9

Evolution Issues in Aspect-Oriented Programming

Kim Mens[1] and Tom Tourwé[2]

[1] Université catholique de Louvain, Belgium
[2] Eindhoven University of Technology, The Netherlands

Summary. This chapter identifies evolution-related issues and challenges in aspect-oriented programming. It can serve as a guideline for adopters of aspect technology to get a better idea of the evolution issues they may confront sooner or later, of the risks involved, and of the state-of-the-art in the techniques currently available to help them in addressing these issues. We focus in particular on the programming level, although some of the issues and challenges addressed may apply to earlier software development life-cycle phases as well. The discussed issues range from the exploration of crosscutting concerns in legacy code, via the migration of this code to an aspect-oriented solution, to the maintenance and evolution of the final aspect-oriented program over time. We discuss state-of-the-art techniques which address the issues of aspect exploration, extraction and evolution, and point out several issues for which no adequate solutions exist yet. We conclude that, even though some promising techniques are currently being investigated, due to the relative immaturity of the research domain many of the techniques are not out of the lab as yet.

9.1 Introduction

Just like the industrial adoption of object orientation in the early nineties led to a demand for migrating software systems to an object-oriented solution—triggering a boost of research on software evolution, reverse engineering, reengineering and restructuring—the same is currently happening for the aspect-oriented paradigm. *Aspect-oriented software development* (AOSD) is a novel paradigm that addresses the problem of the *tyranny of the dominant decomposition* [490]. This problem refers to a software engineer's inability to represent in a modular way certain concerns in a given software system, when those concerns do not fit the chosen decomposition of the software in modules. Such concerns are said to be *crosscutting* as they cut across the dominant decomposition of the software. Consequently, the source code implementing crosscutting concerns gets *scattered* across and *tangled* with the source code of other concerns. Typical examples of crosscutting concerns are tracing [88], exception handling [91] or transaction management [164].

In absence of aspect-oriented programming techniques, crosscutting concerns often lead to duplicated code fragments throughout the software system. As argued by

T. Mens, S. Demeyer (eds.), *Software Evolution*.
DOI 10.1007/978-3-540-76440-3, © Springer 2008

Koschke in Chapter 2, duplication tends to have a negative impact on software quality. Crosscutting concerns are thus believed to negatively affect evolvability, maintainability and understandability, because understanding and changing a crosscutting concern requires touching many different places in the source code. Although the few studies that have explored this negative relation between crosscutting concerns and software quality do not contradict this claim (see Subsection 9.3.3), there is currently not enough empirical evidence of this claim yet. Nevertheless, aspect-oriented programming (AOP) does propose a solution to this acclaimed problem by introducing the notion of *aspects*, which are designated language constructs that allow a developer to localise a concern's implementation, and thus improve modularity, understandability, maintainability and evolvability of the code.

Adopting a new software development technology brings about particular risks, however, and aspect-oriented programming forms no exception. In his article "AOP myths and realities" [302], Laddad refutes 15 often-heard 'myths' that are said to hinder the adoption of AOP, such as "debugging aspects is hard" and "aspects cannot be unit tested". As convincing as his arguments may be, they mainly focus on currently existing AOP technology, and try to prove it sufficiently mature for widespread adoption.

However, a much larger opportunity and obstacle for adopting AOP technology is the fact that it has to be introduced into *existing* software systems. Most software systems today are not developed from scratch, but rather are enhanced and maintained *legacy systems*. Awareness is growing that aspects can and should be used not only to modularise crosscutting concerns in newly developed software; the vast majority of existing software systems suffers from the tyranny of the dominant decomposition as well, making them hard to maintain and evolve. As such, legacy software systems form an important range of applications which may benefit from the advantages that AOP claims to offer. We predict that real widespread adoption of AOP will only be achieved if the risks and consequences of adopting AOP in existing software are studied, and if the necessary tools and techniques are available for dealing with those risks.

This chapter addresses the issues and challenges related to such adoption of AOP from a software evolution perspective. We present a series of questions and challenges that are relevant to new adopters of aspect technology in an evolutionary context and summarise existing research that addresses some of these issues. In this way, potential adopters are given an overview of the existing research efforts and can assess the usefulness and maturity of existing tools and techniques. Additionally, fellow researchers are presented with an overview of the research domain, which can help them in posing new research questions and tackling problems still left open.

As illustrated by Figure 9.1 we distinguish 3 different phases: *aspect exploration*[3], *aspect extraction* and *aspect evolution*.

[3] As in the survey paper [272], we deliberately reserve the term *aspect mining* for the more specific activity of (semi-)automatically identifying aspect candidates from the source code of a software system. We propose the term *aspect exploration* as a more general term which does not imply these restrictions and also encompasses manual approaches as well as techniques that try to discover aspects from earlier software life-cycle artefacts.

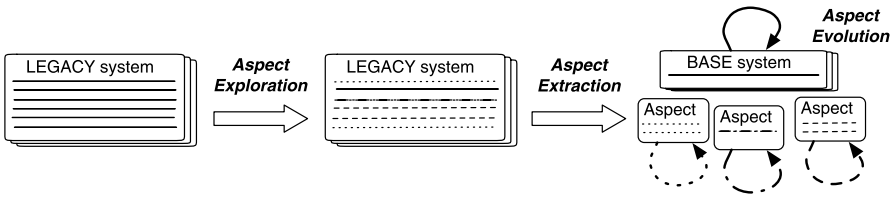

Fig. 9.1. Software evolution and AOSD

Aspect exploration. Before introducing aspects in existing software, one should explore whether that software actually exhibits any crosscutting concerns that are worthwhile being extracted into aspects. The tyranny of the dominant decomposition implies that large software is likely to contain crosscutting concerns. During the aspect exploration phase we try to discover *aspect candidates* in the software, i.e., we try to discover what the crosscutting concerns are, where and how they are implemented, and what their impact on the software's quality is.

Aspect extraction. Once the crosscutting concerns have been identified and their impact on software quality has been assessed, we can consider migrating the software to an aspect-oriented version. We refer to this activity as aspect extraction. If we do decide to migrate the software towards an aspect-oriented solution, we need a way of turning the aspect candidates, i.e., the crosscutting concerns that were identified in the exploration phase, into actual aspects. At the same time, we need techniques for testing the migrated software to make sure that the new version of the software still works as expected, as well as techniques to manage the migration step, for example to ensure that we can still keep on using the software during the transition phase.

Aspect evolution. According to Belady and Lehman's first *law of software evolution* [320], every software system that is used will continuously undergo changes or become useless after a period of time. There is no reason to believe that this law does not hold for aspect-oriented software too. But to what extent is evolution of aspect-oriented software different from evolution of traditional software? Can the same techniques that are used to support evolution of traditional software be applied to aspect-oriented software? Do the new abstraction mechanisms introduced by AOP give rise to new types of evolution problems that require radically different solutions?

Before taking a closer look at each of these phases, in Section 9.2 we provide a brief introduction to AOP for the non-initiated readers. In the subsequent three sections we then discuss the challenges, risks and issues related to each of the three phases of aspect exploration, extraction and evolution. Each section has the same format:

1. *Rationale:* A more precise description and definition of the activity, and why it is important and needed, from an end-user perspective.

2. *Challenges and risks:* What are the challenges and risks that need to be dealt with or will be encountered when conducting this activity?
3. *Existing techniques:* What existing techniques help in supporting the activity? What challenges do these techniques address and to what extent?
4. *Open issues:* To what extent do available techniques address the aforementioned risks and challenges? What challenges and risks are not addressed by any technique?
5. *Case study:* To obtain a better intuition of some of the issues, challenges and risks pertaining to the activity, we discuss some of our experiences gained on a realistic case.

9.2 Aspect-Oriented Programming

The goal of aspect-oriented programming is to provide an advanced modularisation scheme to separate the core functionality of a software system from system-wide concerns that cut across the implementation of this core functionality. To this extent, AOP introduces a new abstraction mechanism, called an aspect. An aspect is a special kind of module that represents a crosscutting concern. Aspects are defined independently from the core functionality of the system and integrated with that *base program* by means of a dedicated *aspect weaver*, a dedicated tool similar to a compiler that merges aspect code and base code in the appropriate way. Figure 9.2 illustrates this idea.

In most current-day aspect languages, of which AspectJ is the most well-known, aspects are composed of *pointcuts* and *advices*. Whereas advices correspond to the

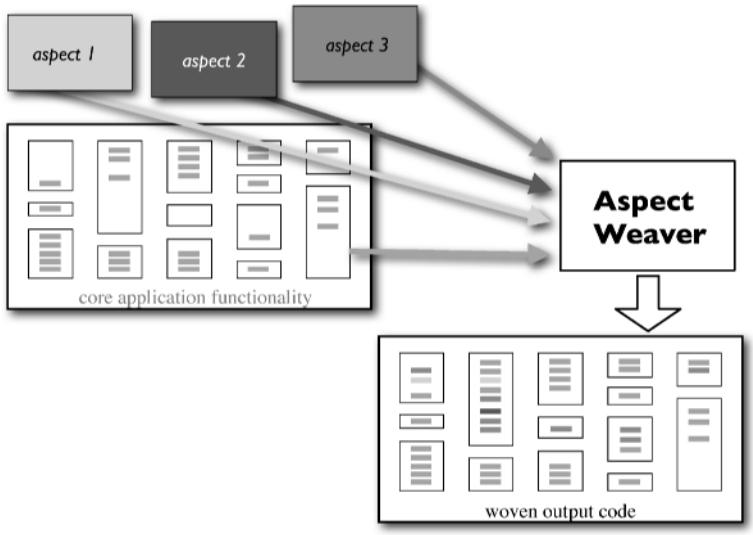

Fig. 9.2. The AOP idea

```
class Point extends Shape {

  public void setX(int x) throws IllegalArgumentException {
    if ( x < MIN_X || x > MAX_X )
       throw new IllegalArgumentException("x is out of bounds.");
    ...
  }
  public void setY(int y) throws IllegalArgumentException {
    if ( y < MIN_Y || y > MAX_Y )
       throw new IllegalArgumentException("y is out of bounds.");
    ...
  }
}

class FigureElement extends Shape {

  public void  setXY(int, int) throws IllegalArgumentException {
    if ( x < MIN_X || x > MAX_X )
       throw new IllegalArgumentException("x is out of bounds.");
    if ( y < MIN_Y || y > MAX_Y )
       throw new IllegalArgumentException("y is out of bounds.");
    ...
  }
}
```

Fig. 9.3. Bounds checking concern for moving points

code fragments that would crosscut an entire program, pointcuts correspond to the locations in the source code of the program where the advice will be applied (i.e., where the crosscutting code will be woven). A pointcut essentially specifies a set of *joinpoints*, which are well-defined locations in the structure or execution flow of a program where an aspect can weave in its advice code. Typical AspectJ joinpoints are method invocations or field accesses, for example.

To illustrate these notions, consider the Java code fragments in Figure 9.3 that implement bounds checking for operations that move points in a graphical drawing application. Because the application has been decomposed into classes and methods according to the different graphical elements that can be drawn, such as points and figures, the bounds checking code cuts across this dominant decomposition and does not align with these classes and methods. This results in scattering and tangling: bounds checking code is implemented in different methods and classes, which clutters and interferes with the other code implemented by those methods and classes.

To modularise this bound checking concern, a `PointBoundsChecking` aspect can be defined as in Figure 9.4. For didactic purposes, we kept the definition of this aspect very simple; a more intelligent definition is given further on. This code defines two pointcuts: the `setX` pointcut captures all executions of methods that change the `x` value, while the `setY` pointcut captures all executions of methods that change the `y` value. In addition to these pointcuts, the advice code is defined: the first ‘before’

```
aspect PointBoundsChecking {

  pointcut setX(int x):
    (execution(void FigureElement.setXY(int, int)) && args(x, *))
    || (execution(void Point.setX(int)) && args(x));

  before(int x): setX(x) {
    if ( x < MIN_X || x > MAX_X )
       throw new IllegalArgumentException("x is out of bounds.");
  }

  pointcut setY(int y):
    (execution(void FigureElement.setXY(int, int)) && args(*, y))
    || (execution(void Point.setY(int)) && args(y));

  before(int y): setY(y) {
    if ( y < MIN_Y || y > MAX_Y )
       throw new IllegalArgumentException("y is out of bounds.");
  }
}
```

Fig. 9.4. An extensional bounds checking aspect

advice specifies that before every execution of a method captured by the `setX` pointcut, the appropriate bound needs to be checked. The advice for the `setY` pointcut is defined analogously.

Note that the advice code references an actual parameter of the method it advices in order to check its value, i.e., the `x` and `y` parameters. The pointcut exposes this parameter to the advice code, by providing an appropriate name. Moreover, when the method has more than one parameter, as is the case for the `setXY` method, the pointcut needs to make sure that the appropriate parameter is exposed. All this is achieved by using the `args` construct. For example, `args(x, *)` exposes the variable `x`, which corresponds to the first argument of the method, to the advice code. Similarly, `args(*,y)` exposes a variable `y` that corresponds to the last argument of the method.

The `setX` and `setY` pointcuts defined in Figure 9.4 are examples of *extensional* or *enumeration-based* pointcuts, since they explicitly enumerate the signatures of *all* methods they need to capture. Such pointcuts are brittle and can break easily when the base program evolves. More robust pointcut definitions can be obtained by mentioning explicitly only the information that is absolutely required and using wildcard patterns to hide implementation details that do not matter. For instance, in Figure 9.5 we use the wildcard `*` to hide the exact return types of the method joinpoints and the pattern `Shape+` to hide the precise name of the implementing class while still requiring that it belongs to the `Shape` class hierarchy. However, we do not use a wildcard in the method names, but match against the exact names, because their intension-revealing nature is likely to help us in capturing the correct pointcuts. Using more abstract names might result in accidentally capturing the wrong joinpoints.

```
pointcut setX(int x):
    (execution(* Shape+.setXY(int, int)) && args(x, *))
     || (execution(* Shape+.setX(int)) && args(x));

pointcut setY(int y):
    (execution(* Shape+.setXY(int, int)) && args(*, y))
     || (execution(* Shape+.setY(int)) && args(y));
```

Fig. 9.5. An intensional pointcut definition for the bounds checking aspect

Pointcuts that use wildcards patterns or other mechanisms to abstract over certain implementation details are called *intensional* or *pattern-based* pointcuts. They are said to be more robust toward evolution, because of the abstractions they use. For example, when a `setXY` method would be added to another class in the `Shape` hierarchy, it will still be adviced and the bounds of its parameters will be checked. Similarly, when the return type of such a method would change, it would still be captured by the pointcut.

In summary, pointcuts, whether they are extensional or intensional, specify those places in the code or its execution where the advice code needs to be woven. This means that aspects are not *explicitly* invoked by the program. The *base program* (i.e., the program without the aspects) is not aware of the aspects that apply to it. Instead, it is the aspects themselves that specify when and where they act on the program. This has been referred to as the *obliviousness* property of aspect orientation [176], and is one of the most essential characteristics of an aspect-oriented programming language.

9.3 Aspect Exploration

Migrating a legacy software system into an aspect-oriented one is a non-trivial endeavour. The sheer size and complexity of many existing systems, combined with the lack of documentation and knowledge of such systems render it practically infeasible to *manually* transform their crosscutting concerns into aspects. To alleviate this problem, a growing body of research exists that proposes a number of tools and techniques to assist software engineers in semi-automatically migrating crosscutting concerns to aspects. Most of these approaches distinguish two phases in this migration process: aspect exploration and aspect extraction. Whereas Section 9.4 will focus on the aspect extraction phase, the current section discusses issues and challenges related to aspect exploration, as well as existing techniques to address some of those issues.

9.3.1 Rationale

We define *aspect exploration* as the activity of identifying and analysing the crosscutting concerns in a non aspect-oriented system. A distinction can be made between

manual exploration supported by special-purpose browsers and source-code navigation tools, on the one hand, and aspect mining techniques that try to automate this process of aspect discovery and propose their user one or more aspect candidates, on the other hand.

9.3.2 Challenges and Risks

What (Kind of) Crosscutting Concerns Can Be Discovered?

Examples of crosscutting concerns that are often mentioned in literature, and exemplified by small-scale example projects, include simple and basic functionalities like tracing, logging or precondition checking. Do such simple concerns actually occur in industrial code? Is AOP only suited to implement such simple concerns? Do industrial software systems contain more complex crosscutting concerns? How good is AOP at tackling those?

How Are Crosscutting Concerns Implemented in Absence of Aspects?

Since crosscutting concerns in a traditional software system are per definition not well-localised, they need to be implemented over and over again. To minimise this implementation overhead, developers tend to rely on a variety of programming idioms and naming and coding conventions. What (kind of) crosscutting concerns are implemented by which programming idioms and conventions?

(How) Do Crosscutting Concerns Affect Software Quality?

When crosscutting concerns occur in a software system, (how) do they affect the quality of that software? How can we measure their impact on quality factors like understandability, maintainability and adaptability? How can these measures help us assess whether extracting the concerns into aspects is beneficial?

How to Find Where Crosscutting Concerns Are Located in the Code?

When we want to turn a crosscutting concern into an aspect, we need to know where exactly it is located in the code. This knowledge is important to determine an appropriate pointcut for the extracted aspect. How can we be sure that we have found all relevant crosscutting concerns, that we have covered them completely, that there are no false positives or negatives?

9.3.3 Existing Research

Over the last few years, aspect exploration has become quite an active research domain and a whole range of different approaches, techniques and tools for supporting or automating the activity of aspect exploration have been proposed.

Crosscutting Concerns in Practice

In most research papers, tutorials and textbooks on AOP, the examples of crosscutting concerns given show simple and basic concerns (like logging), implemented in small-scale software systems or as illustrative examples only. Consequently, one may wonder whether more complex concerns exist, whether such concerns actually occur in industrial software systems, and whether AOP is only suited for addressing simple small-scale crosscutting concerns.

Several examples of more complex crosscutting concerns occurring in real-world software have been described in literature as well, however. Bruntink et al. [91] discuss how *exception handling* is a crosscutting concern in a large-scale embedded software system, and show how its implementation is prone to errors. Colyer and Clement [122] present a study in which they separated support for Enterprise JavaBeans from the other functionality contained within an application server of several millions of lines of code. Coady et al. [118] refactored the *prefetching* concern from the FreeBSD UNIX operating system kernel.

This shows that complex crosscutting concerns do occur in practice, and that large-scale industrial software systems also exhibit such concerns. Hence, when exploring a system for crosscutting concerns, one should not limit the search for well-known concerns only, but one should look for scattered and tangled code of any nature.

Implementing Crosscutting Concerns

Since crosscutting concerns are not well-modularised, to reduce the effort of implementing, maintaining and evolving them developers rely on *structural regularities* like naming and coding conventions, programming idioms and design patterns.

Bruntink et al. [88, 91, 90] discuss an industrial software system that implements crosscutting concerns by means of programming idioms. Idioms are simple code templates for the implementation of a concern, that are prescribed in architecture manuals and that a developer can copy-paste and then adapt to his particular needs and wishes. The bounds checking concern presented in the previous section is an example of such an idiom: all methods that move points need to test the new value first, and need to raise the appropriate exception whenever the value is not within a specified range. As the authors observed, such an approach leads to code duplication, is prone to errors and is time- and effort-consuming.

In [364], we observed that, when implementing crosscutting concerns, developers often rely on naming conventions and thus provide valuable hints about the locations of such crosscutting concerns. We studied the JHotDraw application framework and grouped classes and methods that share identifiers in their name. This lightweight approach turned out to be capable of detecting several interesting concerns, such as for example an *Undo* and a *Persistence* concern. Shepherd et al [460] discuss a similar approach based on *lexical chaining*, a natural language processing technique, to identify crosscutting concerns in the PetStore application.

Marin et al. [347] introduce *crosscutting concern sorts*, a classification system for crosscutting functionality. For each sort of crosscutting concern they indicate how it could be implemented by using traditional modularisation mechanisms. For example, they define a *role superimposition* sort, a construction that implements a specific secondary role or responsibility for a class, and observed that this is often implemented by using interfaces (in Java). Other examples of implementation techniques that are used to implement sorts are several design patterns, such as the Observer, Decorator and Adapter design pattern, and a design by contract approach in a language that supports explicit pre- and postconditions.

Again, when exploring a software system for crosscutting concerns and reasoning about them, discovering such regularities helps. Additionally, such information is also useful for program comprehension, as it provides interesting information on how the software is structured.

Crosscutting Concerns and Software Quality

Few studies exist that explore the relation between crosscutting concerns and software quality.

Bruntink et al. [91, 90] assessed the quality of the *exception handling* and *parameter checking* concerns, implemented by means of idioms in an industrial context. They observed that the implementation of both concerns exhibited several faults, but were unable to conclude whether these were due to the crosscutting nature of the implementation or to the inherent complexity of the concern itself. Moreover, they acknowledge that faults are not failures, and hence they are not sure about the severity of the discovered faults.

Kulesza et al. [300] performed a study in which they computed metrics for both object-oriented and aspect-oriented versions of a medium-scale software system, and compared them in order to quantify the difference. They observed that the aspect-oriented versions resulted in fewer lines of code, improved separation of concerns, weaker coupling and lower intra-component complexity. However, they also found that the number of operations and components in the aspect-oriented version increased, and observed a lower cohesion for the aspect-oriented components.

A number of authors have studied whether the implementation quality of popular design patterns could be improved by using aspect-oriented programming [222, 193]. It turns out that such an improvement can be achieved, and comes primarily from enhanced modularisation, which makes the implementation more localised, reusable, composable and pluggable. However, these results have been observed in small and illustrative cases only, and no evidence has yet been provided that these results can be generalised to large-scale industrial software.

Gibbs et al. [198] conducted a broad case study where they compared the maintainability and evolvability of a version of a software system that was restructured with traditional abstraction mechanisms against a version of that same system which was restructured by means of aspects. They then considered a ‘big bang’ type of evolution that implied many changes to the code being crosscut. Their conclusion was that, in the particular case they studied, overall the aspect-oriented version performed

either better or not worse than the other (non aspect-oriented) version at dealing with those changes.

Locating Crosscutting Concerns

Kellens et al. [272] distinguish three main categories of techniques that can help in locating the crosscutting concerns in a software system:

Early aspect discovery techniques: Research on 'early aspects' tries to discover aspects in the early phases of the software life-cycle [35] like requirements and domain analysis [34, 431, 492] and architecture design [42]. Although these techniques can help to identify some of the crosscutting concerns in a software system, early aspect discovery techniques may be less promising than approaches that focus on source code, when applied to existing software systems where requirements and architecture documents are often outdated, obsolete or no longer available.

Dedicated browsers: A second class of approaches are the advanced special-purpose code browsers that aid a developer in manually navigating the source code of a system to explore crosscutting concerns. These techniques typically start from a location in the code, a so-called "seed", as point-of-entry from which they guide their users by suggesting other places in the code which might be part of the same concern. This way, the user iteratively constructs a model of the different places in the code that make up a crosscutting concern. Examples of such approaches are Concern Graphs [442], Intensional Views [363], Aspect Browser [211], (Extended) Aspect Mining Tool [221, 563], SoQueT [347] and Prism [564].

Aspect mining techniques: Complementary to dedicated browsers, a number of techniques exist that have as goal to automate the aspect identification process and that propose their user one or more aspect candidates. To this end, they reason about the system's source code or execution traces. All techniques seem to have at least in common that they search for symptoms of crosscutting concerns, using either techniques from data mining and data analysis like formal concept analysis and cluster analysis, or more classic code analysis techniques like program slicing, software metrics and heuristics, clone detection and pattern matching techniques, dynamic analysis, and so on. For an extensive survey and an initial classification of aspect mining techniques which semi-automatically assist a developer in the activity of mining the crosscutting concerns from the source code of an existing system, we refer to [272].

9.3.4 Open Issues

From the discussions in the previous subsection, it is clear that several researchers have studied complex crosscutting concerns that occur in real-world industrial software systems, and that they are starting to get an idea about how such concerns

are implemented in the absence of aspect-oriented programming techniques. Nevertheless, more such studies on industrial-size software systems would be welcome. Similarly, although preliminary research attempts have been undertaken for the remaining two challenges (i.e., how crosscutting concerns affect software quality and how to locate them in the software), more research is needed in order to come up with satisfying answers and solutions.

For example, more empirical work and quantitative studies are needed on how crosscutting concerns affect software quality. The impact of crosscutting concerns on software quality factors like evolvability is not yet clear, and has been investigated mostly on small-scale example software only. Part of the problem stems from the fact that AOP is a relatively young paradigm, and hence little historical information (in the form of revision histories etc.) is available for study. Another problem is that, more often than not, a traditional version and an AOP version of the same software system are not available, making it hard to conduct objective comparisons.

As for the identification of crosscutting concerns, all known techniques are only partly automated and still require a significant amount of user intervention. In addition, most aspect mining techniques are only academic prototypes and, with few exceptions, have not been validated on industrial-size software yet. Although this may hinder industrial adoption, the existence of such techniques is obviously a step forward as opposed to having no tool support at all. Another issue with applying automated aspect mining techniques is that preferably the user should have some knowledge about the system being mined for aspects. Indeed, different aspect mining techniques rely on different assumptions about *how* the crosscutting concerns are implemented.

9.3.5 Exploration in Practice

As a concrete practical case study of aspect exploration, in this subsection we summarise a larger experiment that was conducted by Bruntink et al. [92, 93] to evaluate the suitability of clone detection techniques for automatically identifying crosscutting concern code. They considered a single component of a large-scale, industrial software system, consisting of 16,406 non-blank lines of code.

In a first phase, the programmer of this component manually marked five different concerns that occur in it, consisting of 4,182 lines of code, or 25,5% of the total lines of code. The concerns that were considered were memory handling, null pointer checking, range checking, exception handling and tracing. The details are in the second column of Table 9.1.

In a second phase, three different clone detection techniques were applied to the component: an AST-based, a token-based and a PDG-based one. In order to evaluate how well each of the three techniques succeeded in finding the code that implemented the five crosscutting concerns, the third phase then consisted of measuring precision and recall of the results of each of these clone detection techniques with respect to the manually marked occurrences of the different crosscutting concerns. *Recall* was used to evaluate how much of the code of each crosscutting concern was found by each clone detector, while *precision* was used to determine the ratio of

Table 9.1. Line counts and average precision for the five concerns

Concern	Line Count (%)	AST-based	Token-based	PDG-based
Memory handling	750 (4.6%)	.65	.63	.81
Null pointer checking	617 (3.8%)	.99	.97	.80
Range checking	387 (2.4%)	.71	.59	.42
Exception handling	927 (5.7%)	.38	.36	.35
Tracing	1501 (9.1%)	.62	.57	.68

crosscutting concern code to code unrelated to the crosscutting concern found. Table 9.1 shows the average precision of the three clone detection techniques for each of the five concerns considered, whereas Table 9.2 shows their recall.

The results of this experiment were rather disparate. For the null pointer checking concern, which is somewhat similar to the bounds checking example presented earlier, all clone detectors obtained excellent results, identifying all concern code at near-perfect precision and recall, as can be seen from the corresponding rows in Table 9.1 and 9.2.

For the other concerns, such as the exception handling concern, none of the clone detectors achieve satisfying recall and precision, as can be seen from the corresponding rows in Tables 9.1 and 9.2. It appeared that this was related to the amount of tangling of the concerns. Clone detectors achieved higher precision and recall for concerns that exhibited relatively low tangling with other concerns or with the base code, than for concerns that exhibited high tangling.

This experiment illustrates several of the issues identified in subsection 9.3.2. First of all, it shows that simple concerns, such as logging, as well as more complex concerns, such as exception handling, are present in industrial software systems. Second, the experiment shows a particular way of implementing crosscutting concerns in the absence of an aspect-like language constructs: cloning small pieces of idiomatic code. This knowledge was used to verify whether clone detection techniques can be used to identify *where* crosscutting concerns are implemented. Last, the experiment shows some of the effects of crosscutting concerns on software quality. In particular, it confirms the common belief that crosscutting concerns are, at least in some cases, implemented by using similar pieces of code, that are scattered throughout the software. It also shows that up to 25% of the code can be attributed to crosscut-

Table 9.2. Recall for the each of the clone detection techniques on the five concerns

Concern	AST-based	Token-based	PDG-based
Memory handling	.96	.95	.98
Null pointer checking	1.0	1.0	1.0
Range checking	.89	.96	.92
Exception handling	.79	.97	.95
Tracing	.76	.85	.90

ting concern code. The negative effect of code duplication on software quality, and in particular on maintainability and evolvability, has already been investigated (see Chapter 2).

9.4 Aspect Extraction

Once we have identified the crosscutting concerns and have obtained an idea of their impact on code quality, a decision needs to be made whether or not to extract the concern code into aspects. Concerns which occur in only a few places or with limited scattering and tangling, may be less important to extract into aspects than concerns that have a high impact on software quality factors like understandability, modularity and maintainability. You should not feel compelled to migrate towards aspects if there is no real need to. It may be that aspect exploration revealed that there are no significant opportunities for introducing aspects, or that there is no clear evidence that introducing them will improve the quality of your code. Also, even if during aspect exploration some interesting crosscutting concerns were discovered, maybe you are happy with just documenting these crosscutting concerns, and keeping them in sync with the code, using a dedicated environment based on *multi-dimensional separation of concerns* [490], *concern graphs* [442] or *intensional views* [363]. However, when you do decide that it would be useful to actually turn the identified crosscutting concerns into aspects, then you enter the aspect extraction phase.

9.4.1 Rationale

Aspect extraction is the activity of separating the crosscutting concern code from the original code, by moving it to one or more newly-defined aspects, and removing it from the original code. Since an aspect is typically defined as a collection of pointcuts and associated advice code, extraction entails the identification of suitable pointcuts and the definition of the appropriate advice code corresponding to the crosscutting concern code.

Although aspect extraction is often referred to as *aspect refactoring* in existing literature, we believe that term to be ambiguous. Indeed, Fowler [183] defined refactoring as "the process of modifying source code without changing its external behaviour". When applying this definition to aspect-oriented programs, the term "source code" could either refer to the code from which an aspect is extracted, or to the code of an existing aspect that evolves. Therefore, we will use the term *aspect extraction* for the activity of turning a traditional crosscutting concern into an aspect and reserve the term *aspect refactoring* for the activity of refactoring an already existing aspect.

Research in aspect extraction thus focusses on how to automate the activity of extracting aspects from existing source code. Only with an automated approach, an extraction that is both *efficient* and *correct* can be achieved. Existing software systems often consist of millions of lines of code, and a real-world crosscutting concern thus easily consists of thousands of lines of code. Manually extracting aspects from these

crosscutting concerns, if feasible at all, would not only be very time-consuming, but prone to errors as well. A correct aspect weaves the appropriate code at the appropriate joinpoints, and hence requires correct advice code and correct pointcuts. These are hard to construct, given the scattered and tangled nature of crosscutting concerns, and the size of current-day software systems.

9.4.2 Challenges and Risks

In order to be able to extract crosscutting concerns code from the original code into the appropriate aspects, the following questions need to be addressed:

How to Separate Crosscutting Concerns from Original Source Code?

Sophisticated program analysis and manipulation techniques are needed to separate crosscutting concern code, since by definition such code is tangled with other code. Depending on the kind and the amount of tangling, some code is easier to separate than other code. Tracing code, for example, is often relatively independent of the code surrounding it, whereas Bruntink et al.'s experiment [91] showed that exception handling code exhibits significantly more tangling.

How to Determine Appropriate Joinpoint(s) for Extracted Aspects?

An aspect needs to specify the exact location where advice code needs to be woven, by means of a (set of) pointcut(s) that select(s) the appropriate joinpoints. However, aspect languages impose certain restrictions on the locations in the static or dynamic software structure that can be made available as joinpoints. Hence, determining the appropriate joinpoints requires significant attention.

How to Determine Appropriate Pointcut(s) for Extracted Aspects?

Assuming that appropriate joinpoints can be found, the next problem is that of determining the appropriate pointcut(s) that describes these joinpoints. Additionally, the pointcuts need to expose the appropriate context information for the advice code.

How to Determine Appropriate Advice Code for Extracted Aspects?

The crosscutting concern code typically cannot be transformed "as is" into advice code. Small modifications are often required, due to the code being defined in a different context, but also due to small variations in the scattered snippets of crosscutting concern code.

How to Ensure Correctness of Extracted Code?

The correctness requirement is of course related to *behaviour preservation*, i.e., extracting aspects from existing source code is expected to preserve the external behaviour of that code. Even for traditional refactorings this is already considered a non-trivial problem [404]; when extracting aspects from traditional programs the problem only becomes harder. Obviously, automating the transformations that are applied can help meeting this requirement, as automated transformations can be proven correct by using preconditions [404]. Additionally, appropriate test suites are of great value, but are not always present in (legacy) software. Furthermore, since the extraction process affects the original code structure, certain tests that rely on that structure may need to be restructured as well. In particular, certain tests may need to be transformed into their aspect-oriented equivalent.

9.4.3 Existing Techniques

Research on aspect extraction is still in its infancy, as most researchers focussed primarily on aspect exploration first. Nonetheless, work exists that contributes to the growing body of aspect extraction research [88, 375, 376, 163, 67, 223, 220]. Most of this work does not clearly distinguish between aspect extraction and aspect evolution, however. In this section, we only consider those parts of this work that deal with extraction.

Separating Crosscutting Concern Code

Separating the crosscutting concern code from the original code requires taking tangling into account: the code might use local variables that are defined by the ordinary code, or might modify variables that are used by the ordinary code. Hence, all separation techniques need to include a way to deal with such local references.

Both Monteiro and Fernandes [375] and Hanenberg et al. [220] discuss an *extract advice* transformation, that is responsible for separating the concern code but is not automated. Both mention that particular attention should be paid to local variables used in the crosscutting concern code. Hanenberg et al. take the position that either the developer should check whether such variables are not referenced outside the crosscutting code, in which case the variable declaration can be moved safely to the advice code, or else the transformation cannot be applied. Monteiro and Fernandes suggest that the code fragment should be isolated first using *Extract Method* or *Replace Method with Method Object* refactorings [183]. Binkley et al [67] present automated transformations, but propose the same approach as Monteiro and Fernandes. It is not clear, however, if this would work in practice, as these refactorings themselves might not be applicable when dealing with the problem of local variables.

The work of Ettinger and Verbaere [163] is currently the only one proposing an automated solution to the problems encountered when separating concern code from the original code. They propose to use program slicing [538] to untangle concern code and ordinary code. Program slicing is a technique that singles out those statements that may have affected the value of a given variable and that outputs a set of

statements, called a *slice*. The idea is that this slice contains all code that is related to the concern, including references to local variables and how their values are computed, and can be factored out by means of an *extract slice* transformation [352]. This transformation can either fully extract all statements from the original code, or can leave some statements where they are, if they are relevant for the original code. It is not clear whether such a transformation is feasible to implement, however.

Determining Appropriate Joinpoints

After having separated the crosscutting concern code from the original code, we need to map those locations where that code was originally located to an appropriate set of joinpoints. The possible joinpoint locations that can be specified by a given AOP language are often limited: not every node in the structure or execution flow graph can be selected by an aspect. Hence, the required mapping is not always possible.

A possible solution for this problem is to extend the pointcut language so that more joinpoints can be exposed. However, a trade-off exists between the completeness of the joinpoint model and the performance of the produced software. The execution of aspect-oriented software would slow down considerably if an aspect could select any node in the structure or execution flow graph. Consequently, a complete joinpoint model is considered impractical.

Another alternative is to restructure the code before extracting the crosscutting concern code, to make it fit the joinpoint model offered by the AOP language. This is the approach taken by both Binkley et al [67] and Monteiro and Fernandes [375], who suggest to apply traditional refactorings first in order to make the code more "aspect friendly". For example, concern code occurring in between a set of statements is impossible to separate using most existing AOP languages. Hence, as depicted in Figure 9.6, this concern code can be extracted first using an *Extract Method* refactoring, for example, producing additional joinpoints that an aspect can use. There is considerable discussion in the AOSD community about this issue, as it interferes with the obliviousness property of AOSD, as explained in Section 9.2: the ordinary code should not "know" about the aspects that apply to it. Clearly, transforming the code with the sole intent of making it "aspect friendly" breaks this assumption. However, the experiments of Binkley et al. [67] suggest that only 20% of the cases requires performing a traditional refactoring first. The authors acknowledge the fact that performing such transformation should be seen as the "extreme recourse that solves all problems", since the transformation might reduce code familiarity and quality in general.

Determining Appropriate Pointcuts

Having determined the appropriate joinpoints, we need to define the appropriate pointcuts that capture those joinpoints. The simplistic solution is to use extensional pointcuts which merely enumerate all joinpoints. However, as explained in Section 9.2, we prefer more intensional pointcut definitions which are more robust towards evolution.

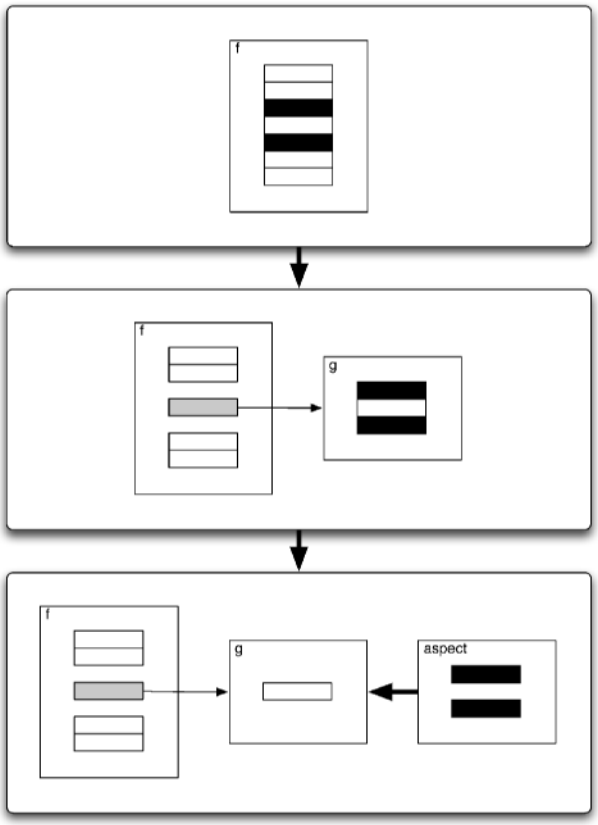

Fig. 9.6. Making code aspect friendly

Authors that propose non-automated extraction transformations generally do not pay sufficient attention to the definition of appropriate pointcuts. Hanenberg et al [220] consider extracting crosscutting concern code from a single method only, and describe that "a pointcut that targets the relevant method" has to be defined. Monteiro and Fernandes [375] provide a bit more sophistication, saying that a pointcut "should capture the intended set of joinpoints", and that if the intended pointcut is already under construction, it should be extended so that it includes the joinpoint related to the code fragment currently being extracted. The responsibility of defining a good pointcut thus rests completely with the developer, who needs detailed knowledge of the structure and the behaviour of the software.

Binkley et al. [67] tackle the problem of determining "sensible" pointcuts automatically, and describe 7 extraction transformations with the particular pointcuts they generate. For example, they define an *Extract Before Call* transformation, depicted in Figure 9.7, that extracts a block of code that always occurs before a particular method call. In the aspect `B`, the pointcut `p` intercepts the call to `h` that occurs within the execution of method `f`. A before-advice reintroduces the call to `g` at the proper execution point.

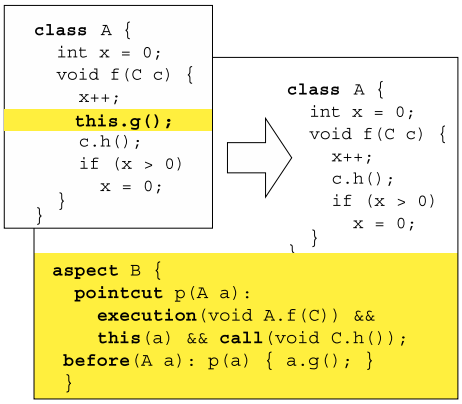

Fig. 9.7. The Extract Before Call transformation

Although not explained explicitly in the paper, it is clear that applying their extraction transformations yield extensional pointcuts: when extracting code from many different locations, the transformations extract the code from one location at a time, and combine the pointcut of each individual location with the already existing pointcut, in order to form a new pointcut.

Braem et al. [79] present an experiment where they use *inductive logic programming* in order to uncover "patterns" in, and generate intensional pointcuts from, a given set of joinpoints. Inductive logic programming is a machine-learning technique that requires positive as well as negative examples and background information, so as to define a logic rule that captures all positive but none of the negative examples. For this experiment, the authors use joinpoints corresponding to the crosscutting concern code as positive examples, all other joinpoints occurring in the program as negative examples, and structural information about the program, such as the classes in which methods are defined and which methods a particular method calls, as background information. The resulting induced pointcuts look similar to a pointcut that a developer would define when confronted with the same task.

Determining Appropriate Advice Code

The advice code of an aspect definition consists of the code that should be woven at the joinpoints selected by the aspect's pointcuts. Although we discuss the problem of how to determine that advice code separately here, it is strongly overlapping with the problem of separating the crosscutting concern code from the original code, which we discussed earlier on. The advice code corresponds to the crosscutting concern code that was separated from the original source code, but cannot be used as advice code as is. In general, the crosscutting concern code makes use of the context in which it is implemented: it may contain references to local variables or use instance variables or methods of a class. To determine the appropriate advice code, the crosscutting concern code needs to be inspected for such context-specific references, and the pointcut and advice code need to be adapted adequately to the new (aspect) context.

Most aspect languages provide dedicated constructs to allow aspects to expose context information associated to the joinpoint at which the aspect applies, such that this information can be used in the advice code. The `args` construct used in Figure 9.4 was an example of such a construct and allowed a method joinpoint to pass the actual value of the method's argument to the advice. Other examples are constructs to expose the name of the method corresponding to the joinpoint, the names of its formal parameters, or a reference to its defining class. In general, the pointcut definition that captures the appropriate joinpoints is extended with dedicated predicates and parameters in order to be capable of exposing the necessary information to the advice code.

This can be a quite complex undertaking, however, due to limitations in the context information exposed by aspects. For example, the crosscutting concern code may use temporary variables local to the method or function in which it is contained, and most aspect languages do not provide constructs to expose such information. Additionally, in an object-oriented language, the crosscutting concern code may reference private instance variables and/or methods, and visibility rules may prevent an aspect from accessing or extracting such private information.

Hanenberg et al. [220] and Monteiro and Fernandes [375] touch upon the problem of references to (private) instance variables and methods when dealing with their *extract advice* and *extract introduction* transformations. Their solution consists of declaring an aspect `privileged`, meaning it can bypass visibility rules, and of using additional `this` and `target` pointcuts in order to resolve self and super calls in the advice code. Additionally, Monteiro and Fernandes [375] consider the problem of crosscutting concern code that uses local variables, and propose to turn such variables into instance variables if necessary. The consequences of adapting the code in this way with the sole intent of making it "aspect friendly" is not elaborated upon, nor is made clear what its impact would be on large code bases or on the code quality, and whether this solution is always feasible.

Binkley et al. [67] explicitly mention the context exposure problem when defining their extraction transformations, and provide a precise description of how these transformations generate pointcuts that expose the necessary context. Because these transformations are automated and reason about the crosscutting concern code, they either generate a correct pointcut that exposes the necessary context, or are not applicable at all. Hence, the resulting aspect is always correct, which is not the case for the other (manual) approaches.

9.4.4 Open Issues

The issues identified above and our overview of the current state of the research show that the major issues and problems related to aspect extraction have been identified, but that no satisfactory solutions exist yet. Most existing techniques touch upon a specific part of a particular problem, but no single technique provides a complete solution to all problems identified. This is no surprise, as research on aspect extraction is only just emerging.

First of all, the level of automation of current extraction techniques is poor, and all issues touched upon in the previous subsection could benefit significantly from more automation. Clearly, more effort is needed in this area, since automated extraction techniques are indispensable when dealing with large-scale software, in order to achieve efficiency and correctness. A technique such as the use of inductive logic programming to automatically produce intensional pointcut definitions [79] is definitely a step forward. However, this technique was validated only on a single example and it remains to be investigated how it performs on more complex cases.

Second, the issue of preserving the behaviour of the software after extraction has not yet been tackled explicitly. Proving the correctness of aspects that were extracted manually is practically impossible. Automated techniques, however, could be proven correct. Given Opdyke's experience in this matter [404], it is clear that constructing formal proofs for the complex extraction transformations is far from trivial. However, formally defining the necessary preconditions for such transformations should be feasible, but has currently not yet been realised.

Related to testing behaviour-correctness of performing aspect extraction, the issue of migrating the original test suites to the migrated software system remains. Unfortunately, little work exists on testing aspect-oriented systems (notable exceptions are the works of Xu and Xu [555] and Xie and Zhao [549]), let al.one on the migration of the original tests to their aspect-oriented equivalent. Chapter 8 of this book also mentions this explicitly as a topic that warrants further investigation.

Finally, little or no empirical validation of the proposed techniques on large-scale, real-world software systems has been performed. This makes it hard to assess whether the techniques actually work in practice, what their advantages and disadvantages are, whether they scale to large industrial software, and whether the extraction actually improves the quality of the software.

9.4.5 Extraction in Practice

As is apparent from the previous subsections, most work on aspect extraction is focused on the technical level, i.e., it describes new transformations that extract crosscutting concerns into aspects. With the notable exception of the work by Binkley et al. [67], none of these transformations have been applied extensively on real-world systems. Binkley's work does not present any concrete details of the case study either, and focuses mainly on the transformations themselves.

In this subsection, we summarise an experiment by Bruntink et al. [88], where they studied the *tracing* crosscutting concern in a 80.000 lines subset of an industrial software system. The goal of their experiment was to study whether this concern was implemented in a sufficiently systematic way, so that it could be expressed easily in terms of appropriate pointcuts and advice. Such an investigation could be regarded as a preliminary step before performing an actual extraction.

As an illustration of their approach, taken from [88], consider the idiomatic implementation of the tracing concern in Figure 9.8. A developer needs to trace input parameters of a function at the beginning, and output parameters at the end of that function. The `trace` function implements tracing and is a variable-argument function.

```
int f(chuck_id* a, scan_component b) {
    int result = OK;
    char* func_name = "f";
    ...
    trace(CC, TRACE_INT, func_name, "> (b = %s)",
          SCAN_COMPONENT2STR(b));
    ...
    trace(CC, TRACE_INT, func_name, "< (a = %s) = %d",
          CHUCK_ID_ENUM2STR(a), result);
    return result;
}
```

Fig. 9.8. Code fragment illustrating the tracing idiom in Bruntink et al..'s case study

Its first four arguments denote, respectively, the component in which the function to be traced is defined, whether the tracing is internal or external to that component, the name of the function for which the parameters are being traced, and a `printf`-like format string that specifies the format in which the parameters should be traced. Optional arguments specify the input or output parameters that need to be traced. Parameters of a complex type (as opposed to a basic type like `int` or `char`) need to be converted to a string representation. Typically, this is done by using a dedicated function or macro, such as `SCAN_COMPONENT2STR` and `CHUCK_ID_ENUM2STR` in the example of Figure 9.8.

In order to study whether the concern was implemented consistently throughout their case study, Bruntink et al. proposed a method based on formal concept analysis, applied on typical attributes associated with the concern under study. For the tracing concern, they studied both *function-level* and *parameter-level* variability, and tuned the concept analysis algorithm so that it grouped all functions that invoked tracing in a similar way, and all parameters that are converted in the same way, respectively.

Without going into all details, the results of running the experiment on four components of the software system are described in Table 9.3. The most striking observation (second row) was that only 40 out of 704 (5.7%) of all functions invoke tracing in the 'standard' way. The first row shows that 29 different tracing variants are used in the four components. In addition, the authors observed that none of these 29 variants could be considered as the 'standard' variant, with the other variants being simple deviations from the general rule.

As for parameter-level variability (lower half of the table), the study showed that 37.7% of the parameter types (94 out of 249) were traced in an inconsistent way, i.e. a single parameter type is converted into a string representation in more than one way. Only 16% (40 out of 249) was traced consistently, and 115 parameter types were not traced at all. Some inconsistency arises because not all functions need to trace, however, and hence some parameter types are converted using one single converter function in many different functions, while not traced in other functions. To take this into account, those parameter types are excluded from the number of inconsistently-

Table 9.3. Function-level and parameter-level variability results (taken from [88], ©ACM, 2007)

	CC1	CC2	CC3	CC4	total	global
Function-level variability						
#tracing variants	6	4	19	2	31	29
#functions w. std. tracing	13	1	26	0	40	40
% of total functions	4	0.7	15	0		5.7
Parameter-level variability						
#not traced	61	49	4	16	130	115
#consistently traced	15	5	16	19	55	40
#inconsistently traced	32	17	45	14	108	94
#w.o. not traced	11	6	39	8	64	57

traced parameter types. Hence, the fourth row shows the parameter types that are converted using more than one converter function, and the authors concluded that 42.5% (57 out of 134) of all parameter types were not traced consistently, and 57.5% (77 out of 134) were traced consistently.

An additional advantage of their method is that formal concept analysis produces concept lattices that can be visually inspected. Figure 9.9, again taken from [88], illustrates this: it clearly shows that the component uses three different ways to specify the component name (`CC`, `"CC2"` and `CC2_CC`), and that there is one function that uses both `CC` and `"CC2"`, i.e. there are two trace statements in that function, and each statement specifies the component name in a different way.

The most appealing result of this experiment was the observation that the implementation of the tracing concern was not consistent at all, contained much more variability than expected, and could thus not be expressed as one single aspect. This came as a surprise, given the fact that tracing is a relatively simple concern, which is often used as the prototypical example of a concern that can easily be turned into

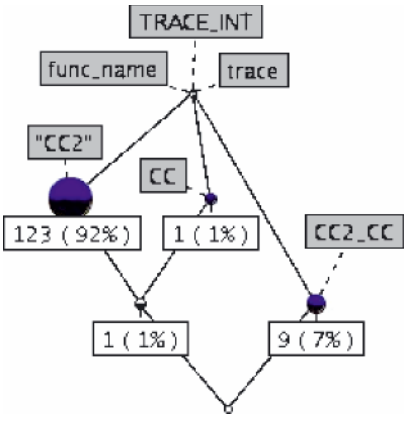

Fig. 9.9. Function-level variability in the CC2 component

an aspect. Since current aspect extraction tools and techniques that are proposed do not take these observations into account, they are not yet ready to be used on real-world systems. Granted, since a study of this kind was never conducted before, these issues could not have been identified yet. But it clearly illustrates the complexity of the activity of aspect extraction.

9.5 Aspect Evolution

Once the crosscutting concerns in the original software system have been explored and the system has been migrated to a new aspect-oriented version, the system enters a new phase in which it will need to be continuously maintained and evolved in order to cope with changing requirements and environments. In this section, we highlight some of the issues and problems related to such evolution of aspect-oriented systems.

9.5.1 Rationale

As was argued in the introductory section, AOSD overcomes some of the problems related to software evolution, in particular the problems related to maintaining and evolving independently the different (crosscutting) concerns in a system. But since all software systems are subject to evolution (remember the first law of software evolution), aspect-oriented systems themselves too will eventually need to evolve. We define *aspect evolution* as the process of progressively modifying the elements of an aspect-oriented software system in order to improve or maintain its quality over time, under changing contexts and requirements.

9.5.2 Challenges and Risks

While research on aspect exploration is only starting to produce its first results and research on aspect extraction is still gaining momentum, research on aspect evolution is even younger. This is largely due to the fact that few large-scale aspect-oriented software systems exist today. Even if they would exist, they would be too young in order for them to be the subject of a rigorous scientific study regarding their long-term evolution problems.

Despite the immaturity of the field, some initial research questions have been raised, related to how the evolution of aspect-oriented software differs from evolving traditional software and whether techniques and tools, successful in supporting traditional software evolution, can still be applied to the evolution of aspect-oriented software. It seems that the very techniques that AOP provides to solve or limit some of the evolution problems with traditional software, actually introduce a series of new evolution problems. This phenomenon is sometimes called the *evolution paradox* of AOP [505].

9.5.3 Existing Techniques and Open Issues

Evolving aspect-oriented software differs in at least two ways from evolving traditional software:

1. First of all, evolving the base code in any way may impact the aspects that work on that code (see Figure 9.10). Evolution normally involves adding and removing classes, methods and instance variables, or changing them in some way. By doing so, the set of joinpoints associated to the program changes too: new joinpoints are added and existing joinpoints are removed or changed. This clearly affects the aspects which select joinpoints by means of pointcuts. Hence, when evolving the base code, care has to be taken to assess the impact this evolution has on the aspects.
2. Conversely, the aspects themselves can be subject to evolution too (Figure 9.11). Since concerns are easier to evolve when they are separated into aspects instead of being implemented by means of coding conventions and idioms, it seems natural to assume that aspects may therefore evolve more often. However, like any other software artefact, aspects evolve for a variety of different reasons. For example, pointcuts could be generalised to make them less brittle, abstract aspects could be introduced to make the aspect-oriented code more reusable, or advice code could be restructured to make it more comprehensible. Hence, the introduction of AOP introduces new types of evolution that where previously impossible or difficult to achieve.

These issues were already identified by a number of authors. Hanenberg et al. [220] introduce *aspect-aware* and *aspect-oriented* refactorings. The former are traditional refactorings that are extended to take aspects into account, such as *Rename method* and *Extract method* that need to make sure an aspect's pointcuts are updated appropriately. Such refactorings tackle the first issue presented above. The latter refactorings are newly-defined and refactor the aspect code instead of the base code. They

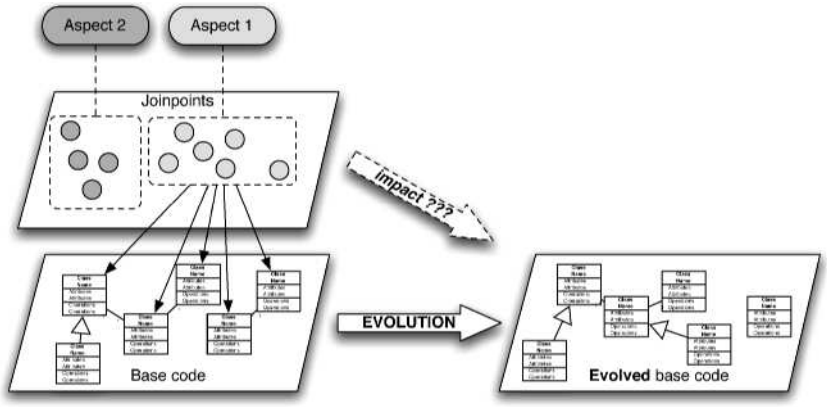

Fig. 9.10. Impact of base code evolution on the aspects

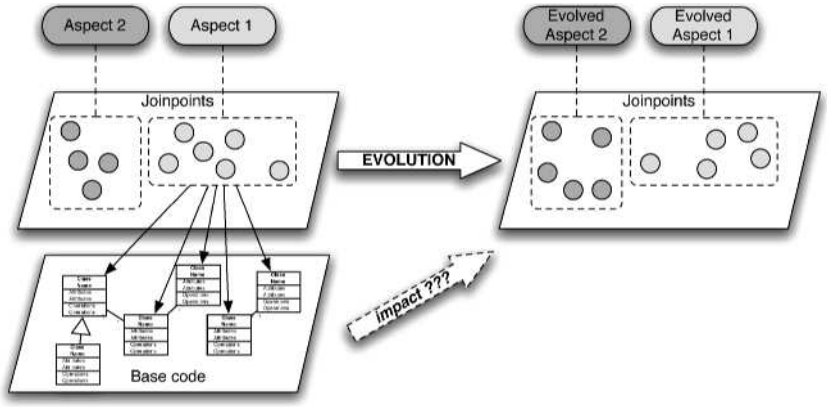

Fig. 9.11. Impact of aspect evolution on the base code

thus tackle the second issue. An example is the *Separate pointcut* refactoring, that extracts the common parts of several pointcuts into a new pointcut, so that it can be reused properly.

Monteiro and Fernandes [376] follow an approach similar to Fowler [182] where they identify several *bad (aspect) smells*, and define refactorings that alleviate these. They divide the different refactorings into three categories, as shown in Table 9.4: for extracting crosscutting concerns, for restructuring the internals of aspects, and for dealing with generalisation. The first category of refactorings has been explained in Section 9.4. The second category contains refactorings that are often applied after an aspect has been extracted from a crosscutting concern, and needs tidying up, whereas the third category contains refactorings that should make an aspect definition more general and hence more reusable. The distinction between the last two categories is rather arbitrary, and the refactorings presented are not automated. Since they are rather high-level refactorings, they are probably difficult to automate at all.

Not only does AOP lead to new types of evolution, it also introduces new kinds of evolution problems. In particular, several authors have identified and suggested solutions for the *fragile pointcut problem* [293, 477, 482, 362]. This problem occurs when pointcuts accidentally capture or miss particular joinpoints as a consequence of their fragility with respect to seemingly safe modifications to the base program. We will illustrate and discuss this problem in more detail in Subsection 9.5.4.

Another danger to evolution of aspect-oriented programs is what is sometimes called the *aspect composition problem* [226]. When combining into the same application two aspects that have been developed independently, they may interact in undesired ways. For example, suppose we want to combine a simple logging and synchronisation aspect. For those joinpoints that are captured by both aspects, do we only want to log and synchronise the base code? Do we want the logging aspect to log the synchronisation code as well? Or do we want the synchronisation aspect to synchronise the logging code? Languages like AspectJ propose language

Table 9.4. Three categories of refactorings as defined by Monteiro and Fernandes (Adapted from [376]), ©ACM, 2005

Restructuring aspect internals
Extend marker interface with signature
Generalise target type with marker interface
Introduce aspect protection
Replace inter-type field with aspect map
Replace inter-type method with aspect method
Tidy up internal aspect structure

Extracting crosscutting concerns	Dealing with generalisation
Change abstract class to interface	Extract superaspect
Extract feature into aspect	Pull up advice
Extract fragment into advice	Pull up declare parents
Extract inner class to standalone	Pull up inter-type declaration
Inline class within aspect	Pull up marker interface
Inline interface within aspect	Pull up pointcut
Move field from class to inter-type	Push down advice
Move method from class to inter-type	Push down declare parents
Replace implements with declare parents	Push down inter-type declaration
Split abstract class into aspect and interface	Push down marker interface
	Push down pointcut

constructs to define how to combine aspects, for example by providing priority rules and permitting a developer to declare in what order to apply the aspects. However, when combining more complex aspects, often these constructs do not suffice and more intricate compositions are desired. Lopez-Herrejon et al. [331] and others propose alternative composition models, based on program transformations, that support step-wise development, retain the power of AspectJ and simplify program reasoning using aspects. Such composition models align aspect-oriented software development with component-based software engineering in order to offer the best of both worlds. Chapter 10 also briefly touches upon these issues.

To conclude, it is clear that aspect evolution is still an emerging research area, in which not all important research questions have been identified, let al.one answered. Nevertheless, it is important to mention that an awareness of the problem is growing inside the AOSD community, and that more and more researchers in that community are starting to investigate such problems.

9.5.4 Aspect Evolution in Practice: The Fragile Pointcut Problem

As a concrete example of an aspect evolution problem, this section touches upon the *fragile pointcut problem*, proposes a possible solution, and discusses a small case study on which an initial validation of this solution was conducted.

In Section 9.2 we already illustrated the distinction between extensional pointcuts, which merely enumerate the joinpoints in the source code, and intensional pointcuts that are defined in terms of more high-level structural or behavioural properties of the program entities to which they refer. The tight coupling of extensional

pointcut definitions to the base program's structure hampers evolvability of the software [482] since it implies that all pointcuts of each aspect need to be checked and possibly revised whenever the base program evolves. Due to changes to the base program, the pointcuts may unanticipatedly capture joinpoints that were not supposed to be captured, or may miss certain joinpoints that were supposed to be affected by the aspect. This problem has been coined the fragile pointcut problem [293, 477].

Kellens et al. [362] address the fragile pointcut problem by replacing the intimate dependency of pointcut definitions on the base program by a more stable dependency on a conceptual model of the program. This is illustrated schematically in Figure 9.12. Their *model-based pointcut* definitions are less likely to break upon evolution, because they are no longer defined in terms of how the program happens to be structured at a certain point in time, but rather in terms of a model of the program that is more robust to evolution.

To validate their approach, they defined two simple aspects on an initial release of the SmallWiki application. The 'action logging' aspect extended SmallWiki with basic logging functionality for the different actions that occur in the wiki system. A second 'output' aspect altered the way (font) in which text in wiki documents was rendered. They implemented each of these two aspects once with aspects defined in terms of traditional pointcuts and once in terms of model-based pointcuts defined over a conceptual model of the application. Then they considered two more recent versions of the SmallWiki application (a version one month and another about one year after the initial release) and assessed the impact of the aspects in those versions.

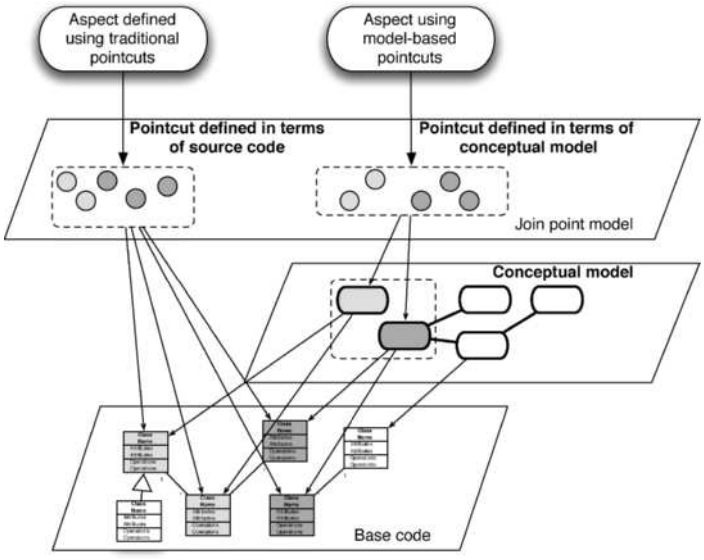

Fig. 9.12. Managing the fragile pointcut problem with model-based pointcuts

As it happened, some occurrences of the fragile pointcut problem appeared. In the solution with traditional aspects this resulted in an erroneous behaviour of the aspectualised application. More specifically, some actions that should have been logged were not and some text outputting that should have been altered was not. With the model-based pointcut approach, however, these fragile pointcut problems were detected as mismatches between the evolved code and the conceptual model that was defined on top of it. As such, the conflicts could be detected early and solved by the aspect programmer before actually applying the aspects.

On the downside, the approach does not detect all occurrences of the fragile pointcut problem: a lot depends on the level of detail of the conceptual model in terms of which the pointcuts are defined. The more detailed the model, the more mismatches can be detected. Also, since the approach has only been illustrated an a relatively small case on two simple aspects, it remains to be investigated how well it performs on real aspect-oriented systems.

9.6 Summary

In this chapter, we summarised important evolution-specific issues and challenges related to the adoption of aspect-oriented programming, and presented an overview of the state-of-the-art research that addresses these. We identified three different stages that adopters of AOP may need to go through: exploration, extraction and evolution.

The exploration stage is a preliminary phase that studies whether the software actually exhibits important crosscutting concerns that can or should be extracted into aspects. We showed that exploring a software system for crosscutting concerns means looking for more than the well-known and simple crosscutting concerns often documented in the research literature. Moreover, particular coding conventions and idioms can help in identifying important crosscutting concerns, as they are often used to make up for the lack of aspects. Additionally, we described how some of the existing exploration tools make use of the very same information in order to automatically mine a software system for crosscutting concerns. However, even those automated tools typically require quite a lot of manual inspection of the produced results, due to the relatively low precision and recall of the proposed techniques. Regarding crosscutting concerns and software quality, we discussed some preliminary work that hints at a positive impact on the software quality of implementing crosscutting concerns by means of aspects, but no definitive conclusions can be drawn yet and more experimental validation is clearly needed.

The extraction stage follows the exploration stage, and considers how crosscutting concern code can be extracted from the ordinary code and defined into the appropriate aspects. We identified the issues related to this extraction, in particular separating the concern code from the base code and turning it into advice code, and determining the appropriate joinpoints and pointcuts. Existing work that tackles (part of) these issues was described, which showed that this area of research is still young and needs significantly more work before it can be useful in an industrial context.

Nonetheless, these techniques show the feasibility of automating, at least partly, the process of aspect extraction.

The evolution stage then deals with the evolution of the final aspect-oriented software, and how this differs from the evolution of ordinary software. We showed that evolving aspect-oriented software involves evolving the ordinary code as well as the aspect code, and that this gives rise to extensions of existing techniques that support evolution, as well as new techniques to support aspect evolution. Additionally, we explained that the adoption of AOP gives rise to new evolution-related problems, such as the fragile pointcut problem and the aspect composition problem. Solutions to those problems are under active research by the AOP community.

The overall conclusion that can be drawn is that aspect-oriented software development is still a young paradigm, that still needs to mature and requires much more rigourous research. Nonetheless, it is a promising paradigm that receives a lot of attention, and gives rise to several tools and techniques that already provide at least some kind of support for early adopters.

Acknowledgement. This chapter builds on the work of a vast community of people working on evolution-related issues in the domain of AOP. We are grateful to all authors of the work referred to in this text for having implicitly or explicitly provided us the necessary material for writing this chapter. Given the broad range of topics and issues covered by this chapter, it is inevitable that some important references may be missing. We are equally grateful to those researchers for advancing the state-of-the-art in this exciting research area.