ISSN 1407-7493 COMPUTER SCIENCE DATORZINĀTNE 2009-7493

APPLIED COMPUTER SYSTEMS
LIETIŠĶĀS DATORSISTĒMAS

PROBLEMS OF TEST-DRIVEN ASPECT-ORIENTED DEVELOPMENT

TESTU VADĀMĀS ASPEKTORIENTĒTĀS IZSTRĀDES PROBLĒMAS

Aleksandr Sukhorukov, M.sc., assistant, Riga Technical University, Meza 1/3, Riga, LV 1048, Latvia, phone: +371 26321842, Aleksandrs.Suhorukovs@rtu.lv

Test-driven development, aspect-oriented programming, AspectJ

1. Introduction

During last years aspect-oriented programming (AOP) paradigm [1] is gaining popularity among programmers. AOP fits well in traditional object-oriented programming (OOP) providing some additional possibilities. It does not require one-time complete switch of approaches, since aspect-oriented (AO) program can be developed in object-oriented (OO) way just adding AO-specific features as such need arises. This factor simplifies adoption of AOP in existing development communities. Several implementations of AOP were developed, but the most popular still is AspectJ language – based on Java it provides its own compiler that understands AOP-specific constructs [2].

Test-driven development (TDD) is an agile development method based on idea that tests can be used as a specification of code being developed, so the tests should be developed before the actual code [3]. TDD gained popularity in OO world because it allowed to achieve aimed quality of developed software in shorter time because of clearer design and extensive test coverage.

Both approaches are recognized as valuable tools to improve software development. However, combining these techniques leads to new emergent problems because of several contradictions between fundamental principles of both. This article focuses on those problems describing the most important of them and proposes several ways to lower their negative impact allowing TDD and AOP better used together.

The article is organized as follows. Section 2 provides short introduction to TDD and AOP. Section 3 identifies most notable problems that arise when using AspectJ in TDD. Section 4 provides some empirically discovered principles that help to soften impact of those problems. Section 5 concludes the discussion.

2. Test-driven development and aspect-oriented programming

In this section we provide short introduction to TDD and AOP and estimate how they fit together.

TDD is a software development approach in which unit tests are developed before the actual code. According to [3] TDD process consists of repeating sequence of five steps:

1) add a test (a little automated unit test that tests a small piece of functionality);
2) run all tests and see the new one fail (as there is no implementation yet);
3) make a little change (develop an implementation, or fix a problem);
4) run all tests and see them all pass;
5) refactor (improve the design of already working code) and retest.

TDD helps programmer to think in terms of contract first (how class will be used) and only then to switch attention to implementation (how class will work). It results in cleaner design, better test coverage and designing loosely coupled units.

AOP is relatively new software development paradigm aimed to better separation of concerns. It allows implementation of concerns that crosscut whole system or significant part of system to be

implemented separately as aspects. Classical examples are logging and permission control functionality. AOP builds upon OOP adding more abstract layer to it. Base concerns are developed in traditional OO classes (for example, in Java) and crosscutting concerns are added to specific points of those classes (called join points) during weaving process thus extending their functionality. Aspect encapsulates pointcuts (rules that select a specific set of join points), advice (method-like structures, actual code that will be added to join points corresponding to specific pointcut), and introductions that add new methods or fields to specific classes [2].
The most used AOP language is AspectJ, which is based on Java (each Java program is valid AspectJ program) and adds additional language constructs to define aspect-oriented features. AspectJ compiler takes sources of Java classes and aspects, weaves them together and compiles them into bytecode that can be run in Java virtual machine like any bytecode created by Java compiler itself.
TDD can be successfully applied to traditional OO programs because tests define contract for classes and this contract implementation is enclosed in class itself. In AOP certain features of class are extracted in aspects which may apply to multiple classes. It means that implementation of class behaviour may be distributed between class itself and aspects that affect this class. This introduces new problems in applicability of TDD for AOP programs.
Nevertheless benefits of TDD are worthy of considering its application to AOP generally and to AspectJ program development in particular. In order to do it successfully we need to realize emergent problems and find ways to eliminate them or at least soften their impact. In the next section we identify those problems.

3. Arising problems

In this section we provide list of most notable problems that arise in TDD projects where AspectJ is used.

3.1. Performance

TDD practice of retesting code after each small modification requires frequent updates of class binaries. This is not an issue with pure Java as tools like Ant [4] automate process of recompilation of only those classes whose code changed since the last recompilation. For AspectJ programs this is not a straightforward process. AspectJ compiler ajc needs to analyze each class in order to determine whether some aspect should be woven into it. So, compilation process must involve multiple source files regardless of their last change time. This leads to significantly longer compile times if compared to Java compiler [5, 6].
In practice slow compilation performance leads to psychological resistance to code retesting. A programmer tends to make more changes at once and retest less frequently. As a result introduced problems are more difficult to trace and speed of development may slow down. If execution of tests takes more than a few seconds, programmer's attention may switch to other targets during this process. But insufficient concentration on code being developed results in less quality.
Latest versions of AspectJ allow choosing compile-time or load-time weaving. In case of load-time weaving compilation time can be reduced. But it leads to slower execution time, as weaving needs to occur during execution. Therefore total time of retesting is not much different for compile-time and load-time weaving [6].

3.2. Test identification

Tests should be written before the actual program code in TDD. Program code in AOP consists of base code and aspect code, which are usually physically separated. In OO programs class is the basic testable unit where all its fields and methods are defined (except fields and methods inherited from ancestor classes). In AO programs some features of class fields and/or methods can reside in one or several aspects: aspect can augment or even replace method behaviour, it can introduce new fields and methods etc. So class after weaving may not be the same as before weaving. It leads to question how to define tests for such class. Different strategies are imaginable:

1. Create tests just for completely woven class with all intended aspects applied (illustrated at Figure 1a). This strategy is close to OO approach – we define tests (as a single jUnit TestCase class, for example) as for traditional OO class, and tests shouldn't know whether (or how) implementation is partitioned between class and aspects. However this strategy also has its drawbacks. If requirements for crosscutting concern change, removing aspect from the system can be painful – many tests are likely to fail. Reusing such class in another context with other aspects also would require writing new tests for that context.

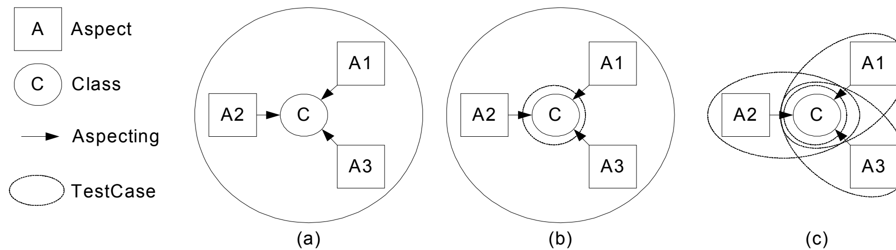

Fig.1. Test definition strategies for aspected class

2. Create separate tests for unwoven class and woven class (illustrated at Figure 1b). In this strategy one TestCase can focus on base code, but another one – on class version affected by all related crosscutting concerns. However aspects may affect class in such a way that base code tests would fail for woven class version. Solution could be a two-stage testing: first compiling classes without aspects and executing base code TestCases, then recompiling classes with aspects and executing woven code TestCases. Thus changes in crosscutting concerns will not affect base code tests. However we still are tied to specific combination of aspects affecting the class, trying to change the combination (remove some aspects or add new ones) will cause tests to fail.

3. Create separate TestCase for unwoven class and separate TestCases for each aspect affecting the class (illustrated at Figure 1c). This strategy may work especially well if aspects are orthogonal [7] and do not introduce interference problems [8]. It allows making different combinations of aspects, when such need arises. However this strategy requires more careful design – to achieve aspect orthogonality and to avoid interference.

Other strategies are possible, however none of them is as easy as in traditional pure OO Java.

3.3. Test code structuring

Problem of test code structuring is related to previous one. Common approach in Java is to create single TestCase for single class, name it ClassNameTest where ClassName is name of tested class and put it in the same package (usually in parallel physical directory structure). In case of first test definition strategy, the same approach is applicable. In other strategies there are many questions about structuring: in which package TestCase should be placed if class and aspects reside in different packages, how to name TestCases, etc.

However the most difficult question is how these tests should be run in respect of weaving. In the simplest case all tests are designed to run on completely weaved code. So, code is compiled first with AspectJ compiler, and then all tests are run. If one group of tests is designed to work only for unweaved code, some second group – on completely weaved code, but third group – on partially weaved, then each group execution should be preceded by compilation with different options. Situation is even more complicated as there are multiple possibilities to weave code partially and aspect precedence can be important both for partial and complete weaving.

Writing Ant targets for each kind of compilation required and selection of TestCases to run for these compilations can be complex and time-consuming work.

3.4. Invalidated assertions

Introducing new aspects into the system can cause initially unforeseen behaviour. Classes that already have been covered by tests may be affected by new aspect if its pointcut captures some of join points in those classes. As a result assertions that passed before may fail now. For example, RoundingAspect may be introduced that for all methods returning values of type double in certain package will apply advice to round the return value to two decimal places. Many tests that tested return value of such methods will fail now if such behaviour was unforeseen.
This brings us back to problem described in 3.2 – again we have a choice of strategy – whether to change existing tests so that they take into account new aspect, or leave those tests unchanged (and run them only in environment without new aspect being weaved) and create new tests for new aspect.
This problem here is approached from different side. Newly introduced aspect requires programmer to write new tests for all classes that this aspect can affect in order to ensure reliable coverage. As in TDD test for some feature should be written prior to feature implementation, all those tests should be written before the aspect itself. As it could require much more time than writing aspect, programmer will have temptation to write (or change) tests only for those classes whose existing tests would fail otherwise. Leaving other code without additional tests for newly introduced feature decreases test coverage and makes test suite less reliable.

3.5. Uncontrolled aspects weaving

In OOP behaviour of class depends on author of the class. In AOP behaviour of class may be changed without making direct modifications to the class code. Therefore it is not possible for author to foresee all possible ways how the class can be affected by aspects. Two possible scenarios that lead to this problem are:
1) At some point of time some external library is imported to the project. If weaving is applied to this library, aspects that reside in this library can affect our class. This leads to foreign aspect problems [9].
2) Our project is imported into another external project as a library. Aspects that reside in the external project can be woven into our classes, resulting in similar problems.
Result of both these scenarios in insufficient test coverage of our class – author of the class cannot provide sufficient tests for originally unforeseen behaviour. If described integration scenarios happen when source code is available for both sides, then the problem is solvable. In first scenario author of class can look up for library aspects that affect the class code and add necessary tests. In second scenario author of external aspect can look for classes that are affected by new aspect and add tests for those classes.

3.6. Refactoring affected tests

In OOP if tests are developed for public interface of class, changing just implementation details of the class does not require to introduce modifications to tests. So, for refactoring class code without breaking its contract with external world, the same tests can be used to verify correctness of refactoring.
In AOP refactoring can involve extracting some part of class implementation to aspect. It means that though public interface remains unchanged, the actual behaviour of class will depend whether aspect will be woven. Depending on selected test definition strategy, it may require redefinition of tests.
Thus tests for AspectJ program are less stable if compared with traditional OO Java and may require more frequent changes according to changing structure of program.

3.7. Break of simplest implementation principle

Common approach in TDD states that code should be developed in simplest possible way that allows tests to pass. If this simplest implementation does not correspond to requirements, it means that there

is insufficient amount of tests. New tests should be added one by one followed by implementation and refactoring until the simplest possible solution that passes the tests appears to be the required one. [3]
In AOP this principle is difficult to follow. For our example of RoundingAspect, the following scenario is possible. At some stage the simplest possible implementation appears to be introducing such aspect instead of rounding inclusion in each method of some class. The aspect then is introduced. Then existing code in other classes can also be refactored to this aspect – rounding feature is removed from classes themselves and aspect's pointcut is redefined to apply to all classes in some package.
This again leads to problem that existing tests will fail if the aspect is not woven. The new problem here is that RoundingAspect is adding features not only to classes that already exist in the package, but also to classes that will be developed in future. TDD process breaks here – we added functionality for new classes before tests for these classes were developed.

3.8 Unintended weaving

This problem is a consequence of the previous one. At some point the simplest implementation to pass existing tests may be introducing an aspect affecting classes for which tests are not yet developed.
This implementation may be correct from requirements perspective, just not covered by tests sufficiently. But it may also lead to incorrect pointcut strength problem [10]. The aspect may affect some classes that it shouldn't affect, but tests for this class may still pass. New functionality is introduced at some point unintentionally without appropriate test definitions.

3.9. Control-flow dependant pointcut testing

At some point introducing aspect with cflow or cflowbelow construct may seem the simplest possible implementation from requirements perspective. However sufficient testing of such implementation may require defining tests checking correct behaviour in both situations – when some code runs in selected execution context and when this code runs out of the context.
Simulating execution under certain execution context may not be easily implemented in tests which always run in simplified context comparing to real program execution. This limitation of possibility to construct appropriate tests leads to contradiction. The simplest possible implementation that will pass tests may not be the simplest one from requirements perspective. So, programmer will have dilemma – whether to adhere to TDD principles and develop classical (and more complex) solution or break those principles and develop intended features in AO-way in simplest way that satisfy actual requirements.

4. TDD and AspectJ harmonization

Described problems emerge due to contradictions between TDD and AOP principles. So, they cannot be totally eliminated in current state of AOP technology. Particularly AOP inevitably breaks encapsulation and makes implementation of objects distributed throughout different modules. Nevertheless AOP can considerably improve simplicity of design but should be applied carefully. Here we formulate some empirically discovered principles that can help to use AspectJ in TDD in a way that these contradictions are softened though with trade-offs from both sides.

1. Don't overuse AspectJ-specific constructs. Introduce only those aspects that will significantly improve overall design. In case of doubts, prefer OO-constructs. Crosscutting concerns should be implemented as aspects only if they really crosscut large amount of classes.
2. Prefer orthogonal aspects that cause as little interference as possible. Aspects could add more steps but should not change logic of base code. Base code should be totally functional also when no aspects are woven into it. This also will allow base code tests to execute regardless of whether aspects are woven or not.
3. Avoid changing existing method behaviour (e.g. with around-advice) for "normal" conditions. Though doing so is acceptable for "special" or "exceptional" cases paying special attention for such code maintenance. In this case separate tests for base code for "normal" conditions and tests for woven code for "special" conditions.

4. Prefer development of small components (libraries) and prevent aspect impact on external components where possible. Small components are compiled much faster and usually there are not many crosscutting concerns that really crosscut the whole application. Often they crosscut just some limited part of application which then is a candidate for separate component.
5. Consider combining tests related to specific aspect into one TestCase. In this scenario tests for base code is placed in its corresponding TestCase but tests for woven class reside in corresponding aspect TestCases. This keeps tests close to place where tested feature resides and allows simpler selection of TestCases for certain weaving scenario.
6. Prefer using explicit aspect precedence rules. Even if there are no conflicting aspects, it introduces stronger determinism into system and avoids possible side effects when the code changes.
7. Avoid overuse of wildcards in pointcut patterns. Prefer specifying exact join points or join point matching based on annotations. It significantly reduces possibility of unintended weaving, as requires simple but yet conscious action to apply aspect to a certain join point.
8. Avoid control-flow dependant pointcuts. Decision to apply them should be taken with special care and only if it significantly simplifies design of application taking into account that it will make the code slower and much more difficult to test.

We believe that this list is not complete and new principles will be discovered for more specific situations. However those alone help avoiding many typical problems arising in TDD using AspectJ.

5. Conclusion

Traditional TDD cannot be applied to AspectJ development without change due to many contradictions. It however does not mean that TDD is not applicable at all. It needs just some adaptation work. We provided the most apparent problems that arise when AspectJ is used in TDD process. List of principles we provided helps to soften many of those problems. We believe however that this list is not complete and more practical experience data is necessary to improve it.

These principles are just slight adaptations of TDD, they do not change fundamental TDD process as it is described in [3]. Further investigation is required to find out an improved form of TDD that could be more appropriate for AOP. Probably TDD needs more fundamental changes in its principles in order to be effectively used in AOP or even adapted outside OO-world.

References

1. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-oriented programming. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, pages 220–242, Springer-Verlag.
2. Laddad, R, 2003. AspectJ in Action: Practical Aspect-Oriented Programming. Manning.
3. Beck, K., 2002. Test Driven Development: By Example. Addison-Wesley.
4. Loughran, S., Hatcher, E., 2007. Ant in Action, 2nd Edition. Manning.
5. Setty, R., Dyer, R., Rajan, H., 2008. Weave Now or Weave Later: A Test Driven Development Perspective on Aspect-oriented Deployment Models. Technical Report 08-02, Computer Science, Iowa State University.
6. Hilsdale, E., Hugunin, J., 2004. Advice weaving in AspectJ. Proceedings of the 3rd international Conference on Aspect-Oriented Software Development. Lancaster, UK.
7. Mortensen, M., Alexander, R.T., 2005. An approach for adequate testing of AspectJ programs. Proc. 1st Workshop on Testing Aspect-Oriented Programs (WTAOP), Chicago, IL, USA.
8. Störzer, M., Krinke, J., 2003. Interference analysis for AspectJ. In Workshop on Foundations of Aspect-Oriented Languages (FAOL 2003).
9. McEachen, N., Alexander, R.T., 2005. Distributing classes with woven concerns: an exploration of potential fault scenarios. Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD 2005, Chicago, IL, USA.
10. Alexander, R.T, Bieman, J.M., Andrews, A.A., 2004. Towards the Systematic Testing of Aspect-Oriented Programs. Technical Report CS-4-105, Colorado State University.

Suhorukovs A. Testu vadāmās aspektorientētās izstrādes problēmas
Testu vadāmā izstrāde un aspektorientētā programmēšana ir salīdzinoši jaunas izstrādes metodes, katrai no tām ir savas priekšrocības. Taču aspektorientētās valodas, piemēram, AspectJ, izmantošana testu vadāmajā izstrādē noved pie jauniem problēmu veidiem, kas neizpaužas, ja tās pieejas tiek izmantotas atsevišķi. Šādas problēmas rodas, galvenokārt, tāpēc, ka aspektorientētā paradigma pārkāpj iekapsulēšanas principu – klases uzvedība tiek implementēta nevis tikai pašā klasē, kā tas notiek tradicionālajā objektorientētajā programmēšanā, bet var tikt būtiski ietekmēta ar vairākiem ārējiem aspektiem. Tradicionālie vienībtesti, kas tiek veidoti vienas atsevišķas vienības testēšanai, vairs nav pietiekami, jo testējama uzvedība ir sadalīta vienlaicīgi starp vairākām vienībām, kas padara testu projektēšanu par sarežģītāku. Lai efektīvi izmantotu abas šīs metodes kopā, šādas problēmas ir jāidentificē, kā arī jāatrod to apiešanas ceļi. Šis raksts ir veltīts šādām problēmām, nozīmīgākās no kurām ir aprakstītas. Problēmu apraksti ir balstīti uz pieredzi ar AspectJ izmantošanu, taču lielākā to daļa ir raksturīga jebkurai aspektorientētajai valodai ar līdzīgām iespējām. Tiek piedāvāti daži empīriski atrasti principi, kas palīdz mīkstināt šo problēmu ietekmi.

Sukhorukov A. Problems of test-driven aspect-oriented development
Test-driven development and aspect-oriented programming are relatively new development techniques each having its own benefits. However, using aspect-oriented language like AspectJ in test-driven development leads to new types of problems that do not appear if these two approaches are applied separately. These problems arise mainly because aspect-oriented paradigm breaks encapsulation principle – behaviour of class is not implemented just in class itself like in traditional object-oriented programming but can be heavily affected by many external aspects. Traditional unit tests designed to test a single unit are not sufficient any more because behaviour being tested is distributed through several units at once, so test design becomes more complicated. In order to efficiently use these two techniques together such problems have to be identified and ways to workaround them have to be discovered. This paper focuses on those problems, describing the most notable ones. Problem descriptions are based on experience with AspectJ, but most of them are common to any aspect-oriented language with similar features. Some empirically discovered principles are provided that help softening negative impact of those problems.

Сухоруков А. Проблемы управляемой тестами аспектно-ориентированной разработки
Управляемая тестами разработка и аспектно-ориентированное программирование – сравнительно новые методы разработки, имеющие свои преимущества. Однако использование аспектно-ориентированного языка, такого как AspectJ, в управляемой тестами разработке приводит к новым типам проблем, которые не проявляются, если эти подходы используются по отдельности. Такие проблемы возникают в основном из-за того что аспектно-ориентированная парадигма нарушает принцип инкапсуляции – поведение класса реализуется не только в самом классе, на него могут серьезно воздействовать многие внешние аспекты. Традиционные блочные тесты, проектируемые для тестирования отдельных блоков, в данном случае недостаточны, поскольку тестируемое поведение класса распределено между несколькими блоками одновременно, поэтому проектирование тестов усложняется. Чтобы совместно применять эти два подхода эффективно, такие проблемы должны быть идентифицированы и найдены пути их устранения. Статья посвящена таким проблемам, самые существенные из которых здесь описываются. Описания проблем основано на опыте использования AspectJ, но большинство из них относится к любому аспектно-ориентированному языку с похожими возможностями. Приводятся некоторые эмпирически найденные принципы, которые помогают смягчить эффект указанных проблем.