementation of variable software product line parts to their tests.

1 Introduction

Software product lines are a very promising approach to software reuse, they have proven their applicability in a broad range of situations and produced impressing results. A software product line is a collection of software products sharing a common set of features that address the specific needs of a defined domain. These products can be derived from a common product line infrastructure or, in other words, from a common managed set of assets [3]. In order to accomplish this, variability is key, implying that the assets that comprise the product line infrastructure are variable enough to be adaptable to different environments, user groups etc.

Similar to products developed as single systems, traceability among the work products from different life cycle phases is necessary for product line development too. But additionally to single system development, the traceability has to support the variability within the product line infrastructure. This comprises the consistent usage of variable assets or asset parts, implying that requires/excludes/etc. dependencies among the assets are considered when resolving variation points for binding assets together in a specific product. Decision models [5] have been proposed and applied to solve the related problems.

This paper explores the special situation where the development environment allows the integration of program code with (component) test code for regression tests. The Eclipse platform [4], used for Java development by a growing community of developers, supports JUnit, a plug-in that supports the programming of regression tests in Java to test single classes (as components) and - up to a certain limit - integration of classes [7].

Aspect-oriented programming (AOP, [9]) has been evaluated as being an appropriate mechanism to realize variability in software product lines *if* there is tool support available [10]. In the case of Java as programming language, there exist a couple of AOP implementations; for the example used in this paper (see Section 3) we are using AspectJ [2] which is another plug-in that is embedded perfectly in the Eclipse environment.

The rest of this paper is structured as follows: Section 2 of this paper discusses the design of AOP programs wrt. their specific testing problems. In Section 3 an example (a software for a parking ticket machine) with different kinds of variability is introduced. Section 4 discusses implementation and test code of the variable features using AOP. Lessons learned from the use of AspectJ together with JUnit are summarized in Section 5 and Section 6 concludes the paper.

2 Testing Aspect-Oriented Programs

In industrial practise, the variability within software product lines is often realized using macro preprocessors. Probably the main reason for this is the widespread availability and know-how resulting from the distribution of such a tool with C and C++. Studies have proven that their application is – while possible – often not recommendable, especially within large systems where the effects of single variabilities affect many places in the product line assets [1]. Muthig and Patzke point out in [10] that preprocessors do not provide direct language support, tend to clutter the code, impede comprehensibility, and do thus not scale up for large programs and/or many product variants.

For the modular implementation of crosscutting concerns aspect-orientation with its most popular technologies aspect-oriented programming (AOP) and multi-dimensional separation of concerns (MDSOC) seems to be more appropriate. The availability of aspects on the design and implementation level allow for the modular design of aspects that are closely related in the problem domain (i.e., on the semantic level). Thus, AOP decreases the complexity of large systems by supporting modularity. Muthig and Patzke evaluate aspect-orientation as being an appropriate mechanism to realize variability in software product lines *if* there is tool support available [10].

One drawback from using AOP is that linking variable program code to the respective (variable) test code becomes more difficult than, for example, when using a preprocessor for conditional compilation. This results from the fact that the locations where additional code is woven in are described by patterns that can not be reused to identify the corresponding test code locations. These test code locations have to be described by completely different patterns. We have explored this problem in detail and use a small example in Section 3 to illustrate the related problems. A possible implementation including test code is sketched in Section 4 and lessons learned are given in Section 5.

3 Example: Software for a Parking Ticket Machine

3.1 The Basic PTM Model

The example we use to illustrate variable program and test code that should be directly linked together is a Parking Ticket Machine (PTM). Figure 1 gives an idea of how a basic PTM could look like; devices like a slot for coins, the display and the coin slot where printed tickets are given are emulated by software. The features of our basic PTM are:

1. Different coins from 1ct to 50ct can be used to buy a ticket; the total value of coins inserted so far and the end of paid parking time are shown in the display after each coin.
2. The chargeable parking time ranges from Monday to Friday from 8am to 6pm; costs for the first parking unit (30 minutes) are 25ct, for later units (also 30 minutes) 50ct; maximum parking time is 2 hours.
3. Money for more than two hours is accepted (in case somebody does not have enough small coins) but no change is given. If the parking time paid extends the 6pm, the remaining time is granted at the beginning of the next chargeable period (starting at 8am in the next morning or on Monday).
4. If the Ticket button is pressed, a ticket is printed and ejected, the display is cleared, and the coins inserted are consumed.
5. There exists a Cancel button to interrupt and stop the buying process at any time, all coins inserted so far are returned.

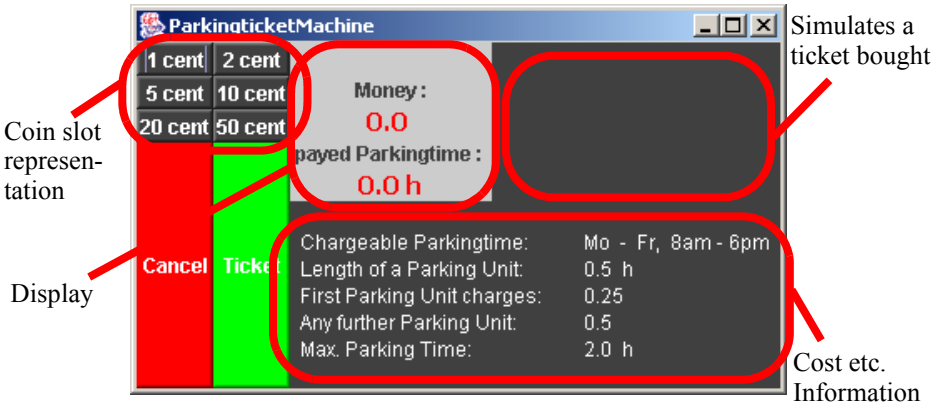

Figure 1: Parking Ticket Machine (PTM), basic model

For demonstration purposes, our PTM is built in Java using Swing to implement the GUI. In reality the different PTM units like the coin slot, the display etc. would be hardware units accessible by drivers whereas all internal calculations would still be realized by software. For the implementation (see Section 4) we focus on the internal, that is, software parts.

3.2 Variants of the PTM

Van Gurp, Bosch, and Svahnberg distinct three (basic) different kinds of variability [6]: (a) optional variability, (b) alternative variability (XOR), and (c) multiple coexisting variability (OR). Using the PTM as described so far as base system, we describe four different extensions, that represent these kinds of variability:

1. An optional feature (type (a) variability) is the ability to print a receipt after having requested a ticket. For this purpose, a receipt button has to be added which becomes active after ticket printing and which is disabled again after any other interaction with the PTM, including receipt printing.
2. The PTM allows for the use of *either* coins (as in the basic model, see Figure 1) *or* credit cards (type (b) variability). This implies that the coin slot unit can be replaced

with a card reader unit. Inserting a credit card automatically leads to charging the amount for minimum parking time.

3. The PTM allows for the use of coins and credit cards *in parallel* (type (c) variability).[1] This implies that coin slot unit and card reader unit can both be present in one PTM model. Consequences for the software are that activating one of these devices at runtime by either inserting a coin or a card leads to deactivation of the other one until a ticket has been printed or the cancel button has been pressed.
4. Since the receipt option is very simplistic, we consider one more optional feature (type (a) variability): if (and only if) the PTM accepts credit cards, it is possible to choose different numbers of parking units (whereas having the credit card alternative only just allows to buy a ticket for one parking unit). This feature implies two buttons that can be used to select more or less parking units.

Figure 2 shows a PTM with card reader (instead of coin slot) including buttons for choosing the parking time (option 4) and receipt button.

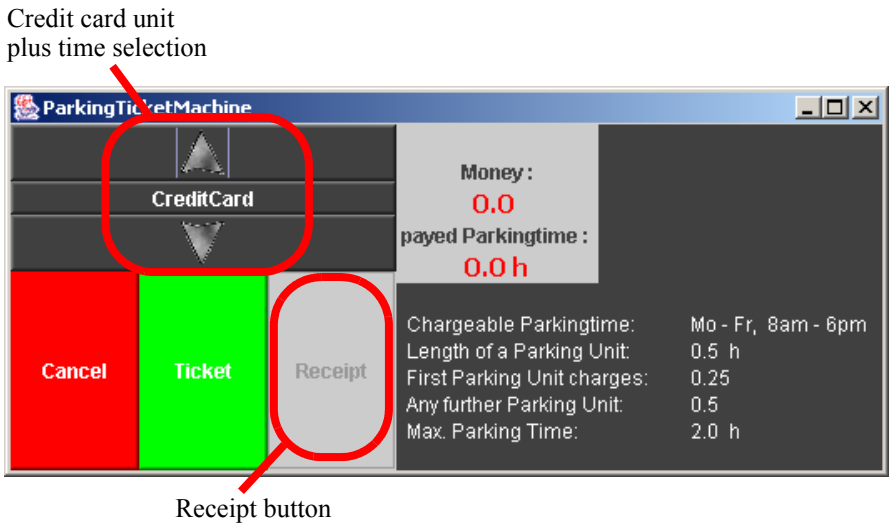

Figure 2: Parking Ticket Machine (PTM) with variable features

4 Implementation and Test of the PTM-Software

Section 4.1 gives an overview of a typical implementation of the PTM, in Section 4.2 the characteristics of the different variants and their implementation together with their tests are briefly explained; focus is always on where to place the corresponding variable code. Lessons learned during design, implementation, and test are summarized in Section 5.

1. It may seem strange that we reuse the components from our XOR example in the OR example. In fact, we found that using AOP for variability implementation reduces the difference between XOR and OR alternatives: basically, in the case of XOR only one aspect (out of a certain group) can to be added to a system whereas in the case of OR more than one aspect (from that group) can be added. If the aspects are designed carefully, this works perfectly well because all these aspects can be added one after the other!

4.1 The Basic PTM Model

Figure 3 shows a UML class diagram of our PTM using the variability notation from [8] to indicate optional and alternative components. Some of the non-variable classes represent the hardware units (`CoinSlot`, `TicketButton` and `CancelButton`, `TicketPrinter`, and `Display`), the class `ParkingTicketMachine` is responsible for their coordination, the internal (software) calculations (using the `ParkingTimeCalculator`) and storing parameters (class `ParkingParameters`).

Realizing component tests for the software parts using JUnit results in two test classes: `ParkingTimeCalculatorTest` and `ParkingTicketMachineTest`. Examples for tests of the class `ParkingTimeCalculator` are

- correct determination if a certain point in time is within free or chargeable parking time,
- correct mapping of amount of money inserted to number of parking units, and
- correct calculation of end of parking time depending on money inserted and current time, especially for cases like 2 hours paid beginning on a Friday at 5pm: parking time should end at next Monday 9am

The JUnit tests of the class `ParkingTicketMachine` focus on class interaction aspects like

- inserting a coin should result in recalculation of money inserted and end of parking time, the display and the internal variables should be updated accordingly,
- pressing the cancel button should result in resetting the internal state and ejection of the coins inserted so far, also the display should be updated, and
- pressing the ticket button should only be possible if at least the minimum amount of money has been inserted, then, the coins should be consumed, a ticket should be printed, the internal state should be reset, and the display should be updated.

Using JUnit, these kinds of test situations are typically encoded in one method each where each method can subsume a number of different test cases for these situations.

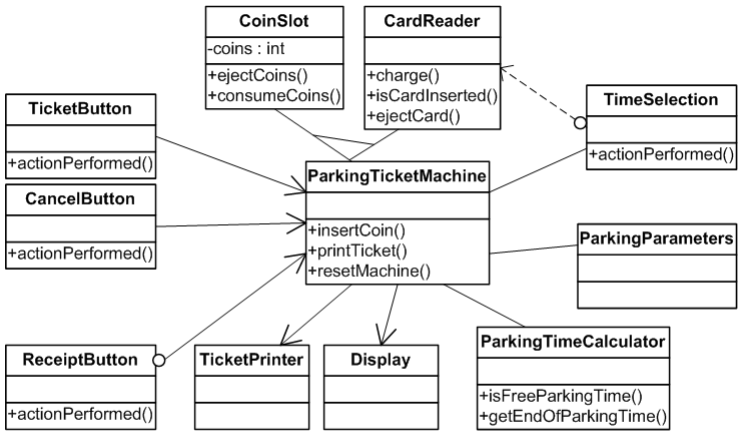

Figure 3: PTM Class Diagram, with variable components

4.2 Implementation and Test of PTM Variants

In this section we describe the implementation of the PTM variants as they are sketched in Section 3.2. For each variant the positions are given where existing program code is extended or changed using AspectJ (called *join points*[1]).

The interesting part in the context of this paper is, of course, which join points are used for extending or changing the existing test code in order to adapt it to the respective variants; only this way each aspect can combine variable code with its corresponding test code. The intention is to reuse existing test code as far as possible, that is, invariable code should be tested by invariable test code and variable test code should only cover the variable parts within the PTM code. We do not discuss extensions of the GUI (code) here because in a real PTM these would be realized by adding or removing hardware components (figures 1 and 2 are given for illustration purposes only). Figure 3 shows the variable components and their relationships.

4.2.1 Variant 1: Optional Receipt Button

The aspect which implements the receipt printing, has to address the following issues:

- The receipt button must be activated after a ticket has (successfully) been printed; this code would be placed at the end of the `printTicket` method of the `ParkingTicketMachine` class.
  This is checked at the place where the `printTicket` method is tested (usually in a method called `testPrintTicket`).
- Pressing the activated receipt button leads to printing a receipt and deactivation of the button; the respective code is placed in an additional method which is registered as event handler for the receipt button.
  The test code is added in an additional test method.
- The receipt button must be deactivated after any interaction; respective code has to be added in the event handlers for the ticket and the cancel button and in either the `coinInserted` method (if the coin slot is present, class `ParkingTicketMachine`) or the event handler for card insertion (if the card reader is present). In our case this would happen in the `printTicket` method.
  The corresponding test code is added to the event handler tests for the ticket and the cancel button and to the `printTicket` method.

4.2.2 Variant 2: Alternative Coin Slot or Credit Card Reader (XOR alternative)

It is obvious that the coin slot specifics and the card reader specifics are implemented using two aspects that exclude each other. Both devices have a similar purpose, thus, they have a similar interface:

- Both devices have to react to a `printTicket` request: the coin slot would consume the coins inserted so far, the card reader would charge the respective amount and eject the card after it has been successfully checked that a card is inserted.
  This is checked at the place where the `printTicket` method is tested (usually in a method called `testPrintTicket`). Additionally, for the card reader alternative is

1. *Join points* are well-defined points in the execution of a program.

has to be checked that a ticket is only printed after the method `isCardInserted` has returned `true`.

- After pressing the cancel button, the devices have to be able to eject (potentially) inserted coins resp. a (potentially) inserted card.
  The corresponding test code is added to the event handler methods.

4.2.3 Variant 3: Coin Slot and Credit Card Reader together (OR alternative)

Allowing coin slot and card reader together results in the following implementation and test issues:

- At startup of the PTM, both devices are active. As soon as a coin is inserted the card slot has to be closed and as soon as a credit card is inserted the coin slot has to be closed, at the same time the PTM is informed about which device is active (stored in an additional attribute `presentDevice`). The code is added to the end of the resp. event handlers.
  Test code for checking that the devices are correctly disabled and that the PTM has been informed is added to the test methods for the event handlers.
- The `printTicket` and the cancel requests are implemented and tested similar to Variant 2. In the `printTicket` method only the activated device is asked to `charge`/`consumeCoins` depending on the setting of the attribute `presentDevice`. Additionally both devices must be enabled afterwards.
  Test code for checking about the correct usage of the `presentDevice` attribute is added to the `testPrintTicket` method, in the test methods for the event handlers it is checked that the devices are correctly enabled.

4.2.4 Variant 4: Selection of Parking Units (with Credit Card alternative only)

This aspect depends on the credit card alternative, that is, it does not work with the coin slot unit. The following issues have to be addressed:

- Two buttons allow the selection of more or less parking time units (at least one, at most as many as fit into the maximum parking time).
  It should not be possible to select less than one parking time unit or more parking time units than fit into the maximum parking time by pressing the time selection buttons. The resp. code is inserted into a new test method for the button event handlers.
- When requesting a ticket, the amount of money corresponding to the selected unit number must be charged. This is not compatible with the implementation introduced by the card reader aspect (see Section 4.2.2) where always one parking time unit is charged. At this point, there are two alternatives to realize the additional aspect: (a) to charge a second fee if there is more than one parking unit selected or (b) to replace the original code for charging with other code which charges the correct amount of money. Alternative (a) is not really a good option; alternative (b) can easily be realized using an around advice in AspectJ: the original code is hidden by new code.
  It has to be tested that the correct amount of money is charged (corresponding to the number of selected parking time units) instead of the standard charge for one parking time unit. The resp. code is inserted in the test method `testPrintTicket` for the printTicket method. Additionally, it should be made sure that pressing the new but-

tons multiple times in different order results in the correct number of selected parking time units.

5 Lessons Learned from Test Design for AOP Programs

The first and most important lesson tells us that it is possible to combine variable code with its corresponding test code in one aspect. This way, the impressing modularity of aspects for the implementation of cross-cutting concerns can be extended to test code. This works, as sketched in Section 4, for component tests of classes. We demonstrated it with a small example in Java and using JUnit but one can easily imagine that the principle is not limited to this language and the JUnit framework. We think that is even possible to use the same principle for integration testing, of course, to the extend only where it does make sense to do integration testing using hard coded tests.

The first optional variant we introduced was very simplistic: a new button was added that introduced some new functionality. The only real interaction with the existing code was that the button had to be activated and deactivated from the existing code. It is not very surprising that the tests could be implemented in a straight forward manner as well.

This is why we introduced a second optional variant which was a little more sophisticated: two buttons for selecting the number of parking units to be charged from a credit card. For the implementation of this option, an around advice was used to override the existing code which would always charge one unit. If this option could be foreseen, one would have used subclassing and subtype polymorphism in object-oriented programming for its implementation. But because we explicitly wanted to explore AOP as mechanism for variation implementation we did not choose this option. In our specific situation the around advice did work well but in principle we recommend to not use this mechanism too often: changing existing code easily results in unexpected problems due to interference when combining different aspects. Instead, we strongly recommend to not rely on one single mechanism for variant implementation only, other (in our case object-oriented) mechanisms should be used where appropriate! Compared to the implementation of the feature itself, adding the test code for this option was relatively easy.

Summarizing we recommend that a good AOP design should always take care that all aspects rely on the code of the base system. If that base code is overridden by some aspects (using around advice) other aspects may experience problems when trying to contribute their part.

A similar statement can be made for implementation of the XOR alternative: thinking ahead when designing the object-oriented PTM one would probably have designed an interface for some `ChargingDevice` which the `CoinSlot` and the `CardReader` classes would have implemented. This way, the PTM would not have needed to know which device actually was built in one specific variant. This way, the test code could abstract from specific variants as well. No AOP mechanism is necessary to implement this. For the test code again, AOP proved to be a easy-to-use mechanism.

For the implementation of the OR alternative, we actually liked the AOP mechanism: it was quite easy to add the runtime variation point and parametrize it depending on two (or even more) alternatives present. The same holds for the corresponding test code that could easily be added to the existing code of the base PTM system.

6 Conclusion

In the context of this paper the integration of aspect-oriented programming using AspectJ with component testing using JUnit worked quite well. The modularity of aspects for introducing variants into existing code could be extended to the corresponding test code. This way, tests for the variable assets (or asset parts) had to address only the variable parts, the implementation and the tests of invariable product line assets (or asset parts) could be reused completely when adding optional of alternative (XOR, OR) variants. Some lessons learned from our experience are sketched briefly in Section 5.

For the future, we think in different directions. The first thing we plan is to extend our current work to industrial case studies, which will take a lot of convincing because aspect-oriented programming is still in its beginnings and not very widely known so far. We are convinced that the integration of this new technology into existing and well-established development environments like Eclipse which JUnit will lower the entry barrier. Our second activity planned is the definition and application of a more formal schema for the evaluation of product line implementation techniques applied for testing code variants. These techniques include techniques like preprocessors, polymorphism, and genericity.

References

[1] M. Anastasopoulos, C. Gacek: Implementing Product Line Variabilities. Proceedings of the 2001 Symposium on Software Reusability (SSR'01) Toronto, Canada, 2001

[2] The AspectJ™ Programming Guide, see www.eclipse.org/aspectj/

[3] P. Clements, L. Northrop: Software Product Lines: Practices and Patterns. Addison-Wesley, 2001

[4] The Eclipse Platform, see www.eclipse.org

[5] O. Flege: Using a Decision Model to Support Product Line Architecture Modeling, Evaluation, and Instantiation, The First Software Product Line Conference (SPLC1), Denver, CO, USA, 2000

[6] G. van Gurp, J. Bosch, M. Svahnberg: On the Notion of Variability in Software Product Lines. In Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA), 2001

[7] JUnit, see www.junit.org

[8] K. Kang, S. Cohen, J. Hess, W. Nowak and S. Peterson: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin: Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. 1997

[10] D. Muthig, T. Patzke: Generic Implementation of Product Line Components. Proceedings of NetObjectDays, Erfurt, 2002