CAPITULO II

Mantenimiento de Aplicaciones

1. Concerns en un Sistema de Software

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 e inclusive *programming idioms* [1] [23]. Dichos concerns deben estar implementados en el sistema a fin de satisfacer su objetivo general y pueden ser clasificados en dos categorías [2]:

- 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.

Esta clasificación se realiza con el fin de reducir la complejidad del diseño y la implementación de un sistema. Para 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 identificados no pueden implementarse en módulos

1 Los programming idioms son patrones de bajo nivel específicos a un lenguaje de programación. Estos patrones describen cómo solucionar ciertos problemas específicos a la implementación en un lenguaje en particular [24].

independientes [2]. El programa derivado de dicha implementación presenta dificultades de mantenimiento 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 carecen de mecanismos para realizarlo en la fase de implementación. [2]

2. Evolución de las Metodologías de Programación

La ingeniería de software ha atravesado un largo camino comenzando en los lenguajes a nivel máquina, pasando por la programación procedural y llegando a la programación orientada a objetos (POO). Esta evolución de las metodologías de programación permite a los ingenieros enfrentarse con problemas de mayor complejidad y nivel de abstracción que en décadas anteriores [2]. Si bien todos los lenguajes de programación proveen un conjunto limitado de abstracciones, estos mecanismos no permiten mantener la separación de concerns en el código de una aplicación [3].

Actualmente, la programación orientada a objetos (POO) es la metodología elegida en los nuevos proyectos de desarrollo de software. La mayor ventaja de este paradigma reside en el modelado del comportamiento común [2], por lo que los sistemas orientados a objetos son desarrollados mapeando las entidades del mundo real del dominio de la aplicación en una jerarquía de clases [4]. Sin embargo, no todos los requerimientos de la aplicación pueden mapearse a una única unidad modular (clase). La POO no cumple un buen papel en abordar estos requerimientos ya que quedan dispersos en varios módulos, generalmente no relacionados entre sí [2]. Como consecuencia, se observan dificultados en el mantenimiento estos sistemas.

2.1. 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]. Está comúnmente aceptada la premisa de que la mejor manera de solucionar la complejidad es simplificándola. En diseño de software, la mejor manera de simplificar un sistema complejo es identificar cada concern y luego asignarlo a un único módulo que lo realice [2].

La programación orientada a objetos fue desarrollada en respuesta a la necesidad de dicha modularización, en particular, este paradigma permite realizar una buena modularización de los core concerns, aunque falla cuando se trata de modularizar crosscutting concerns.

Los módulos centrales en las aplicaciones orientadas a objetos por lo general se encuentran débilmente acoplados gracias al uso de interfaces. No sucede lo mismo para los crosscutting concerns, debido a que la implementación 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 que proveen un conjunto de servicios y objetos que usan estos servicios respectivamente. En este tipo de sistemas se modulariza en clases e interfaces los servicios provistos. Sin embargo, el pedido del servicio se encuentra disperso en todos los clientes, provocando un fuerte acoplamiento entre los módulos que necesitan los servicios y el módulo que lo provee.

La Fig. II-1 A muestra cómo un sistema bancario implementa el registro de eventos de la información utilizando el paradigma orientado a objetos. El módulo de logging representa el servicio provisto (servidor) y los módulos de “accounting”, “ATM” y

“database” utilizan este servicio. El módulo de logging puede implementarse utilizando una interface y así proveer los siguientes beneficios:

- Disminuir el acoplamiento entre los clientes y la implementación 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 de logging, los módulos clientes necesitan el código para invocar la API del servicio. En color gris se denota el acoplamiento de cada módulo [2].

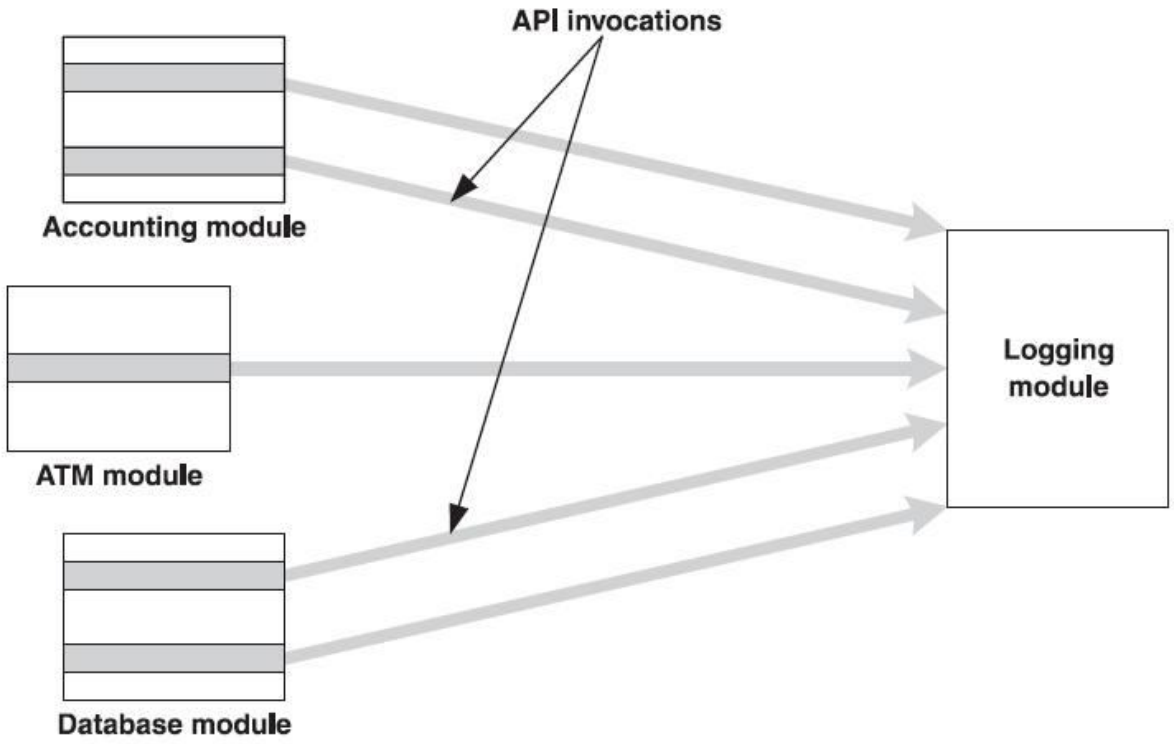

Fig. II-1.Implementación del concern logging utilizando las técnicas convencionales.

En consecuencia, se observa 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 crosscutting concerns no es lo suficientemente independiente y el código que invocan a los servicios se encuentra entremezclado con el código de la lógica de los clientes. Por esta razón, la mantenibilidad del sistema en referencia a los crosscutting concerns es más dificultosa y costosa, conllevando a problemas a la hora de modificar, agregar o reutilizar este tipo de concerns [3]. Estos problemas tienen su origen en 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) [2] es un paradigma que permite dar solución al 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 un crosscutting concern. La combinación de los aspectos con los módulos centrales de la aplicación se denomina weaving la cuál es llevada a cabo con el fin de formar la versión final del sistema final.

La Fig. II-2 muestra la implementación del mismo ejemplo de logging expuesto anteriormente (Fig. II-1) utilizando la orientación a aspectos. La lógica del logging reside dentro del aspecto "logging" y los clientes no tienen código que haga referencia al mismo. Con esta modularización, cualquier cambio al requerimiento de logging afecta solo al aspecto logging aislando completamente a los clientes. En consecuencia, la implementación presenta los siguientes beneficios en comparación con el ejemplo citado previamente: mayor modularización y claridad en las responsabilidades de cada módulo, mayor reusabilidad y flexibilidad y menor acoplamiento de los módulos.

3. Implementación Orientada a Aspectos

La metodología de programación debe ser implementada con el fin de poder aplicarla a sistemas de software concretos. Para ellos, se deben definir los siguientes componentes:

- Lenguaje de especificación: describe los constructores y sintáxis del lenguaje que va a ser usado para implementar tanto la lógica de los core concerns como 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.

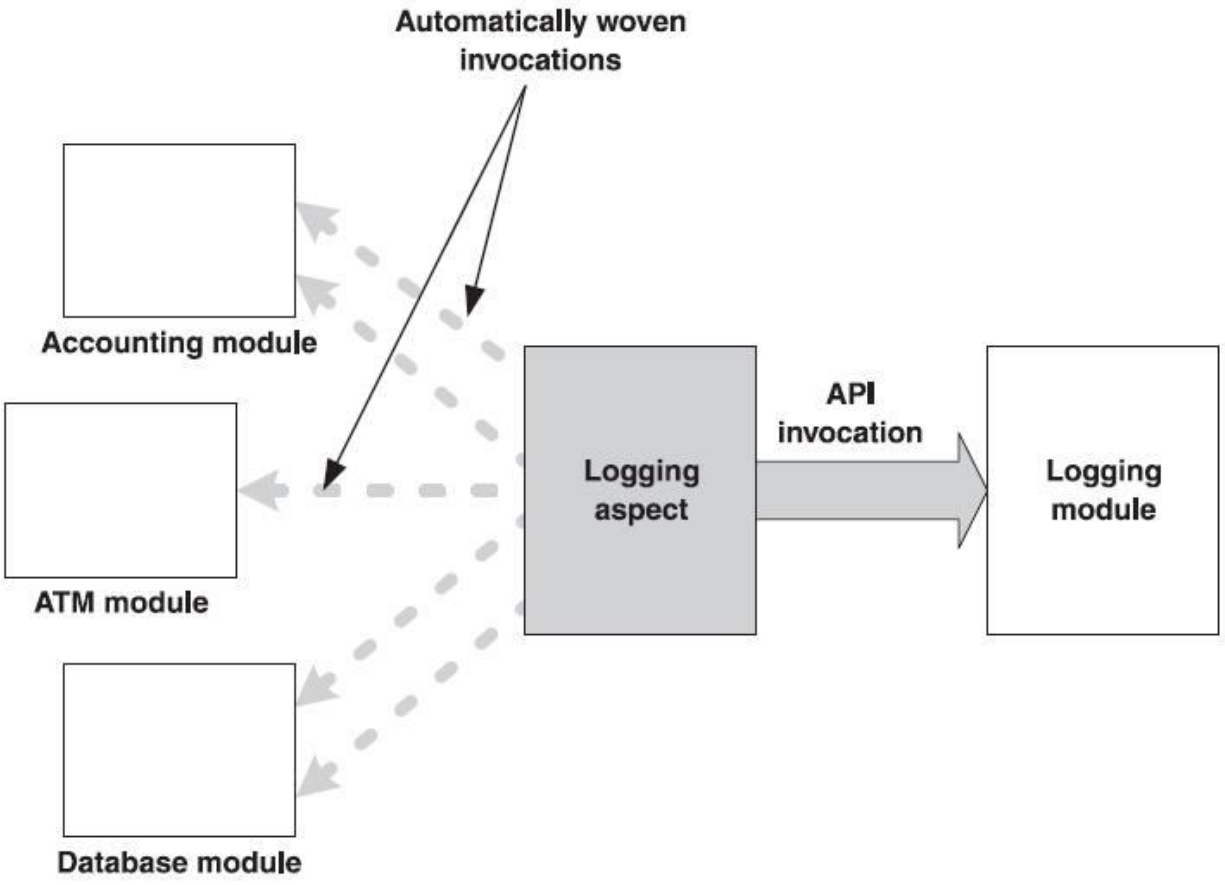

Fig. II-2.Implementación del concern logging utilizando las técnicas de AOP.

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. Las reglas de weaving especifican cómo integrar los concerns implementados a fin de formar el sistema final. El proceso que

combina los concerns individuales de acuerdo a estas reglas se denomina *weaving* y el responsable que realiza está tarea es llamado *weaver*. El poder de AOP proviene de la forma en que las reglas de weaving pueden ser expresadas ya que se pueden especificar unas pocas líneas de código.

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

- AspectJ: es una extensión de propósito general orientada a aspectos del lenguaje de programación *Java*. El primer paso en la compilación, weaving, adhiere los aspectos y las clases como si el código de los aspectos estuviera dispersado a través de las clases centrales. El segundo paso realiza la compilación normal de Java utilizando el comando *javac* [20]. AspectJ utiliza aspectos, join-point, pointcuts y advices como constructores del lenguaje. Los aspectos son unidades modulares que encapsulan los crosscutting concerns. Los joint-points representan cualquier punto de ejecución del programa en el que su comportamiento puede ser modificado por un aspecto. Los pointcut se utilizan dentro de los aspectos para especificar uno o más join-points en los módulos principales o incluso en otros aspectos. Un advice define un tipo de método que se utiliza para encapsular código de crosscuttings [3].

- SpringAOP: Un componente clave en Spring es el framework de AOP. Si bien no existe la restricción de utilizar aspectos, esta funcionalidad está soportada en el framework. SpringAOP provee solamente joinpoints a nivel de métodos y los aspectos son configurados utilizando la sintáxis de definición normal de los beans [22].

- JBossAOP: es un framework orientado a aspectos construido completamente en Java. Puede ser usado en cualquier ambiente de programación e integrado en el application server. JBossAOP no es

simplemente un framework, sino que provee un conjunto de aspectos que pueden ser aplicados mediante anotaciones, pointcuts o dinámicamente en tiempo de ejecución. Esto último, denominado weaving dinámico, es una de las principales ventajas del mismo [21].

4. 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 qué partes del código corresponden a un crosscutting concern es muy dificultoso [4].

Se pueden distinguir tres fases diferentes para realizar y evolucionar la migración a aspectos: “aspect exploration”, “aspect extraction” y “aspect evolution”’ [38].

- Aspect Exploration: Previo a la introducción de aspectos en el código de un software existente, se debe poder explorar si el sistema exhibe o no algún crosscutting concern al cual valga la pena ser extraído a aspectos. La tiranía de la descomposición dominante [6] implica que es probable que grandes sistemas los contengan. Durante esta etapa se trata de descubrir aspectos candidatos, se intenta discernir qué representan los crosscutting concerns, dónde y cómo están implementados y cuál es su impacto en la calidad del programa.

- Aspect Extraction: Una vez que los crosscutting concerns han sido identificados en un sistema de software y su impacto ha sido evaluado, se puede considerar migrar el mismo hacia una versión orientada a aspectos. En el caso de decidir realizar dicha migración, se necesita una forma de convertir los aspectos candidatos, esto es los crosscutting concerns identificados en la

fase de exploración, en aspectos reales. Al mismo tiempo, hay una necesidad de técnicas para testear el software refactorizado con el fin de asegurarse de que la nueva versión continúa trabajando como se esperaba y para manejar los pasos de las modificaciones durante la fase de transición.

- Aspect Evolution: De acuerdo a la primera ley de evolución de software de Belady y Lehman [39], todo sistema de software que está siendo usado se someterá continuamente a cambios o se convertirá en obsoleto luego de un período de tiempo. No hay razones para creer que esta ley no se aplica a los sistemas de software orientado a aspectos, por lo que surgen preguntas referidas a su evolución. Esta fase intenta responder preguntas tales como: ¿cuán diferente es la evolución de un sistema orientado a aspectos de uno tradicional?, ¿pueden las mismas técnicas de evolución de software tradicional ser aplicadas en el nuevo paradigma?, ¿los nuevos mecanismos de abstracción de AOP pueden incurrir en nuevos problemas de evolución de software que requieren nuevas soluciones?

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

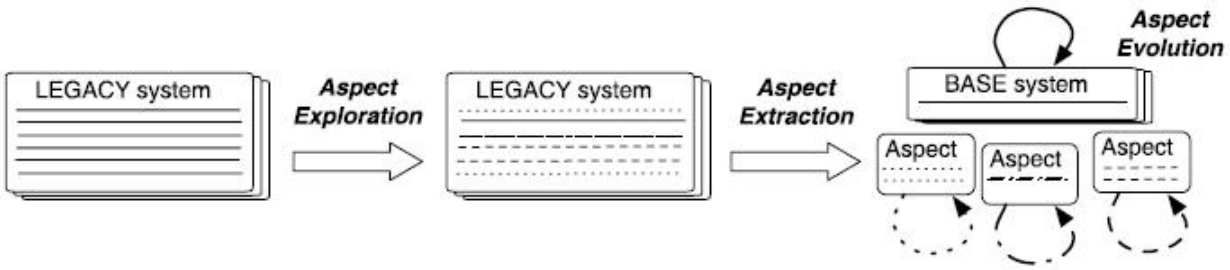

Fig. II-3. Migración y evolución de un sistema legado a un sistema orientado a aspectos.

4.1. Proceso de Migración

Debido al tamaño y complejidad de los sistemas orientados a objetos, existe la necesidad de técnicas y herramientas que automaticen la migración de estos a la

orientación a aspectos [8]. La mayoría de las herramientas distinguen dos fases en el proceso de migración: aspect exploration and aspect extraction [38]. Se define a aspect exploration como la actividad de identificar y analizar crosscutting concerns en sistemas no orientado a aspectos. Aspect extraction es la actividad de separar el código de los crosscutting concerns del sistema moviéndolo en uno o más aspectos y removiéndolo del código original [38].

4.1.1. Aspect Exploration

Kellens et Al. [25] distinguen tres categorías principales de técnicas que pueden ayudar a localizar 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 de vida de un software [26] tales como requerimientos y análisis de dominio [27, 28, 29] y diseño de la arquitectura [30]. A pesar de que la técnica mencionada puede ayudar a identificar ciertos crosscutting concerns en un sistema de software, se la considera menos prometedora que aquellas en las cuales el enfoque se centra en el código fuente. Esto se debe a que los documentos de requerimientos y la arquitectura generalmente se encuentran desactualizados u obsoletos.

- **Dedicated Browsers:** Una segunda clase de enfoque son los browsers de código avanzado de propósito especial, los cuales ayudan al desarrollador a navegar manualmente el código fuente de un sistema para explorarlo en busca de crosscutting concerns. Esta técnica comienza típicamente desde una ubicación en el código llamada “seed” como punto de entrada para guiar a los usuarios mediante la sugerencia de otros lugares en el que el mismo concern pueda aparecer. De esta manera, se construye de forma interactiva un modelo de las diferentes zonas del código que constituyen un crosscutting concern. Se pueden listar los siguientes ejemplos clasificados bajo dedicated

browsers: Concern Graphs [31], Intensional Views [32], Aspect Browser [33], (Extended) Aspect Mining Tool [34, 35], SoQueT [36] y Prism [37].

- **Técnicas de 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 concerns simultáneamente [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.

4.1.2. Aspect Extraction

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 descrito 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.