9 Case Studies in Aspect Mining

Silvia Breu
Lehrstuhl für Softwaresysteme, Universität Passau
breu@fmi.uni-passau.de

9.1 Motivation

A major problem in software re-engineering based on aspect-oriented principles lies in finding and isolating crosscutting concerns. This task is called *aspect mining*. The detected concerns can be re-implemented as separate aspects. This reduces the complexity, and improves the comprehensibility of software systems. Thus, aspects facilitate software maintenance and extension. Aspect mining can also provide us with insights that enable us to classify common aspects which occur in different software systems, such as logging, timing, and communication.

Several approaches based on static program analysis techniques have been proposed for aspect mining [vDMM03]. Our approach [BK03] is the first dynamic program analysis approach. It mines for aspects based on program traces that are generated during program execution, and monitor the run-time behaviour of a software system. These traces are then investigated for recurring execution relations. Different constraints specify when an execution relation is "recurring", such as the requirement that the relations have to exist more than once or even in different calling contexts in the program trace. The dynamic analysis approach has been chosen because it is a very powerful way to make inferences about a system: It dynamically monitors actual program behaviour (run-time behaviour) instead of potential behaviour—as static program analysis does.

The approach has been implemented in a prototype called DynAMiT (*Dyn*amic *A*spect *Mi*ning *T*ool) and evaluated in several case studies over systems with more than 80 kLoC. As we will see, the technique is able to identify *automatically* both seeded and existing crosscutting concerns in software systems.

9.2 Case Study "AspectJ example telecom"

A case study has been conducted in order to verify how successful the developed analysis approach can be applied to a new field: Can DynAMiT also detect crosscutting concerns in Java programs which are already extended by aspects written in AspectJ?

For that purpose the telecom example (included in the distribution of AspectJ) has been chosen: There, people can make telephone calls with different connection types (local and long-distance). The simulation can be executed at three different levels: `BasicSimulation` just performs the calls with the basic functionality needed for making phone calls (call, accept, hang up etc.) `TimingSimulation` is the extension with a timing aspect which keeps track of a connection's duration and cumulates a customer's connection durations. `BillingSimulation` is a further extension with a billing aspect that adds functionality to calculate charges for phone calls of each customer based on connection type and duration. Due to space limitations we describe only some of the detected aspect candidates.

Analysis Results for BasicSimulation. This analysis provides us with crosscutting concerns as well as some insights in the usual sequence of actions in phone calls. The application of the execution relation constraints tells us that the simulation visualises the steps a customer is doing, such as calling someone, answering the phone or hanging up. When someone calls another person, the addition of the call to the pipeline of the customer is done as last thing. The same applies when a called person picks up the phone—the call is added to his pipeline. Another detected crosscutting concern reveals that after a customer has hung up, the call has to be removed from his pipeline.

Analysis Results for TimingSimulation. The resulting sets of aspect candidates in the `TimingSimulation` include those already detected in the `BasicSimulation` (except for some re-namings). The analysis also discovers functionality added by the application of the timing aspect. For instance, the introduction of new fields is discovered: A timer which keeps track of connection times is needed. Before the timer can be started, stopped (or asked for the time), one has to get hold of the timer belonging to the correct connection. Additionally, the timer is needed after a connection is completed or dropped, and when caller and receiver of a connection are determined.

Analysis Results for BillingSimulation. In the third simulation (which includes the timing aspect and a billing aspect on top of that) additional crosscutting concerns introduced by the billing aspect are found. We find, for instance, that before the call rate (local or long distance) for a connection or the receiver of a connection is determined, the current time of the timer is needed. Furthermore, the analysis tells us that—after the correct call rate for a connection is determined—the connection's payer has to be found out. After the paying customer is identified, the charge for the phone call is added to that customer.

9.3 Case Study "Graffiti"

Graffiti[1] (written in Java) is an industrial-sized editor for graphs and a toolkit for implementing graph visualisation algorithms. The software system currently has about 450 interfaces and (abstract) classes, over 3.100 methods and comprises about 82.000 lines. The results of the analysis applied to the Graffiti traces are huge. Only some interesting findings get a closer look as a complete interpretation of all results would be too elaborate. Besides, it would be nearly equivalent to re-factoring the software system.

First of all, DynAMiT has detected a typical, well-known crosscutting concern: logging. Several calls to a method `format` of class `SimpleFormatter` as first and/or last call inside several set- and add-methods were found. A code investigation reveals that all executions of those methods are logged in a log-file. For that, a logger provided by Java's class `Logger` is used. Although we have not traced Java API classes, we know that the logger uses a formatter to transform the system's log messages into human readable messages. For this purpose, Graffiti provides a class `SimpleFormatter` which implements method `format`. Therefore, the analysis detects the formatting of the log-messages, and thus provides us with the information that logging exists. The crosscutting logging functionality is discovered and can be encapsulated into an aspect in a re-engineering process.

The before-aspect candidate desribed in the following is more of an informative nature. Graffiti can easily be extended with graph algorithms by writing plugins. Every algorithm has to implement method `getName` of interface `Algorithm`. If a user wants to use a certain algorithm, e.g. Dijkstra's algorithm, this algorithm has to be added following the plugin principle of Graffiti. In general, every plugin a user wants to have available is registered on startup or (dynamically) later on when loaded. Thus, for every used algorithm, an appropriate plugin has to be added. To be able to determine the kind of the plugin, and in order to have a unique string for each algorithm plugin, the algorithm's name is used, which is gained by calling method `getName` of the corresponding algorithm. Thus, DynAMiT discovers that `getName` in the appropriate algorithm class is always preceded by an execution of `getAlgorithms` of class `GenericPluginAdapter`.

[1] Graffiti version November 2003; now renamed to Gravisto. `http://www.gravisto.org`

Some other retrieved first- and last-aspect candidates tell that method `isSessionActive` of class `MainFrame` is the first method executed inside methods `isEnabled` in each of the classes `FileCloseAction`, `ViewNewAction`, and `RunAlgorithm`. We also observe that this very call is the last one inside `isEnabled` in those three classes. An investigation of Graffiti's source code confirms these analysis results: In the system's architecture it can be seen that class `FileCloseAction`, class `ViewNewAction` as well as class `RunAlgorithm` extend abstract class `GraffitiAction`. Therefore, the question arises why this functionality has not been encapsulated into `GraffitiAction` following object oriented design principles. The reason for that is quite simple: There are a lot more classes extending class `GraffitiAction` which do not have the same functionality, e.g. class `EditUndoAction` or class `ExitAction`. So, the detected pattern (either first- or last-aspect candidate) is a distinct crosscutting concern and thus a candidate for encapsulating this functionality into an aspect.

Method `isSessionActive` was also found as first-aspect candidate in a method called `update` in class `EditRedoAction` as well as in class `EditUndoAction`. A look into the code tells that `EditRedoAction` and `EditUndoAction` both extend abstract class `GraffitiAction`. So the question is again why the developers did not choose a better design. But the analysis algorithm has detected this pattern in only those two but not all of the classes which extend `GraffitiAction`. A further investigation of the source code confirms that result. This suggests that the developers have not been able to provide a different design by encapsulating this concern into the superclass without overriding methods in subclasses (which would be considered bad practice). The introduction of more inheritance levels would not cure the problem either. There is no perfect solution in the sense of OOP, especially in Java as it is not designed to provide multiple inheritance.

9.4 Conclusions

In summary we can say that the results for the three telecom simulations clearly show that the presented approach identifies basic functionality and the functionality added by the two different aspects. The analysis is performed automatically, and did not produce any false positives.

In Graffiti it can be seen that a lot of the functionality is crosscutting its architecture, e.g. actions like to open, save or edit a file or a graph, respectively. It is worth thinking about encapsulation using aspects to satisfy the need for a proper design. Of course, this decision remains to the programmer.

Acknowledgements

Thanks to Jens Dörre for his valuable comments.

Bibliography

[BK03] Silvia Breu and Jens Krinke. Aspect Mining Using Dynamic Analysis. 5. Workshop Software-Reengineering, Bad Honnef. (Published in: GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik, 23(2), pp. 21-22), May 2003.

[vDMM03] Arie van Deursen, Marius Marin, and Leon Moonen. Aspect Mining and Refactoring. In *First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.