# Mantenimiento de Aplicaciones

## Concerns

Un sistema de software es la realización de un conjunto de “concerns”. Se define a un concern como todo lo que un stakeholder quiera considerar como una unidad conceptual, incluyendo características, requerimientos no funcionales y decisiones de diseño. [1]. Los mismos deben ser correspondidos a fin de satisfacer el objetivo general del sistema. Pueden ser clasificados en dos categorías:

* core concerns: son aquellos que capturan la funcionalidad central de un módulo
* crosscutting concerns: son aquellos que capturan requerimientos a nivel de sistema que atraviesan múltiples módulos. Ejemplos de estos son la autenticación, logging, seguridad integridad en las transacciones, etc.

Con el fin de reducir la complejidad del diseño y la implementación de un sistema se diferencia la presencia de ambos tipos de concerns. A fin de realizar esta separación se descompone el conjunto de requerimientos en concerns. Independientemente de la metodología que se use, dicha separación e identificación es un ejercicio importante en el desarrollo del software. El problema surge cuando los concerns no pueden implementarse en módulos independientes [2]. Un programa que presenta este tipo problema es difícil de mantener debido a que un simple cambio en uno de ellos puede impactar en muchas partes del sistema [3]. A pesar de que la separación mencionada pueda ser natural, las metodologías de programación actuales no la permiten en la fase de implementación. [2]

## Evolución de las metodologías de programación

La ingeniería de software ha atravesado un largo camino comenzando en los lenguajes a nivel máquina, atravesando la programación procedural y llegando a la programación orientada a objetos (POO). Esta evolución de las metodologías de programación permite a los ingenieros lidiar con problemas de más alto nivel que en décadas anteriores [2]. A pesar de esto, todos los lenguajes de programación proveen un conjunto limitado de abstracciones, no permitiendo así la separación de concerns de la forma en que se conciben informalmente [3].

La programación orientada a objetos (POO) es la metodología elegida en los nuevos proyectos de desarrollo de software. La fortaleza del mismo reside en el modelado del comportamiento común [2]. Los sistemas de este tipo son desarrollados mapeando las entidades del mundo real del dominio de la aplicación en una jerarquía de clases, alrededor de las cuales el software es desarrollado [4]

Sin embargo, no todos los requerimientos de la aplicación pueden mapearse a una sola unidad modular (clase). POO no cumple un buen papel en abordar este tipo de comportamiento ya que estos requerimientos quedan dispersos en varios módulos, generalmente no relacionados entre sí [2].

## Mantenimiento de Sistemas Orientado a Objetos

El mantenimiento es la parte central del ciclo de vida del software y comúnmente representa más de la mitad del costo del desarrollo del sistema. Es por esto que no es sorprendente que la capacidad de mantenimiento para los programas haya sido un punto clave en el diseño de lenguajes de programación [2].

Para modificar una aplicación, los desarrolladores deben identificar la idea de alto nivel, o concepto a ser transformado, y luego localizar, comprender y modificar el concern que representa a dicho concepto en el código [5]. Es por esto que probablemente el factor más importante que determina la mantenibilidad de un programa es la estructuración del mismo [3]. Esta comúnmente aceptada la premisa de que la mejor manera de lidiar con la complejidad es simplificándola. En diseño de software, la mejor manera de simplificar un sistema complejo es identificar y luego modularizar los concerns.

La metodología POO fue desarrollada en respuesta a la necesidad de dicha modularización. La POO es buena en la modularización de core concerns, pero falla cuando se trata de modularizar crosscutting concerns:

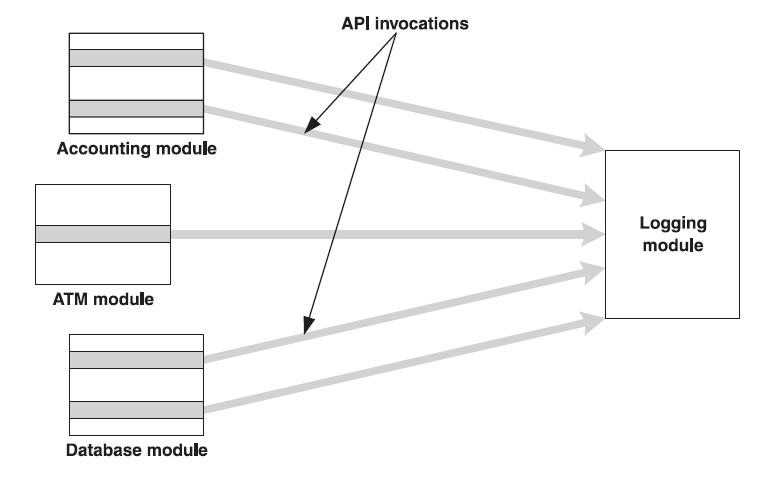
Los módulos centrales en las aplicaciones orientadas a objetos se pueden representar débilmente acoplados mediante el uso de interfaces.

No sucede lo mismo para los crosscutting concerns debido a que la implementación de estos últimos se lleva a cabo en dos partes: la pieza perteneciente al lado del servidor y la pieza perteneciente a la de los clientes. Los términos “cliente” y “servidor” son usados en el sentido clásico de POO, los cuales definen objetos proveyendo un conjunto de servicios y objetos usando estos servicios respectivamente. POO modulariza en clases e interfaces los servicios provistos. Sin embargo, el pedido del servicio se encuentra esparcido en todos los clientes. En consecuencia a lo mencionado, el efecto global es un enredo indeseable entre los módulos que necesitan los servicios y el módulo que lo provee.

A continuación se muestra un ejemplo que muestra como un sistema bancario implementa el logueo de la información utilizando las técnicas convencionales (Fig. II-1). El módulo de logging representa el servicio provisto (servidor) y los módulos de accounting, ATM y database utilizan este servicio. Este mismo puede implementarse utilizando una interface y así proveer los siguientes beneficios:

* disminuir el acoplamiento entre los clientes y las implementaciones del logging. Cualquier cambio en la implementación del servicio no afectará a los clientes
* permitir el reemplazo de la implementación del servicio con solo instanciar la interface del logging

A pesar del buen diseño del módulo del logging, los módulos clientes necesitan el código para invocar la API del servicio. En color gris se denota el acomplamiento de cada módulo [2].

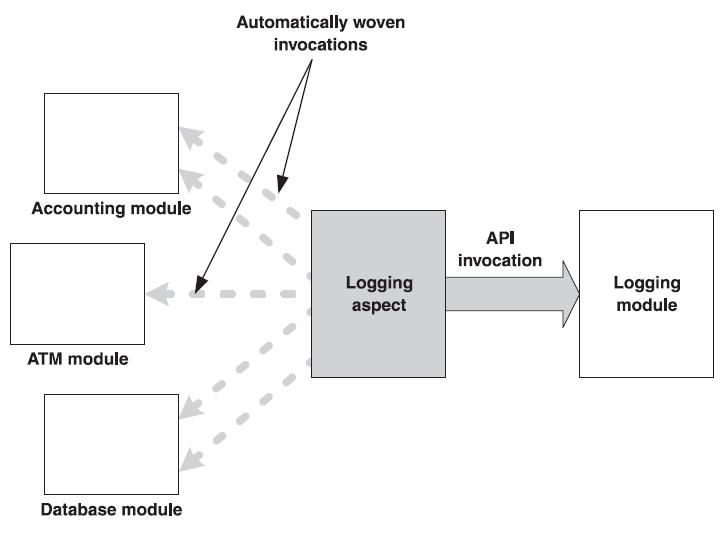


**Fig. II-1**.Implementación del concern logging utilizando las técnicas convencionales: el módulo de logging provee la API para el logueo. Sin embargo cada cliente necesita el código embebido para invocar el servicio.

En consecuencia a lo citado anteriormente se puede notar que la implementación de los crosscutting concerns en POO no es la más adecuada si se piensa en relación a la mantenibilidad del sistema. La modularización de los mismos no es lo suficientemente independiente y los códigos que invocan a los servicios se encuentran entrelazados entre la implementación de los módulos clientes. A razón de esto la matenibilidad del sistema en referencia a los concerns que lo cruzan no será adecuada, conllevando a problemas a la hora de modificar, agregar o reutilizar los crosscutting concerns [3]. Esto se corresponde con la llamada “tiranía de la descomposición dominante”, la cual determina que no importa cuán bien una aplicación se descompone en unidades modulares, siempre existirán concerns que atraviesen dicha descomposición [6].

El Desarrollo de Software Orientado a Aspectos (DSOA) surge en función de resolver el problema de la separación de la funcionalidad central de un sistema de software de los concerns que atraviesan la descomposición del mismo. Para esto, el paradigma provee de un nuevo constructor denominado aspecto, cuyo objetivo es encapsular crosscutting concerns. Se resuelve de esta manera el problema de la “tiranía de la descomposición dominante” [6].

La Fig. II-2. muestra la implementación del mismo ejemplo de logging mostrado en la figura Fig. II-1. utilizando Aspectos. La lógica del logging reside dentro del aspecto “logging” y los clientes no tienen código referenciando al mismo. Con esta modularización, cualquier cambio al requerimiento de logging afecta solo al aspecto logging aislando completamente a los clientes.



**Fig. II-2**.Implementación del concern logging utilizando las técnicas de AOP: el aspecto logging define los puntos de intersección e invoca la API del módulo logging sobre la ejecución de estos puntos. Los módulos clientes ya no contienen código relacionado al logging.

## Implementación orientada a aspectos

La metodología de programación orientada a aspectos es solo eso, una metodología. En función de usarla en el mundo real, debe ser implementada o realizada. Como cualquier otra metodología de programación, la implementación de POA consiste en dos partes:

* lenguaje de especificación: describe los constructores y sintaxis del lenguaje que va a ser usado para realizar tanto la lógica de los core concerns y el weaving de los crosscutting concerns.
* lenguaje de implementación: verifica la adherencia del código con la especificación del lenguaje y traduce el código a una forma ejecutable. Esto es comúnmente logrado mediante un compilador o un intérprete.

### Lenguaje de implementación orientado a aspectos

La implementación de un lenguaje orientado a aspectos realiza 2 pasos lógicos.: primero combina los concerns individuales usando las reglas de weaving, y luego convierte la información resultante en código ejecutable. El proceso que combina los concerns individuales de acuerdo a las reglas de weaving se denomina *weaving* y el proceso que realiza está tarea es llamado *weaver*.

Las reglas de weaving especfican como integrar los concern implementados a fin de formars el sistema final. El poder de POA proviene de la manera económica en que las reglas de weaving pueden ser expresadas. Citando el ejemplo del loggeo, se pueden especificar todas las operaciones públicas en el sistema que deben ser loggeadas en unas pocas líneas de código.

Las plataformas más conocidas para el desarrollo orientado a aspectos son las siguientes:

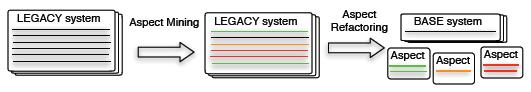
## Migración de Sistemas Orientado a Objetos hacia Sistemas Orientado a Aspectos

La adopción de un nuevo paradigma de programación conduce evidentemente a la pregunta de cómo migrar los sistemas existentes al nuevo paradigma. Esto mismo sucede con la programación orientada a aspectos [7]. La modularización de los crosscutting concerns en aspectos de un sistema legado es potencialmente beneficiosa. A pesar de esto, el trabajo dificultoso se encuentra en decidir que funcionalidades deben ser consideradas como aspectos [4].

Debido al tamaño y complejidad de los sistemas orientados a objetos, existe la necesidad de técnicas y herramientas que automaticen la migración de estos al paradigma orientado a aspectos. Existen dos tipos de técnicas para automatizar esta migración: “aspect mining” y “aspect refactoring” [8]:

* Aspect mining es la actividad de descubrir aquellos crosscutting concerns desde el código fuente o las trazas de ejecución de una aplicación que podrían ser encapsulados como aspectos del nuevo sistema [8].
* Aspect refactoring es la actividad de transformar los aspectos candidatos identificados en el código orientado a objetos en aspectos reales en el código fuente [8].

La siguiente imagen (Fig. II-3.) visualiza el proceso de migración de un sistema legado en a sistema que utiliza aspectos. El pasaje consiste en dos pasos: la identificación de aspectos candidatos y el refactoring de estos aspectos candidatos en aspectos.



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

### 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 de la misma. 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. Dentro de las técnicas de análisis estático se identifican varios enfoques, algunos de estos se listan a continuación:

* Formal Concept Analysis sobre en los nombres de clases y métodos [11].
* Procesamiento del Lenguaje Natural sobre el código fuente [10].
* Detección de métodos únicos [9].
* Clustering jerárquico de métodos relacionados [14].
* Análisis de Fan-in [12].
* Detección de clones como indicadores de crosscutting concerns [13].

Por otra parte, las técnicas basadas en análisis dinámico buscan patrones de ejecución durante la ejecución del programa. Se identifican las siguientes técnicas:

* Análisis de patrones recurrentes en trazas de ejecución [15].
* Análisis formal de trazas de ejecución (FCA) [16].

### 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].

Se define al refactoring como el proceso de cambiar un sistema de software orientado a objetos para mejorar la estructura interna del código de cierta manera de no alterar el comportamiento externo del mismo [18].

Entre los diversos refactorings existentes, se encuentra el denominado aspect refactoring. 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 28 refactorings de aspectos. Para todos ellos especifica su nombre, situación típica, descripción de la acción recomendada, motivación, mecanismos y ejemplos de códigos. 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.

# Referencias

[1] A Comparison of Logic Based Infrastructures for Concern Detection and Extraction – Kniesel

[2] [Laadad 2003] Ramnivas Laadad. “AspectJ in Action”. ©2003 by Manning

Publications Co. All rights reserved.

[3] Suitability of Object and Aspect Oriented Languages for Software Maintenance-Jasser, Schachte, Kazmierczak

[4] Aspect Mining through the Formal Concept Analysis of Execution Traces – Tonella, Ceccato

[5] A. Marcus, R. Koschke, A. van Deursen, V. Rajlich, P. Tonella, and H. Sneed. Identification of concepts, features, and concerns in source code. Panel Discussion at the International Conference on Software Maintenance, 2005.

[6] P. Tarr, H Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In ICSE, 1999.

[7] “Pitfalls in aspect mining” – Mens,Kellens,Krinke

[8] Kellens, A., Mens, K. A survey of aspect mining tools and techniques. Technical report, INGI 2005-07, Universite catholique de Louvain, Belgium (2005)

[9] Gybels, K. and Kellens, A. “Experiences with Identifying Aspects in Smalltalk Using Unique Methods,” in: International Conference on Aspect Oriented Software Development. Amsterdam, The Netherlands 2005.

[10] Shepherd, D., Pollock, L. L., Tourwé, T. “Using Language Clues to Discover Crosscutting Concerns,” ACM SIGSOFT Software Engineering Notes 30 (4), 1--6 2005.

[11] Tourwe, T., Kim Mens, K. Mining Aspectual Views using Formal Concept Analysis. In: 4th IEEE International Workshop on Source Code Analysis and Manipulation, pp. 97—106. (2004)

[12] M. Marin, A. Van Deursen, and L. Moonen. "Identifying crosscutting concerns using fan-in analysis," ACM Trans. Softw. Eng. Methodol., vol. 17, no. 1, pp. 1-37, December 2007.

[13] ”On the Use of Clone Detection for Identifying Crosscutting Concern Code” - Magiel Bruntink

[14] ”Interfaces, Aspects, and Views” - The Discoveries of a Clustering Aspect Miner and Viewer

[15] Breu, S., Krinke, J. Aspect Mining Using Event Traces. In: 19th IEEE International Conference on Automated Software Engineering, pp. 310--315. IEEE Computer Society, Washington DC, USA (2004).

[16] Tonella, P., Ceccato, M. Aspect Mining through the Formal Concept Analysis of Execution Traces. In: 11th Working Conference on Reverse Engineering, pp. 112--121. IEEE Computer Society, Washington DC, USA (2004)

[17] T. Mens, T Tourwé. A Survey of Software Refactoring. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. XX, NO. Y, MONTH 2004.

[18] M. Fowler, Refactoring: Improving the Design of Existing Programs, Addison-Wesley, 1999.

[19] M.P. Monteiro. Catalogue of refactorings for AspectJ. Technical Report UM-DI-GECSD-200401, Universidade do Minho, 2004.