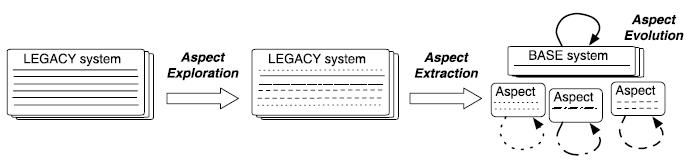
## Migración de Sistemas Orientado a Objetos hacia Sistemas Orientado a Aspectos

La adopción de un nuevo paradigma de programación conduce a la pregunta de cómo migrar los sistemas existentes al nuevo paradigma, pregunta que en la actualidad se aplica al paradigma orientado a aspectos [7]. Si bien, el encapsulamiento de los crosscutting concerns de un sistema legado en aspectos es potencialmente beneficioso, decidir que partes del código corresponden a un crosscutting concern es muy dificultoso [4].

Se puede distinguir tres fases diferentes para realizar y evolucionar la migración a aspectos: “aspect exploration”, “aspect extraction” y “aspect evolution”’ [libro].

* **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?

La Fig. II-3 visualiza el proceso de migración y evolución de un sistema legado a un sistema que utiliza aspectos.



**Fig. II-3**. Migración y evolución de un sistema legado a un sistema orientado a aspectos.

### Proceso de Migración

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.

Aspect Exploration

Se define a ASPECT EXPLORATION como la actividad de identificar y analizar crosscutting concerns en sistemas no orientado a aspectos.

Kellens et. Al. [272] distingue tres categorías principales de técnicas que pueden ayudar a lozalizar los crosscutting concerns en un sistema de software. Estas técnicas son:

* Early aspect discovery techniques: la investigación de esta técnica trata de descubrir aspectos en las fases tempranas del ciclo del vida de un software [35] como por ejemplo requerimientos y análisis de dominio [34,431,492], diseño de la arquitectura [42]. A pesar de que la técnica mencionada puede ayudar a identificar ciertos crosscutting concerns en un sistema de software, se considera una técnica menos prometedora que aquellas que aquellos enfoques que se centran en el código fuente. Esto se debe a que los documentos de requerimientos y la arquitectura generalmente se encuentran desactualizados u obsoletos.
* *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].
  + - 1. Aspect Mining

Las técnicas de aspect mining automatizan el proceso de descubrimiento de crosscutting concerns y proponen al usuario uno o más aspectos candidatos. Para este fin, las técnicas evalúan el código fuente o datos adquiridos ejecutando o manipulando el sistema. Estas técnicas tienen en común, al menos, que buscan síntomas de crosscutting concerns tales como código disperso y código entremezclado mediante la aplicación de técnicas de data mining, comprensión de software o de análisis de programas [8] . El código disperso corresponde a concerns cuya implementación abarca diferentes módulos del sistema. Por otra parte, el código entremezclado corresponde a módulos que manejan múltiples concernssimultáneamente [2].

Las técnicas de aspect mining pueden clasificarse en dos grupos diferentes: técnicas de análisis estático y técnicas de análisis dinámico. Las técnicas basadas en análisis estático analizan la frecuencia de los elementos del programa y se basan en la homogeneidad sintáctica de los crosscutting concerns. Por otra parte, las técnicas basadas en análisis dinámico buscan patrones de ejecución durante la ejecución del programa.

Aspect Extraction

*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.

**Existing Techniques**

**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 themselvesmight 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 interfereswith 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.

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. 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.

### Aspect Refactoring

Debido al alto costo de mantenimiento de los sistemas de software, existe la necesidad de técnicas que reduzcan la complejidad e incrementen la calidad interna de los mismos. Se conoce al dominio de investigación que comprende a este problema como reestructuración. En el caso específico del desarrollo de software orientado a objetos se denomina refactorización [17], el cual se define al refactoring como el proceso de cambiar un sistema de software orientado a objetos para mejorar la estructura interna del código de manera de no alterar el comportamiento externo del mismo [18].

Entre los diversos refactorings existentes, se encuentra el denominado aspect refactoring. El cuál, como se mencionó anteriormente, define la migración de código orientado a objetos hacia código orientado a aspectos. Los refactorings son organizados sistemáticamente en catálogos. En [19] Monteiro define un catálogo de 28 refactorings de aspectos, para todos ellos especifica su nombre, situación típica, descripción de la acción recomendada, motivación, mecanismos y códigos de ejemplo. En motivación describe cuándo debería usarse el refactoring, en mecanismos se describen una serie de pasos a seguir para poder aplicar el refactoring. Finalmente, en ejemplos de código se plasma en concreto lo descripto en los puntos anteriores, ilustrando de esta manera el refactoring. Luego, el catálogo divide los refactorings en cuatro grupos:

* Código java a aspectos: Comprende el encapsulamiento de diferentes elementos del código en un aspecto.
* Estructura interna de los refactorings de aspectos: implica mejorar la estructura interna de un aspecto
* Generalización de los aspectos: transformaciones en la jerarquía de aspectos.
* Código legado: se utiliza cuando existen interfaces de código legado que no pueden ser modificadas.