Transactions and Schema Evolution in a Persistent Object-Oriented Programming System

Rui Humberto Ribeiro Pereira

Escuela Superior de Ingeniería Informática

University of Vigo

Supervisor: Dr. J. Baltasar García Perez-Schofield

A thesis submitted for the degree of

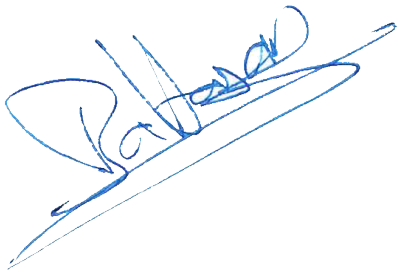
Doctor by the University of Vigo

June 2015

AUTORIZACIÓN DE DEPÓSITO

Dr. José Baltasar Garca Perez-Schofield en calidad de director de la Tesis Doctoral titulada "*Transactions and Schema Evolution in a Persistent Object-Oriented Programming System* realizada por D. Rui Humberto Ribeiro Pereira, autoriza y dan el visto bueno para su depósito y defensa en la Universidad de Vigo.

Vigo, 23 de Xuño de 2015



José Baltasar Garca Perez-Schofield

Resumen

Las aplicaciones sufren un proceso de evolución continua con profundo impacto en el modelo de datos, lo cual exige actualizaciones frecuentes tanto de la estructura de clases de la aplicación como de la estructura de la base de datos. En las bases de datos orientadas a objetos, la evolución es afectada por dos problemas, cada uno en diferentes capas: (1) la evolución del esquema, en la capa de metadatos, y (2) la adaptación de instancias de objetos, en la capa de datos. En esta tesis doctoral se aborda este doble problema (evolución del esquema y adaptación de instancias), también conocido como evolución de las bases de datos, así como los problemas de concurrencia y de recuperación de errores, para los que se propone un nuevo metamodelo y su implementación orientada a aspectos.

Aunque estudiada exhaustivamente en las últimas décadas, la evolución de las bases de datos orientadas a objetos continúa siendo un proceso tedioso y propenso a errores para programadores y administradores de bases de datos. Las actuales bases de datos orientadas a objetos contienen funcionalidades que ayudan al programador a lidiar con la persistencia de objetos, así como con otros problemas relacionados, como la evolución de la base de datos, la concurrencia y el control de errores. La mayoría de los sistemas cuenta con mecanismos transparentes para atacar estos problemas y reducir el trabajo del programador, al mantener sincronizadas las capas de datos y de la aplicación. Sin embargo, algunas actualizaciones, como mover clases en la jerarquía de herencia, renombrar atributos y efectuar cambios semánticos en el contenido de los atributos, exigen la intervención del programador para convertir los datos de la estructura anterior. Pese

a esos mecanismos transparentes, los tipos más frecuentes de actualizaciones de estructuras de clases identificados en trabajos anteriores [Advani *et al.*, 2006; Piccioni *et al.*, 2011], al no tener soporte, necesitan programas auxiliares para leer secuencialmente todos los objetos, convertir los contenidos y volver a almacenarlos en la nueva versión de la clase. Además, cuando se mueve una clase dentro de su propia jerarquía de herencia, estos sistemas requieren una clase temporal con nombre propio, la cual debe renombrarse al final del proceso de conversión. En consecuencia, los programadores deben dedicar considerable esfuerzo a la evolución de la base de datos. Este problema de la evolución de la base de datos, así como los relacionados con la persistencia, fue precisamente lo que motivó la búsqueda de soluciones para mejorar el trabajo del programador, haciéndolo más productivo y menos propenso a errores.

Algunas investigaciones anteriores [Kiczales *et al.*, 1997] han demostrado que las técnicas de programación orientada a aspectos (POA) permiten el desarrollo de sistemas reutilizables y flexibles. Tanto cuanto sabemos, la principal investigación relacionada con la aplicación de técnicas POA en la evolución de bases de datos orientadas a objetos apunta soluciones parciales, y en ocasiones totales, del problema de la evolución de la base de datos. Se trata del trabajo realizado por Rashid & Leidenfrost [2004, 2006]; Rashid & Sawyer [2001, 2005], quienes han contribuido a mejorar la flexibilidad y las características de personalización de las técnicas de evolución existentes. Estos autores proponen la integración de todas las técnicas aisladas en una única y potente solución aplicando técnicas POA, lo cual demuestra que la orientación a aspectos de la evolución de la base de datos permite un mayor grado de flexibilidad y personalización. Sin embargo, ni éste ni ningún otro trabajo anterior [Kusspuswami *et al.*, 2007] se ocupa del paradigma de persistencia ortogonal [Atkinson & Morrison, 1995; Dearle *et al.*, 2010].

Con respecto a la gestión de transacciones y al control de fallos,

Kienzle & Guerraoui [2002]; Kienzle *et al.* [2009] y Fabry [2005] demostraron que puede lograrse transparencia sintáctica entre el principal programa de base y aspectos.

Esta tesis doctoral aborda el problema de la evolución de la base de datos, en particular en el ámbito de sistemas de persistencia ortogonal orientados a objetos, en sus dos vertientes: evolución de esquemas y adaptación de instancias. Estos sistemas motivaron nuestra investigación por sus características intrínsecas, que en algunos casos son exclusivas y no se observan en sistemas no ortogonales, lo cual permite un abordaje innovador del problema ya mencionado.

En teoría, la persistencia ortogonal [Atkinson & Morrison, 1995; Dearle *et al.*, 2010] resuelve problemas de transparencia y de mecanismos nativos de consulta, lo que mejora la productividad y la calidad del trabajo. Además, este paradigma de persistencia de datos crea las condiciones para otros abordajes de la evolución de las bases de datos, ya que sus características permiten una evolución progresiva de los esquemas de la aplicación, así como una integración perfecta de la aplicación con la base de datos. Nuestra contribución se centra en identificar las ventajas y desventajas de la persistencia ortogonal para ese fin, sosteniendo que estas características crean condiciones especiales para la modularización de la persistencia y los problemas de la evolución de las bases de datos al aplicar técnicas POA. Así, esta investigación pretende demostrar los beneficios de combinar el paradigma de persistencia ortogonal con técnicas POA. Asimismo, pretendemos proponer un nuevo enfoque orientado a aspectos para modularizar los aspectos de la evolución de la base de datos: los técnicos y los que dependen del dominio de la aplicación. En consecuencia, esta tesis presenta nuestro metamodelo y su prototipo, que explora los beneficios de esta combinación. Nuestro metamodelo y correspondiente *framework* siguen un enfoque orientado a aspectos, centrado en el contexto de persistencia ortogonal orientada a objetos. Caracterizado por su simplicidad para obtener mecanismos eficientes y transparentes de evolución de bases

de datos, este metamodelo se inspira en el de Rashid & Sawyer [1999], si bien define un conjunto de metaclases mucho más limitado. Estas metaclases constituyen las clases, relaciones y todos los metadatos de las aplicaciones necesarios para emular los objetos de las aplicaciones en todas las versiones de clases existentes. Nuestro metamodelo también toma en cuenta cuestiones importantes como la consistencia e integridad de los datos y la identidad de los objetos. Para lograr estos objetivos, se definió un conjunto de entidades: objetos, metaobjetos y metaclases. El metamodelo tiene seis metaclases:

(1) CVMO Class Version Meta-Object - Cada CVMO soporta los metadatos de una clase en una versión específica; (2) RMO - Root Meta-Object - Cada RMO representa un objeto raíz persistente; (3) IMO - Instance Meta-Object - Representa una instancia lógica de objetos de las aplicaciones, es decir, las instancias de objetos de la aplicación independientemente de su versión física en la base de datos; (4) AMO - Attribute Meta-Object - Representa una relación entre dos objetos, o entre un objeto y una matriz; (5) UBMO - Update/Backdate Meta-Object - Los metaobjetos UBMO soportan expresiones de punto de corte para representar asignaciones explícitas entre versiones de clase; (6) AspMO - Aspect Meta-Objects - Permiten el almacenamiento de aspectos, es decir, los aspectos son persistentes en la base de datos.

Además de estos metaobjetos, encontramos también los objetos de datos, que consisten en objetos de las aplicaciones almacenados en la base de datos según las correspondientes versiones de las clases. Los metaobjetos, que, no debe olvidarse, son instancias de las metaclases, forman la capa de metadatos de la base de datos. Por otra parte, los objetos componen la capa de los datos. Cada uno soporta un objeto de datos de la aplicación (*facet* [Clamen, 1992]) de acuerdo con una determinada versión de clase.

Con respecto a la identidad de los objetos, cada instancia de objetos de la aplicación tiene un Identificador de Objeto Lógico (IOL, o LOID por

sus siglas en inglés), mientras que los objetos y metaobjetos de la base de datos tienen Identificadores de Objetos (IO, u OID por sus siglas en inglés). Lo que aquí se sostiene es que nuestro metamodelo reduce el número de metaobjetos afectados (metadatos) en cada actualización de la estructura de clases de la aplicación.

Nuestro metamodelo soporta múltiples versiones de una estructura de clases a través de la aplicación de una estrategia de versionado, lo cual permite la compatibilidad bidireccional de la aplicación entre diferentes versiones de cada clase. En otras palabras, la estructura de la base de datos puede actualizarse, dado que las versiones anteriores de la aplicación continúan funcionando, del mismo modo que las aplicaciones posteriores reconocen la estructura actualizada. El diseño del metamodelo se completa con las características específicas de los sistemas de persistencia ortogonal, así como con una estrategia de enriquecimiento de metadatos en el código fuente de la aplicación.

De una perspectiva de la programación, más que de la base de datos misma, nuestro enfoque pretende dar apoyo a los programadores a través de servicios relacionados con la persistencia y la evolución de la base de datos, lo que les permite concentrarse únicamente en la lógica de la aplicación y en la definición de la estructura de clases dentro del ámbito de la aplicación.

Con respecto a la viabilidad de esta propuesta, se desarrolló un prototipo que consiste en un *framework* mediador de la interacción entre las aplicaciones y la base de datos através de mecanismos de persistencia ortogonal. Estos mecanismos son introducidos en las aplicaciones como aspectos (es decir, en un sentido orientado a aspectos): los objetos no requieren extensiones de súper clases o implementación de interfaces ni contienen anotaciones especiales. Además, el *framework* también maneja correctamente las clases de tipo paramétrico. Sin embargo, las clases pertenecientes a entornos de programación deben ser consideradas no versionables, debido a las restricciones impuestas por la *Java Virtual Machine*.

En lo que se refiere a la concurrencia, el *framework* dota a las aplicaciones de un entorno *multithreading* que soporta las transacciones y la recuperación de errores de las bases de datos. Proporciona asimismo una API (interfaz de programación de aplicaciones) para que los programadores controlen la concurrencia. Esta API permite delimitar las transacciones de la base de datos y las operaciones de *commit* y *rollback*, de una perspectiva de programación.

El sistema tiene una arquitectura orientada a aspectos, lo que permite la modularización de algunas cuestiones transversales a las aplicaciones y al *framework*. Tanto el funcionamiento del *framework* interno como el modo en que se dota de persistencia a las aplicaciones tienen una orientación a aspectos. Esta arquitectura permite el fácil reemplazo de los mecanismos de evolución del *framework*, así como la reutilización del *framework* gracias a su flexibilidad. El *framework* se compone de un módulo principal, un conjunto de aspectos, una base de datos y un preprocesador.

Nuestro *framework* orientado a aspectos permite la modularización de la persistencia de las aplicaciones y la evolución de la base de datos. Los programadores pueden hacer actualizaciones en la estructura de clases de la aplicación, dado que el *framework* va a crear una nueva versión de estas estructuras en la base de datos. Nuestra propuesta de evolución de esquemas es acumulativa, es decir, las nuevas versiones son siempre añadidas a la capa de metadatos de la base de datos. Este abordaje acumulativo se hace posible gracias al paradigma de persistencia ortogonal, el cual exime al programador de tener que intervenir al nivel de la base de datos y evita el uso de primitivas de evolución de esquemas.

Con respecto a la adaptación de instancias, el *framework* suele resolver este problema de forma autónoma por medio de un mecanismo por defecto basado en la asignación directa. Tales mecanismos por defecto sólo son incapaces de desempeñar la adaptación autónoma de las instancias si ocurren variaciones estructurales o semánticas, en

cuyos casos los programadores deben escribir el código de conversión como si fueran módulos orientados a aspectos, lo cual ampliará los mecanismos por defecto del *framework*. Para escribir estos módulos orientados a aspectos, hemos desarrollado un nuevo tipo de expresiones XML de punto de corte. Al usar tales mecanismos durante el funcionamiento normal de las aplicaciones, nuestro *framework* adapta los objetos cargados de la base de datos.

Este enfoque *lazy* de la adaptación dota a las aplicaciones de instancias de objetos de acuerdo con las estructuras de sus clases internas.

La adaptación de instancias es una incumbencia cruzada de las clases sujetas a evolución. Los aspectos de adaptación de instancias, de la incumbencia del dominio de la aplicación, se resolvieron gracias a nuestras expresiones de punto de corte, mientras que para los aspectos técnicos se usaron los mecanismos base del *framework*. Al usar las expresiones XML de punto de corte, se amplía el mecanismo de adaptación de instancias del *framework*, que de este modo permanece indiferente al problema.

Estas expresiones se estructuran en dos partes: las condiciones de disparo y la acción. Durante el tiempo de ejecución, una acción es disparada cuando la instancia de objeto cargada de la base de datos y la acción requerida por la aplicación reúnen las condiciones de una expresión. La información de la acción es usada para ampliar el aspecto de adaptación de instancias del *framework* con el comportamiento de conversión necesario. Este código fuente definido por el usuario le permite al *framework* conocer la semántica de la actualización aplicada a las estructuras de clase de las aplicaciones. Estas definiciones del usuario establecen una asignación explícita entre dos instancias de un objeto en dos versiones de clase diferentes: una en la base de datos y la otra que es la esperada por la aplicación. Se denomina a ello Asignación Definida por el Usuario.

Aunque estos módulos dependen del código fuente de las aplicaciones, su interfaz con el *framework* está bien definida. Además, todas las in-

terfaces referenciadas en su implementación son verificadas estadísticamente por el tejedor de la base de datos. Por tanto, el *framework* y la aplicación se mantienen indiferentes al problema de la adaptación de instancias, al tiempo que el *framework* resuelve la evolución de clases de forma transparente.

Las expresiones están escritas en lenguaje XML para permitir una edición más fácil por medio de una herramienta gráfica o incluso de un editor de texto. Cada expresión de punto de corte está soportada en la capa de metaobjetos como un metaobjeto UBMO.

En nuestra implementación, la acción (el cuerpo del consejo) está escrita en lenguaje de programación Java, embebida en un nodo XML. Las condiciones de disparo de la acción se especifican por medio de un grupo de parámetros de correspondencia (entre versiones de clases) representados como atributos de un nodo XML. Conviene notar que nuestra propuesta no requiere otro lenguaje de programación como Vegal [Rashid & Leidenfrost, 2004]. El programador escribe un código de conversión en el mismo lenguaje de programación utilizado por el código fuente de la aplicación. Más aún, dentro del cuerpo del consejo (la función de conversión definida por el usuario), el programador puede usar información local y no local perteneciente al objeto que se pretende convertir. Por tanto, nuestra propuesta vuelve las funciones de conversión más ricas y expresivas.

Las potenciales ventajas del prototipo fueron sometidas a una prueba de rendimiento en un estudio de caso. Durante nuestro estudio de caso, los resultados confirmaron que la transparencia de los mecanismos tienen efectos positivos en la productividad de los programadores, al simplificar todo el proceso de evolución tanto a nivel de la aplicación como de la base de datos. El metamodelo mismo fue también cotejado con respecto a su complejidad y agilidad. Aunque inspirado en la propuesta de Rashid & Sawyer [1999], nuestro metamodelo obtiene mejores resultados en términos de complejidad. Por otra parte, comparado con otros metamodelos, éste requiere menos modificaciones

de metaobjetos en cada paso de la evolución del esquema. En el escenario específico del estudio de caso, nuestros resultados muestran que el metamodelo propuesto requiere menos metaobjetos y menos metaobjetos modificados. Es de suponer, entonces, que nuestro metamodelo está mejor adaptado para soportar servicios persistentes, ya que requiere menos metaobjetos para representar los datos de las aplicaciones, así como las actualizaciones a sus esquemas. Además el desempeño global del sistema puede beneficiarse gracias a la simplicidad del metamodelo. Sin embargo, nuestro metamodelo exige el uso intensivo de reflexión, que afecta el desempeño del sistema y cuyo coste resulta injustificado. El desempeño del prototipo deberá ser objeto de mayor investigación.

También se realizaron otras pruebas para validar la robustez del prototipo y del metamodelo. Para llevar a cabo estas pruebas, se usó una base de datos 007 [Carey *et al.*, 1993], debido a la gran complejidad de su modelo de datos. Toda vez que el prototipo desarrollado contiene algunos rasgos que no se observan en otros sistemas conocidos, no fue posible comparar su rendimiento directamente. Sin embargo, sí se han realizado pruebas de rendimiento, que están disponibles para futuras comparaciones de la solución con otros sistemas equivalentes.

Para evaluar nuestro abordaje en un escenario real, se desarrolló una aplicación de prueba de concepto. Esta aplicación usa datos obtenidos de la base de datos geográfica en línea OpenStreetMap (http://www.openstreetmap.org), que permite la exportación de una superficie definida por el usuario. Los archivos OSM exportados, en formato XML (http://wiki.openstreetmap.org/wiki/OSM XML), contienen todos los datos relacionados con esa superficie geográfica. Cada uno contiene las coordenadas de los límites y se estructura como un conjunto de objetos como *Nodes*, *Ways* y *Relations*. Nuestra aplicación geográfica permite que el usuario importe esos archivos OSM para la base de datos local. El usuario puede explorar estas superficies geográficas almacenadas localmente y por tanto tener acceso a toda

la información (sus puntos de interés).

En el desarrollo inicial de esta aplicación geográfica no se implementaron los mecanismos de persistencia de datos. Por medio de pequeños cambios en el código fuente, se dotó a los objetos de la aplicación del aspecto persistencia. El resto del código fuente de la aplicación se mantiene inalterado, ajeno al problema de persistencia y desacoplado del *framework*. Además, el aspecto persistencia no viola el comportamiento especificado de los módulos base, volviéndose así un espectador [Leavens & Clifton, 2007; Przybylek, 2013]. Pese a dotar a las aplicaciones de comportamiento adicional, al ser desconectadas continúan funcionando indiferentes a la falta de mecanismos de persistencia de datos.

Estas experiencias reales mostraron que las aplicaciones continúan funcionando indiferentes a la persistencia y evolución de la base de datos. En este estudio de caso, el *framework* resultó una herramienta útil para programadores y administradores de bases de datos.

Cuando comparados con otros sistemas [Corporation, 2010; Ltd, 2011; Paterson *et al.*, 2006], esta propuesta le proporciona al programador mecanismos más transparentes de soporte a la evolución de bases de datos. En algunos sistemas como los ya mencionados, cuando ocurren variaciones estructurales o semánticas el programador debe escribir programas auxiliares que lleven a cabo la migración de los datos. Por el contrario, en el *framework* propuesto, pueden añadirse expresiones de punto de corte en el sistema para extender sus mecanismos base.

En investigaciones anteriores, fue posible modularizar la persistencia y la evolución de las bases de datos. Sin embargo, nuestra propuesta supera esos abordajes anteriores en los siguientes aspectos:

- Permitirle al programador mantenerse semi-indiferente a la evolución de la base de datos, dado que solo deberá prestar atención a las actualizaciones semánticas en los esquemas de la aplicación. El programador puede modificar las estructuras de datos persis-

tentes en el código fuente de la aplicación, dado que el *framework* producirá una nueva versión del esquema de la base de datos. No se requieren primitivas para la evolución del esquema.

- Permitir que las aplicaciones sean indiferentes a la persistencia de los datos y la evolución. Con este *framework*, las aplicaciones siguen funcionando sin alteraciones, incluso después de la actualización de estructuras de datos persistentes. En la propuesta de Rashid & Leidenfrost [2006], las aplicaciones deben escribirse en un lenguaje de programación especial que pueda reconocer todas las versiones de clases existentes.
- Con respecto al metamodelo de Rashid *et al.*, en nuestra propuesta el soporte de la capa de metaobjetos del sistema es más sencillo y por tanto más capaz de soportar elevados niveles de complejidad en esquemas multiversionados.
- En la propuesta de Kusspuswami *et al.* [2007], los *aspectos de rol* de las clases son implementados como aspectos *update/backdate*. Si bien tal abordaje es en esencia similar al que se propone en esta tesis, no ofrece sin embargo una interfaz sistemática para declarar e implementar aspectos. El programador debe implementar esos aspectos usando AspectJ u otro lenguaje de programación orientado a aspectos.

Las limitaciones principales del *framework* se refieren a cuestiones de rendimiento y a la restricción del uso simultáneo de varias JVM.

Debido a la emulación de la versión de los objetos, la adaptación de las versiones de clases de las aplicaciones produce un retardo considerable (si los objetos aún no están en cache). Aunque esta degradación del desempeño es inevitable e inherente al versionado de clases, el sistema puede ser diversamente optimizado para mejorar su desempeño global. Es nuestra convicción que la flexibilidad y personalización que permite la orientación a aspectos de la evolución de bases de datos permite otros abordajes a la adaptación. Esta flexibilidad y personalización

facilita los medios necesarios para aplicar las estrategias adecuadas a cada caso.

En cuanto al desempeño, algunas limitaciones actuales del prototipo resultan del uso de una base de datos orientada a objetos como repositorio de objetos. Esta base de datos proporciona mecanismos de interacción de alto nivel en un esquema de versión única, por lo que no se adapta demasiado bien a los requisitos de nuestro *framework*. En teoría, un almacén de objetos específico para nuestro *framework* y metamodelo mejorará el desempeño global y la ortogonalidad, evitándose numerosas operaciones adicionales. Este repositorio especializado de objetos podrá ser un perfeccionamiento futuro del *framework*.

El uso de una única JVM constituye una limitación derivada del paradigma de persistencia ortogonal, en el que la persistencia de los objetos depende de su accesibilidad. Dos objetos instanciados en dos JVM diferentes no comparten la memoria. Por ello, cuando uno de estos objetos es actualizado, el otro no se encontrará sincronizado. La sincronización de la memoria y la exclusión mutua a través de varias JVM son cuestiones aún sin resolver y que serán abordadas en futuras investigaciones.

Palabras Clave: Aspect-Oriented Programming, Schema Evolution, Instance Adaptation, Database Evolution

Nota: En el Appendix E - Resumen, puede encontrarse un resumen corto, en español.

Abstract

Applications are subject of a continuous evolution process with a profound impact on their underlining data model, hence requiring frequent updates in the applications' class structure and database structure as well. This twofold problem, schema evolution and instance adaptation, usually known as database evolution, is addressed in this thesis. Additionally, we address concurrency and error recovery problems with a novel meta-model and its aspect-oriented implementation.

Modern object-oriented databases provide features that help programmers deal with object persistence, as well as all related problems such as database evolution, concurrency and error handling. In most systems there are transparent mechanisms to address these problems, nonetheless the database evolution problem still requires some human intervention, which consumes much of programmers' and database administrators' work effort.

Earlier research works have demonstrated that aspect-oriented programming (AOP) techniques enable the development of flexible and pluggable systems. In these earlier works, the schema evolution and the instance adaptation problems were addressed as database management concerns. However, none of this research was focused on orthogonal persistent systems. We argue that AOP techniques are well suited to address these problems in orthogonal persistent systems. Regarding the concurrency and error recovery, earlier research showed that only syntactic obliviousness between the base program and aspects is possible.

Our meta-model and framework follow an aspect-oriented approach focused on the object-oriented orthogonal persistent context. The

proposed meta-model is characterized by its simplicity in order to achieve efficient and transparent database evolution mechanisms. Our meta-model supports multiple versions of a class structure by applying a class versioning strategy. Thus, enabling bidirectional application compatibility among versions of each class structure. That is to say, the database structure can be updated because earlier applications continue to work, as well as later applications that have only known the updated class structure. The specific characteristics of orthogonal persistent systems, as well as a metadata enrichment strategy within the application's source code, complete the inception of the meta-model and have motivated our research work.

To test the feasibility of the approach, a prototype was developed. Our prototype is a framework that mediates the interaction between applications and the database, providing them with orthogonal persistence mechanisms. These mechanisms are introduced into applications as an *aspect* in the aspect-oriented sense. Objects do not require the extension of any super class, the implementation of an interface nor contain a particular annotation. Parametric type classes are also correctly handled by our framework. However, classes that belong to the programming environment must not be handled as versionable due to restrictions imposed by the Java Virtual Machine. Regarding concurrency support, the framework provides the applications with a multithreaded environment which supports database transactions and error recovery.

The framework keeps applications oblivious to the database evolution problem, as well as persistence. Programmers can update the applications' class structure because the framework will produce a new version for it at the database metadata layer. Using our XML based pointcut/advice constructs, the framework's instance adaptation mechanism is extended, hence keeping the framework also oblivious to this problem.

The potential developing gains provided by the prototype were bench-

marked. In our case study, the results confirm that mechanisms' transparency has positive repercussions on the programmer's productivity, simplifying the entire evolution process at application and database levels. The meta-model itself also was benchmarked in terms of complexity and agility. Compared with other meta-models, it requires less meta-object modifications in each schema evolution step. Other types of tests were carried out in order to validate prototype and meta-model robustness. In order to perform these tests, we used an OO7 small size database due to its data model complexity. Since the developed prototype offers some features that were not observed in other known systems, performance benchmarks were not possible. However, the developed benchmark is now available to perform future performance comparisons with equivalent systems.

In order to test our approach in a real world scenario, we developed a proof-of-concept application. This application was developed without any persistence mechanisms. Using our framework and minor changes applied to the application's source code, we added these mechanisms. Furthermore, we tested the application in a schema evolution scenario. This real world experience using our framework showed that applications remains oblivious to persistence and database evolution. In this case study, our framework proved to be a useful tool for programmers and database administrators. Performance issues and the single Java Virtual Machine concurrent model are the major limitations found in the framework.

Keywords: Aspect-Oriented Programming, Schema Evolution, Instance Adaptation, Database Evolution

---

Note: Appendix D - Extended abstract - contains an extended abstract of the thesis. Additionally, Appendix E - Resumen - contains a short abstract in Spanish.

Contents

List of Figures

Acronyms

ACID Atomicity, Consistency, Isolation and Durability

AOAF Aspect-Oriented Application Framework

AOP Aspect-Oriented Programming

AOSD Aspect-Oriented Software Development

API Application Programming Interface

BX Bidirectional Transformations

CF Cross-cutting Framework

CRUD Create, Read, Update and Delete

DAG Directed Acyclic Graph

DAO Data Access Object

DTD Document Type Definition

GUI Graphical User Interface

IoC Inversion of Control

IPC Inter-Process Communication

JCR Java Core Reflection

JDO Java Data Objects

JPA Java Persistence API

JVM Java Virtual Machine

LOID Logical Object IDentifier

OID Object IDentifier

OODB Object-Oriented Database

OODBMS Object-Oriented Database Management System

OOP Object-Oriented Programming

ORM Object-Relational Mapping

PAS Persistent Aplication Systems

PEP Path Expression Pointcuts

SoC Separation of Concerns

XML eXtensible Markup Language

Chapter 1

Introduction

Applications are subject to a continuous evolution process due to many factors such as changing requirements, new functionalities or even the correction of design mistakes. In many of these cases, there are strong implications on the underlying data models. Furthermore, these data models also have specific application interfaces for each of the existing data types. This formalism imposed by the database schema has a significant impact on the whole evolution process, requiring the intervention of programmers and database administrators, in addition to application service disruption. Database and application complexity maintenance clearly grows with the size and complexity of the schema.

In object-oriented databases, the database evolution problem has two distinct parts at two distinct layers:

- (1) schema evolution - at the metadata layer; and
- (2) instance adaptation - at the data layer.

The schema evolution approach determines application compatibility, which can minimize the process's negative impact if a versioned schema approach is chosen. Despite the chosen approach, the entire database must preserve its structural, semantic and behavioural consistency. In this field, a proper instance adaptation approach must be applied in order to ensure that previous existing objects continue to be accessible under the new schema and that no anomalous behaviours occur during the applications' normal functioning [Zicari, 1991].

This PhD thesis addresses the database evolution problem, in the particular context of orthogonal object-oriented persistent programming systems, on both sides of the problem: schema evolution and instance adaptation. Other issues, such concurrency and error recovering, are also addressed as part of the persistence concern.

Orthogonal persistent systems, due to their intrinsic characteristics, have motivated the presented research work. During our research work some of these characteristics were considered as being distinctive to non-orthogonal systems, hence enabling a novel approach to deal with the aforementioned problem. The proposed approach aims to obtain a significant level of transparency in database evolution mechanisms. Yet, that approach can even benefit from the use of aspect-oriented programming (AOP) techniques and the enrichment of the applications' schemas with additional metadata into their source code. The use of these techniques and metadata in the source code enables the system to remain syntactically oblivious to all those problems. Additionally, they enable the semantic and structural aspects of the database evolution to be modularized and, in this way, able to keep the system semi-oblivious to the entire problem. Moreover, the adoption of a versioned schema still enhances the achievement of the proposed approach goals in terms of mechanisms transparency, because it enables backward and forward application compatibility. Our proposal is completed with a meta-model adjusted to the described environment.

A prototype was developed in order to test the feasibility of such approach. The developed prototype is a framework that supports multiple applications sharing a common schema in different state versions. A successful production system based on our prototype and meta-model could provide programmers with a powerful developing tool that sets him/her free from dealing with the referred problem at the database layer. In such scenario, programmers must concentrate only on the application's logic and definition of the data schema in the scope of the application. We argue that in those circumstances the programmer's work, in terms of quantity and quality, is highly benefited.

1.1 Motivations

The object-oriented data-centric applications must know the objects' interfaces in the underlying data model at the database level. Hence, these interfaces work as a contract between the schemas in the applications and the database. However, the applications' class structures, which are embedded in their source code, are usually not exactly the same as represented in the database's metadata layer. For several reasons programmers map applications' objects into the database as persistent objects of other types. The most important one is the general lack of databases in terms of full orthogonality. In the case of relational databases, this problem is all the more unavoidable due to the well-known *impedance mismatch* problem [Dearle *et al.*, 2010].

Contrasting with this scenario, in orthogonal persistent systems [Atkinson & Morrison, 1995] due to the orthogonality principles, the embedded schema in the application's source code and the one that is persistent in the database can be exactly the same. As will be discussed in Chapter 2, the orthogonal persistence paradigm was tested in the PJama [Atkinson, 2000] system, OPJ [Marquez *et al.*, 2000], Visual Zero [Perez-Schofield *et al.*, 2008], Thor [Liskov *et al.*, 1996] and Grasshopper [Dearle *et al.*, 1994]. More recent systems, such as db4o [Paterson *et al.*, 2006], objectdb [Ltd, 2011] and Object-Relational Mapping (ORM) tools, also provide some transparent persistence mechanisms and some orthogonality.

However, in terms of database evolution, we consider the results of these systems as limited. In the PJama system, the programmer must invoke an evolution tool, while in the db4o system few schema updates are managed transparently. Basically, these db4o's evolution mechanisms are based on matching fields by name, in most cases requiring double interventions: in the application and at the database level using helper programs.

The objectdb database is not an orthogonal persistent system, however, it provides some transparency as to how objects are stored and activated. This system also provides an automatic schema evolution mechanism, which is also based on matching fields by name. When class structure evolution includes the renaming of fields, classes or packages, these updates must be explicitly specified in the configuration to avoid data loss. In our opinion, just like in the db4o's

schema evolution mechanisms, these also have transparency limitations for programmers and applications. Additionally, none of these three systems support multiple schema versions, hence compromising backward and forward application compatibility.

In pure orthogonal systems, the close-coupled interaction between applications and the database management system enables a seamless sharing of the schema. This persistence paradigm also enables the application's schema to be propagated transparently into the database after each class first appears to the database management system. This transparency, in the initial propagation of the schema, is a simple technical problem since the reflection API of the programming language can gather all information regarding class structure and generate its metadata. The following schema evolution steps are also propagated in the same manner, producing new versions of the schema or classes (depending on the schema evolution approach), *i.e.* in this type of system, schema evolution can be incremental. However, when the structure or the semantics of a class is updated in the application's source code, database evolution becomes a very complex problem in terms of instance adaptation. Some type of updates can be transparently dealt with by the default conversion functions. On the other hand, structural changes and in particular semantic ones require a manual definition of the conversion mechanism. Notice that existing object instances follow previous versions of the schema. Thus, database evolution can be semi-transparent to the users because the schema updates are totally transparent, while instance adaptation may require user intervention.

That possible semi-transparency in database evolution mechanisms contrasts with non-orthogonal systems, which have a very different approach to update the database schema. In those systems, simultaneously to the update in the application's source code, a set of schema evolution primitives must be applied to the database to reconcile both schemas. Additionally, helper programs may be necessary in order to convert data from the older schema version to the new one.

As far as we aware of, the main research work carried out on the field of object-oriented database evolution that applies aspect-oriented programming techniques, which is discussed in Chapter 2, points to solutions that partially solve and, in

some cases, totally solve the mentioned problem. The research work carried out by Rashid [1998, 2001, 2002]; Rashid & Chitchyan [2003]; Rashid & Leidenfrost [2004, 2006]; Rashid & Pulvermueller [2000]; Rashid & Sawyer [1999, 2001, 2005] has been an important contribution to the improvement of the flexibility and customization features based on known evolution techniques, some of which are also discussed in this thesis. The authors of this earlier work have proposed an approach that enables the combination of all isolated techniques into one unique, powerful and integrated solution applying aspect-oriented programming techniques. Nonetheless, neither Rashid *et al*'s approach, nor any previous one is focused on the orthogonal persistence paradigm, as well as on the transparency of the entire evolution process from the programmer's point of view.

Those particular characteristics observed in the orthogonal persistent systems and the earlier results obtained in related research works, applying aspect-oriented programming techniques, have motivated our work in order to explore this combination: orthogonal persistence and aspect-oriented programming techniques. We argue that such combination provides very capable means to address the database evolution problem, with multiple schema versions, through semi-transparent and oblivious mechanisms. Additionally, other problems regarding the persistence mechanisms, such as transactions and error recovering, may also benefit from this combination. A successfully developed system could overcome the earlier systems, referred above, in terms of overall persistence mechanism transparency and reusability.

1.2 Hypothesis formulated in this thesis

As mentioned previously, the line that guides this research is the transparency of the database evolution mechanisms and how such transparency could improve the work of IT professionals. Therefore, a major effort has been put on the conception of an adjusted meta-model and the development of a prototype to test its feasibility. In particular, we have tried to pursue the following research questions, in order to formulate our hypothesis:

1. *Does the orthogonal persistence paradigm offer special conditions to implement transparent and oblivious mechanisms of persistence, which include database evolution, concurrency and error recovering?*

2. *Which is the most adjusted meta-model to support such mechanisms?*

3. *How can aspect-oriented programming techniques be applied in order to develop systems with such transparency and obliviousness?*

4. *How could that transparency and obliviousness improve the work of programmers and database administrators?*

These questions helped us to formulate our hypothesis.

Hypothesis 1 *Earlier research has proven that aspect-orientation of the persistence concern has customization and flexibility benefits.*
Our hypothesis is that obliviousness and transparency, in those mechanisms, are also possible to achieve in the context of orthogonal persistent systems by applying aspect-oriented programming techniques. Furthermore, its practical implementation is possible as a reusable framework.

Hypothesis 2 *If Hypothesis 1 is confirmed, IT professionals' work can benefit in terms of quality and productivity.*

1.3 Contributions of this thesis

This PhD thesis focuses on the study of the feasibility of transparent and oblivious database evolution and concurrency control in persistence mechanisms supported by aspect-oriented programming. Additionally, this work also intends to prove the usefulness of such systems for programmer's productivity and work quality. To achieve that goal, a novel object-oriented database meta-model was conceived. Additionally, an aspect-oriented crosscutting framework prototype was developed to test the proposed approach. In order to evaluate our approach, we performed experimental tests and developed a proof-of-concept application.

The main scientific contributions are centered on answering the research questions formulated above. Therefore we highlight the following contributions:

1. It formalizes a meta-model for object-oriented and aspect-oriented frameworks of persistence (Section 3.2);

2. It presents the design and implementation of an aspect-oriented framework for orthogonal persistent programming systems (Section 3.7).

3. It presents our pointcut/advice constructs that enable programmers to support the database evolution of their applications (Section 3.2.6.3).

4. It evaluates the benefits of the proposed framework upon a scenario of database evolution of a proof-of-concept application (Chapter 5).

5. It evaluates the framework in terms of its internal meta-model and architecture in a case study involving other competitor systems (Section 4.1).

6. It evaluates the combination of the orthogonal persistence paradigm and AOP, hence contributing to the debate regarding trade-offs between Modularity and Obliviousness (Section 2.1.3.1 and Section 6.3.3).

1.4 Structure of this thesis

This PhD thesis is organised as follows: Chapter 2 presents the current state-of-the-art in the fields of aspect-oriented programming and object-oriented database, focusing on persistent programming systems, as well as related research works. The proposed approach, based on a new object-oriented database meta-model and its prototype, is discussed in Chapter 3. Chapter 4 presents valuable comparative results with other equivalent systems, as well as other tests in terms of both the meta-model and the framework's robustness. Chapter 5 presents a geographical application that was developed without any concern regarding persistence, database evolution and concurrency control. We coupled it with our framework with minor changes applied to its source code. Finally, Chapter 6 contains all conclusions drawn during the development of this research work.

Chapter 2

Literature Review

2.1 Background

In this section we address some important and useful concepts to contextualize our meta-model proposal and prototype, which are presented in Chapter 3. It is assumed that readers are familiarized with all concepts related with object-oriented programming and data persistence. However, orthogonal persistence and aspect-oriented programming (AOP) are considered somewhat specific concepts, and therefore deserve a short introduction.

2.1.1 Orthogonality in object persistence

Persistence has been studied by several researchers in the last decades. The main mission of this research work was to solve the well-known *"impedance mismatch"* problem between object-oriented applications and databases. That mission has been so difficult to achieve that it already been referred to as the Vietnam of the Computer Science [Neward, 2006][Dearle *et al.*, 2010].

Orthogonal persistence is an approach to solve that ancient problem. Atkinson *et al.* [1983] conceptualised this kind of data persistence, first introducing the concept and later adapting it to object-oriented contexts [Atkinson & Morrison, 1995][Atkinson, 2000]. Following three principles, programmers can totally abstract from their data in objects, allowing code reuse, and the focus on application logic and data consistency. These three principles are desirable characteristics of

systems that manage persistent data. In short, these principles are:

- Persistence Independence - The same code should be applicable to both persistent and non-persistent objects.
- Type Orthogonality - All objects can be persistent or non-persistent irrespective of their type, size or any other property.
- Persistence Identification or Reachability - A given object is persistent because it is reachable, directly or through other objects, from a persistent root.

2.1.1.1 Orthogonality consequences

Despite the carelessness offered to programmers, this persistence paradigm does also imply several technical challenges to the designers of these orthogonal systems. In fact, this paradigm of data persistence makes the problem very complex and, moreover, it also introduces performance issues, specially regarding the system's main memory management and concurrency.

Considerable research work has been done regarding the cost of orthogonality. Atkinson & Jordan [1999] discusses issues raised by three years of research work on developing PJama, a Java-based orthogonal persistent system. During this research work, the authors have identified several issues that posed many design problems: achieving orthogonality with classes that have a special relationship with the Java Virtual Machine (JVM); treatment of static variables and keyword `transient`; concurrency; schema evolution; and performance issues.

Nettles [1993]; Nettles & O'Toole [1996], Zigman & Blackburn [1998], Marquez *et al.* [2000], Blackburn & Zigman [1999] and Atkinson & Jordan [1999] have researched concurrency and transactional issues in the context of orthogonal persistent systems.

Atkinson & Jordan [1999] proposed that each Java Virtual Machine execution should act as a single and flat transaction. Inside the context of a Java Virtual Machine, each transaction commit moves the data checkpoints forward, but does not release control of the resources it is using. Furthermore, the concurrency model of programming languages (like Java) does not serve the requirements

of the orthogonal persistence properly in terms of transaction isolation, which suggests the need to add transaction support to programming languages. This also enables the *undo* ability.

Blackburn & Zigman [1999] argue that the aversion to transactional approaches within the orthogonal persistence community stems from the challenge on orthogonality in the transaction model, since such model has implicit the notion of two distinct worlds: application environment and persistent data. Those two worlds do not exist in orthogonal persistent systems. Thus, most of these systems follow "open" approaches [Blackburn & Zigman, 1999] which are based on checkpointing and explicitly synchronization.

The schema evolution and instance adaptation are other challenges when designing orthogonal systems. In section 2.3 the database evolution issue is discussed, with particular focus on orthogonal persistence.

The aforementioned issues motivated some authors, such Cooper & Wise [1996], to criticize orthogonal persistence and advocate another alternative, less restrictive model named Type-Orthogonal Persistence, opposing Atkinson & Morrison [1995]'s model. Analyzing Cooper and Wise's arguments we conclude that these essentially reflect performance issues and not restrictions made to the programmer or the language.

2.1.1.2 Orthogonality benefits

Despite the many challenges posed by this paradigm, its advantages are many, not only in terms of code reuse, as already mentioned, but also in other perspectives such as data type safety checking, coding error reduction, better code organization promotion and improvement to the applications' refactoring processes. Atkinson & Morrison [1995] summarized the benefits of orthogonal persistence as:

- improving programming productivity from simpler semantics;
- avoiding *ad hoc* arrangements for data translation and long-term data storage;
- providing protection mechanisms over the whole environment;
- supporting incremental evolution; and

- automatically preserving referential integrity over the entire computational environment for the whole life-time of a Persistent Application Systems (PAS).

Other concepts, like Safe Queries [Cook & Rai, 2005] and Native Queries [Cook & Rosenberger, 2005], also may provide a better understanding of orthogonality on persistence and its potential in terms of code quality and productivity.

Despite the current generalization as to the use of persistence frameworks, which provide some orthogonality, the orthogonal persistent programming paradigm is still a strange concept for many programmers, novice or senior. Indeed, more experienced programmers, that have their thinking formatted to follow the model "input, process, output" and mappings between the "internal" and "external" data structures, have some difficulty in understanding orthogonality [Atkinson & Jordan, 1999].

2.1.1.3 Conventional persistence approaches

Atkinson [2000] did a survey on persistence mechanism options for Java. Regardless of the article's age and its restrictive technological embracing, it can be generalized to today's reality and programming technologies, in most common programming practices. Based on that earlier study, Balzer [2005] categorized the conventional persistence approaches for object-oriented programming. That categorization is presented next with some adjustments, which we consider relevant.

- Object Serialization: This mechanism is based on encoding and decoding object graphs, respectively, into and from binary (or other such as XML) representations. The mechanism serializes whole object graph structure transitively reachable by a root object. Most common object serialization implementations, like Java, require objects to implement a `Serializable` interface. This approach limits a large number of Java core classes to be serialized. Furthermore, it does not preserve previously common substructures; it only provides navigational access to the serialized objects starting from the root object; and it does not scale very well. Thus, object

serialization breaks orthogonal persistence principles, being only suited for a limited number of cases, such as remote method invocation, where sharing of sub-structures is not desired. Object serialization is a valid complement to a persistence mechanism, but not a replacement thereof.

- Relational Database Interface: This two-tiered architecture, based on an object-oriented programming language and a relational database management system, suffers from the impedance mismatch between the object model of the programming language and the relational model of the database. Consequently, the programmer must manually maintain all complex mapping code between those two worlds. Programmer performs that mapping through a well-defined Application Programming Interface (API) which is provided by the programming language. This API offers methods to connect to databases and methods to store, update, and retrieve data contained in application's objects.

- Object Database Interface: Object database interfaces do not suffer from the impedance mismatch as relational interfaces do. Apart from the easy mapping, however, object database interfaces provide persistence-related operations, such as for deletion or transaction control, that rather defeat persistence independence.

- Persistence Frameworks: Persistence frameworks provide a huge selection of persistence facilities, such as access to a wide variety of heterogeneous data sources in case of Java Data Objects (JDO) or distributed persistence in case of Enterprise Java Beans (EJB) and Object-Relational Mapping tools that allow the separation of the object-relational mapping concern from code to specialized XML files. Although the majority of those frameworks provide an object database interface, they do not comply with persistence independence.

2.1.1.4 Orthogonal Persistent Systems

Orthogonal persistence was applied to the presented prototype [Pereira & Perez-Schofield, 2010, 2011, 2012, 2014a,b] and others implementations, which in some

cases were not totally compliant with the concept. PS-Algol [Atkinson *et al.*, 1983], PJama [Atkinson & Jordan, 1999], OPJ [Marquez *et al.*, 2000], Visual Zero [Perez-Schofield *et al.*, 2008], and Thor [Liskov *et al.*, 1996], are examples of those systems.

In Grasshopper [Dearle *et al.*, 1994], the file system and memory management components of the operating system were unified in order to provide a seamless orthogonal environment. Orthogonal persistence is provided by the operating system to the programming language. The authors of this work argue that the approach overcomes the problems of implementing orthogonal persistence at the programming language level. Furthermore, this approach theoretically enables any programming language to provide orthogonal persistence. Following that direction, the authors of this work have ported the framework to several programming languages [Dearle *et al.*, 1996].

Some object-oriented databases, as well some object-relational mapping tools, also implement some level of orthogonality.

2.1.2 Discussion of the Java genericity implementation

In the early versions of the Java platform, the lack of parametric polymorphism support led to an intensive research[Agesen *et al.*, 1997][Bank *et al.*, 1997][Bracha *et al.*, 1998][Solorzano & Alagić, 1998][Odersky *et al.*, 2000][Alagic & Nguyen, 2001] in order to find solutions. However, the final adopted solution [Bracha *et al.*, 1998][Gosling *et al.*, 2005] was not consensual. Several researchers have studied the same problem, as well as the pros and cons of the adopted solution which is based on type erasure.

The Java generic idiom [Bracha *et al.*, 1998], supported by the standard libraries based on casts, type tests and the `Object` class as a generic type, enables programmers to deal with the data type generiticity, including covariance and contravariance [Castagna, 1995] type variations. However, the expressiveness and the type safety of the programming language is very compromised.

The native support for genericity in the Java platform appeared in its 5.0 version, solving the aforementioned problems. For example, before generics, the object references when put in an `ArrayList` collection were considered as `Object`,

the top level class in the hierarchy of the Java classes. The type of these collected objects are not tested, therefore any type could be added to that collection. Furthermore, a cast is mandatory in the opposite phase when objects are retrieved from that data structure. This is because the compiler does not allow direct covariance when assigning an object pertaining to a super class to a reference of a subclass type.

This Java native support for genericity effectively improves the expressiveness and static type checking of the programming language. However, the chosen implementation approach raises two issues which are relevant for the present work: (1) The type erasure compromises the orthogonality of the object persistence. (2) On the other hand, the method genericity provides means to improve the persistence mechanisms' transparency of the developed framework. These two issues are discussed in the following two sections.

2.1.2.1 Type erasure

Since version 5.0 of the Java platform classes can be generic, it requires a set of typing parameters when calling their constructors. Using the `ArrayList` example, given previously, when this generic object collection is instantiated, it must be parameterized with the type of the collected objects (*e.g.* `ArrayList<Integer> al=new ArrayList<Integer>();`). In this case, the compiler does consider that all objects inside that collection pertain to a single type (*e.g.* `Integer`), the class specified as type parameter. Thus, it will only accept that class of objects as input and, when objects are retrieved, it will not require any cast because the compiler will insert it automatically.

In truth, when a generic class is instantiated, the compiler statically replaces (erases) that parametric data type with a raw data type, typically an `Object` class. Besides that, the compiler introduces the necessary type checks at compile-time and uses bridge methods to ensure type security of the retrieved objects. This approach, based on a type erasure idiom [Gosling *et al.*, 2005], frees the programmer from all those concerns and also ensures static type checking.

The authors [Bracha *et al.*, 1998] of this approach, based on type erasure using raw types, justify their option on the fact that it serves two important purposes.

These authors' arguments are: (1) the support of interfacing with legacy code, retrofitting all existing and used libraries in many production applications; and (2) they support writing code in those few situations (such as the definition of an equality method) where it is necessary to downcast from an unparameterized type (like `Object`) to a parameterized type (like `ArrayList<A>`), and one cannot determine the value of the type parameter.

Regarding the binary format, the adopted solution in the Java platform 5.0 allows a normal coexistence of non-generic and generic code. That is achieved because of the compatibility of the binary class files that represents each class. That compatibility is a complex issue to solve, which results in the multiple structure versions that a polymorphic class may have, depending on the type parameters. Those multiple representations of the same class are not suitable to be represented within a pure homogeneous translation [Odersky & Wadler, 1997; Odersky *et al.*, 2000], which is applied in previous versions of the Java platform. Another considered alternative was the heterogeneous translation [Odersky & Wadler, 1997; Odersky *et al.*, 2000]. This one maps a parameterised class into a separate class for each instantiation. As an example, in the Pizza system, the heterogeneous translation of the class `Hashtable<Key,Value>` replaces the instance `Hashtable<String,Class>` with the class `Hashtable$_String_$_Class_$` [Odersky *et al.*, 2000]. However, this other type of translation, besides the obvious disadvantages such as the extra needs for disk and memory space, is incompatible with the class file structure used in the previous versions of the Java platform.

Despite all advantages of the adopted solution, many authors are very critical regarding this implementation option, arguing that it compromises type safety, type orthogonality and other important characteristics of the Java language [Cabana *et al.*, 2004][Radenski *et al.*, 2008][Alagić & Royer, 2008]. These criticisms are specially harsh about implementation decisions (based on the type erasure idiom [Gosling *et al.*, 2005]), whereas the chosen syntax has been well received [Radenski *et al.*, 2008].

Cabana *et al.* [2004] have studied the limitations of the Java type erasure and found several pitfalls: violations of Java Type System; violations on subtyping rules; problems in method overloading and in the Java Core Reflection (JCR). In order to address these problems, the authors proposed a technique mainly based

on the representation of the parametric class or interface in the standard class file format with some subtle changes on: (1) The Java class files - introducing optional attributes without affecting the compatibility with older versions, since those are ignored on a legacy JVM; (2) extending the JCR to be able to obtain information about the type parameters; and (3) modifications on the class loading process.

The relevance of the aforementioned work and others [Alagic & Nguyen, 2001][Radenski *et al.*, 2008][Alagić & Royer, 2008] is in orthogonal persistence in parametric polymorphic classes. The adopted approach compromises the implementation of Java orthogonal persistent systems [Solorzano & Alagić, 1998][Alagic & Nguyen, 2001][Cabana *et al.*, 2004][Alagić & Royer, 2008]. The erasure process, which consists of eliminating the type parameters during compile-time, affects reflection on parametric polymorphic classes because that type of information is not available at run-time. The reflection features are particularly important to persistence mechanisms and database systems. Moreover, the incorrect run-time type information also affects the objects' reachability [Alagić & Royer, 2008]. For example, if an object contains references to a `Collection<Person>`, there is the risk that all these `Person` objects will be stored as pertaining to class `Object`. Hence, if a query is applied to this collection of objects, looking for `Person` objects, it obtains an empty and wrong set of results, in addition to requiring a cast.

Those earlier works have shown the relevance of parametric classes in orthogonal persistence systems, when supported by the Java platform or any others that are based on type erasure. The developed framework discussed in Chapter 3, while being supported by a Java platform, addresses those identified problems within an aspect-oriented approach. Thus, the parametric classes were researched in order to address those platform dependent problems in our prototype and meta-model. In Section 3.3.6.5, we will discuss our approach.

2.1.2.2 Generic methods

Parametric polymorphism is also applied in methods in order to provide them with genericity. A given method is identified as generic, if it is declared to have one

or more type parameters (using the same syntax, as the one in class constructors [Gosling *et al.*, 2005]). The generic methods have the inference ability of their return type value in some scenarios. These characteristics were explored in our framework and they will be discussed in detail in Section 3.3.6 and Section 3.3.2.1.

2.1.3 Aspect-Oriented Programming

Object-oriented programming (OOP) paradigm is the most widely used in software development nowadays. This paradigm enables the system's partitioning as a set of modules (the classes), which work together in order to provide the system with its features, while enhancing code reusing and programming productivity, as well as reducing maintenance costs. Despite the well-known advantages of the object-oriented programming paradigm, it does not provide the efficient means to compartmentalise software transversal requirements, leading to code scattering and tangling. Programmers modularize the code using the best organization, in order to minimize the scattering and tangling of the requirements, and maximizing code reuse. Procedures, inheritance and classes, as well as programming design patterns, provide a good level of code reusing to non-transversal requirements. However, these programming techniques are unable to handle cross-cutting requirements efficiently.

Usually, software applications have one or more transversal concerns: Presentation, distribution and persistence are good examples of non-functional transversal requirements. Dijkstra [1976] identified these common transversal requirements as cross-cutting concerns which should be modularized in unique conceptual code units. That pioneer work lead to the conception of several approaches: Metaobject protocols [Kiczales & Rivieres, 1991], Subject-Oriented Programming [Harrison & Ossher, 1993], Composition Filters [Akit *et al.*, 1992], Adaptive Programming [Mezini & Lieberherr, 1998] and Aspect-Oriented Programming (AOP) [Kiczales *et al.*, 1997]. This last one being the most promising approach.

Object-oriented programming enables users to model real-world objects, enhancing abstraction, modularity and code reuse. Aspect-oriented programming retains all these goals and goes a step further in order to avoid the problem of code scattering and tangling. This programming technique aims to encapsu-

late cross-cutting concerns into *aspects* to retain modularity. Thus, we can see aspect-oriented programming as an extension of the object-oriented paradigm. This programming paradigm isolates secondary or supporting functions from the main program's business logic. It aims to increase modularity by enabling the separation of cross-cutting concerns. Lopes [1997] sees aspect-oriented programming as a language framework for the separation of concerns in a system. For Lorenz [1998], aspect-oriented programming is more than a programming paradigm; it is a design framework for the separation of concerns. In the absence of linguistic support of the base programming language though, aspect-oriented patterns can provide the novice with simple and elegant aspect-oriented solutions to specific problems. It all makes sense because aspect orientation is not bound to object-oriented programming. It can be applied in any other programming paradigms such as the procedural ones.

With aspect-oriented programming some new concepts emerge: *join point*, *pointcut*, *advice* and *aspect*:

- *Join point* - It is the specific point in the application such as method execution, exception handling, changing/reading object's variable values and so on;
- *Pointcut* - Groups a set of join points exposing some of the values (*e.g.* the target object) of the execution context in the join points. These program's execution points are defined through *pointcut expressions*;
- *Advice* - It is the piece of code that is executed at each join point, identified using a *pointcut*;
- *Aspect* - An *aspect* is the modularization unit of aspect-oriented programming. These modularization units encapsulate cross-cutting features as *pointcuts* and *advices*. The aspects are integrated into the system using a special tool called weaver.

The weaver is the central piece in this technique. This element is responsible for merging the program base and *aspects* at the joint points defined through the *pointcut* expressions. This mechanism could be statically or dynamically woven.

- Static weaving [Popovici *et al.*, 2002] means that the application's aspect-oriented behaviour is static. After the application and the *aspect* are put together, they remain fixed. As a consequence, adapting or replacing aspects is a very time-consuming process. A static weaver acts at compile-time, before or after the compilation process. In the first case, it merges the *aspect* and application source code to then deliver the result to the compiler. On the other hand, post-compile weaving is done after the compilation, when application and aspects are in binary format.

- Dynamic weaving [Popovici *et al.*, 2002] is performed at load-time or run-time, when an application is already running. The weaving, when done when classes are being loaded (load-time weaving), allows dynamically different aspects to be attached to them. Load-time approaches are supported by object Instrumentation, enabling application binary code transformation at load-time. In a Java environment, they replace the default class loader of the Java Virtual Machine by another, which updates classes when they are loaded. In a run-time approach, aspects can be replaced when classes are already loaded. Those application updates can be achieved using meta-object protocols, virtual machine weaving interfaces, reflection [Popovici *et al.*, 2002] or even using coding techniques that can dynamically support the enabling and disabling advice in aspects [Kiczales *et al.*, 2001a]. Both dynamic approaches open a wide spectrum of solutions enabling, for instance, the replacement or the choice of the application's behaviour after it starts. As a disadvantage, it introduces overheads and can be used to inject malicious code in the applications.

Kersten & Murphy [1999] identified four aspect/class types of association: (1) Class-directional - Aspect knows about the class but not vice-versa; (2) Aspect-directional - Class knows about the aspect but not vice-versa; (3) Closed - Neither the aspect nor the class know about the other; and (4) Open - Arbitrary.

2.1.3.1 What is distinctive in AOP

Filman & Friedman [2000] found two major characteristics in aspect-oriented programming: *Quantification* and *Obliviousness*.

Quantification [Filman & Friedman, 2000] consists in the definition of the join points where application and special methods, called *advices*, resident in the *aspect* object, come together. That definition of the localization is made through pointcut expressions. Depending on aspect-oriented programming language those expressions are capable of identifying a broad type of joint points such as method calls or definitions, properties reads or writes, object constructors and other strategic application points.

The application's concern, in the form of code written in the *aspect*, can be introduced or even replace an existing implementation of a specific behaviour of the application without being previously prepared for that procedure. Programmers and applications do not need to have knowledge about that. This second characteristic present in aspect-oriented programming is called *Obliviousness* [Filman & Friedman, 2000].

In their paper, Filman & Friedman [2000] posit these two properties are necessary and distinctive for aspect-oriented programming. Later, in another publication [Filman, 2001], Filman clarifies his position and responds to some comments that were made about the previous work.

For Steimann [2005] *Quantification* and *Obliviousness* are also fundamental properties of aspect-oriented software development (AOSD). Based on his observations of the generalized accepted definitions for the domain model and aspect, and what he considers to be the fundamental properties of AOSD, the author argues that aspects are always second-order statements and domain models are first-order; therefore, it could be argued that aspects (in the aspect-oriented sense) are technical and extrinsic to the modelled problem domain and its models. That is to say, aspects are technical and they are few [Steimann, 2004]. However, Steimann's perspective is not consensual. In his paper [Steimann, 2005], Steimann launched a debate [Gabriel *et al.*, 2006b; Leavens & Clifton, 2007; Rashid & Moreira, 2006; Steimann, 2006] on domain models' aspects and what is distinctive in AOSD and AOP. In our research, we do not intend to participate in this debate by giving a crosscutting position over the entire spectrum of domain aspects and technical aspects. We only expect to contribute with new ideas regarding the category of aspects, which we address in our work, within a very particular context of orthogonal persistence.

Rashid and Moreira have another perspective regarding the fundamental characteristics of an AOSD approach [Rashid & Moreira, 2006]. In their perspective, it is the systematic support for *abstraction*, *modularity* and *composability* of crosscutting concerns that distinguishes AOSD techniques from other SoC mechanisms [Rashid & Moreira, 2006]. *Abstraction* allows the hiding of the details of how a specific concept or feature may be implemented within a system. By abstracting a system's concept or feature, we are in condition to modularize that part of the system within a module (*Modularity*). *Composability* complements *modularity*, *i.e. the various modules need to relate to each other in a systematic and coherent fashion so that one may reason about the global or emergent properties of the system* [Rashid & Moreira, 2006].

In Rashid and Moreira's perspective, the modularization of a system's aspect requires a global reasoning over the entire system and its requisites, which must begin in the early phase of the design process. Early aspects [1] are crosscutting concerns that are identified in the early phases of the software development life cycle [Rashid *et al.*, 2004; Tekinerdogan *et al.*, 2004]. Identifying these aspects at this early phase of the design is fundamental, as aspectual requirements have influence on other requirements in the system.

In early work on aspect-oriented programming [Kiczales *et al.*, 1997, 2001b], modularity was conjectured to be a direct consequence of aspect-oriented SoC. However, in more recent research [Aldrich, 2004; Bodden *et al.*, 2014; Gabriel *et al.*, 2006a; Hoffman & Eugster, 2013; Kiczales & Mezini, 2005; Leavens & Clifton, 2007; Przybylek, 2013; Steimann, 2004] several authors have debated on the implications of aspect-oriented programming for modularity. In particular, obliviousness has been considered as a source of difficulties to reason about the behaviour of aspect-oriented programs.

Modular reasoning means that we may abstract from the specific details of a module and also reason about these details in isolation. That is to say, one can *make decisions about a module while looking only at its implementation, its interfaces and the interfaces of modules referenced in its implementation or interface* [Kiczales & Mezini, 2005].

[1] For more information about Early Aspects research topic see http://www.comp.lancs.ac.uk/computing/aop/EarlyAspects.php

The problem with aspect-oriented programming is that while base modules are syntactically oblivious to aspects, they are not semantically oblivious to aspects [Przybylek, 2013]. Therefore, when studying some module, a programmer has to consider all aspects that can possibly interfere and change the module's logic. To mitigate this problem, several extensions to aspect-oriented programming have been proposed. Most of them restrict obliviousness by introducing various forms of interface or contract.

Gudmundson & Kiczales [2001] proposed to place named pointcuts in class definition and then to export them in the class's interface (so called pointcut interface). The pointcuts are then used by *aspects* when defining *advices* which apply to the class. By having the pointcut interface, the programmer is aware that the class may be influenced by *aspects*. Exported pointcuts form a contract between a class and its client *aspects*, allowing the class to be evolved independently of its clients as long as the contract is preserved. The work of Gudmundson and Kiczales was continued by Aldrich [2004] and Hoffman & Eugster [2013], while the idea of explicit interfaces was further developed by Kiczales & Mezini [2005].

Kiczales & Mezini [2005] introduced the concept of *aspect-aware interfaces* to annotate method declarations with the aspects that may apply to them. In their approach, a class's aspect-aware interface is automatically computed using reverse engineering through a global analysis of the aspects and the classes once a system has been composed.

In turn, Clifton & Leavens [2002, 2003] and Przybylek [2013] proposed to distinguish between harmful and harmless *aspects*. Their proposals allow programmers to ignore harmless *aspects* when reasoning about the base code. For Clifton & Leavens [2003], the *aspects*' obliviousness makes the programmer's job very difficult in terms of debugging and code comprehensibility. They observed that there are conflicting demands of obliviousness and comprehensibility within the context of object-oriented programming. They also define a notion of modular reasoning that corresponds to Parnas's comprehensibility benefit [Parnas, 1972]. Their work was continued by Przybylek [2013] who demonstrated that even though aspectization of crosscutting concerns removes some lexical dependencies between classes within base code, new semantic dependencies are introduced between *aspects* and the base code. Since these dependencies are not explicit, programmers

need more effort to get a mental model of the source code [Przybylek, 2011].

Pointcut fragility [Störzer & Koppen, 2004] is another challenge for the aspect-oriented research community. Current aspect-oriented programming languages rely on referencing structural properties of the program. These structural properties are used by pointcuts to define intended conceptual properties about the program. Thus, maintenance changes that conflict with the assumptions made by pointcuts may introduce defects [Przybylek, 2011]. These defects occur when a pointcut unintentionally captures or misses a given join point as a consequence of seemingly safe modifications to the base code [Störzer & Koppen, 2004].

Moreover, detection and resolution of undesirable aspect interactions is an important and ongoing field of research. When *advice*'s declarations made in different *aspects* affect the same join point, a wrong execution order can break the program [Przybylek, 2011]. To summarize, new techniques are still needed to overcome the aforementioned challenges. Additionally, the discussion about the trade-offs between obliviousness and modularity is yet to be accomplished.

2.1.3.2 Java based aspect-oriented tools

The AspectJ [Kiczales *et al.*, 2001a] is one of the existing aspect-oriented languages which accompanies the aspect-oriented programming paradigm. It is based on the object-oriented Java programming language, extending its capabilities with aspect-oriented features. This language enables the separation of the applications' cross-cutting concerns, which are normally scattered and tangled over several classes, in order to put them into specialized objects called *aspects*, while the remaining concerns that are specific for each class remain implemented in these classes. The language provides a specific syntax, which allows a full description of both joint points (through pointcuts expressions) and *advices* (the behaviour) of an *aspect*. The AspectJ framework provides a compiler `ajc` (or `iajc`, an incremental version of `ajc`) that supports three weaving types: compile-time, post-compile and load-time weaving.

JBoss AOP [Inc., 2009] is a 100% pure Java framework. All aspect-oriented programming constructs are defined as pure Java classes and bound to the application code through an XML file or Java annotations. Contrasting with AspectJ,

which has its own language to define aspects, in the JBoss AOP framework an aspect class is a plain Java class that can define zero or more *advices*, pointcuts, and/or mixins.

These two aspect-oriented environments follow distinct approaches with the same purpose: the former as a language and a special compiler; the latter as a framework, which enables the injection of the cross-cutting concern into the applications, through pointcuts defined in a central configuration file or annotations.

2.1.3.3 Aspect-Oriented Frameworks

Fayad & Schmidt [1997] define software framework as a reusable, "semi-complete" application that can be specialized using system definitions to produce custom applications. In contrast to object-oriented reuse techniques based on class libraries, frameworks are targeted at particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or real-time avionics). Furthermore, contrasting with these libraries, in a framework-based application it is the framework that controls the flux, inverting the control.

Martin [1994, 1996] presents a simple problem of a Copy program, which can be implemented following that paradigm of inversion of control (or IoC). This program copies keyboard characters to a disk file, supposing that it is working in an environment without an operating system where all input/output operations must be implemented by the programmer. Figure 2.1 presents a classical implementation approach of that program based on three modules. One is the main Copy program while the other two are device interfaces. Although it is a very simple problem, it helps us to better understand what the inversion of control concept means.

It would certainly be useful to reuse the Copy module since it is the main part of the system, which contains all system intelligence and business logic. The authors proposed a solution to this example based on an object-oriented framework approach. This framework provides a set of abstract classes which define the interface between the main module and the other two. These two

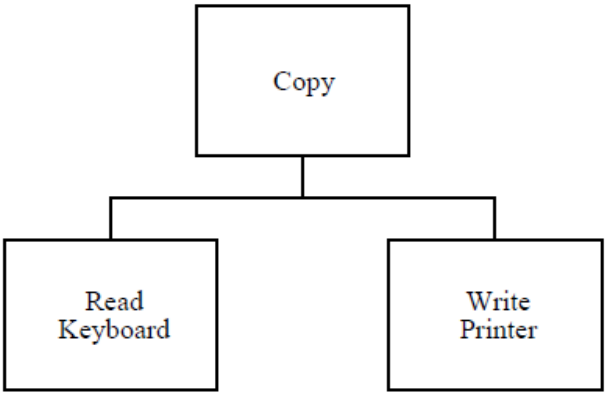


Figure 2.1: Inversion of Control - Copy program example [Martin, 1996]

modules may have any kind of input/output devices, while the main module remains unchanged.

Johnson [1997] defines an object-oriented framework as a collection of organized abstract classes and interfaces that can be used to implement a specific family of problems, such as the one previously presented. However, this type of framework suffers from the same limitations as the ones discussed previously regarding object-oriented programming languages and the fact that they are unable to efficiently deal with cross-cutting concerns. Aspect-oriented programming, because it is an extension of that paradigm, when applied to this kind of framework, could provide additional means to deal with concerns that are transversal to many classes.

Hanenberg [2000] defines the aspect-oriented framework as a collection of abstract and concrete aspects. Like object-oriented frameworks, they also contain hooks and templates but, in addition to methods, abstract pointcuts may be overwritten. Certain aspects may already be implemented but they also can be abstract in order to be specialized through concrete aspects. Thus, the combination of object-oriented and aspect-oriented approaches provides very powerful means to develop configurable, versatile and pluggable frameworks.

Vieira de Camargo [2006] identified and classified aspect-oriented frameworks. The author considers that there are two types of aspect-oriented frameworks:

- *Aspect-oriented application framework (AOAF)* - This kind of framework is a generic architecture of an application domain, which, when instantiated,

produces an application of that specific domain. These kinds of frameworks based on classes and aspects are very similar to object-oriented frameworks in the way they apply abstract and concrete classes to obtain concrete implementations.

- *Cross-cutting framework (CF)* - The author coined the term to classify those frameworks that implement one or more cross-cutting concerns. According to the author, this type of framework is the most common aspect-oriented framework found in the literature. Usually these frameworks implement one cross-cutting concern such as persistence, cryptography, competition and security.

The cross-cutting framework [Vieira de Camargo, 2006] concept is explored in this PhD Thesis in order to develop an orthogonal persistent programming environment, which provides applications with transparent mechanisms of persistence, database evolution, as well as transactional control mechanisms.

2.2 Persistence as an aspect

Persistence is frequently referred to in the literature as a good example of a concern that can be modularized as an *aspect* in terms of aspect-oriented programming [Rashid, 2001][Rashid & Chitchyan, 2003][Ortin *et al.*, 2004][Ramos *et al.*, 2004][Hohenstein, 2005][Soares *et al.*, 2006][Soares & Borba, 2007][Al-Mansari *et al.*, 2007]. Theoretically, such approach should enable: (1) *aspects*' code reutilization, regardless of the application and (2) development of applications that are oblivious regarding their data persistence. These previous research works focus on modularization of persistence, in order to provide applications with persistence mechanisms, applying aspect-oriented programming techniques or through reflective systems.

Soares *et al.* [2006] present their experience while refactoring a web application, a Health Watcher system, modularizing all code related with distribution and persistence concerns using AspectJ.

Rashid & Chitchyan [Rashid, 2001; Rashid & Chitchyan, 2003] addressed persistence modularization as an aspect. Rashid & Chitchyan [2003] demonstrated

that, in the context of a database application, persistence can be aspectized effectively. However, these authors argue that only partial obliviousness is desirable. They argue that persistence has to be accounted for as an architectural decision during the design of data-consumer components [Rashid & Chitchyan, 2003]. For example, graphical user interface (GUI) components need to be aware of large volumes of data so that they may be presented to users in manageable chunks. Yet these two previous studies did not focus on orthogonal persistence. We argue that orthogonal persistence provides special conditions that may make obliviousness desirable with regard to persistence. In the next section we discuss our points of view.

2.2.1 Is AOP suitable for the orthogonal persistence?

Aspect-oriented programming techniques enable quantified programmatic assertions over the code, which are oblivious regarding these assertions. We argue that this ability makes these techniques well suited to provide systems with orthogonal persistence as an aspect.

Regarding the Type Orthgonality principle, all objects of any class could be persistent or non-persistent. In most persistence frameworks (*e.g.* Hibernate, EJB or JDO), object persistence is achieved by inheritance or by implementing special class or interfaces. This approach compromises Atkinson's principle of type orthogonality. When using aspect-oriented programming techniques, since we can quantify any jointpoint in any type of object, the weaving mechanism can weave the required persistence code into objects. This prevents objects from extending any super class or implementing any interface. Finding which persistence behaviour objects are required is then the main challenge.

Regarding the Persistence Identification principle, the object's persistence state depends only on whether or not it is directly or indirectly related to a persistent root object. Additionally, the *aspect's* code can find if an object is reachable by examining the data contained in objects and its relationships. Hence, there are conditions to find which persistence behaviour should be applied to objects.

Considering the two previous arguments that any object can be persistent, despite its type, and that persistent object state identification is possible through

reachability, no special care regarding the remaining code to handle those objects is required. In other words, the *aspect's* code can, in itself, distinguish a persistent from a non-persistent object by examining its relationships from a persistent root, hence making the remaining code of the program oblivious to the problem. Thus, it is also possible to achieve the third principle of Persistence Independence.

Using simple pointcut expressions these `get`, or `set` access operations to object's attributes can be easily quantified. In our prototype, we just use two AspectJ pointcuts expressions (in Figure 3.17 one can see these two pointcuts within the application's execution). In the context of the orthogonal persistence, these two pointcuts are effective because they do not suffer from the pointcut fragility problem [Störzer & Koppen, 2004]. Notice that in an orthogonal persistent system any object can be persistent. It only requires an analysis in order to check its state of persistence and then it applies the adjusted behaviour. Thus, by applying these pointcuts, we achieve the goals of an aspect-aware interface [Kiczales & Mezini, 2005]. One can reason about the persistence concern without interfering with any other application concerns, because it is a technical problem that does not depend on the applications. Thus, these observations allow us to argue that *quantification* is also desirable in this category of systems.

This is distinctive from earlier works [Rashid, 2001; Rashid & Chitchyan, 2003; Soares *et al.*, 2006] in which the studied systems are non-orthogonal persistent. Although persistence has been successfully modularized, the reusability of the *aspects* is poor, because in those systems the persistence is a *role aspect* [Steimann, 2005]. Let us consider the example referred by Rashid & Chitchyan [2003] regarding GUI issues: GUI issues are part of the modelled problem domain. This contrasts with orthogonal persistence environments where persistence is a technical problem. Using AOP techniques, one can develop (technical) *aspects* of persistence for applications that are not aware of their existence. Thus, the combination of AOP and orthogonal persistence enables high levels of reusability, as well as all the other advantages identified in those earlier works. We base our argument on the intrinsic nature of *obliviousness* of orthogonal persistent systems. Such *obliviousness* enables reusability.

Al-Mansari *et al.* [2007] presented a solution for orthogonal persistence based on two new concepts: *Path Expression Pointcuts* (PEP) and *Persisting Contain-*

ers. The *Path Expression Pointcuts* [Al-Mansari & Hanenberg, 2006] are a new pointcut construct that applies path expressions on object relationships using a well-known technique [Campbell & Habermann, 1974], similar to XPath to find a node in an XML file. These expressions provide *aspects* with non-local information and relationship information of the objects, and are thus crucial to achieve the reachability principle. Figure 2.2 illustrates how a *Path Expression Pointcut* can be used in order to support persistence by reachability.

The *Persisting Container* is an object maintained by the *aspect* that provides persistence services, as an *ad hoc* functionality to all objects that it contains. When an object is added to a container, it is given persistence capabilities and it loses them when it is removed. Thus, classes do not need to be prepared to be persistent or non-persistent. Consequently, type orthogonality and persistence independence principles are met.

```
pointcut trapUpdates(PersistentList pl,Object o):
set(* *) && target(o) && path(pl -/->o);
```

Figure 2.2: Path Expression Pointcut (extracted from [Al-Mansari *et al.*, 2007])

Unfortunately, none of the current aspect-oriented tools provide this type of pointcut constructs. The authors just propose denotational semantics for *Path Expression Pointcuts*. Their contribution is a guide for future *Path Expression Pointcuts* developments and its integration with AOP systems. Thus, in our framework, discussed in Chapter 3, an object cache and a mappings based solution was adopted. These system's data structures are able of provide the framework with all required information regarding the objects' persistence state and relationships. In Section 3.3.4.1 we will discuss how applications are provided with persistence services in our framework.

2.2.2 Transactions supported through AOP

Concurrency control and failure management are two important facets of the persistence concern. These two issues are referred by authors [Kienzle & Guerraoui, 2002] [Rashid & Chitchyan, 2003] [Soares *et al.*, 2006][Soares & Borba, 2007] [Al-Mansari *et al.*, 2007] as impossible to be totally aspectizable and turn the

programmer oblivious to both.

Castor *et al.* [2009] presents an in-depth study of the adequacy of the AspectJ language for modularizing and reusing exception-handling code.

Fabry *et al.* [2008] argue that general-purpose aspect languages lacks a formal foundation to express advanced models of transactions. In order to overcome these limitations, the authors propose a domain-specific aspect language for advanced transaction management, called KALA. The KALA language is based on ACTA [Chrysanthis & Ramamritham, 1990], which is a formalized framework developed to characterize the whole spectrum of interactions. The ACTA formalism enables the specification of the structure and behaviour of the transactions, as well as the reasoning about the concurrency and recovery properties of the transactions [Chrysanthis & Ramamritham, 1990]. ACTA is a framework that is intended to unify the existing transaction models, hence opening these models to KALA.

Kienzle & Guerraoui [2002] made a detailed study about the aspectazition of these two concerns and classifies them according to three levels of aspectization ambition:

- *Transaction semantics* - All semantics related to transactions are hidden. The programmer does not have to worry about transactions. The authors argue that it is impossible to achieve;

- *Transaction interfaces* - In this approach, the transaction interfaces (*begin*, *commit*, *abort*, and others) are transferred from the functional parts to specific *aspects*. With this solution, a method can be made transactional by encapsulating it in a *around advice* surrounding the `proceed` statement with the transaction interfaces. This approach leads to some problems. The roll back operation must be done externally to the *aspect*, turning it into an intricate code where part of them is in the *aspect* and the failure treatment code remains in the functional part of the program. The authors also refer this level of aspectization as a hindrance to collaboration among threads, due to the fact that they can't enter into each others transactions. They also argue that it makes no sense to turn all application objects into transactional objects; the only methods that have transactional behaviour

must be intercepted and aspectized;

- *Transaction mechanisms* - This less ambitious goal is focused on the separation of mechanisms needed to ensure the ACID (*Atomicity, Consistency, Isolation and Durability*) properties of transactions from the main application and objects, and have these encapsulated within the code invoked through the respective *aspects*.

These issues were tested using the OPTIMA transactional framework [Kienzle, 2001], an object-oriented framework that provides the necessary run-time support for open multithreaded transactions. This framework supports, among others, optimistic and pessimistic concurrency control, different recovery strategies (*i.e.* Undo/Redo, NoUndo/Redo, Undo/NoRedo), different caching techniques, different logging techniques (*i.e.* physical logging and logical logging), and different storage devices.

The AspectOptima [Kienzle *et al.*, 2009] system is an aspect-oriented framework that uses aspect-oriented programming to decompose transaction models and their implementations into many individually reusable aspects. The overall goal of AspectOptima is to serve as a case study for aspect-oriented software development, in particular, to evaluate the expressivity of aspect-oriented programming languages and how they address complex aspect interactions and dependencies.

Kienzle *et al.* [2009]'s study was not focused on orthogonal systems, which have specific requirements. Even so, this research contributes with results that were very useful to the achievement of the proposed approach discussed in section 3.5.

2.3 Database Evolution

Database schema defines the interface which object-oriented applications must use to access persistent data. In the database layer, these interface definitions are represented through a metadata layer (schema). In the application layer, this interface is embedded inside the applications' source code or, in case of relational

databases, using design patterns, such as the DAO Design Pattern, or in a separate Object-Relational Mapping (ORM) layer. Our research work is focused on object-oriented environments in the particular context of the orthogonal persistence. In this kind of systems, because there is no *impedance mismatch*, both representations of the data schema are identical in these two layers: application and database.

The schema of an application is not immutable due to many factors. Organizations' processes are dynamic, consequently, their requirements are dynamic as well. People analyzing organization requirements and building applications make mistakes that must be corrected, sometimes when the development process has been completed. Also there are cases when new features are added to an application, implying new changes at the data model level. All these reasons result in frequent updates in the data structures, forcing a constant redesign of the schema with an important impact on the application, with particular complexity on legacy ones. Additionally, this is a time-consuming problem for programmers and database administrators.

The object-oriented database evolution occurs at two different layers: the metadata and the data layer. The former is referred to as schema evolution while the second is known as instance adaptation.

2.3.1 Schema updates (metadata layer)

The literature mainly refers three strategies that had been applied to deal with updates on the metadata structures of object-oriented databases. These strategies derive from the combination of tree main approaches: modification, evolution and versioning. As to versioning there are two different levels of granularities: schema level or class level. These three strategies are:

- Schema Evolution [Banerjee *et al.*, 1987a][Ferrandina *et al.*, 1995];
- Schema Versioning [Kim & Chou, 1988],[Lerner & Habermann, 1990]Lautemann [1997];
- Class Versioning [Bjornerstedt & Britts, 1988][Clamen, 1992][Monk & Sommerville, 1993].

Roddick [1995] has clarified the distinction between modification, evolution and versioning of a database schema. The author considers that schema modification is accommodated when a database system allows changes to be made to the schema of a populated database. The evolution implies schema modification without the loss of existing data. Finally, the versioning strategy goes further, enabling the maintenance of relevant historical states of the schema, as well as data object instances.

The O2 [Ferrandina *et al.*, 1995] and Orion [Banerjee *et al.*, 1986, 1987a,b] are examples of systems that implement the Schema Evolution approach. In this approach, after any step taken on the evolution of the database, the information about previous versions of the schema is lost. This method, while conceptually simple, does not allow parallel versions of the database schema. The loss of that schema information may cause the inoperability of existing applications.

Relational databases that follow this schema evolution approach can deal with that problem by applying views which emulate the previous versions of the relational schema. The database administrator applies schema updates and then defines a virtual data structure that supports the previous schema version. A similar approach was also explored in object-oriented databases in the Transparent Schema-Evolution System (TSE) [Ra & Rundensteiner, 1997]. This system supports schema evolution by adding virtual classes that emulate the changes to the data model. In truth, it does not apply any change to classes in the underling base schema. Each application version has a different and specific virtual schema view. This approach is somewhat limited by the fact that it does not allow the addition of new schema information and is therefore, an important constraint to the semantic updates [Benatallah, 1999][Lee *et al.*, 2006].

Another similar proposal [Lee *et al.*, 2006], the Rich Base Schema (RiBS) model, is also based on view mechanism but is less restrictive, allowing the execution of arbitrary schema evolution operations (defined in the Orion taxanomy [Banerjee *et al.*, 1987a,b]) against the current schema version. Additionally, this model supports schema version merging operations. This approach overcomes the model applied in the TSE system, as new semantic information can be added to the base schema.

Schema versioning approaches are well suited to provide the necessary information for backward and forward application compatibility because they can maintain the historical information regarding the schema (metadata) and object instances (data). This schema's information is necessary to provide a consistent view of the data in each state.

In this versioning category there are two different levels of granularity: at the schema and class levels. In the first case, a new schema is derived just when a single class is updated, while in the second, only that class is derived. Both approaches can give a consistent view of data structures. However, class versioning tends to make global schema very complex. In contrast, schema versioning does not suffer from this complexity problem but has poor granularity, forcing the definition of a new complete schema, even if that change is insignificant. This second approach is also very expensive in terms of space usage.

2.3.1.1 Schema consistency

Whatever the chosen strategy, updates applied to a schema must ensure database consistency, under the penalty that anomalous behaviours may occur. Zicari [1991] identified two types of consistency: *structural consistency* and the *behavioural consistency*. The first one is concerned with the static part of the database. Informally, it means that a schema must obey a set of invariants [Banerjee *et al.*, 1987b][Lerner & Habermann, 1990][Rashid & Sawyer, 2005]. In the literature, those invariants have been presented with slight differences but, in essence, they point to the same issues. One of them is the following:

- *Class Lattice Invariant* - The class lattice (inherence graph) is a rooted and connected Directed Acyclic Graph (DAG);
- *Unique Name Invariant* - All names must be unique. Each class must have a unique name and, inside a class, all properties, whether locally defined or inherited, must also have a unique name;
- *Full Inheritance Invariant* - A class must inherit all properties from each of its super classes; and

- *Type Compatibility Invariant* - The type of an inherited attribute must be the same type or the subtype of the attribute it inherits. Also the type of a property must correspond to a class in the class lattice.

The loss of consistency in the static part of the schema, during a schema update, implies that objects created under one schema could be unavailable in the other.

Behavioural consistency, on the other hand, is related with the dynamic part of the schema. This type of consistency refers to the behaviour of the schema at run-time. The changes can result in errors or other unexpected results while an application is running. This means that each method must respect its signature and its code must be aware of all static changes.

Structural and behavioural consistency refers to the schema level. However, semantic consistency also must be ensured [Monk & Sommerville, 1993] at the application level. Even when those two types of consistency are ensured at the schema level, the loss of semantic consistency implies erratic assumptions regarding the data. As an example, an object's attribute in one version may refer the weight of a person in pounds and, in another application version, that same attribute represents kilograms. Although this type of consistency is not exactly a schema issue, we consider it an important part of the application schema evolution problem.

2.3.1.2 Primitives for Schema Evolution

The update types applied to a schema can be categorized. Several taxonomies have been proposed for object-oriented databases [Banerjee *et al.*, 1987a,b][Tresch & Scholl, 1992][Ferrandina *et al.*, 1995][Rashid & Sawyer, 2005]. They depend on the underlying data model and architecture of the system implementation. For each taxonomy there are a set of primitives provided by the system which can be applied to the schema in order for it to perform the desirable evolution. All those operations should leave the schema in a consistent state. If they do not obey the schema invariants, their execution should be rejected by the system. The discussion of those taxonomies is beyond the scope of this research work, since our work relies on a different approach which is discussed in next two sections.

2.3.1.3 Evolving the schema's transparent programming systems

The db4o [Paterson *et al.*, 2006] system addresses schema evolution by offering some transparent mechanisms (termed as Automatic Refactoring [1]), which, in some cases, do not need any intervention on the database or the use of any tool. New classes as well as new attributes can be freely added. Deleted classes are simply ignored. Every attribute in the new schema, for which there is a matching attribute in the old schema, is initialized using the value of the matching old attribute in the original object. Attributes in the new schema that do not have matching attributes in the old schema are initialized with default values (`0, false or null`). Attributes in the old schema that do not have matching attributes in the new schema (deleted attributes) are simply ignored.

For those cases that do not obey these rules, this system also offers a renaming API which supports class and attribute renaming, as shown in Figure 2.3.

```
Db4o.configure().objectClass("MyClass.class").objectField("oldField").
    rename("newField");

Db4o.configure().objectClass("Mypackage.MyClass").
    rename("NewPackage.NewClass");
```

Figure 2.3: Renaming a class and attribute in db4o

Attribute operations, like merging or splitting, or semantic changes, are dealt with helper programs that transfer, and if necessary transform, the data from one class to another. Refactoring Class Hierarchy [2] also requires a helper program to copy data from old classes to the new one with a new inheritance-hierarchy and different naming. After all data is transferred to the new structures, old classes can be dropped. Finally, if there is the need for these new classes to preserve their old name, they must be renamed.

The ObjectDB system [Ltd, 2011] is another pure Java object-oriented database management system, which provides a non proprietary API based on standard

[1] http://community.versant.com/documentation/reference/db4o-8.1/net/reference/Content/advanced_topics/refactoring_and_schema_evolution/automatic_refactoring.htm

[2] http://community.versant.com/documentation/reference/db4o-8.1/net/reference/Content/advanced_topics/refactoring_and_schema_evolution/refactoring_class_hierarchy.htm

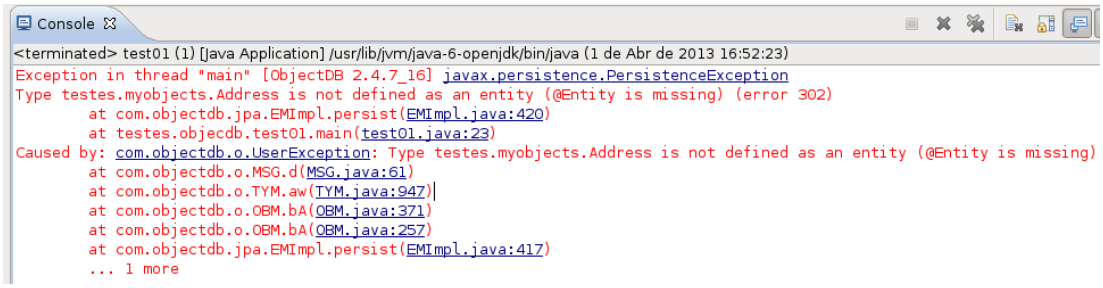


Figure 2.4: Run-time error due to a missing `@Entity` annotation in ObjectDB

Java APIs: *Java Persistence API* (JPA 2) and *Java Data Objects* (JDO 2).

Regarding the mechanisms' transparency, the ObjectDB system requires object source code to contain additional metadata as annotations. Figure 2.4 shows a run-time error due to a missing `@Entity` annotation. This requirement contrasts with db4o and our framework that do not require metadata or any other information in the definition of the classes.

This system implements an automatic schema evolution mechanism that enables the transparent use of old objects after the schema update. When an object of an old schema is loaded into the memory, it is automatically converted into an instance of the up-to-date class version. This is automatically done in the memory each time the object is loaded. The persistent object is lazily updated to the new schema version when that object is stored in the database again. The object conversion to a new schema version is done on an attribute-by-attribute basis, following the same principles of the db4o system. A matching attribute might also be located in a different place in the class hierarchy. That makes automatic schema evolution very flexible and very insensitive to class hierarchy changes (*e.g.* moving attributes across the classes' hierarchy, or removing an intermediate class in the hierarchy). This approach is applied to the default instance adaptation mechanisms of the proposed framework in Chapter 3.

However, more complex schema updates that require renaming can be addressed through a system's configuration file, an XML file within the default configuration. The ObjectDB system tries to apply the specified schema updates every time a database is opened.

In the <`schema`> element presented in Figure 2.5, one can see how this sys-

```
<schema>
  <package name="com.example.old1" new-name="com.example.new1" />
  <package name="com.example.old2" new-name="com.example.new2">
    <class name="A" new-name="NewA" />
    <class name="B">
      <field name="f1" new-name="newF1" />
      <field name="f2" new-name="newF2" />
    </class>
  </package>
  <package name="com.example.old3">
    <class name="C" new-name="NewC" >
      <field name="f3" new-name="newF3" />
    </class>
    <class name="C$E" new-name="NewC$E" />
  </package>
</schema>
```

Figure 2.5: ObjectDB's evolution mechanism (extracted from [Ltd, 2011])

tem supports schema evolution. Attributes, classes and packages can be freely updated following a strict hierarchy of <package>, <class> and <field> elements. Notice that in the db4o system, attributes operations, like merging or splitting, or semantic changes are dealt with helper programs.

PJama system [Dmitriev, 1998; Dmitriev & Atkinson, 1999; Dmitriev & Hamilton, 2001] is an experimental persistent programming system based on Java programming language. The PJama programmer simply modifies and substitutes the classes, outside the store on the application source code, recompiles them and passes the list of modified classes to an evolution tool.

2.3.1.4 Evolving the schema in orthogonal persistent systems

Those transparent persistent systems, discussed in the previous section, provide some orthogonality. The way schema updates are applied is very different from others discussed previously, which apply schema evolution primitives.

In pure orthogonal systems, however, the schema invariants are checked by the programming language compiler at compile-time. The absence of evolution primitives is replaced by the freedom of the programming language to perform the desirable changes, as well as the accuracy of the compiler to check schema invari-

ants. Our prototype follows this approach to evolve the application's schemas.

As demonstrated in PJama, db4o and ObjectDB, transparency and orthogonality shape the schema evolution problem very differently than other non-orthogonal persistent systems. Two main characteristics can be observed in this category of systems that explain the phenomenon. Those characteristics are:

- The schema is embedded in the application's source code (the class structures); and
- There is a close-coupled interaction between the application and the database management system.

In Chapter 3, this discussion will be continued in the context of our framework and meta-model. The proposed framework pertains to this category of systems. It explores their benefits in order to provide applications with semi-transparent mechanisms of database evolution, as well as all the mechanisms related with persistence concern.

Regarding the instance adaptation problem, in orthogonal systems their adaptation mechanisms are conceptually similar to the ones generally used in ODBMS systems. In orthogonal persistent programming systems no relevant characteristics have been found that sets them apart from other systems. In the next section we will discuss these mechanisms.

2.3.2 Instance adaptation (data layer)

The requirements in terms of application compatibility dictate the decision about the schema evolution strategy. However, if after a reorganization at the schema level the existing stored data of each object instance is not according the new schema definitions, the schema update process is not a closed issue. Thus, the instance adaption mechanism is an important part of the database evolution process and has significant implications.

An instance adaptation process must be able to deal with structural and semantic differences among class versions. A new class version may have a new attribute or lose one, only affecting its structure. Other updates may also affect data semantics: an attribute can be replaced by two (ex: name to first and last

name) or an attribute with the weight of a person could be changed from kilograms to pounds. In this second case, there are no structural variations. However, a new version definition, as well as mechanisms of conversion are required.

2.3.2.1 Application compatibility

Object-oriented applications expect to see retrieved objects from the database with a structure according a specific class definition. As discussed in previous sections, these classes have their structure definitions as being the application's schema. When the object-oriented database supports only one schema at each time, there is no compatibility between versions, implying a simultaneous evolution of database and application. This is a strong constraint for programmers and database administrators because their intervention should be synchronized. For users, this is also a constraint because they must stop their work during the technical intervention.

On the other hand, in applications that use databases that support multiple versions of the same schema, changes could be applied without affecting other applications. This type of database can provide each application with a view of the data as it expects. To ensure this compatibility, the database management system must adapt the retrieved information of objects before delivering them to the applications. In others words, the database must provide a specific interface that is compatible with the one in the application. Additionally, the bidirectional (backward and forward) compatibility also depends on the existence of enough historical information about the various states of object instances.

The compatibility of applications among different schema versions can be analyzed on four levels of ambition: (1) the system does not provide any adaptation mechanism and so data is simply lost; (2) the data is converted from the old schema to the new and consequently old applications stop working; (3) the data can be viewed by old and future applications, but all schema changes must guarantee certain conditions; (4) the data can be used by old and future applications without restrictions.

In the first case, it is the programmer that must write special conversion applications that transfer and convert the data from one schema to another.

In the second approach, the database already provides native instance adaptation mechanisms. Here, object instances are simply converted from the old to the new class version. This technique is based on a schema evolution strategy. Programmers could provide user-defined conversion functions that transform data from one structure to another, in addition to some default conversion behaviour natively provided by the system.

Both third and fourth approaches require the maintenance of historical information regarding the various states through which the schema has passed. The schema evolution approaches, based on schema versioning and class versioning, can deal with these requirements in terms of compatibility.

In these scenarios, techniques like screening [Banerjee *et al.*, 1987b] provide conditions to emulate [Clamen, 1992] objects in their classes in the respective schema version. As we will discuss, the screening *per se* has some limitations that impose some restrictions on schema changes under the penalty of data loss. Other approaches [Skarra & Zdonik, 1986] put limitations on the semantic variations among different versions of the object.

Combined emulation techniques can prevent data loss and semantic variations, supporting a fully versionable and emulated environment. Such is the case of the object versioning [Clamen, 1992] approach, which preserves object information in each existing version. The object versioning technique overcomes all emulation limitations, providing full support for bidirectional application compatibility.

2.3.2.2 Conversion

The conversion of object instances in a schema evolution strategy, or even combined with emulation (while maintaining the objects in the most pertinent class version), can be done in two distinct modes: (1) immediately (or eager or early) (2) deferred (lazy). The former initiates the propagation of the schema update on objects without delay. On the other hand, the lazy mechanism delays the conversion until it is necessary or to a more appropriate moment, like a data update.

The O2 [Zicari, 1991] system has a special command, `transform database`, forcing immediate execution, placing the choice of the appropriate time to perform

the conversion in the hands of the database user.

Conceptually, the immediate approach is very simple. The schema updates are applied in a stop-the-world model, like the PJama [Dmitriev & Atkinson, 1999] system, or without requiring the system to stop. Naturally, for an object-oriented database with a medium or large dimension, the immediate conversion of all instances could introduce a very significant and intolerable waiting state while the transaction is in execution. Thus, deferred approaches are well suited for a large database scenario.

Later, in Section 2.3.2.6, other issues related with the lazy approach when used with complex functions are discussed.

2.3.2.3 Object versioning emulation

Object versioning emulation, as the name itself suggests, consists in the emulation of the object to the desired version independently of its real persistent state version in the database. Obviously, this technique has a great advantage over a strict conversion approach. First, because it allows application compatibility and secondly, because it avoids, or delays, the need for physical conversion, a characteristic that is especially relevant in large databases.

However, in an emulation approach, objects are stored with a structure that may not respect the running application interface. Thus, the emulation of object versions also requires a conversion process to convert and create a virtual view of the object version as the applications expect.

The object versioning emulation is commonly based on the screening technique. It consists in associating an object instance to the class version under which it was created. After creation, the image structure of an object never changes. The object is screened to the applications as they expect, hiding or not some attributes.

The Orion system [Banerjee *et al.*, 1987b] uses this technique to avoid object update when one of their attributes is dropped. The O2 system [Ferrandina *et al.*, 1994] also applies the screening technique to avoid information loss, when the termed complex functions need to correctly retrieve several objects in different old versions in deferred physical conversions.

Although the screening technique has advantages over versioned approaches, as will be discussed, in terms of user space (since each object only occupies one slot space), it has two severe limitations. The first one is the overhead, due to the cost of the emulation process that occurs when the persistent version of the object is different from the one expected by the application. The second limitation is a serious problem. Consider a class `Person` in version "A". In the following version "B" of that class, a new attribute called `age` is added. If an update of an object instance created under the class version "A" occurs, the information about the person's age is lost. When emulating for version "B", the best that can be done is to fill that attribute with a default value because there is no place to save the person age value. This second limitation makes this technique unsuitable for backward and forward application compatibility.

In the Encore [Skarra & Zdonik, 1986; Zdonik, 1987] system, a wrapping scheme that extends the version set interface of a class with extra attributes is proposed. A version set is an ordered collection of all incarnations of a particular object. The version set is initialized with the first definition of a type, it is expanded with a new version with each modification of the type, and it is terminated when the type is deleted. When a type is instantiated, the user may specify a version. If the user does not, the default version of the object is obtained. One of the versions of a version set is termed as current version, this being the one most recently created. Each version set of a specific class has an interface which is viewed by the application.

In the Encore system, since virtual classes somewhat preserve all removed attributes pertaining to old versions, any application using their specific interface expects the existence of all those attributes which it knows. For this reason, associated to each one of those attributes, of the version set, there is a handler that treats an error condition [Skarra & Zdonik, 1986]. These handlers catch errors that may occur in an attempt to access an attribute that does not exist in the class version of the instance object. For these two specific problems, reader and writer, Encore has read and write handlers.

The CLOSQL [Monk & Sommerville, 1993] system supports multiple versions of a class with screening. To solve the problem of information loss, each time that an object is converted to a version where an attribute is dropped, the sys-

tem automatically saves that lost information, which is then restored when the instance is re-converted. The authors of this work have also proposed that when a new version of the class is created, a backdate method in that new class version should be defined to convert to the previous one, as well as an update method in the n-1 class version. Following this path of update/backdate methods, it is possible to convert an instance from any version to another. These backdate/update methods have the knowledge regarding the semantic differences between two adjacent versions, contrasting with Encore's approach, which is based on version sets where the mapping is restricted to the equivalence of attribute names. Thus, the backdate/update methods approach improves emulation mechanisms with semantic adaptation.

2.3.2.4 Object versioning

In the object versioning techniques, the derivation of a new class version leads to the maintenance of the respective object's version in the database. Thus, all application interfaces of the object's information are preserved. However, the usage storage space increases in the same proportion as the number of versions. In large databases, this limitation becomes a very problematic issue. In terms of overhead, we consider that there are advantages and disadvantages. While it is true that it is necessary to manage the whole complexity associated with versioning, it is also true that any emulation or conversion of objects to the required application version is unnecessary.

Clamen [1992] proposed that each instance could be composed by multiple *facets*: as many as the existing versions of its respective class. Each *facet* encapsulates the instance's state of each version of the application interface. To ensure coherence between the same information, replicated in the multiple *facets* of an object, this system uses a trigger mechanism that propagates updates of an attribute. A deferring strategy, which updates the *facet*'s attributes only when it is needed, is provided in order to improve the performance as much as possible.

In order to overcome the usage space limitation, Benatallah [1999] proposed a unified framework that takes the best of the schema evolution and versioned approaches. That is to say, it avoids information loss with a new version of

the schema and object versioning when possible. To reduce the proliferation of versions at both levels, data and metadata, it applies schema evolution using object conversion and screening. In fact, this unified approach controls version proliferation and information loss, but could compromise the backward application compatibility if no cautions were taken. Thus, the authors have introduced the concept of class pertinence levels [Benatallah, 1999], which prevents certain class versions from being lost: A class may be pertinent or obsolete. This pertinence level depends on the number of applications that use this class version and whether or not it pertains to the most recent schema version.

2.3.2.5 How adaptation is made

An instance adaptation mechanism is required in both conversion and emulation approaches. Both cases are based on very similar instance adaptation mechanisms. This instance adaptation process requires a transformation mechanism capable of migrating, and, when necessary, semantically transforming data.

Similarly, as in schema evolution, at the metadata level, the instance adaptation process must also obey a set of transform invariants [Lerner & Habermann, 1990], which dictate the rules of data transformation. These transform invariants: (1) completeness; (2) correctness and (3) sharing, preserve the object state (within 1 and 2) and object identity (within 3), in order to preserve structural, semantic and behavioural consistency.

Object identity has particular importance considering that a composite object (*e.g:* a `Person` or `Invoice`) referred by other objects must continue to be referred by them after any conversion. In orthogonal persistent programming environments, object identity is central because objects are accessible by reachability. If they lose their identity, they become inaccessible and consequently they are deleted.

Figure 2.6 gives a very simple and concrete example of a conversion function of a `Person` object, in which its weight is represented in kilograms, in the old class version, while in the new one it is represented in pounds.

In the O2 system, this function is given as part of a transaction that performs the schema update [Ferrandina *et al.*, 1995]. That is to say, a set of primitives

```
conversion function conv_person(old: tuple(name:string,weight:real))
in class Person
{
    self.name=old.name;
    self.weight=old.weight * 2.20462262;
}
```

Figure 2.6: O2 conversion function

for schema updates and instance adaptation are given to the system as a single transaction (delimited with begin/end modification syntax).

The PJama, an example of a persistent programming system where schema evolves differently, has another mechanism [Dmitriev & Atkinson, 1999] which is also based on conversion code written in the system's native programming language (Java). The old classes are renamed with `$$_old_ver_` suffix to remain available to the programmer in the programming environment together with the new versions. Functions are defined as special methods in the conversion classes that have special signatures that are automatically recognized by the evolution system.

```
public static Person convertInstance(Person$$_old_ver_ old)
{
    Person newP=new Person();
    newP.name=old.name;
    newP.weight=old.weight * 2.20462262;
    return newP;
}
```

Figure 2.7: PJama conversion method

For each supported type schema update in the taxonomy of a PJama system [Dmitriev, 1998], there is a specialized method with a dedicated signature for that operation. Figure 2.7 presents an example of a conversion method.

Barbados [Perez-Schofield *et al.*, 2002] is a research prototype of an object-oriented persistent programming system which integrates a persistent object store and a development environment with a C++ based programming language. This

system applies a container-based persistent model, dividing the persistent store into groups of objects, each one visible to users through the abstraction of directories of a file system. This persistent programming system uses an instance conversion mechanism inspired in PJama, using a similar approach to the renaming old classes with suffix `$$_old`.

These conversion functions are introduced in other systems in very different ways: As backdate/update methods, in the CLOSQL system, or handlers, in the Encore system, or even as an *aspect* in other approaches [Kusspuswami *et al.*, 2007; Rashid, 1998] based on aspect-oriented programming.

On the other hand, the Orion system can only perform evolutions for which it has a rule defined. Its evolution taxonomy is restricted to the definition of a default value and to the domain generalization of an attribute.

Default and user-defined conversion - These conversion functions defined by the user are required when structural and/or semantic differences between versions could not be inferred transparently by the system. In many cases it is possible for the system to use the so-called default functions that should be applied if no user-defined function is explicitly given. Generally, the system default conversion behaviour is dictated by a set of rules, based on the principle that attributes have the same name and their types are compatible (`int` to `long` or `float` to `double`). In these cases, the values are just copied from the old attribute to the new one. Other rules could handle the conversion when those desirable conditions do not occur. For example, a new attribute is initialized for a default value, generally zero or null, depending on its type. In the opposite case, when an attribute is removed in a new class version, it is ignored.

Simple and complex conversion functions - Ferrandina *et al.* [1994] have classified conversion functions as follows:

- *Simple conversion function* - The object adaptation process evolves only object local information. In this case, the conversion generally consists in a simple data migration with or without any semantic conversion. That

function could be defined as

$$newObj = f(oldObj) \tag{2.1}$$

- *Complex conversion function* - In this case, the adaptation process uses local and non-local information. The new object instance could be the result of several operations involving several other object instances. This second type of function could be defined as

$$newObj = f(oldObj1, , oldObjn) \tag{2.2}$$

2.3.2.6 Lazy conversion with complex functions

Earlier we discussed the lazy conversion approach as a deferred mechanism that delays conversion to the most appropriated moment. A conversion process has implicit that object instances evolve following the same order as the schema evolution steps. However, this approach rises some issues, requiring special attention when used with complex functions because they are confronted with multiple objects, possibly in different versions. It is possible that another application may have already triggered the conversion of one or more of those function's inputs to versions where some data may have been lost.

In order to make this problem clearer, consider the following example: The complex function `f` has, as input, an object from class `x`, but also uses other objects from classes `y` and `z`.

$$f(x, y, z) \tag{2.3}$$

It is possible that `y` is in an upper version when an attribute has been dropped, breaking its structural consistency. The O2 system solves this problem, supporting lazy conversion with complex conversion functions, using the screening technique [Ferrandina *et al.*, 1994]. When some information is deleted and/or modified in the schema, it is only screened out without being physically deleted from the database. The applications see a structure while the conversion process sees another, more complete, with all versions of the class.

2.3.2.7 Object conversion in multi version schemas

In a scenario of sequential schema updates, lazy conversion with complex functions is a simple problem because old schema versions are dropped. However, in emulated multi-version schema scenarios this problem is very complex. In these types of systems, each time an emulated object version is required, which depends on other objects, the problem referred to in Section 2.3.2.6 becomes more complex. This is because these input function objects can be found in any future or older version. Additionally, those objects may even refer to other objects in any version.

Behavioural consistency is also an important issue in a deferred conversion strategy. In addition to the schema's static information, conversion functions could use methods of the converted objects if needed, empowering the expressiveness of these functions and the semantic of the upgrade. Such complexity has been avoided by limiting the expressiveness of the applied conversion functions, *i.e.* by disabling the use of non-local object references and the methods invocation. In these cases, the conversion functions are only faced with data under the previous schema version, making the problem quite simple.

Our prototype aims to provide users with such expressiveness in conversion functions, enabling access to non-local information and object's behaviour. This prototype addresses this problem with a specially designed meta-model (see Section 3.2) and our pointcut/advice constructs (see Section 3.2.6.3).

Boyapati *et al.* [2003] researched the lazy upgrades issue and have formulated what they call *upgrade modularity conditions*. These are a set of conditions that constraint the behaviour of an upgrade system. The conversion functions of a system that runs under those conditions only encounters objects whose interfaces are known and where object states are coherent with these interfaces. That is to say, the *modularity conditions* provide the needed structural and behavioural consistency of the data to the conversion functions, under a scenario where objects dispersed among different class versions can be found. That is achieved while granting the correct order and the encapsulation of the transactions. When an upgrade is installed, it must know all interfaces of all upgraded classes. Thus, the developed system only accepts the upgrade - a set of class-upgrades -, if it is

complete.

In the authors' approach, an upgrade transaction transforms all objects of the old class and each conversion function runs its own transaction. Thus, an upgrade transaction consists in the execution of all conversion transactions of that upgrade. The application transactions are interleaved with individual conversion function transactions, thus avoiding long waiting periods. When an application transaction is about to use an object that is waiting to be adapted, the system interrupts that application transaction and runs the conversion function at that point.

The authors of this work assume that the majority of the functions are well-behaved (*i.e.* are simple conversion functions). Their approach exploits that fact, providing an efficient way to enforce *modularity conditions* without requiring versions. For other situations where *modularity conditions* are not possible, they foresee the use of additional mechanisms based on triggers and versions. By using triggers, the programmers can explicitly define the order of the transactions. When there is no way to ensure the correct order of the transactions, an object versioning strategy is adopted as a solution to the problem.

2.4 Earlier related work

In the last years, the database evolution problem, using aspect-oriented programming techniques, has attracted the attention of several researchers [Rashid, 1998][Rashid & Leidenfrost, 2004][Rashid & Sawyer, 2005][Rashid & Leidenfrost, 2006][Kusspuswami *et al.*, 2007][Jie, 2010][Song *et al.*, 2012]. In this section two of those earlier related works, which most contributed to a successful implementation of database evolution mechanisms as an *aspect* (in the aspect-oriented sense), are discussed.

2.4.1 SADES and AspOEv

Rashid [1998], in the SADES system, introduced the concept of aspect in the object-oriented databases, making the system itself an aspect-oriented database [Rashid, 1998]. In this research work some cross-cutting concerns that could

be aspectized in terms of aspect-oriented programming were identified. These cross-cutting concerns exist at the database manager system and database levels. Constraints checking, access rights, as well as all related aspects with the database evolution domain problem, such as schema evolution, instance adaptation, inheritance and versioning, are good examples of those concerns.

In SADES system and later in the AspOEv system [Rashid & Leidenfrost, 2004, 2006], aspect-oriented programming techniques were explored in order to implement database mechanisms of schema evolution and instance adaptation. These systems use a meta-object model [Rashid & Sawyer, 1999], based on three types of entities that represent data and schema structures: objects, meta-objects and meta-classes. The objects represent the applications' data and reside in an area called object space. On the other hand, the meta-objects are special objects that represent each element in the object model of the schema: Classes, attributes, methods, parameters, relationships and all other possible types. For them there is the meta-object space. Meta-objects are instances of the meta-classes. Meta-classes reside in the meta-class space. Although in separate virtual spaces, the metadata (in meta-objects and meta-classes), the data (in the objects) and even the aspects coexist inside the same database.

The relationship between objects could be intra-space, when involving objects pertaining to the same space, or inter-space when the objects pertain to distinct spaces. One example of intra-space relationships are the connections between two meta-objects which represent an inherence relationship from a class to its super-class. Examples of inter-space relationships are the instance-of relationships between objects and their classes (the meta-objects), as well as meta-objects with their respective classes (the meta-classes). Rashid & Sawyer [1999]'s meta-model also supports version derivation graphs [Loomis, 1992], which represent the derivation path of class versions and objects. These two types of distinct relationships are intra-space. As previously mentioned, the *aspects* also have their place in this meta-model. Thus, inter-space relationships among *aspects* and classes can also exist.

Schema evolution and instance adaptation are two database concerns that were identified as transversal to schema manager, meta-objects and objects. At the database management system layer, the primitives for schema evolution are

interpreted and executed, activating the mechanisms that operate in the meta-objects, modifying their content in order to represent the new schema. At the database layer, for instance adaptation, the conversion functions are associated to each object version. Simultaneously, the system manager must ensure the structural, semantic and behavioural consistency of the data and metadata. This complexity and the tangled mechanisms spread the system's behaviour across several components of the database management system and data entities.

The authors of this work remark on the high degree of flexibility provided by their approach when compared to other systems. They argue that, traditionally, object-oriented databases only offer a single schema evolution approach coupled with a specific instance adaptation mechanism, which does not serve the application's needs effectively [Rashid & Sawyer, 2001]. They have concluded that the aspect-orientation of a database enables a pluggable and customizable system, which can provide database administrators and programmers with powerful means to do their work [Rashid & Sawyer, 2001]. At the database management system level, it makes it possible to access strategic internal points of the system in order to introduce customizable components. In the scope of database evolution, the most appropriated approach can be chosen or even be a combination of them [Rashid & Leidenfrost, 2004; Rashid & Sawyer, 2001]. In traditional systems this is very difficult or impossible. They have identified the following reasons: (1) These concerns are overlapping and intertwined with other elements such as transaction management, type checking, security and others. Thus, any customization could affect the consistency of the global system; (2) The internal complexity of the database management system and the vendors' reluctance to expose their systems' internal operations; (3) Finally, the implications of the evolution mechanisms in type checking as different schema evolution approaches might have different perceptions of type equivalence [Rashid & Leidenfrost, 2004].

At the database level, the stored entities, both data and metadata objects, can have their own *aspects*. Thus, the authors' meta-model enables these *aspects*, themselves, to be made persistent entities also stored in the database. In the authors' opinion, with which we agree, this approach opens a broad spectrum of customization solutions. For example, in data objects, the specific concerns of the application logic and entities could be aspectized. Moreover, the classes' *role*

aspects, for instance adaptation, are likely to be aspectized. This approach avoids the need to introduce new code in the classes every time the conversion methods are updated, *i.e.* when a new class version is created.

The AspOEv system introduced its own language, the Vejal [Rashid & Leidenfrost, 2006], capable of manipulating the objects in their multiple class versions. An application based on the AspOEv system is written in Vejal programming language. With this approach, Rashid and Leidenfrost intend to overcome the generalized limitation of the object-oriented programming languages, which do not support type versioning. In Vejal, the class `Person` in version `1` is represented as `Person<1>`. In another syntax, `Person<s=1>` means the class version of `Person` that occurs in version `1` of the schema. In Vejal, a class in its version is a type. We concur with the authors' opinion that this approach improves type-checking safety by avoiding incorrect type inferences when evolving the database with the adaptation of types and instances.

The AspOEv system was inspired in SADES. However, SADES uses aspects to directly plug-in the instance adaptation code into the system. Consequently, the complexity of the instance adaptation hot spots has to be exposed to the programmer. With this exposure there is the risk of unwanted interference with the database from the *aspect*'s code. Furthermore, SADES supports customization of the instance adaptation approach only; the schema evolution strategy is fixed. It does not support version polymorphism either.

In AspOEv, the Vejal language provides programmers with the means to deal with class structure variations as version types. Therefore, applications must be aware of semantic and structural variations among class versions. This contrasts with our approach, which offers the possibility to modularize these *role aspects* of classes [Steimann, 2005]. Notice that in our framework the details of how to convert a class among its versions are totally separate from the applications and the framework itself. Each application only knows its own class structure.

The AspOEv system requires programmers to be aware of low-level technical details. Database consistency is ensured using *Reflective Handlers* [Rashid & Leidenfrost, 2006], which handle the mismatch exceptions through *reflective generators* [Connor *et al.*, 1994; Kirby *et al.*, 1996], *i.e.* reflectively altering dependent code. This also contrasts with our framework in which programmers just

write the appropriate conversion functions to adapt objects. In our framework, the pointcut/advice constructs provide the same mechanisms as these handlers. However, programmers only have to write conversion code restricted to the conversion scope, minimizing the risk of interference with the rest of the system. Furthermore, programmers do not need to know about the technical details regarding the framework, or to write code to reflectively manipulate the classes' structures.

2.4.2 AOP in instance adaptation

Kusspuswami *et al.* [2007] have also explored aspect-oriented programming techniques to propose a flexible instance adaptation approach. In their work, the authors have developed a system that supports instance adaptation with two *aspects*: update/backdate and selective lazy conversion *aspects*.

The update/backdate *aspects* implement the concern regarding the emulation of object versioning. The objects are retrieved from the store, in their physical versions, and are then converted to the expected application interface.

The second *aspect*, a selective lazy conversion, is responsible for physically converting the stored objects when they are accessed and conversion is deemed as necessary. This *aspect* operates conversion under a deferred approach to avoid the disturbance of the normal system's functioning. Moreover, this *aspect* is selective about the instances to be converted. In some cases, when necessary, physical conversion occurs and, at other times, objects are kept indefinitely in their versions. In the first case, the conversion occurs in order to accommodate new object's information, when it is updated under a new class version. On the other hand, legal issues may not allow the modification of the information (*e.g.* accounting records) or it may be that the information is obsolete, making physical conversion unnecessary. In the latter, the structure and behaviour of any object instance that remains in an older version is emulated through the former *aspect*.

This system implements its own persistence mechanism based on serialization. The persistence manager provides these persistence mechanisms as an *aspect*. Each time an object is created or updated, that persistence *aspect* invokes serialization and identity mechanisms. Other issues such as data integrity constraints

and adaptation invariants are also implemented as *aspects*.

In this work, the authors also highlight the flexibility provided by the aspect-oriented programming techniques to support database evolution concerns. They conclude that concerns encapsulation in *aspects* enables the easy replacement of the adaptation strategy and code, contrasting with other existing systems that introduce code directly into the class versions.

The approach proposed by Kusspuswami *et al.* [2007] implements *role aspects* as update/backdate *aspects*. Although the basis of this approach is similar to the one proposed in this PhD Thesis, it does not offer a systematic interface to declare and implement these *aspects*. Programmers must implement instance adaptation aspects using AspectJ or another aspect-oriented programming language.

These earlier works show that aspect-orientation of the database evolution enables high levels of flexibility and customization. However, they are not focused on orthogonal persistent systems. We are convinced that the AOP techniques are well suited for the modularization of persistence and database evolution concerns in orthogonal persistent systems. Thus, our contribution is focused on identifying the advantages and disadvantages of orthogonal persistence for that purpose. We also intend to propose a new aspect-oriented approach to modularize both technical and domain aspects of database evolution. We argue that such an approach overcomes earlier works in terms of reusability.

Chapter 3

Developed meta-model and framework prototype

3.1 Initial considerations

In this chapter we discuss our proposal for a meta-model and its prototype [Pereira & Perez-Schofield, 2010, 2011, 2012, 2014a,b], the AOF4OOP (Aspect-Oriented Framework for Orthogonal Object Persistence) framework, an aspect-oriented Java framework capable of providing an object-oriented application with orthogonal persistence and database evolution services.

The main goal of our prototype is to test the feasibility of our approach, as well as the evaluation of the applicability of aspect-oriented programming in the development of transparent mechanisms of persistence and database evolution in orthogonal persistent programming systems.

This prototype is a persistent programming environment, which follows the three principles of orthogonal persistence formulated by Atkinson & Morrison [1995]. This thesis corroborates with this previous work, as well as others, that argued that the orthogonal persistence paradigm allows full transparency and safe querying, while interacting with the database.

Our framework mediates the interaction between the applications and the database, within an aspect-oriented approach. In Figure 3.17 one can observe these three layers: Application, framework and database. In the database, the

applications' class structures are represented through our meta-model. In the current version of our prototype, a db4o [Paterson *et al.*, 2006] database is used as object and meta-object repository. Although this object-oriented database already provides object persistence and has some transparent database evolution capabilities, its role is reduced to a simple object store.

The code listing presented in Figure 3.1 depicts how the applications interact with their data using the framework's API. In this example, in particular in lines 10 and 13, the high level of transparency, as the framework provides applications with persistence mechanisms, is highlighted. In these distinct application points the same persistent root delivers two distinct types of object to their variables without the need for any type of casting. Another important characteristic of this implementation is obliviousness regarding the persistence concern, which is implemented using aspect-oriented programming in a specialized *aspect*. That is to say, our framework was conceived in order to provide applications with persistence mechanisms, even without them having been prepared to manage the persistence of their data because persistence is achieved by reachability. As we will discuss in Section 5.3, minor changes are required in applications in order to provide them with persistence mechanisms.

As shown in lines 10 and 13 in Figure 3.1, applications access objects through named root objects. However, our framework still provides a query API (see Appendix C - API Reference Guide and Section 3.3.2.1), which enables database querying in order to retrieve *ad hoc* collections of objects. The code listing presented in Figure 3.2 exemplifies this form of accessing persistent objects. In this example, the query retrieves from the database all objects of the class `Person`, independently of the class's version. In fact, despite how instances are loaded from the database, each of these persistent objects are roots that enable all others that are reachable from it access.

The developed framework does not require any changes made to the Java Virtual Machine. Its internal operation, based on the aspect-oriented philosophy, allows access to strategic points of the application, as well as the framework itself. This framework is composed by a set of core services and *aspects* (in terms of AOP), as will be discussed later in Section 3.3.5. This set of core services implement all core and default mechanisms, which support applications and the

```
01  CPersistentRoot psRoot;
02  Student student;
03  Student student2;
04  Course course;
05
06  // get a persistent root (psRoot)
07  psRoot=new CPersistentRoot();
08
09  //Get one Student object from the psRoot (the database)
10  student=psRoot.getRootObject("rui");
11
12  //Get one Course object from the psRoot (the database)
13  course=psRoot.getRootObject("TOC");
14
15  //Associate the persistent Student object
16  //with the persistent Course
17  course.addStudent(student);
18
19  // Instantiates a new Student object (still non-persistent)
20  student2=new Student(1234,Student Name,Student Address);
21
22  // Turns the student2 persistent simply because it
23  //is associated to another persistent object
24  course.addStudent(student2);
```

Figure 3.1: Application example

```
List<Person> persons=psRoot.query(new CQuery(Person.class));
```

Figure 3.2: Querying database

framework's functioning itself. On the other hand, these aspects support all concerns that require a flexible and customizable implementation. Additionally, applications' class *role aspects* can be stored in the database in order the extend the framework with new instance adaptation behaviour.

Our framework treats the data types orthogonality, *i.e.* its persistence mechanism does not require applications' classes to extend some superclass or to implement any interface. Thus, in our prototype there is a common *aspect* of instance adaptation that applies the framework's default mechanisms. For each application's class under evolution, programmers can extend this *aspect* using the metadata stored in UBMO meta-objects. These UBMO meta-objects pro-

vide the system with these *aspects* (in terms of AOP), specific for each class version. As will be discussed, the framework's schema manager and the default instance adaptation mechanisms, in conjunction with the enriched schema (by means of metadata) in the application's source code can, in many cases, transparently handle the database adaptation problem. Only in those cases, where these framework's base mechanisms are unable of deal with database evolution problem, is programmer's intervention required. In such cases, programmers must write our pointcut/advice constructs (aspect-oriented modules) which are stored in the data as UBMO meta-objects.

3.2 System meta-model

The established requirements for the developed framework in terms of application compatibility required the adoption of a simple and flexible data meta-model in order to support schema metadata and the applications' data entities. Our model was designed in order to allow the versioning of the schema through class versions, hence enabling forward and backward application compatibility. This meta-model still enables additional metadata which is put into the application's source code, enriching the class structure definition. The framework takes advantage of that additional metadata in order to enhance the transparency of the database evolution process.

The meta-model approach is based on three types of entities: objects, meta-objects and meta-classes. The objects (data entities) are instances of classes in a specific version, which are represented through a special meta-class. Meta-objects are instances of meta-classes. This meta-model considers a limited set of built-in specialized meta-classes to represent the data objects in each class version, the relationships among them and all other data. The existing types of meta-object are:

- Class Version Meta-Object (CVMO) - In our approach, a class may have several versions. Each CVMO meta-object supports the metadata regarding an application's data class in a specific version. Besides metadata, these meta-objects preserve the classes' bytecode (in its class version), which feeds

the system's classloader. Moreover, the classes' bytecode also provides the framework with additional metadata.

- Instance Meta-Object (IMO) - Each one of these meta-objects is a logical representation of an application's object instance. This type of meta-object is associated with one or more CVMO meta-objects, as well as to one or more object's image (*facet* of an object [Clamen, 1992]). This set of CVMO meta-objects provides the framework with historical information regarding the object's class versions, while the set of related objects provides input data for the object emulation process of each one of those class versions.

- Attribute Meta-Object (AMO) - The AMO meta-objects represent the relationships among the data objects, which are represented through IMO meta-objects. Since objects are related through their attributes, this meta-object represents an instance variable that points to another object instance. This meta-object has two distinct forms: a single object reference or a reference object array.

- Root Meta-Object (RMO) - The RMO meta-objects are database starting points, giving access to all other reachable objects. Each one represents a root object [Atkinson & Morrison, 1995] identified by an arbitrary key string. This key string is the name given to a named root of our API, as can be seen in the examples in lines 10 and 13 of the listing in Figure 3.1. The key value is an arbitrary string that univocally identifies the root object. Like AMO meta-objects, these meta-objects also have the same two forms for objects and arrays.

- Update/Backdate Meta-Objects (UBMO) - This kind of meta-object contains all matching information and conversion code needed by the instance adaptation *aspect*. It is required only when the default instance adaptation algorithm is unable to perform the adaptation transparently. Each of these meta-objects support a pointcut/advice construct (in Section 3.2.6.3 we will discuss these constructs) .

- Aspect meta-objects (AspMO) - Each one of these meta-objects support an *aspect* (in terms of AOP), which can have one of two usages: application

aspects (*e.g.* auditing or logging) or database management system aspects (*e.g.* security). These meta-objects enable the persistence of aspects, thus making the database itself aspect-oriented [Rashid & Pulvermueller, 2000].

The existing types of meta-objects of the presented meta-model are reduced and very specific as to their usage. Additionally, the meta-model does not explicitly represent the derivation path of all the classes' versions. In our opinion, this approach reduces the implementation complexity and promotes global system performance.

As previously mentioned, besides those meta-objects, inside the database there are also the application's data objects. Each one of these objects is under the class version it was created in. The framework emulates the object version as required by the running application. If the emulation to a specific object version is not viable, the meta-model enables a copy of the object for that version to be maintained. However, in our prototype only a single image (*facets* object [Clamen, 1992]) is maintained. All these versions of the same object are associated to their respective IMO meta-object.

Figure 3.3 illustrates an example of how these meta-objects are related in order to represent a course associated with its students. The persistent root is an object of class `Course`, which is pointed through a RMO meta-object and its key value. In the presented example, this key has the value `"TOC"` which is the course's code.

The course object has an IMO meta-object that supports its logical identity. Notice that in the database a class may have instances in distinct versions. Thus, a logical object (IMO meta-object) supports an instance in any of the existing class versions. In the example, the `Course` class has three versions: `"A"`, `"B"` and `"C"`. However, only versions `"A"` and `"B"` require an explicit relationship using an UBMO. This meta-object supports user-definitions for object conversion from class version `"A"` to `"B"`. The conversion to version `"C"` can be handled by the default conversion mechanisms as discussed in Section 3.2.6.1. In this example, the course object in the database is in version `"A"`.

The AMO meta-object supports the students enrolled in the course. This meta-object supports the member `students` of class `Course`: an array of objects

pertaining to `Student` class. In turn, this array points to a set of IMO meta-objects. Each one of them is a cell of the array in a specific version of class `Student`. Notice that other AMO or RMO meta-objects may refer to these objects (cell arrays).

The class `Student` is a subclass of the class `Person`. This hierarchical relationship of class is not explicitly represented in the meta-model because it can be obtained from the class's bytecode through reflection (see Section 3.2.2). In our example, both versions of the class `Student` (`"A"` and `"B"`) share the same superclass in its version `"A"`. Finally, *Instance-of* relationships are supported using CVMO meta-objects. These CVMO meta-objects contain the classes' metadata.

Our meta-model was inspired by Rashid & Sawyer [1999]'s meta-model. Nonetheless, it defines a much more limited set of meta-classes. These meta-classes represent the application's classes, the relationships, and all the metadata required to emulate the applications' objects in every existing class version. We argue that our approach reduces the number of affected meta-objects (metadata) each time the class structure is updated. In Section 3.2.2 we will discuss how the model's complexity is addressed. In Section 4.1 we will compare Rashid & Sawyer [1999]'s meta-model with our own.

3.2.1 Object identity

As discussed previously, each IMO meta-object is a logical representation of an application's object. As an Object Identifier (OID) identifies an object in the database, a Logical Object Identifier (LOID) identifies an IMO meta-object. That is to say, a LOID is the identity of an application's object that is supported in the database by one or more object (*facets* Clamen [1992]). In turn, the OID physically identifies an object or meta-object. This approach simplifies the object update process because it avoids the physical replacement of relationships. Furthermore, it also allows an application data entity to have objects in distinct versions.

In the example depicted in Figure 3.3, the course's logical object, as well as the students in each cell of the array, are identified using their LOID identifiers. Data objects and meta-objects are identified using physical OID identifiers.

In regards to the database/framework layer, an object array is treated as a collection of individual object references as in the running environment, according to the same programming language variance rules. Thus, an AMO meta-object, which points to an array, can refer to individual objects in different states of the class version. Furthermore, these objects may be referred to by other arrays, AMO meta-objects, or even RMO meta-objects. In Figure 3.3, one can observe each cell of the array with distinct LOID identifiers, class versions and OID identifiers.

An internal framework data structure provides the mapping between LOID identifiers and references to objects in memory. Thus, two objects with the same LOID inside the running environment have a common reference in the memory. That is to say, they are the same object instance.

3.2.2 Addressing the meta-model's complexity problem

Two types of changes can be applied to a class: at class level, by adding, removing or modify properties (attributes and methods), or, by changing its inheritance hierarchy. Each time any of these changes occurs, a new class version is created. If one superclass is updated, all its subclasses also evolve and a new version is required for each. The proposed schema evolution model is based on class versioning. In this kind of schema evolution approach, depending on the dynamism in the programmers' development work, the version derivation path can increase exponentially in terms of complexity. The path of class version derivation, combined with class inheritance hierarchy, results in an unsupported complexity, compromising system performance and maintenance. These two categories of relationships are illustrated in Figure 3.3, respectively: (1) Among the versions of class `Course`; and (2) between `Person` and `Student` classes.

Regarding the path of class version derivation, this metadata is essencial for the object conversion process among the existing class versions. In the proposed approach, the path of class version derivation is supported through two types of class version relationships: (1) implicit and (2) explicit. These implicit and explicit relationships are based on direct and user-defined mappings, respectively. In Sections 3.2.6.1 and 3.2.6.2, these two types of relationships will be discussed

in detail. The main argument for this strategy is based on the observation that many of the updates in class structures can be inferred autonomously by the system using direct mappings. This results in an implicit relationship between those two versions that do not require an explicit definition given by the user. On the other hand, the explicit ones are required only when structural and semantic updates occur in the classes' structures. This approach minimizes the required metadata in order to represent those paths of class version derivation, consequently, reducing the meta-model's complexity, as well as the programmers' manual intervention. Additionally, our pointcut/advice constructs also reduce programmers' effort. As we will discuss in Section 3.2.6.3, these constructs' approach provide means for a single construct to handle several class versions. In our proof of concept application, presented in the Chapter 5, an example is discussed.

Each class version has its own meta-object (a CVMO instance) in the database, which contains all its metadata. By means of these meta-objects, the upper class inheritance hierarchy can be obtained by interrogating the class version regarding its superclass. On the other hand, by querying the database for CVMO meta-objects, all subclasses of a specific class version can be easily obtained. Thus, inheritance hierarchy information is available for the framework and the applications.

When compared with other meta-models [Rashid & Sawyer, 1999], the proposed approach considerably reduces the complexity of the path of class version derivation, since the registration of many of these paths is not required. This approach deals efficiently with the relationship complexity problem since the same number of equivalent CVMO meta-objects is maintained, while the need for relationship meta-objects for the class inheritance hierarchy representation is totally avoided. Due to direct mapping, the registration of the path of class version derivation, using UBMO meta-objects is also, in many cases, unnecessary. Figure 3.3 illustrates these advantages in terms of the absence of an UBMO meta-object between `Course$B` and `Course$C` and also between `Student$A` and `Student$B`.

As already demonstrated by Monk & Sommerville [1992], the update/backdate methods approach, applied in our meta-model and framework, also provide means for semantic evolution consistency. Our approach deals with this type of consis-

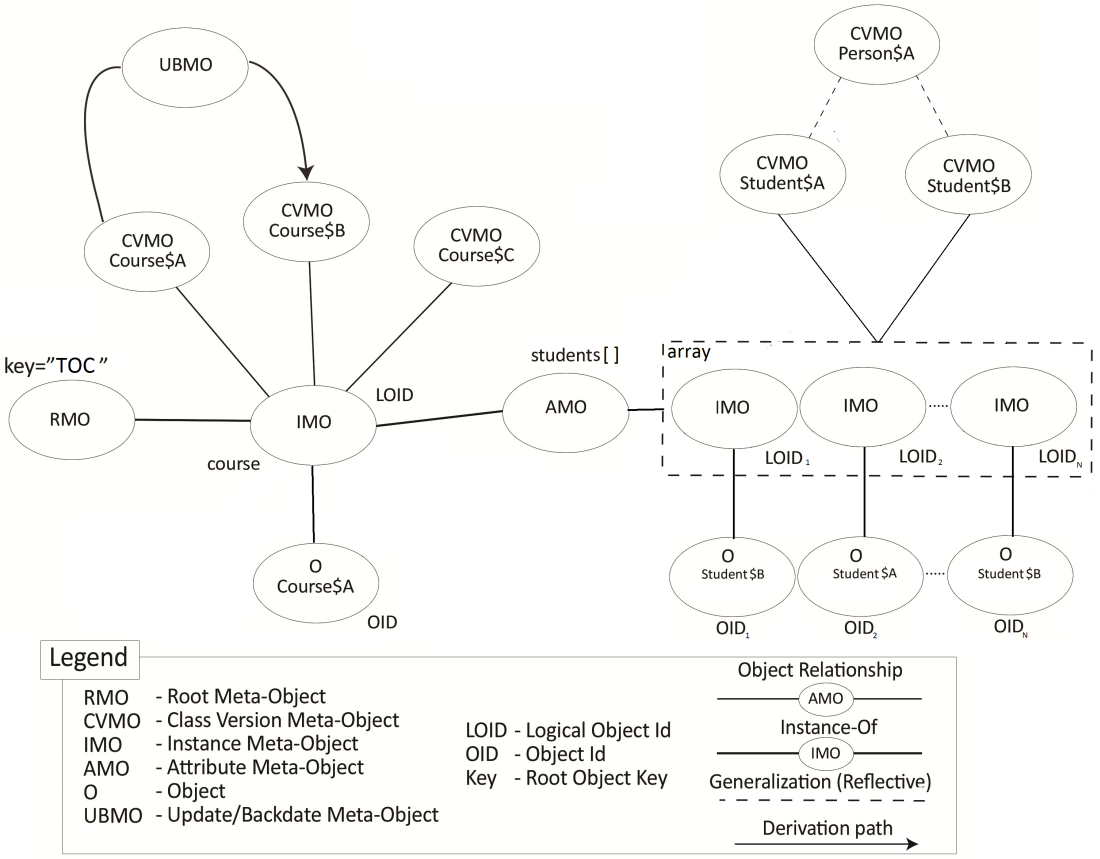


Figure 3.3: System meta-model

tency and others (see 3.2.7), ensuring instance adaptation among class versions.

3.2.3 Class versioning algorithm

In our approach, the schema information is only available in the application's source code in its class structure definitions. Moreover, the applications just know their own class structure ignoring all the others in distinct class versions. Thus, in the database all these applications' class versions are represented through our meta-model. In order to enable the propagation of the classes' metadata to the database, we adopted an incremental schema evolution approach. That is to say, when a new class version is detected, it is automatically preserved in the database as a CVMO meta-object. Additionally, a class version identifier is automatically

assigned to this new class version. In our framework, we generate these identifiers using hash functions based on the class's bytecode. Our framework also enables users to manually define these identifiers (see Section 3.2.4). All information regarding the class, including its inheritance hierarchy in its specific class version, is saved in that CVMO meta-object.

Regarding versioning, our framework treats objects in three distinct ways, depending on its category type:

- Simple object - A primitive type object, such as an `int`, `Integer` or a `String`;
- Composite object - Composite objects are composed by simple objects and other composite objects. This type of object can still be:

  Versioned - User-defined objects such as `Person` or `Invoice`;

  Non-Versioned - In special cases where versioning is not desired or is not possible, such as classes that belong to a Java environment (*e.g.* `java.util.ArrayList` or `java.util.TreeMap`).

Simple objects are primitive data types, hence they do not require versioning. Autoboxing [Gosling *et al.*, 2005] is the automatic process made by the Java compiler in order to enable bidirectional conversion between primitive types and their corresponding object wrapper classes. In our framework, this Autoboxing mechanism enables primitive types, such as `int` and `float`, to be treated as `Integer` and `Float`, *i.e.* simple objects, respectively.

On the other hand, versioned composite objects are objects that have a complex internal structure, composed by simple objects and other composite objects (Versioned or Non-Versioned). Due to Java Virtual Machine limitations, our prototype enables some classes to be treated as non-versionable. In Sections 3.3.2.4 and 3.6.1 we will discuss this issue.

The objects may exist in distinct class versions. Such coexistence poses problems inside the run-time environment because the name of a class must be unique inside the same Java Virtual Machine (shared by application, framework and database management system). Notice that in Java, as well as in many modern object-oriented programming languages, there is no native support for class

versioning. In order to solve these problems, a class `Person` in its version `"A"` is renamed to `Person$A` avoiding conflicts with the current class version (*e.g.* version `"B"`) that may already be loaded in the Java Virtual Machine. This class renaming approach does solve the very same problem inside the database. In the current version of our framework, a object-oriented database db4o is used as repository instead of implementing the basic storage mechanisms from scratch. The repository would not allow storing two types of class with the same name identifier.

The renaming process is applied at run-time to a clone of the application object before being stored in the storage. The reason to use a copy of the object instead of the original instance is the loss of the object identity, which will be replaced by a LOID. Note that in our framework the persistence state of an object is defined by its reachability, where objects identifiers have an important role. Since a renamed copy of the actual object is used for storage concerns, when another application in a different version (*e.g.* version `"B"`) accesses that object (*e.g.* version `"A"`), it must be emulated to the class version as it expects. In this example, the instance adaptation *aspect* performs the conversion of the object stored in version `Person$A` to a copy in version `Person$B`, renames the class to `Person` (the class name in the application version) and finally provides a correct object identity. At run-time, the reference to an object instance is mapped through a LOID in the framework's cache. This cache and mapping completes the puzzle, providing the required association between the running application's objects and the persistent environment with multiple versions of classes and objects.

3.2.4 Enriching the application's class structure

The meta-objects described previously support all gathered information from the applications' classes, as well as the additional metadata provided in the source code. Our meta-model implementation recognizes new classes and versions through the generation of an identifier. These class version identifiers are generated using a hash function. As a result of the hash function, these automatically generated identifiers are not very programmer-friendly. That is why the

`@Aop4oopVersionAlias` Java annotation [Bloch, 2004; Gosling *et al.*, 2005] has been added to the framework. This annotation can be associated to a class in order to have its versions easily named.

```
01  @Aof4oopVersionAlias(alias = "A")
02  class Person
03  {
04     @Aof4oopConstraintNotNull("Invalid name")
05     private String name;
06     @Aof4oopConstraintCheck(message="Invalid Age",expression=">0")
07     private int age;
08     @Aof4oopConstraintCheck(message="Invalid Weight",expression=">0")
09     private float weight;
10  }
11  @Aof4oopVersionAlias(alias = "B")
12  class Student extends Person
13  {
14     private int studentNumber;
15     @Aof4oopConstraintNotNull(message="Invalid Type")
16     @Aof4oopDefault(value = "Ordinary")
17     private String studentType;
18  }
```

Figure 3.4: Schema enrichment through annotations

Figure 3.4 illustrates how the schema inside the application is enriched using additional metadata. The annotations in lines 01 and 11 define alias to class version identifiers, facilitating future class identification. Other annotation categories to support data integrity are also available in the framework. Programmers may impose constraints to an object's attributes following Meyer's principles of Design by Contract [Meyer, 1992]. The annotations in lines 04, 06, 08 and 15 define domain integrity policies. Currently our framework just allows constraints `@Aof4oopConstraintCheck` in numerical attributes. The annotation in line 16 belongs to a third type, which allows the initialization of attributes with constant values. In our plans for future developments this annotation will receive a function, hence providing programmers with powerful means to initialize attributes in a specific class version.

3.2.5 Supporting schema evolution

As discussed in Section 2.3.1.4, in systems based on orthogonal persistence the schema (class structure) is shared by application and database. Thus, any schema update can be easily detected and reflected into system's metadata, enabling an incremental schema evolution. Programmers just need to perform the update within application source code because the framework will produce a new schema version. In our framework, when a class update is detected, its structure and additional meta-data (annotations) are obtained using the reflective capabilities of the programming language and then preserved in the system's meta-objects layer. That is to say, for each updated class a new class version is created and then maintained in the database as a CVMO meta-object.

3.2.6 Supporting instance adaptation

The simplicity of the schema evolution, addressed with an incremental approach, contrasts with the complexity of the adaptation of existing objects in the database. This instance adaptation problem rises complex implementation issues, specially in a multi-version schema, which is our case. In order to address these problems aforementioned in Section 3.2.2, our framework performs two types of mappings: *Direct Mapping* and *User-Defined Mapping*.

3.2.6.1 Direct Mapping

Direct mapping is applied in the absence of any user-defined mappings. This mechanism supports all class updates that can be autonomously inferred by default conversion functions as referred in Section 2.3.2.5. In our framework, direct mapping deals with changing fields to compatible types, movement of fields across class inheritance structure, field removing and new fields initialized at zero or `null` or any fixed value defined through an annotation `@Aof4oopDefault`. Our prototype implements this kind of mapping as its default instance aspect discussed in Section 3.3.2.5.

3.2.6.2 User-Defined Mapping

User-defined mapping is required when direct-mapping is unable to deal with conversion. Examples of these mappings are presented in Figure 3.3. The class `Course` has three versions: `"A"`, `"B"` and `"C"`. Each one of them is represented in the database as a CVMO meta-object. Class versions `"A"` and `"B"` have an explicit mapping between them through the use of an UBMO meta-object. In version `"C"`, no mapping is explicitly defined. This means that any conversion scenario involving class version `"C"` is handled by direct-mapping.

Each UBMO meta-object contains information regarding the conversion of object instances from one version (or a set of versions) to another. This is given as a conversion function that receives the object in its version and returns it in the desired one. These meta-objects follow a pointcut/advice approach in the aspect-oriented sense. Hence, their data structure is organized in two parts: condition and action. The former defines the cutting point, through class, version and other matching parameters. These matching parameters quantify the objects which are subject of conversion. These target objects could pertain to one or more versions. The latter consists in pure programming language source code for conversion. In an aspect-oriented sense this function is an *advice*.

UBMO meta-objects are built using our pointcut/advice constructs (we will present them in Section 3.2.6.3). That is to say, programmers write these pointcut/advice constructs, which are stored as UBMO meta-objects in the metadata layer. This layer of UBMO meta-objects supports user-defined mappings among class versions.

3.2.6.3 Pointcut/advice constructs

Instance adaptation is a class cross-cutting concern. In order to quantify these join points, we developed a new kind of pointcut/advice constructs that follows the *Quantification* definition posited by Filman & Friedman [2000].

> *"AOP is thus the desire to make programming statements of the form*
> *In programs P, whenever condition C arises, perform action A."*

These expressions are structured into two parts: trigger conditions and action. In Table 3.1 we present a quick reference guide for these constructs. Notice that they were improved since our last publication [Pereira & Perez-Schofield, 2014b].

At runtime, when such conditions are satisfied, the user-defined action is triggered establishing an explicit mapping between two instances of an object in distinct class versions. The action information extends the framework's instance adaptation *aspect* with the required conversion behaviour. Each pointcut/advice construct is supported at the meta-object layer as an UBMO.

These constructs are written in XML language in order to enable easier editing through a graphical tool or, simply, a text editor. Furthermore, our approach also benefits from the XML extensibility for future implementation of new features. In our implementation, the action is written in Java programming language, which is embedded into the <`sourceCode`> XML node. We also note that our approach does not require another programming language like Vegal [Rashid & Leidenfrost, 2004]. Programmers write conversion code in the same programming language used within the application source code.

Examples of these expressions are presented in Figures 4.6, 5.7, 5.8 and 5.9. These examples will be discussed in Sections 4.1.1 and 5.4.
A complete description of these constructs is available in Appendix B - Pointcut/Advice Construct reference Guide.

Each class has its own global version identifier. Thus, in complex class structures, after several evolution steps, the number of versions and identifiers may become an unsustainable management problem for programmers. These version identifiers are especially useful in our pointcut/advice constructs, in order to identify the target objects for adaptation. In these user-definitions programmers may use wildcards, ranges of versions, as well as logical operators. By using suitable naming schemas in these identifiers much of this complexity can be overcome because one single construct could handle several conversion scenarios.

3.2.7 Structural, semantic and behavioural consistency

The proposed meta-model allows a class to have several versions. Thus, applications with a specific class structure should be able to use objects that reside in the

	Parameter/node	Description
Conditions	`name`	Mandatory parameter which enables pointcut identification. It is applied when generating the name of the woven classes. This name is very useful for debugging purposes because, in case of a runtime error, the programmer can understand its localization within the source code.
	`matchClassName`	Class canonical name of the advised classes. Emulated objects whose classes match this parameter are advised. This parameter allows the * wildcard. If the class name is followed by `[]`, the adapted object must be an array of this class.
	`matchSuperClassName`	Superclass canonical name of the advised classes. Matching through a superclass is meant to reduce the number of user-definitions. If many target classes that share a common superclass exist, all of them are advised. This parameter allows the * wildcard.
	`matchOldClassVersion`	The class version of the persistent object in the database being emulated/converted. This parameter allows the * wildcard and `or` operator.
	`matchCurrentClassVersion`	Defines the class version identifier of the running application. This parameter is required when many versions of the class already exist. In these cases, we must ensure a correct target application version. This parameter allows the * wildcard and `or` operator.
	`matchParentClassName`	This contextual parameter enables advising objects which at runtime are pointed from a certain class. This parameter allows the * wildcard.
	`matchParentMember`	This is another contextual parameter that enables advising objects which, at runtime, are pointed at from a certain object member.
Action	`applyDefault`	This parameter, when true, applies the default conversion behaviour alleviating the programmers' effort in order to write this action (the body's advice). Note that in many cases just some members of a class require a user-defined code.
	`outputClassName`	This parameter specifies the type of return value. This is very useful when we intend to apply an expression to several classes that share a common superclass. In these cases, the *advice* can convert that superclass or any other.
	`sourceCode`	Conversion code in plain programming language. Must be finished with a return statement.

Table 3.1: Pointcut/advice constructs quick reference guide

database irrespective of their class versions. These requirements raise consistency issues which are addressed in our approach.

The static type checking done by the compiler ensures that each object instance inside the application runtime environment unconditionally pertains to, and follows, the application's schema. Inside the user-defined conversion code, the referred classes are statically checked during the compiling of the weaving process, ensuring that they belong to a known class version. Thus, in both cases, anomalous behaviours are avoided at runtime because there are no invalid references to methods or attributes. Since each new class version is incrementally propagated to the database, in this layer all object instances also pertain to one of those multiple versions of the classes. Hence, the schema evolution invariants are propagated to the database.

While structural and behavioural consistency is verified at compile-time, semantic consistency is addressed with update/backdate conversion code and additional metadata added into the application's classes. Our UBMO meta-objects ensure this type of schema consistency. For example, let us consider the class `Person` in version `"A"` where attribute `weight` represents the person's weight in pounds. In version `"B"`, the same attribute represents that person's information using kilograms as the measuring unit. Although the structure is exactly the same, there is a semantic variation. In our meta-model and framework, this problem can be addressed using our pointcut/advice constructs. The listing in Figure 3.5 presents the required user-definitions to adapt objects from version `"A"` to `"B"`.

Another similar construct is required to convert objects from version `"B"` to `A"`. These user-definitions produce two UBMO meta-objects that ensure bidirectional semantic consistency between those two class versions.

Updates in the class's structure may also compromise semantic consistency. For example, let us consider the same class `Person` in version `"A"` and `"B"`. In version `"A"` the person's name is organized as two attributes: first name and surname. In version `"B"` as first name, middle name and last name. Our framework also addresses this type of inconsistency following the same approach applied in the previous example. In Sections 4.1.1 and 4.2 we will discuss this example in the context of two comparative case studies.

```
<ubmo name="ConvPerson$A_to_Person$B"
      matchClassName="rhp.aof4oop.cs.datamodel.Person"
      matchOldClassVersion="A"
      matchCurrentClassVersion="B">
  <conversion applyDefault="true"
              outputClassName="rhp.aof4oop.cs.datamodel.Person">
     <sourceCode>
         <![CDATA[
        newObj.weight=oldObj.weight * 2.20462262;
        return newObj;
          ]]>
      </sourceCode>
    </conversion>
</ubmo>
```

Figure 3.5: Pointcut/advice construct for conversion of person's weight

Several database evolution taxonomies for object-oriented databases have been proposed [Banerjee *et al.*, 1987b; Rashid & Sawyer, 2005]. The taxonomy of our framework is closely tied to programming language because programmers can freely perform updates in the class structures of their applications. Our proposal to address this problem is, theoretically, able to deal with all kinds of updates at the class level in which there is equivalence. For example, consider class `C` in versions α and β respectively as Cα and Cβ. The f function (in <`sourceCode`>) converts an object in version α to β. So that

$$f : C\alpha \rightarrow C\beta \tag{3.1}$$

Hence, consistency is always achievable at the class level, if there is an f function.

Our pointcut/advice constructs still address another scenario. Consider a class `C` in version α and class `D` in version β, respectively as Cα and Dβ. The equation for this case is

$$f : C\alpha \rightarrow D\beta \tag{3.2}$$

Thus, if there is an f function, then it is possible to convert objects in class Cα to class Dβ. This evolution scenario occurs when in class structure α there is a class `C` which is replaced by `D` in class structure β. From the point of view of

our meta-model and framework, the replacement of a class or its update creates the same problem because an f function will perform the conversion. Notice that our framework invokes an appropriate f function when applications require an object according to its class structure.

As discussed in Section 3.3.2.5, the framework's internal conversion mechanism is provided with non-local information. That is to say, besides information regarding the target object, its parent information is also available for this mechanism. Thus, Equation 3.2 can be generalized as follows:

$$f : (C\alpha, PC\alpha, PD\beta) \rightarrow D\beta \tag{3.3}$$

Where PCα and PDβ are, respectively, the parent object of `C` in version α and the parent object of `D` in version β. As an example, consider class `Person` in class structure α which has its address information in three attributes: `street`, `city` and `zipCode`. Since several persons (belonging to a family) can share the same address, it makes sense to create a class `Address` with these attributes. Thus, in the new class structure β, objects of class `Address` are obtained from

$$f : (null, \texttt{Person}\alpha, \texttt{Person}\beta) \rightarrow \texttt{Address}\beta \tag{3.4}$$

Notice that in this scenario the class `Address` does not exist in version α. Hence, it is passed on as a null value to the f function. In Section 3.3.2.6 we will present a practical example.

Note that our framework emulates objects according to the class structure of the running application. Only when objects are updated by the running application, is its structure in the database updated. Hence, objects of class `Address` do not obtain any LOID after being emulated. They are just marked as being reachable. In this state, objects are considered as being persistent but they don't have a LOID; however, they will get a LOID in a later update.

Based on the previous arguments, one can conclude that our approach to maintain consistency is based on equivalence through the use of conversion functions. Hence, if class structures (before and after the evolution) are not equivalent, then the evolution problem cannot be fully addressed within our approach. As an example, consider a class in which an attribute is added in version `"A"`. In version

`"B"` of this class, this attribute is removed and no other attribute is added or exists in the class in order to provide the same information. In this example there is equivalence from version `"A"` to `"B"`. However, no equivalence exists from version `"B"` to `"A"`. This lack of equivalence breaks version continuity. Applications based on version `"A"` of that class structure will lose information, if objects are updated according to version `"B"`. The problem in this example is well known in the literature [Clamen, 1992]. In order to avoid losing information, the information of the deleted attribute can be maintained in several ways. Maintaining *multiple facets* [Clamen, 1992] of the object is one of these solutions. However, in other complex scenarios, where no equivalence exists, version continuity is effectively broken.

3.3 Framework architecture

Our framework mediates the interaction between applications and database, within an aspect-oriented approach. This aspect-oriented architecture enables the modularization of some crosscutting concerns regarding applications and framework. The internal framework's functioning, as well as the way applications are provided with the persistence mechanisms, is aspect-oriented. Applications rely on our framework in order to be provided with transparent mechanisms of data persistence. A main module, a set of aspects, database and preprocessor compose the framework. These framework's components are discussed next:

- Preprocessor - This component is optional, allowing some dynamic typing features that are not compliant with standard Java compiler rules. It acts as a static weaver, allowing the introduction of additional code into applications that enables the framework to perform a correct persistence of parametric classes [Alagić & Royer, 2008].

- Main module - This core component of our framework implements its API, based on the `CPersistentRoot` class, the meta-model and all basic services such as: persistence management, data integrity, caching, class loading, class versioning calculus, database garbage collecting and all default mechanisms of database evolution.

- Aspects - These aspects are responsible for the real implementation of those services provided to applications by the framework's main module. Notice that the main module only implements some default mechanisms and basic services. For example, persistence, and all issues regarding it, are effectively provided for by the database. Instance adaptation is also implemented as an aspect; the main module only implements basic mechanisms to support adaptation.

- Database - Repository of objects and meta-objects.

Figure 3.6 illustrates how those components enumerated above are related in a layered architecture comprised of three levels: Application, framework and database.

The initial version of our prototype [Pereira & Perez-Schofield, 2010, 2011] only supports persistence services within a unique schema. Later [Pereira & Perez-Schofield, 2012, 2014a,b], the persistence manager, schema manager, class-loader, database weaver and garbage collector were introduced into the system as result of the adopted meta-model, which has enabled the support for database evolution mechanisms as well as transactions. From the applications' point of view, memory heap is seamless extended to the database. Thus, a database garbage collector is needed in order to eliminate objects that lose their direct or indirect reachability from any persistent root object.

The framework distinguishes two categories of classes: the first category are classes that pertain to running schema of the application; a second category of classes are the ones that pertain to other versions of the schema, past or future. The former are initially resident in the application classpath, (*e.g.* the class `Person`), while the classes belonging to the second category, (*e.g.* `Person$A`), are stored into the database as specialized CVMO meta-objects. For this second category of objects a special classloader was developed. It loads any class in any of its known versions at run-time. Notice that these classes stored in the database were renamed in order to enable their coexistence in multiple versions (as discussed in Section 3.2.3).

In our prototype, application, framework and database management system share a common Java virtual machine. This seamless environment has facilitated

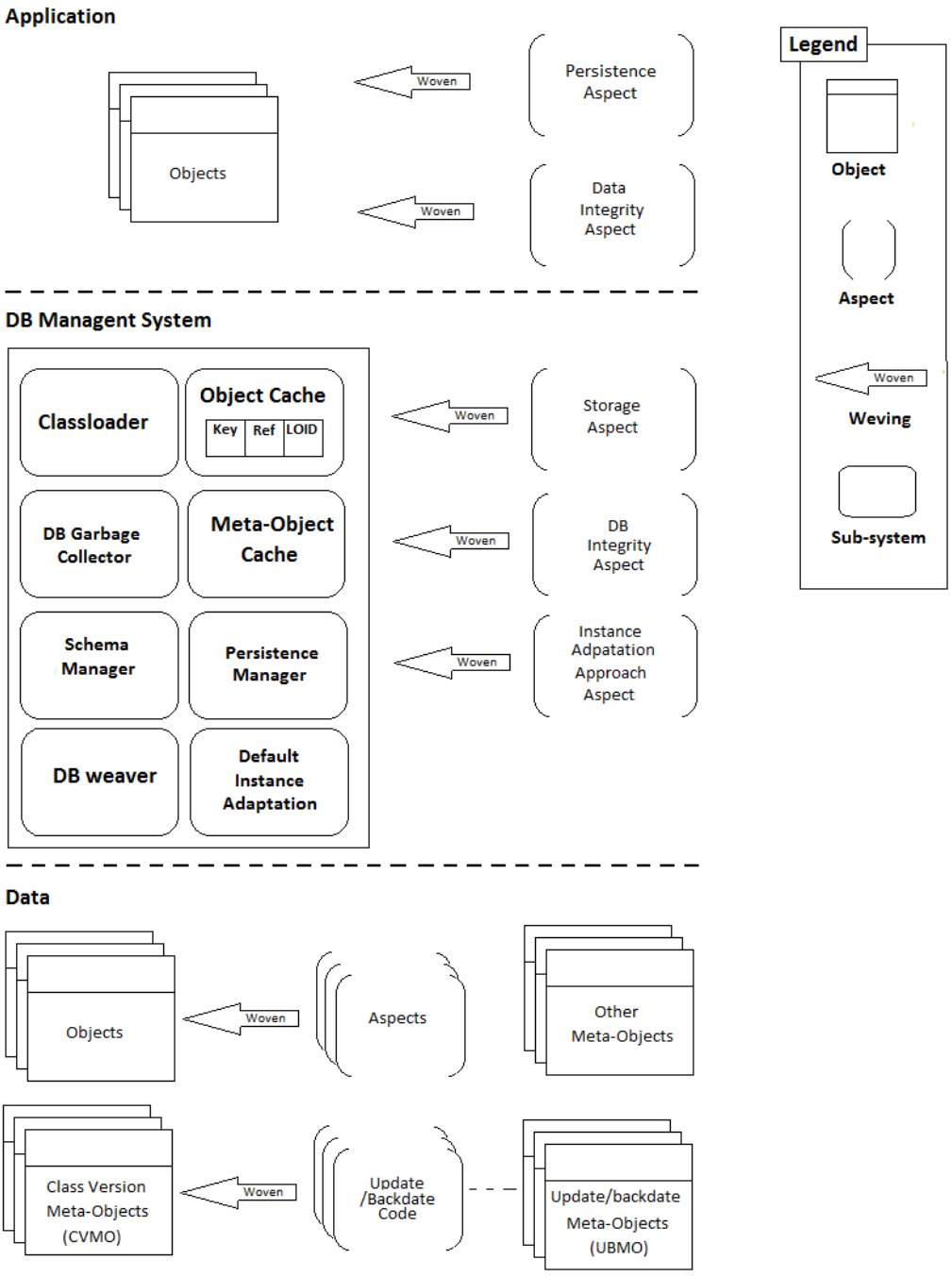


Figure 3.6: System's architecture

the construction of the presented *aspects* in order to intercept transversal concerns to the entire system.

The meta-objects and object storaging is handled by `AObjectStorageDB4O` *aspect* enabling the framework's obliviousness regarding persistence.

In the following sections we will discuss these components in some detail.

3.3.1 Database

The system's database comprises two distinct areas: an object-oriented database and an XML file. This system's component can be observed in Figure 3.17. The object-oriented database feeds the storaging aspect with objects and meta-objects (excluding the UBMO ones). The XML file feeds the framework's weaver with UBMO meta-objects. This implementation strategy has enabled a rapid development of the framework. Objects and meta-objects are easily stored in an object-oriented database like db4o. On the other hand, for our UBMO meta-objects, due to the special editing and extensibility requirements of our pointcut/advice constructs, an XML format was considered very adequate.

3.3.2 Framework's main module

The framework's internal architecture is presented in Figure 3.6. Next, we will discuss the internal details of the framework's main module.

3.3.2.1 Application Programming Interface (API)

The application programming interface consists of a class `CPersistentRoot`, which provides references for any persistent object resident in the database, as well as querying facilities. As presented in Figure 3.1, the method `getRootObject()`, returns run-time references to persistent objects which are associated to an arbitrary key string. This method, when invoked without a key, returns the database's default root. Each one of these roots points to an entry to the persistent object graph. Figure 3.7 presents three of these root objects: the default, a named reference "student1" to a `Student` object and another named reference called "courses" that points to an array of `Course` objects. One can obtain any of these

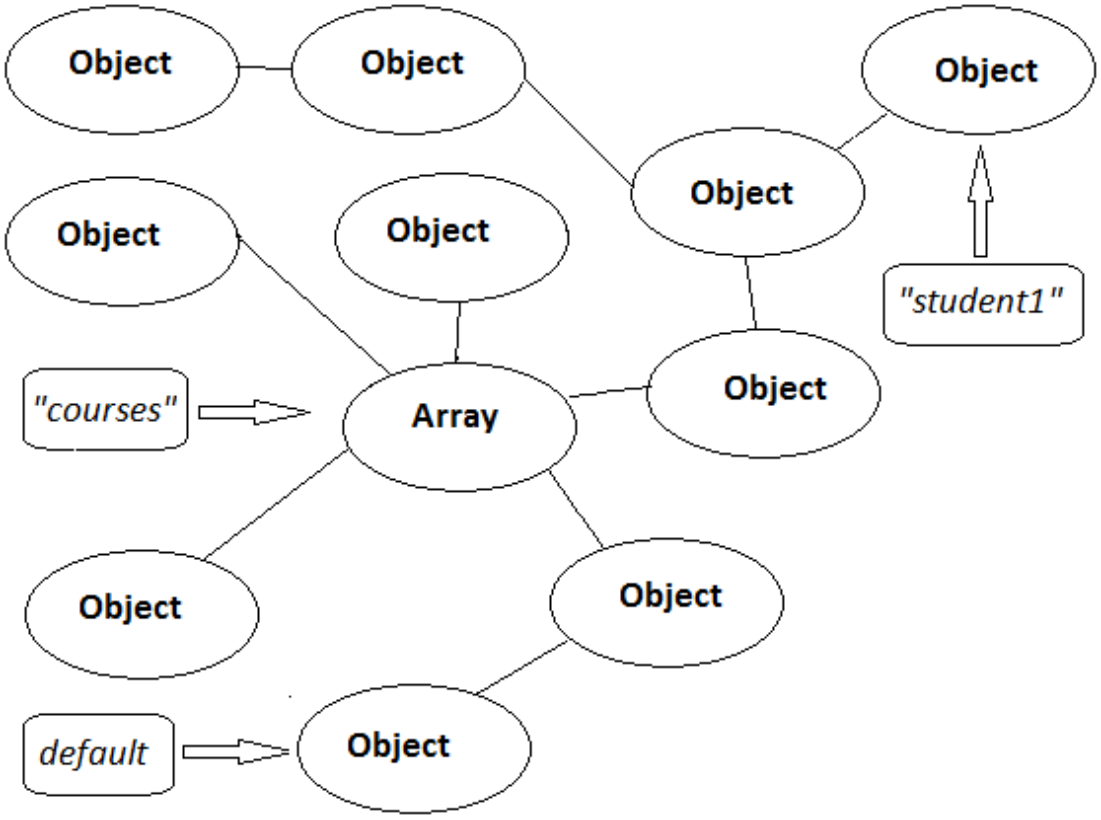


Figure 3.7: Named and default root objects

references simply by invoking the `getRootObject()`, without arguments to the default root or supplying the name of the intended reference.

New named references can be added by invoking the `setRootObject(String name,Object object)` method. If the object is already persistent, only a new name reference is added. Otherwise, an image of that object is saved into the database with the name associated to it.

These roots pointed by named references can be objects or arrays. As to parameterized classes [Gosling *et al.*, 2005], these are also handled without loss of information as will be discussed in Section 3.3.6.1.

The framework's programming interface does provide querying mechanisms, which provide a set of results as object collections. Additionally, these queries can be saved with an associated name making them dynamic views to those sets of results. Currently, the framework's query engine implements these queries

following the Cook & Rai [2005]; Cook & Rosenberger [2005] paradigm through the `CQuery` class, which implements the `IQuery` interface.

Like the objects accessible through named references, these objects also enable access to other objects reachable from them. That is to say, each object pertaining to these collections are root objects as well.

The list in Figure 3.8 exemplifies how these queries can be used in an application. In this example, all persistent objects of `Family` class are retrieved from the database. The query constructor receives one argument, the query's predicate. Currently, only this type of predicate is available in our framework, which intends to be a proof of concept. More sophisticated predicates shall be subject of future research and development.

```
CPersistentRoot psRoot=new CPersistentRoot();
List<Family> families=psRoot.query(new CQuery(Family.class));
for(Family f:families)
{
   System.out.println("Family: "+f.toString());
}
```

Figure 3.8: Query mechanism

Figure 3.9 depicts an example of how a query can be saved as a view to be used later. A view creation requires a unique name. If that name is already used by another view, the framework throws a system exception.

```
CPersistentRoot psRoot=new CPersistentRoot();

psRoot.createView("allPersons",new CQuery(Person.class));
System.out.println("A new view was created");

List<Person> persons=psRoot.view("allPersons");
for(Person p:persons)
{
   System.out.println("Person: "+p.toString());
}
```

Figure 3.9: View mechanism

The framework's API also provides the means to drop views. The code list in Figure 3.10 erases the view created in the previous example. The `dropView()`

system call returns a boolean that acknowledges if it was really dropped. If the view does not exist, a false value is returned.

```
CPersistentRoot psRoot=new CPersistentRoot();
boolean r=psRoot.dropView("allPersons");
System.out.println("The view was dropped:"+r);
```

Figure 3.10: Droping Views

In Appendix C - API Reference Guide a complete reference guide of the framework's API is presented.

3.3.2.2 Caching

The cache subsystem plays an important role in the framework's architecture. Besides performance goals, at run-time, it maintains tracking information regarding persistent objects loaded in memory. It provides important information that allows the persistence *aspect* to distinguish persistent objects from the non-persistent ones. In fact, this cache is a set of specialized and distinct caches of objects and meta-objects.

As to the object cache, each entry maintains the mapping between a memory reference and its LOID identifier in the database. The loaded objects are not inside the cache. Only their references as a `SoftValue` object are. This object cache extends a `SoftReference<Object>`, which is based on soft references [Gosling *et al.*, 2005]. This approach allows its cache entry to be discharged by garbage collector before memory exhaustion occurs, if all strong references to an object are lost. These cache entries pertain to an inner class of `CCache` class, which maps the cached soft references with a key. This key is an internal hash-based data structure of the `CCache` class. Figure 3.11 depicts the code listing that implements the framework's `SoftReference` inner class.

Inside the `CCache` class, a hash map based on the Java data structure `(HashMap <String, SoftValue>)` holds the cache entries.

By using the mapping between memory references and LOID, this cache provides the means to object identity, since it maintains the relationship between each application's object and its logical identifier (LOID) in the database at run-time. When an object is required by the application, if it is already loaded, the

```
private static class SoftValue extends SoftReference<Object>
{
    private final String key;
    private final long LOID;

    // The remaining code was omitted due to space restrictions.
    //...
}
```

Figure 3.11: Soft references to cache entries

framework provides the application with the reference to that instance by sharing the object's identity. Thus, two objects with the same LOID inside the running environment have a common reference in the memory. That is to say, they are the same object instance.

For meta-objects, cache has performance goals only. The system's *aspects* systematically require information about these meta-objects. The framework built-in meta-classes are well know, thus, this cache is organized in several specialized caches, each one for a specific meta-class. This approach reduces the searching overhead without sacrificing any other issue.

Cache implementation also provides statistical indicators, which help to understand the system's behaviour. The `objectCacheHits` and `objectCacheAccess` give, respectively, the number of hits when trying to get an object from cache and the global number of accesses. The `objectCacheHitsPart` and `objectCacheAccessPart` indicators provide the same information since the last count reset.

3.3.2.3 Garbage collector

Framework and database act as an extension of the application itself in a seamless environment. The memory heap is extended to the database and data objects cached into the cache subsystem. Thus, the Java Virtual Machine garbage collector was extended to the scope of the database in order of free unreferenced persistent objects. In truth, they are two independent garbage collectors: one operating in the memory heap, provided natively by the Java Virtual Machine, and another implemented in the framework.

This framework's garbage collecting mechanism can be explicitly invoked

through the framework's API using the `gc()` system call. Currently, it is automatically called only when the database connection is closed. This option lies on the system's high load, as well as the required time to perform this task.

Garbage collecting is a vast field of investigation that raises many performances issues. This problem is not addressed in this PhD Thesis, thus, a simple algorithm was implemented in order to free unused objects references on all existing class versions and related meta-objects.

3.3.2.4 Schema manager

The schema evolution approach was already discussed in Section 3.2.5. In this section only architectural issues are discussed. This adopted approach is very specific and dependent on the orthogonal persistence paradigm, as well as our meta-model. Thus, the schema manager development was oriented to a library that provides the framework with a complete set of schema evolution mechanisms, directly implemented into the framework as a subsystem.

The framework's schema manager deals with the evolution that occurs in the applications' classes (schema), reflecting its changes into the system's metadata at the database. Mainly, this system's component implements all mechanisms related with new class and version recognition, as well as the management of related meta-objects in the database.

Our schema manager uses reflection (see section 3.7.2) and instrumentation (see section 3.7.3) intensively. Respectively, these facilities are provided through the native Java reflection API and the Javassist (Java Programming Assistant) library [Chiba, 1998]. Reflection enables the introspection of the internal objects' structure in order for all metadata to be mapped into respective CVMO meta-object. On the other hand, instrumentation enables the schema manager to manipulate the bytecode of each application class version in order to produce its corresponding persistent form (*e.g.* `Person` and `Person$B`). This operation is required each time that a new class or version appears in the framework. If the schema manager detects a new class version at run-time, it obtains its version identifier, renames it, stores it in the database and delivers it to the system's classloader. Thus, from that moment on, the class is available in two places at

run-time: in the application (*e.g.* `Person`) and in the framework (*e.g.* `Person$B`).

The central element of our schema manager is a built-in meta-class called `CClassVersionMetaObject`. These meta-class instances are CVMO meta-objects that support the metadata of each existing application class version. For each class version, (*e.g.* `Person$B`), the respective bytecode is preserved, enabling reflection on it in order to obtain all internal metadata and their hierarchical lattice of superclasses.

Additionally, the meta-class `CClassVersionMetaObject`, presented in Figure 3.12, also implements functions for version calculation as static methods.

```
public class CClassVersionMetaObject
{
   private String classCanonicalName;
   private String classVersion;
   private byte[] classByteCode;

   public CClassVersionMetaObject(String classCanonicalName,
                                  String classVersion,
                                  byte[] classByteCode)
   {
      super();
      this.classCanonicalName = classCanonicalName;
      this.classVersion = classVersion;
      this.classByteCode = classByteCode;
   }
   // The remaining code was omitted due to space restrictions.
   //...
}
```

Figure 3.12: CVMO Meta-class

The class version calculus is based on a message digest algorithm MD5, which gives it a unique identifier. Since this identifier is not human friendly, optionally, programmers can introduce an annotation in the class's source code superimposing this automatic identifier. The `calcVersion(CtClass clazz)` static method performs this calculation. However, if an annotation is found in a class, its version calculus is shortened. An annotation value is then returned as can be observed in the code listing of this static method in Figure 3.13.

```
public static String calcVersion(CtClass clazz)
throws ClassNotFoundException,
       NotFoundException,
       IOException,
       CannotCompileException
{
    byte [] byte_code;

    //Check for a version alias
    Aof4oopVersionAlias annot = (Aof4oopVersionAlias)clazz.
    getAnnotation(Aof4oopVersionAlias.class);
    if(annot!=null)
    {
        return annot.alias();
    }
    byte_code=__getByteCode(clazz,0);
    try
    {
        return md5(byte_code);
    }
    catch(NoSuchAlgorithmException e)
    {
        throw new NotFoundException("MD5 algotithm not found");// :-((
    }
}
```

Figure 3.13: Class version calculus

3.3.2.5 Instance Adaptation

Contrasting with schema manager implementation, the adopted approach for the instance adaptation problem is just one of the many possible. Besides, the instance adaptation behaviour is not static. Application and framework behaviour must be able to be dynamically extended due to the semantics of evolution. Our framework extends its default instance adaptation behaviour using the metadata provided by UBMO meta-objects. We implemented instance adaptation as an *aspect* enabling a flexible and customizable implementation of the instance adaptation strategy. Based on the observation that a default behaviour, enhanced by the enrichment of the application's class structure (schema version), is capable of providing a useful yet powerful default conversion mechanism, such mechanisms

were implemented as a base module. Thus, this subsystem is composed by default conversion mechanisms that can be dynamically extended, depending on user-defined conversion functions (user-defined pointcut/advice constructs).

The instance adaptation is implemented at two *aspects* levels: (1) An AspectJ base *aspect* and (2) another, which is implemented in the framework extending the base behaviour. The former supports the framework crosscutting instance adaptation concern, allowing data object adaptation. Using an AspectJ *around advice* the base module is called, in a *proceed* statement, in absence of user-definitions.

The second level is implemented by the framework's weaver, which injects the user-defined conversion code at run-time. If these user definitions are provided, new behaviour can replace or be added to the default one. Our pointcut/advice constructs, discussed in Section 3.2.6.3, provide data regarding the target classes, the conversion code, as well as the `applyDefault` parameter that defines if it is addable or not. In Section 3.3.2.6 we will discuss that weaver.

The default conversion behaviour is implemented in the `directMaping` operation presented in Figure 3.14. This operation performs direct mapping between two class versions, as discussed in Section 2.3.2.5 and Section 3.2.6.1. In Figure 3.14 we present the Java's prototype for this operation, which receives as arguments a reference to the instance to be converted, as well as the destination class name (class and version). The output is a reference to a new instance pertaining to the destination class version (`dstClassNameVersion`).

```
public Object directMapping(Object srcObject,String dstClassNameVersion)
throws IllegalArgumentException,
       IllegalAccessException,
       InstantiationException,
       ClassNotFoundException
```

Figure 3.14: Internal system call for direct mapping

Two internal framework operations, `ReaderAdapter` and `WriterAdapter`, use this mapping operation. These read/write operations are called by the persistence *aspect* every time an object is loaded or saved, respectively. These operations have two associated *around advices* implemented inside the instance adaptation

aspect `AObjectAdaptation`, which actually performs the object conversion. This instance adaptation *aspect* is discussed in section 3.3.5.2.

3.3.2.6 Database Weaver

The database weaver plays a key role in the instance adaptation process. This system's component weaves user-defined conversion code and adds it into the framework, customizing the framework according to the user's requirements. The user-defined conversion code is stored in the database as UBMO meta-objects and expressed through our pointcut/advice constructs.

If a UBMO matches the conversion scenario at time-conversion, this weaver generates the source code of a conversion class that implements a static method named `doConversion`. Figure 3.15 depicts an example of a pointcut/advice construct that converts the `Person` class in version "`A`" to version "`B`". The framework's weaver uses the action part of that construct in order to produce the class presented in Figure 3.16.

After being woven, the `doConversion()` system call is then invoked, replacing (or adding) the default instance adaptation *aspect*. This weaving mechanism takes place the first time and each time that the UBMO-meta object is updated.

The `doConversion()` system call receives two pairs of arguments. Their names are reserved words, which are references to the target object in two distinct versions and their parents in the context of conversion. The `oldObj` provides data regarding the object, as loaded from the database. The `newObj` is the application's object being converted. It can be provided as a pre-initialized new object or as an object partly converted by the default instance adaptation module (see `applyDefault` option in Table 3.1). Both parent objects provide non-local information regarding the join point (conversion context that is taking place). These two parent references improve the richness of the user-defined conversion functions. The `oldObjParent` is the object (as loaded from the database) that contains the reference to `oldObj`, *i.e.* its parent object. Notice that, `oldObj` may have parent objects of several classes and versions. Thus, the `matchParentClassName` ensures the right class of this parent object (see Table 3.1). In our example, it belongs to the `Family` class. Similarly, `newObjParent` is the parent object, in the

```
<ubmo name="ConvPersonA_to_PersonB"
      matchClassName="rhp.aof4oop.cs.datamodel.Person"
      matchOldClassVersion="A"
      matchCurrentClassVersion="B"
  matchParentClassName="rhp.aof4oop.cs.datamodel.Family">
  <conversion applyDefault="true"
              outputClassName="rhp.aof4oop.cs.datamodel.Person">
     <sourceCode>
         <![CDATA[
     String tmp=oldObj.getName().substring(0,oldObj.getName()
               .lastIndexOf(" "+newObjParent.getFamilyName()));
     newObj.setFirstName(tmp);
     newObj.setLastName(newObjParent.getFamilyName());
     return newObj;
          ]]>
      </sourceCode>
    </conversion>
</ubmo>
```

Figure 3.15: Pointcut/advice construct for adapting person's last name

scope of the application, that contains the reference to `newObj`.

In our example, in version `"A"` of class `Person`, the person's name is a single property. However, in the current version of the application (lets consider it as being `"B"`) it is split in two parts: first and last name. The `doConversion()` function receives a `Person` object pre-converted through its `newObj` argument. Thus, only the `firstName` and `lastName` members need to be converted, alleviating the programmer's effort. Notice that in this example the remaining attributes are handled by direct mapping (because `applyDefault` is set to true). That is to say, the adaptation process is additive: first the default and then the user-defined one. In the conversion context, a `Family` instance points to the `Person` instance, the one that is being converted. The conversion code takes advantage of the `Family` object, which contains the family name, in order to split the person's name into those two parts.

This example highlights the expressiveness of the conversion code that is enabled by our pointcut/advice constructs: (1) full programming language features, (2) access to non-local data and (3) the application's class behaviour can be reused in the conversion functions.

```
package rhp.aof4oop.framework.tmp.dynweaver;
class ConvPersonA_to_PersonB
{
     Person doConversion(Person$A oldObj, Family$A oldObjParent,
                         Person newObj, Family newObjParent)
     {
         String tmp=oldObj.getName().substring(0,oldObj.getName()
                   .lastIndexOf(" "+newObjParent.getFamilyName()));
         newObj.setFirstName(tmp);
         newObj.setLastName(newObjParent.getFamilyName());
         return newObj;
     }
 }
```

Figure 3.16: Example of a woven `doConversion()` function

3.3.2.7 Classloader

In the application's scope only the current class versions are known. That is to say, only the classes declared in the application's source code and other available in within the Java Virtual Machine and additional libraries are known. Our classloader extends the system's default engine by another that has the ability of load from database any class in any version, past or future. Notice that classes in other versions were previously stored in the database's meta-objects layer using the framework's schema manager. The woven conversion classes (discussed in Section 3.3.2.6) are also loaded at run-time by this classloader.

3.3.3 Aspects

The system's aspects were modularized at three distinct levels: application, framework and database. Therefore, concerns of an *aspect* were modularized at another level.

The system's aspects are presented in Figure 3.17. This figure illustrates a program execution that is intercepted in joint points that require persistence behaviour. In this figure one can observe these joint points (`get(*/*) and set(*.*)`), in the program's execution, which are intercepted by the persistence *aspect*. These *aspects* are attached to the applications using AspectJ.

In regards to the framework, its internal operation is aspect-oriented. In

Figure 3.17 one can observe joint points inside the persistence aspect being intercepted by other aspects (such as integrity, storing and adaptation). These aspects are attached to the framework using AspectJ.

We discuss these aspects of the applications and framework next.

3.3.4 Application's Aspects

In the application's scope, two crosscutting concerns were modularized as two distinct AOP *aspects*: (1) data persistence and (2) data integrity. The former regards all join points where persistent data is read or write. Each accessed object is *advised* according to its state of persistence. The latter enables all integrity checks which are defined through Java annotations as discussed in Section 3.2.4. These two application *aspects* use services implemented at the framework level.

3.3.4.1 Data persistence

The object persistence *aspect* provides applications with orthogonal persistence. This *aspect* consumes the framework's persistence services being woven into applications using AspectJ [Kiczales *et al.*, 2001a]. Basically, this *aspect* intercepts all read and write operations in each object member, rendering the required persistence code at each of these joinpoints . Figure 3.17 depicts how the application and the framework are attached through `get(*/*) and set(*.*)` pointcuts.

Additionally, loaded objects are put in the object cache and mapped through their LOID and runtime references. Thus, in the next object access they are obtained from the cache using the mapping information. If an object is updated, its changes are reflected in the database. During this process, the persistence aspect manages all involved meta-objects. Each object update is handled atomically in a single database transaction. Thus, the database's consistency at the meta-object and data layers is ensured. We acknowledge the ability of this transactional mechanisms to operate over persistent and non-persistent data. Notice that errors also may occur in non-persistent objects due to data integrity violations. In Section 3.5 we will discuss the framework's transactional mechanisms.

3.3.4.2 Data integrity

We implemented data integrity as a concern of persistence at the application level. This data integrity *aspect* intercepts the persistence code (inside the persistence *aspect* as presented in Figure 3.17) that handles the updates of the objects.

This *aspect* acts the same as the database integrity *aspect*, but at the application level and limited to the application's schema version. The interest of this *aspect* is for non-persistent objects, which could also benefit from an integrity checker since persistent object constraints are checked by database integrity *aspect*. Note that our approach follows Atkinson's principles. Thus, all objects (persistent or non-persistent) must follow the same integrity constraints.

3.3.5 Framework's Aspects

Several elements that compose the framework's main module were discussed in the previous sections. This core module implements all components that do not require a flexible implementation or that are not transversal.

In our framework we modularized five concerns as aspect-oriented programming *aspects*. These *aspects*, and the ones within the scope of the application, form two distinct and cooperative layers of *aspects*, which enhance the global system modularity enabling flexible approaches in each scope. Thus, we can consider that the application persistence *aspect* itself is modularized through those framework's aspects. Figure 3.17 depicts this cooperation between aspects.

3.3.5.1 Storage Aspect

The storing *aspect* makes the physical data storing process approach highly customizable. In the current framework's implementation, a db4o database was used as object and meta-object repository. However, this *aspect* can be easily replaced by another which uses a different database, or even be replaced by implementing all basic persistence mechanisms from scratch.

This *aspect* encompasses all physical access to the data and metadata. In fact, it provides persistence mechanisms as an aspect to the application persistent *aspect* and the entire framework. Besides, this storing *aspect* also manages all

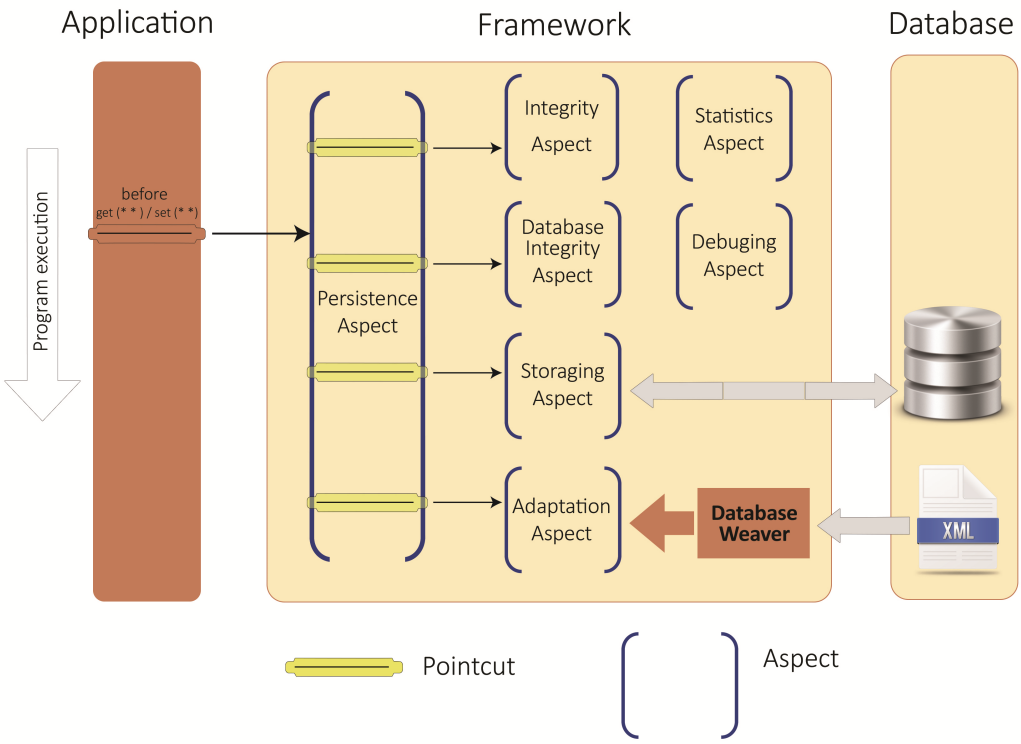


Figure 3.17: System's aspects

interactions with the system's cache, enabling abstraction from the persistence concern.

3.3.5.2 Instance Adaptation Aspect

This *aspect* implements the chosen instance adaptation approach, thus being the glue of several modules. In Section 3.3.2.5 and Section 3.3.2.6, this *aspect* was partially discussed in the context of the framework's weaver, which implements this framework's *aspect*. At the framework level, it intercepts all object read/write database operations by introducing the required code that implements the conversion functions for instance adaptation. In the absence of user definitions, this *aspect* executes the default conversion mechanism provided by the framework's instance adaptation base module. On the other hand, if any UBMO meta-object matches the conversion scenario, an user-defined conversion function is attached to this *aspect* using the framework's weaver. The run-time database weaver uses UBMO meta-object information in order to render the binary conversion code. After the woven process, the framework's classloader loads the class that extends the default conversion behaviour. Note that the run-time weaving procedure is carried out only once the first time, or when the UBMO meta-object is updated.

3.3.5.3 Database Integrity Aspect

The database integrity *aspect* is responsible for data integrity within the scope of the database. Since integrity constraints can be different in each class version, this *aspect* ensures the integrity rules defined in all class versions.

This *aspect* intercepts the system calls that apply data updates in order to ensure the constraints. It cooperates with the one within the scope of the application, since these updated objects have their constraints checked. Thus, when an object instance is updated, two distinct approaches can be adopted: (1) Optimistic and (2) Pessimistic. The optimistic approach simply does nothing when an object is updated. That is, it supposes that the update does not violate class integrity rules in all other existing class versions. Just when the object is emulated to a specific version, is it then checked by this integrity *aspect*. Thus, if a violation occurs at run-time, the framework will throw an exception.

On the other hand, the pessimistic approach performs an individual check to all data integrity rules in the known class versions. This approach can anticipate future run-time errors by informing the user of that violation at insert/update time, through the application's user interface. However, this approach introduces considerable performance degradation since it performs several checks, one for each known class version. Our implementation follows the optimistic approach.

In a multi-version schema environment, as the one being discussed, the database integrity *aspect* is also concerned with the schema evolution problem. Some schema updates could be contradictory with past or future versions. Supposing that an older schema version does not allow null values in a certain object member except in the current version, unexpected problems could happen. Thus, programmers must be aware of these contradicting schema interests.

3.3.5.4 System statistics and Debugging

The framework statistic *aspect* generates all internal statistics. Using the framework's API, users can get time consumptions (cache access, database access and instance adaptation), number of accesses to cache, as well as number of object and meta-objects loaded.

Another *aspect* generates debugging data for development and tuning purposes.

3.3.6 Preprocessor

A preprocessor was introduced into the framework's architecture in order to support some dynamic typing features in the applications' source code, which are not allowed by the normal static checking rules of the Java compiler. Additionally, this preprocessor also provides means for preserving data pertaining to parametric classes.

In Section 2.1.2, the Java generic approach and its limitations were discussed. Next, we will discussed how, using this preprocessor, the framework deals with parametric classes and how Java platform genericity provides the means to enhance the transparency of the framework's mechanisms.

3.3.6.1 Framework transparency and data abstraction

Object references obtained through a system's persistent roots can be of any type. Thus, the method `getRootObject(String rootName)` must return an object from the class `Object`. When that return value is assigned to a `Student`, a `Course` or any other type of reference, the static type checking rules of the Java compiler requires a cast from the `Object` for the actual type of persistent object obtained from the database.

The framework's API, provided through class `CPersistentRoot`, provides the method `getRootObject(String rootName)` that returns a generic data type reference. That enables the compiler to decide, in each case, what kind of class the method effectively returns at run-time by doing type inference [Milner, 1978]. The underlying process of calculating persistence closures continues to be the same. This is achieved through the Java platform support for generics. The following listing presents the signature of this framework's system call.

```
public <T extends Object> T getRootObject(String rootName) {...}
```

This type of return value is of great importance since it enhances the level of data type abstraction and framework's mechanisms transparency, making any cast of data type unnecessary. That transparency is evident in the two following lines of code. In this example, the same API's method returns two distinct classes of objects.

```
...
Student student=psRoot.getRootObject("rui");
...
Course course=psRoot.getRootObject("TOC");
...
```

Type inference is achieved given that the generic type is replaced by the correspondent type at compile-time. An implicit cast is actually taking place through type inference. The generic return type is converted to a specific type of object reference. This happens in the example with the `Student` and `Course` references. This replacing process is the same as the one discussed in Section 2.1.2.1 with

the parametric classes. The method return type is erased and replaced by a raw type, in this case an `Object` class. In the framework, that generic type is explicitly made by declaring `T` as a subtype of `Object`. At the end, the Java compiler inference process does an automatic cast to the corresponding type of variable.

Our application programming interface provides the framework with a considerable level of transparency and orthogonal data type treatment, by freeing the programmer of doing casts and systematically data type tests, improving data abstraction. Furthermore, with an obtained object reference, programmers can call all its methods or access all its properties, as presented in the following example:

```
student=psRoot.getRootObject("rui");
System.out.println(Street:+student.getAddress().getStreet());
```

The object is pointed from a reference of the appropriate type (`Student`). Thus, the entire class structure is available to the compiler or the IDE allowing, for instance, auto-completion facilities.

A common procedure for any programmer is to avoid splitting this code into two lines, writing just one, as follows:

```
System.out.println("Street:"
    +psRoot.getRootObject("rui").getAddress().getStreet());
```

Unfortunately, in this case everything changes making type inference impossible. For a better understanding of what follows, this case will be identified as Case A. As already discussed previously, the compiler infers the obtained reference type through the program variable chosen by the programmer. However, in this new situation, the compiler has no way to apply the inference algorithm and find what class the generic value pertains to. It is actually impossible to obtain that covariance information at compile-time at all. Only at run-time will the system be able to determine which class the activated object from the database pertains to. Since the generic method returns a generic type, the compiler does subtyping, assuming that the result is an `Object` instance, where the `getAddress()` method does not exist. Because of that, the result is an illegal source Java code. The compiler gets an error while parsing that source line.

The use of reflection and a change to Java compiler's normal behaviour enables the framework to deal with this issue [Pereira & Perez-Schofield, 2011]. An alternative reflective code must be generated at compile-time to serve as a replacement code. This technique allows the access to the internal data class of the activated objects and the invocation of all its methods at run-time. That can be achieved through very different approaches. Nonetheless all of them share a common goal: the generation of an alternative version of the code, this one already legal from the compiler's point of view. For the given example, Case A, that alternative code could be similar to the one that follows:

```
Object o1=psRoot.getRootObject("rui");
Object o2=o1.getClass().getMethod("getStudent")
      .invoke(o1,(Object[])null);
Object o3=o2.getClass().getMethod("getAddress")
      .invoke(o2,(Object[])null);
System.out.println("Street:"+o3);
```

Another type inference also does not work in a second kind of situation, identified as Case B. This second case prevents the method overloading to work properly at compile-time. Let us consider the existence of a method `printPersonalData( Student student )` that prints to screen students' personal data. When using this method, as presented below, the compiler considers the argument an `Object` class.

```
printPersonalData(psRoot.getRootObject("rui"));
```

In these cases compilers do not accept an `Object` in the place of a `Student` class. Additionally, the method overloading does not work properly, if another method with the signature `printPersonalData(Teacher teacher)` exists. Cabana *et al.* [2004] have identified a very similar problem as a result of type erasure on parametric classes. Similarly to Case A, in this case the code is also illegal to the Java compiler.

This second Case B can also be solved with reflection. The process is very simple and somewhat similar to the previous one, because at compile-time all these code situations can also be solved by replacing the original code with another

version of it. At run-time, this alternative code should test the generic object returned, determining if there is any method with a correct signature for the corresponding argument type. As already referred, only at run-time is it possible to know which is the class of the loaded object, so method overloading must occur at that moment.

3.3.6.2 Preprocessing algorithms

To find a solution for these two cases (A and B) a preprocessor extension to the Java compiler and prototype was developed, applying code manipulations in order to make the compilation possible. Thus, the genericity of the language and the framework prototype were extended enabling dynamic typing features. Furthermore, this preprocessor also provides mechanisms to preserve the class's parametric information at run-time. In a nutshell, this extension to the standard Java compiler uses a preprocessor to parse the application source code, identifying all statements where new wrapping and replacing code must be woven. This replacing code performs introspective operations at run-time, which enable the system's dynamic behaviour. These applied techniques to source code preprocessing are discussed in detail in the following section. This preprocessor is, in fact, a static weaver, which weaves the code for very specific functioning framework issues: Case A, Case B and parametric class information handling.

3.3.6.3 Method Genericity (Case A)

Case A presented previously is easily handled by searching in the source code for any direct method call from the `CPersistentRoot` instance object. For each point found in the source code, all nested method calls from it are replaced by a special method that accepts a generic object as an argument and invokes each one of these nested methods reflectively. This special method is rendered at compile-time and remains a private method of the class. The following code fragment shows how the problem identified above is addressed by the framework's preprocessor.

```
System.out.println("Street:"
    +_aof4oop$0$getAddress$getStreet(psRoot.getRootObject("rui"));
```

The name of this method follows a rule in order to avoid naming conflicts. It is obtained as a result of the concatenation of all nested method signatures. If a same method sequence occurs again, with different arguments, another method with a different version number is created. The method was already discussed previously.

3.3.6.4 Method overloading(Case B)

This algorithm is very similar to the previous one, Case A, and was already described in some detail. For the given example, the actual method call should be the following:

```
private void _aof4oop$printPersonalData(Object arg1) throws Exception
{
  if(arg1==null)
     throw new NullPointerException();
  else if(arg1 instanceof Student)
     printPersonalData((Student)arg1);
  else if(arg1 instanceof Teacher)
     printPersonalData((Teacher)arg1)
  else
     throw new ClassCastException("No such method");
}
```

Besides its simple working principle, Case B raises some complex implementation requirements. The algorithm of overloading method inference must be able to deal with all types of method signatures, requiring a very sophisticated parsing process. In this example the implementation is quite simplistic, but in other studied examples that is not the case. Presently, the framework preprocessor is able to handle the first case (A) presented, but new developments are needed to solve Case B. These will be considered in our future work.

3.3.6.5 Parametric classes persistence support

The Java 5.0 chosen approach, based on type erasure, provides good mechanisms for type safety at compile-time. However, parametric classes type information is

discarded after compiling, not being available at run-time. This option does not guarantee a correct object storage and retrieval in the repository, compromising two of Atkinson's three principles: type orthogonality and, consequently, object persistence by reachability [Pereira & Perez-Schofield, 2011].

The AOF4OOP framework gathers the objects' parametric information available at compile-time and saves it into a respective IMO meta-object, enabling its persistence. This information is also cached, thus always available at run-time. The framework provides a system call, `String[] findRuntimeTypeParameters( Object obj )`, which gets the instantiation parameters. That is achieved during the preprocessing phase when parametric class instantiation in the application source code is wrapped by the framework added code, which saves the information gathered. The following code presents the internal system call that wraps all parametric class instantiation.

```
public static <T> T wrapper(T obj,String[] paramTypes)
{
        saveRuntimeTypeParameters(obj, paramTypes);
        return obj;
}
```

As can be observed in the source code, the input reference (`T obj`) is returned without being manipulated. It just needs to be associated to the gathered information.

Next, an example is presented. The application source code that initializes an `ArrayList` of integers is replaced by the following one:

```
ArrayList<Integer> list =
rhp.aof4oop.framework.core.wrapper(new ArrayList<Integer>(),
                                   new String[]{"Integer"});
```

For more complex parameters sequences, all parameters are saved sequentially in the same order that appears in the original source code.

3.3.6.6 Static Weaver

Analyzing the three features discussed previously, we can conclude that our approach follows the same basic aspect-oriented programming principles and techniques, making this prototype extension, as well as the rest of the system, an aspect-oriented component. The rendered methods can be considered as *advices*, which are invoked in join points [Kiczales *et al.*, 1997]. Due to that, the preprocessor works as a static weaver [Kiczales *et al.*, 1997] that applies dynamic data type mechanisms at run-time (for Cases A and B) and enables completeness in orthogonal persistence regarding parametric classes.

With this new static weaver (the preprocessor) a gap that exists in all studied aspect-oriented programming frameworks was bridged. During this research work, a syntax of pointcut expression capable of meeting all requirements of the problems discussed beforehand was not found. Due to the advantages of dynamic weaving, this gap is currently exacerbated by the actual tendency in all aspect-oriented programming frameworks that apply *aspects* at load-time or run-time in the byte-code after compilation. It must be noted that in those two cases (A and B), where the reflective code must be injected, the source code at the beginning is not even legal for the compiler. Because of that, this weaving process must be performed at compile-time.

3.3.6.7 Side effects

The discussed approaches apparently have two drawbacks. The first is the disturbance on the error exception handling. If an exception occurs within the code that was replaced (by an *advice*) the information about the error is not correct, because it will give information about a code that does not exist from the programmer's point of view. However, this problem is already known in aspect-oriented environments, which inject strange code as *advices* into applications. For Clifton and Leavens Clifton & Leavens [2003], the aspects' obliviousness makes the programmer's job very difficult in terms of debugging and code comprehensibility. They observed that there are conflicting demands of obliviousness and comprehensibility within the context of object-oriented programming.

The second drawback regards performance penalty due to the additional code

generated by the framework, which is heavily supported through reflection.

Both drawbacks are inevitable, if that dynamic type behaviour is required in the framework.

3.3.6.8 Potential risk of the total data type abstraction in cases A and B

Besides the two drawbacks already discussed, the level of data type abstraction and dynamic typing achieved by our compiler extension reduces the type safety granted by the standard Java language due to the absence of static type checking mechanisms. Requiring explicit casts on returned values, programmers are fully conscience about what type is expected. Furthermore, the correct method invocation syntax can be statically verified at compile-time. Thus, this anticipates numerous possible run-time errors while compiling.

Considering these facts, it is questionable if that level of abstraction, in both cases A and B, enabling the dynamic typing in Java by changing the normal behaviour of the compiler, is actually desirable. Naturally, we argue that it should be the programmer's decision, as happens in other programming languages. The programmer should decide wether or not to apply the type inference and if it is decidable or undecidable.

3.4 Data Integrity, Constraints and Garbage Collecting

Most of the modern object-oriented languages maintain referential integrity information at run-time through the instances' attributes for the internal use of the garbage collector. Thus, there is no need to check whether an instance referred by an attribute exists in the memory heap. Besides, objects are not explicitly deleted: only when an object is detected to have lost all references pointing to it, is it collected, meaning that its memory is marked as free. Furthermore, its entries in the symbol table are removed. The presented framework takes this approach and extends it into the database. When an object is referred by one or more persistent objects, then it is automatically marked as persistent. However,

when those references are changed to point to some other object, or simply the object to which they pertained is deleted, then the database garbage collector destroys it. This approach makes the barrier between primary and secondary memory very tenuous, since the same mechanisms are employed on both sides under similar circumstances.

The value domain integrity of object attributes is generally handled by the logic encapsulated inside the object and materialized into class through its methods, although not always in a setter method. These data constraints are tangled in the source code, only accessible by reflection. This makes it very complex or even impossible to query that information. As described previously, in the our meta-model these domain constraints can be defined through metadata introduced directly in the application class structure. This approach is very similar and somewhat inspired in the constraint definition syntax used on relational tables. The metadata added to the class version takes place in their definition, creating a complete data integrity constraint description. This approach creates the required conditions for the instance adaptation process effectively, to ensure the domain integrity on all class versions. Additionally, it also removes the tangled code from classes, substituted by very clearly defined rule points, implemented as *aspects* (in terms of AOP). Programmers may impose constraints to an object's attributes following Meyer's principles of Design by Contract [Meyer, 1992].

Currently, the AOF4OOP framework supports two types of domain integrity constraints, which we already discussed and are presented in lines 4 and 6 of Figure 3.4, respectively, ensuring no null values and user-defined intervals of numerical values. A complete description of all constraints enabled by our framework can be found in Appendix A - AOF4OOP Annotation Reference Guide.

3.5 Concurrency, transactions and error handling

Traditionally, databases and programming languages have very different approaches to support concurrency control. In programming languages, concurrency control is based upon the concept of coordination by using inter-process communica-

tion (IPC) methods, which allow thread and/or process synchronization. These methods include programming language constructs such as semaphores, monitors, mutual exclusion, signals, sockets, message passing, shared memory and others. Contrasting with programming languages, in databases the concurrency is supported by transactional mechanisms, which provide isolation and atomicity on parallel activities, serializing them while accessing data.

In pure persistent programming systems, the database acts as a memory extension. On the other hand, the databases' transactional model was coined to interact between persistent data (databases) and the application environment (programming languages). Thus, in persistent programming systems the natural approach to deal with concurrency and transactions would be the one used by programming languages. However, nothing invalidates the existence of programming constructs to manage transactions within the same database paradigm. These mechanisms would also be very useful since it enable isolation, atomicy and total error recovering, over non-persistent data.

In order to support concurrency, transactions and error recovery, in our framework two run-time exceptions and transactional basic support mechanisms were implemented. These are discussed next. Moreover, we also discuss our approach to provide the framework with programming constructs to controll transactions over persistent and non-persistent data. Both approaches are inspired on *aspectization of transaction mechanisms* [Kienzle & Guerraoui, 2002].

3.5.1 Framework exceptions

The AOF4OOP framework aims to be integrated into applications seamlessly without them being conscious of that. Consequently, strange code to an application is executed as if it belongs to it. The nature of the persistence services turns systems prone to run-time errors. Thus, entire system must be aware of this phenomenon and be prepared to recover from any unexpected error.

In the Java programming language, and others, run-time exceptions represent problems that are the result of a programming problem, which one cannot reasonably be expected to recover from them or to handle them in any way. Such problems include arithmetic exceptions (such as dividing by zero), pointer excep-

tions (such as trying to access an object through a null reference) and indexing exceptions (such as attempting to access an array element through an index that is too large or too small). Due to a large number of possible types of errors, those exceptions are not part of the method signature, because it would reduce program clarity, although it can be done. Thus, the compiler does not require a try/catch block statement surrounding those methods. Contrasting with these run-time errors, exceptions arising from an error while establishing access to a database or to open a file, are prone to being recoverable or prone to retrying. This second type of errors usually throws a checked exception that must be declared in the method signature, requiring code to handle that error condition.

According to Atkinson's persistence independence principle, the same code should be applicable to both non-persistent objects and persistence objects. This invalidates any error predicting at compile-time because two objects instances of the same class could have very distinct behaviours in terms of persistence. A getter method, which requires a database access on a persistent object, may throw an exception while undergoing operations that occur at the framework level. However, that same method invoked through a non-persistent object never throws that kind of exception. Thus, all internal framework errors that may affect the normal application functioning must be handled as unchecked exceptions (run-time exceptions). Additionally, the inclusion of these exceptions within the methods' signature would require the revision of the entire application.

In order to allow a suitable error treatment, the framework's core library provides two run-time exceptions, which are thrown every time that an error occurs within the scope of the framework. The following code listing presents their class signature.

```
public class EIntegrityFault extends RuntimeException
public class EFrameworkFault extends RuntimeException
```

The `EIntegrityFault` is thrown every time that a data integrity violation occurs. On the other hand, the `EFrameworkFault` is used in any kind of error arising from database access or class manipulation within the scope of the framework. These two exceptions can be handled in order to allow a program to recover, or it results in an abnormal program termination. In both cases, in the

implementation of the current framework, any object member update is executed within an individual transaction, which is committed in the absence of errors or is rolled backed if any error occurs.

These observations let us conclude that orthogonal persistent applications cannot be completely oblivious regarding error recovering. This conclusion is concurrent with earlier researches [Kienzle & Guerraoui, 2002; Kienzle *et al.*, 2009][Fabry, 2005]

3.5.2 Framework concurrency support

In our framework, the adopted approach to support concurrency is focused on Atkinson's persistence independence principle, which treats all types of objects in the same manner. Such approach requires the introduction of a set of transactional programmatic constructs into the framework, which can reconsolidate both worlds (application run-time environment and database), as referred by Blackburn & Zigman [1999]. This enables an orthogonal transaction model, as the one proposed next. Furthermore, it also enables parallel usage of concurrent programming models (*open approaches* [Blackburn & Zigman, 1999]).

3.5.2.1 Atomicity, Consistency and Durability

In the current prototype of our framework, in order to obtain that orthogonal behaviour over both types of references, non-persistent and persistent, each object member update is performed in an isolated and atomic transaction. We highlight the fact of our approach enables support for controlled concurrent access on any kind of data, persistent or non-persistent. Thus, the object's persistent state is always consistent if any abnormal termination execution occurs because during the transaction's life cycle (an object's member update) the application's thread remains locked. If any error occurs, a framework's exception is thrown and the entire transaction, involving objects and meta-objects, is reverted. At the end of the transactions, the framework returns the control to the applications. Using this approach, ACID properties are always ensured.

This decision, to interpret successful update termination as an implicit checkpoint and abnormal termination as abort, is similar to the commit and abort

operation of a database transaction [Gray & Reuter, 1992]. Regarding this simple type of updates, our approach is the same as PJama [Atkinson & Jordan, 1999; Atkinson *et al.*, 1996].

These characteristics of atomicity, consistency and durability in updates of object members is given by the framework's persistent *aspect*, which handles that concurrent access from several threads. This mechanism is implemented inside the *advice* method which intercepts all member updates in a synchronized and serialized form. Whenever more complex concurrency support is required, those *open approaches* [Blackburn & Zigman, 1999] should be applied to the applications. Additionally to those *open approaches*, our framework still provides other mechanisms for delimiting larger transactions and error recovery, which are discussed next.

The framework's architecture provides good means for implementing transactional mechanisms, enabling a seamless integration between application run-time environment and the database. Thus, the development of three system calls, which lock the program execution thread in an atomic transaction, was possible. These implemented system calls are: `beginTransaction()`, `commitTransaction()` and `rollBackTransaction()`. The code listing in Figure 3.18 presents an example of their usage in the current version of our prototype.

The `beginTransaction()` system call changes the transactional behavioural of the storing *aspect* (see section 3.3.5.1), which is responsible for the storing mechanisms. After starting a new transaction, each object member update is added to a database transaction and the framework's transactional log. When a commit or a roll back occurs, that transaction is finished. In fact, these three system calls are facades to their real implementations in the database engine, as well as the transactional log mechanisms of the framework. If another thread performs any concurrent access to same objects' attributes, the framework and database transactional mechanisms support the serialization of operations. Notice that in our framework, the locking granularity is done at the attribute level of the objects.

Due to the persistence independence principle, the framework's behaviour should be orthogonal. That is to say, it should be applicable to both non-persistent and persistent data. Thus, if any error occurs within a transaction,

```
01   try
02   {
03      Product p1=new Product("Mouse Model X1",35);
04      Procuct p2=psRoot.getRootObject("mouseX2");
05
06      psRoot.beginTransaction();
07      ...
08      p1.setPrice(p1.getPrice()*1.1); // Increase price in 10%
09      p2.setPrice(p2.getPrice()*1.1); // Increase price in 10%
10      ...
11      //An Exception ocurs here
12      ...
13      psRoot.commitTransaction();
14   }
15   catch(EIntegrityFault e)
16   {
17     //A data integrity violation occurs
18     psRoot.rollBackTransaction();
19   }
20   catch(EFrameworkFault e)
21   {
22     //An internal error on framework or database
23     psRoot.rollBackTransaction();
24   }
25   catch(Exception e)
26   {
27     // Any other type of run-time error
28     psRoot.rollBackTransaction();
29   }
```

Figure 3.18: Transaction and failure

a roll back should put objects (non-persistent and persistent) in the same state as when they enter the transaction. Or, if a commit occurs, changes should be durable. We present this case in Figure 3.18. The objects, `p1` and `p2`, respectively non-persistent and persistent, are the subject of updates in lines 8 and 9. In line 11, an exception occurs which is caught in line 25 and rolled back in line 28. After roll back takes place, the state of both objects returns to the one as before line 6.

Although our transactional approach enables nested transactions, inner transactions are finished only when the outer one ends. Our approach to complex transactions is different from that of PJava. This system has a concept of *Trans-*

actionShell, enabling two distinct types of behaviour by means of class specialisations [Atkinson *et al.*, 1996]. Additionally, in our system, non-persistent objects are treated as persistent ones, while in PJava and other systems just persistent objects belong to the scope of transactions [Atkinson & Jordan, 1999].

3.5.2.2 Isolation

As to thread isolation, in our framework concurrent threads share the data being updated. If one thread begins a transaction, all others share it only when reading data. In case of write operations, the objects' attributes are locked in their transactions, hence blocking other threads that intend to update them. Our approach is coherent with concurrent programming models in which threads share data in a synchronized access model.

According to Atkinson & Jordan [1999], since the Java language provides concurrency through lightweight threads, the multiple-process model is strictly unnecessary. We partially agree, however, the single-process model has drawbacks such as limited support for load balancing and lack of thread isolation. Since all threads are running in a common and shared object space, it is important that they do not accidentally interfere with one another.

Implementing orthogonal persistence in multiple-process model based systems raises complex issues. In our prototype's current state, these multiple-process model issues were not considered. They were left for future research work. Our framework's environment is restricted to a single virtual machine.

In terms of data access concurrency, we argue that our approach merges the two paradigms referred previously: (1) concurrent programming models (*open approaches*) and (2) database transactional model. The framework enables programmers to orthogonally apply both techniques over persistent and non-persistent data. Considering this framework's behaviour, we argue that our framework follows the ACID (Atomicity, Consistency, Isolation and Durability) model in a multithreading system.

3.6 Framework limitations

3.6.1 Versioning's orthogonality

The current framework's prototype is unable to handle some of Java's classes, such as the ones that belong to the Java Collections Framework (`ArrayList`, `Hashtable` and others). This particular type of classes, which under our classification are composite objects (see Section 3.2.3), have a complex internal structure that requires their versioning and representation as an IMO meta-object in the database's metadata layer. In our framework, these classes must be manipulated in order to be renamed to their corresponding version. However, in the classes that belong to the Java framework, such type of class manipulation is not allowed by the Java Virtual Machine. Thus, three approaches can be applied in order to address this problem: (1) consider this particular type of classes as not versionable. Such approach can make the framework dependent on the Java Runtime Environment; (2) Implement special handlers for each class of this type; and (3) modify the Java Virtual Machine. The second approach requires additional and very specific code pertaining to these classes. In our opinion, the third approach is a reasonable solution; however, it compromises Java Virtual Machine updates. The former approach is considered the most reasonable because, from the application's point of view, they do not have versions because they don't belong to its schema. Thus, in order to support the persistence of these objects, we implemented that former approach.

3.6.2 Performance

Despite our concerns regarding the meta-model's performance, the prototype was not designed with this issue as its main goal. Thus, the performance issue was not specifically evaluated and discussed in Chapter 4, which is focused on experimental tests. However, an OO7 based benchmark [Carey *et al.*, 1993] is presented in Section 4.3.1 of this Chapter. The results just allow us to observe the performance degradation of the framework when it is faced with objects in different class versions.

During the testing phase of our geographical application (see Chapter 5), we

also observed a performance degradation when the framework is faced with objects in different versions other than what is expected by the running application. Notice that our framework follows a lazy approach on instance adaptation. Although most of this performance degradation is unavoidable and inherent to the multi-version schema approach, many optimizations may yet be introduced in order to improve overall system performance. Some of the current prototype's limitations, in terms of performance, arise from the usage of an object-oriented database as object repository. This object-oriented database provides high-level mechanisms of interaction in a single version schema, not being well adapted to the requirements of our framework. Theoretically, a new specialized object store for our framework and meta-model would be able to enhance its overall performance and orthogonality, avoiding some reflective operations. Additionally, such object store might also facilitate the achievement of an optimal solution in terms of fully versioning orthogonality, as discussed previously.

3.7 Framework development

In this section, the technologies applied on the development of the AOF4OOP framework will be discussed. We present those technologies as well as how they have contributed to a quick and easy development, shortcutting many implementation challenges.

3.7.1 Aspect-oriented extension to the Java platform

The framework was written in Java programming language to be used by applications written with same language. As already discussed in Section 2.1.3.2, the two major aspect-oriented tools for Java are AspectJ and AOP JBoss. The chosen tool was AspectJ due to a previous existing knowledge and because this tool responds positively to almost all of the framework's requirements.

As far as we know, no existing aspect-oriented Java extension or tool is able to deal with the very specific requirements discussed in sections 3.3.6.3, 3.3.6.4 and 3.3.6.5, which raised the need to develop our preprocessor based on a static weaver. In the following sections the applied technologies in this preprocessor are

presented.

3.7.2 Reflection

Reflection played a major role during the entire development, being present in every part of the system. The most relevant was in object inspection, in order to gather its internal structure information. This internal object information is crucial in modules like: schema manager, persistence *aspect* and instance adaptation *aspect*. These reflection features are also available to programmers in their applications.

A recognized limitation, due to the Java type erasure approach, is the lack of support to parametric classes. As already discussed in Section 3.3.6.5, that information is not available at run-time. This limitation placed an additional challenge that was surpassed with the framework's preprocessor. This optional element of the framework extends the Java reflection API, providing introspection features in order to obtain information regarding parametric classes for application and framework usage. Furthermore, it also enables the framework to correct performance persistence in parametric classes.

3.7.3 Instrumentation

The Javassist (Java Programming Assistant) library [Chiba, 1998] helps programmers with bytecode manipulation. It is a library to edit bytecodes in Java, enabling Java programs to define new classes at run-time and to modify a class file when the Java Virtual Machine loads it. Javassist provides two levels of API: source level and bytecode level. If the source-level API is used, a class file can be edited without knowledge regarding the specifications of the Java bytecode. The entire API was designed to use only the vocabulary of the Java programming language. This library also enables an easier insertion of the bytecode in the form of source text; the Javassist compiles it on the fly. Additionally, the bytecode-level API enables users to directly edit a class file.

3.7.4 Language Recognition

ANTLR, ANother Tool for Language Recognition [Parr, 2007], is a language tool that allows the construction of recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. ANTLR provides excellent support for tree construction, tree walking, translation, error recovery and error reporting. In our framework, we used version 3 of this tool, although, at the present date the latest version is 4.5. ANTLR v3 is a completely rewritten version and is the culmination of nearly 20 years of experience building language tools. The goal of version 3 was to provide a really clean source base and to significantly clean up the syntax and semantics of ANTLR grammar meta-language.

This language recognition tool sped up the development of the framework's static weaver discussed in Section 3.3.6 because it easily parses the application's source code, recognizing all Java syntax structures. The framework's preprocessor works as a static weaver, analyzing the application's source code using ANTLR to search the joint points, in order to inject in it the required code through the Javassist's bytecode manipulation mechanisms.

3.7.5 Object repository

The AOF4OOP framework aims to enhance the persistence mechanisms' transparency, enabling an orthogonal persistent programming environment, while mediating the interaction between applications and database. Thus, by using our framework, the database role is reduced to a simple repository for objects and meta-objects.

As already mentioned, the framework's object repository is based on a db4o database [Paterson *et al.*, 2006] in version 7.12. This object-oriented database provides good means for querying, data retrieval and storing. These query and object manipulation mechanisms were important, enabling a quick development of the mechanisms for meta-objects management. However, it does not provide all necessary means for our framework, hence, compromising the overall system's performance. Thus, in future work, a native object repository should be developed, enhancing the system's performance, as well as its orthogonality.

Chapter 4

Experimental results

The AOF4OOP framework prototype was developed in order to test the proposed approach based on a new meta-model, an aspect-oriented paradigm and the orthogonal persistence principles.

To evaluate this prototype four categories of experiments were done. Three of them are discussed in this chapter while the fourth, a real-world application, is discussed in Chapter 5. The former focuses on the impact at meta-model level, under an application data model redesign scenario. This category of experiments intended to evaluate the meta-model performing an analysis based on an earlier case study [Rashid, 2002], extending its results with the ones achieved in the present research work. The second category intended to demonstrate how the framework can be helpful to programmers in their work. The framework was applied in order to accelerate the work developed in the scenario used in the previous experiment. In the third category of experiments, the concerns were about robustness issues. To achieve this third purpose, an OO7 benchmark database was used due to its high data structure complexity to test both meta-model and framework robustness.

The system's performance was not evaluated in this chapter, besides the considerations done in Section 3.6.2. Two main reasons justify the absence of this kind of tests: (1) The current prototype was developed without major concerns regarding performance (there are many optimizations that could be done); and (2) as far as we are concerned there is no known comparable system with ours. The db4o, for instance, does not support multiple schema versions. Besides, it

is this system that supports the prototype's object storing mechanisms (itself persistence).

However, the developed OO7 benchmark is now ready to be used in a future performance benchmark with any compatible system that supports backward/forward application compatibility on an object-oriented database.

4.1 Meta-model and instance adaptation comparative case study

An earlier case study [Rashid, 2002] provides a comparative evaluation of four different systems based on a design correction scenario. This scenario consists of a data structure of seven classes, which are redesigned so that a new class `Staff` appears. This new class was added into the inheritance hierarchy of two of them. Figures 4.1 and 4.2 show that structure before and after the schema evolution, respectively. From now on, we will refer to these two versions of the class structure as version `"A"` and version `"B"`, repectively.

This scenario was reused in order to extend that previous work with our results and to enable a more direct and easier comparison. This comparative case study was structured into two parts. The former analyzes the impact on meta-objects, at schema level, while the second part provides an analysis of the instance adaptation process. The compared systems were: Orion [Banerjee *et al.*, 1986], Encore [Skarra & Zdonik, 1986], TSE [Ra & Rundensteiner, 1997] and SADES [Rashid, 1998]. Some of the techniques applied in these systems were previously discussed in this PhD Thesis.

4.1.1 Schema

At each schema update the relationships among classes are modified. Depending on how the respective meta-model represents relationships, the impact could be more or less complex, affecting a proportional number of meta-objects to that complexity. Classes, attributes, methods and other types of model entities are represented in the meta-model through their respective meta-objects. Thus, all these meta-objects are interconnected according to the data structure that is

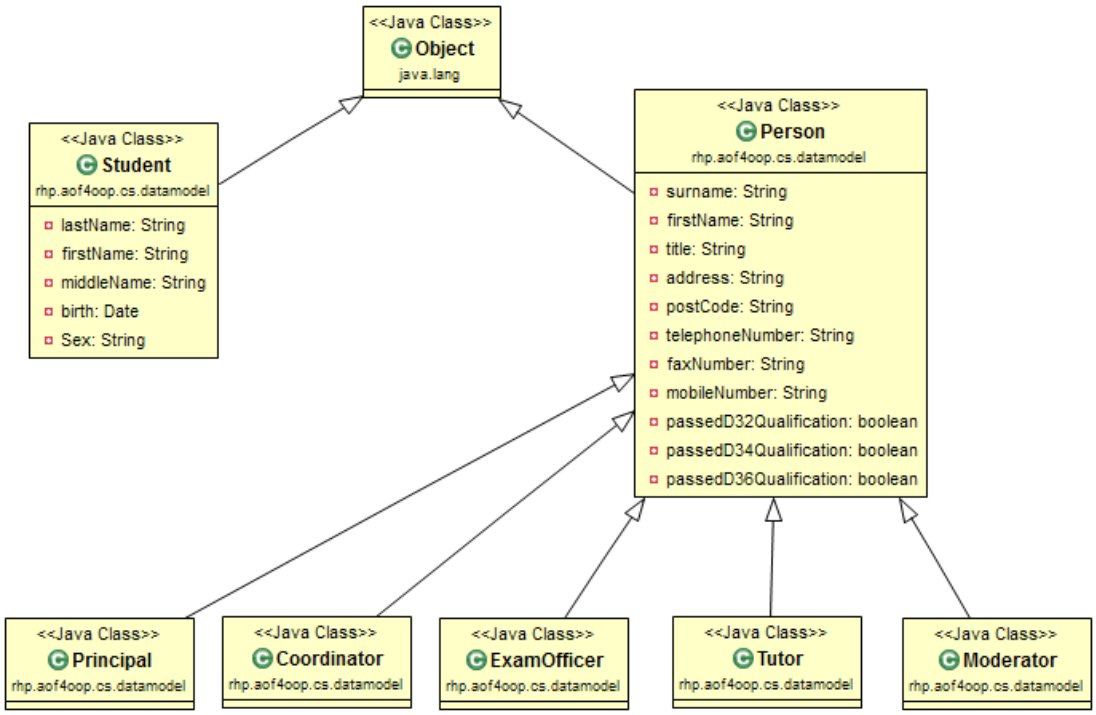


Figure 4.1: Before the evolution - Version `"A"` (extracted from [Rashid, 2002])

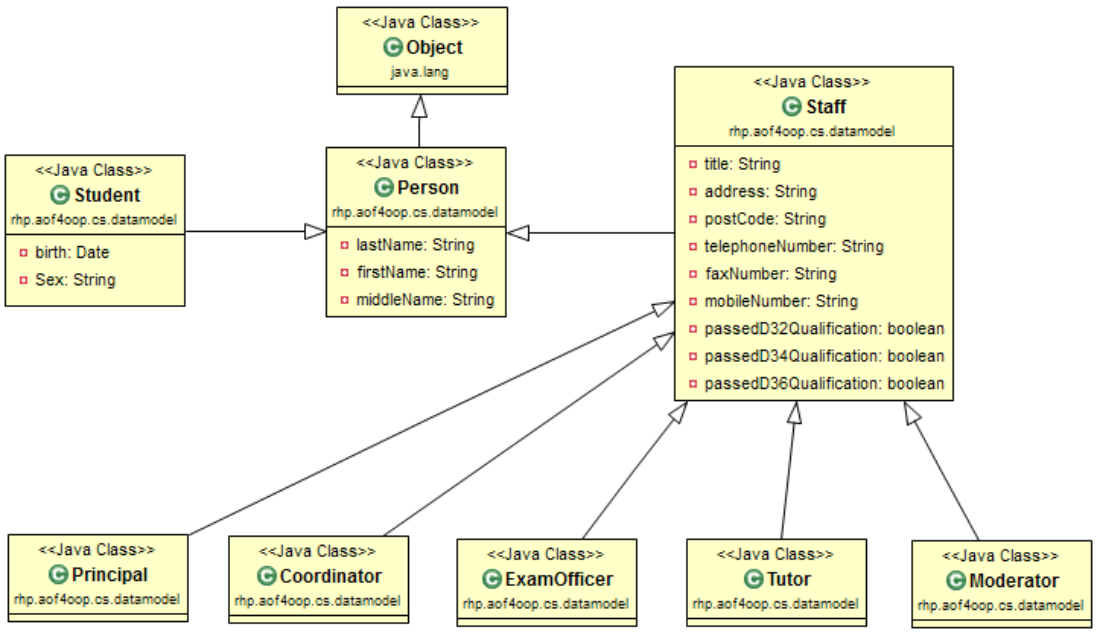


Figure 4.2: After the evolution - Version `"B"` (extracted from [Rashid, 2002])

represented. For example, a class's meta-object has an inheritance relationship with its super class's meta-object, and an aggregation relationship with all its subclasses, attributes and methods. Orion, Encore and TSE use attributes at the meta-object level to implement those relationships among meta-objects. In SADES, those connections are represented as relationship objects [Rashid, 2002]. The AOF4OOP framework approach presented in Section 3.2 is much simpler because much of the objects' relationships are avoided. In our meta-model, the data objects' relationships always require AMO meta-objects. The class inheritance relationships do not require meta-objects because each class version has its own metadata in a CVMO meta-object. This enables the framework to know its inheritance relationships. Regarding the class version relationships, only when semantic or structural variations occur, are the UBMO meta-objects required.

In the comparative case study the introduction of a non-leaf class `Staff` is given as an example. It has a different impact on each system affecting several meta-objects. The graph in Figure 4.3 combines the results of the earlier case study with the ones obtained in the present research work. The respective number of affected meta-objects are presented for each system.

In comparison with Orion, Encore and TSE, SADES presents the best results in terms of affected meta-objects due to the use of relationships objects. These first class objects, that encapsulate the information of connections among the meta-objects, contrast with the remaining systems that have that information embedded within the meta-objects. A detailed description on the implications of `Staff` super class introduction in each system can be found in the referred case study. Since updates in our meta-model are incremental, only new class versions are added to the schema. Thus, just eight new CVMO meta-objects are inserted: seven for the existing classes and an additional one for the new `Staff` class. Additionally, two UBMO meta-objects to support update/backdate compatibility are also required: One is required due the need for a user-defined conversion function from `Person` class, with the `surname` attribute, on the former version, to the new one with two distinct attributes (`middleName` and `lastName`); The second UBMO meta-object performs the same job in the opposite direction of class versions. In this scenario, all remaining changes are transparently handled by the default instance adaptation mechanism, avoiding any extra metadata. In

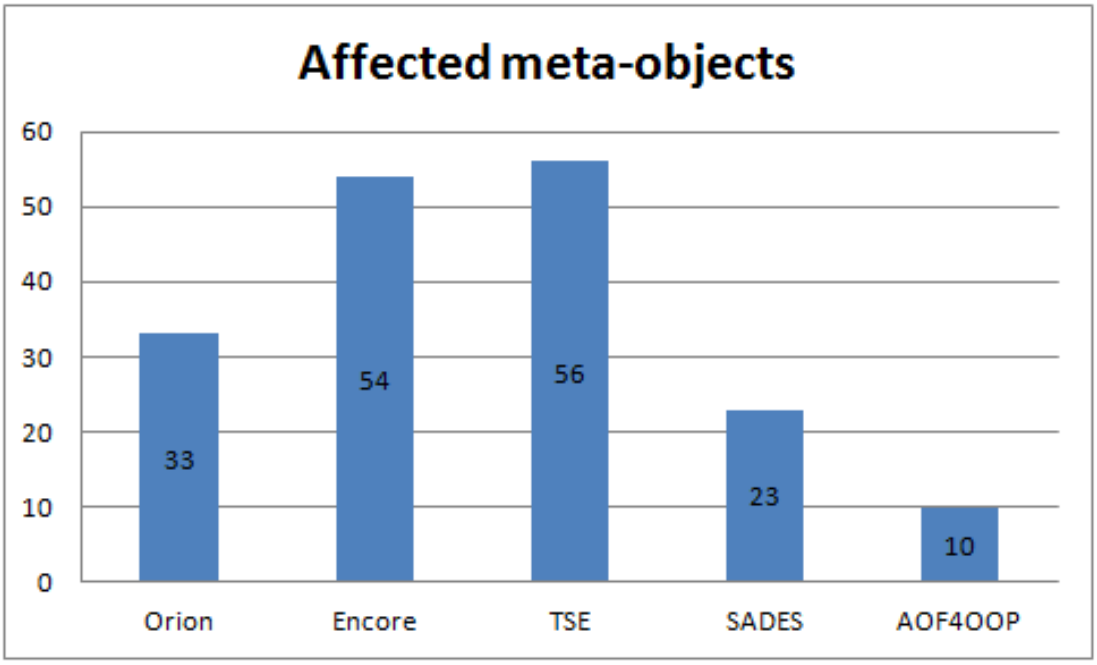


Figure 4.3: Affected meta-objects

Section 4.2 this scenario is again reused, where these two UBMO meta-objects will be discussed in detail.

This design correction scenario, while being very simple, allows a good understanding of the advantages present in the meta-model, discussed in Section 3.2. In this example, we observe a smaller overhead in terms of affected meta-objects when compared with the other studied systems. The proposed approach requires less than half of the meta-objects that are affected in the SADES system. Although this example is not representative of all cases and cannot be generalized, it evidences the advantages of our meta-model in terms of affected meta-objects during a schema evolution scenario. Additionally, its incremental approach to the definition of new schema versions allows a semi-transparent schema evolution. This contrasts with the other four systems, which require a much more complex human intervention at the database, providing schema evolution primitives.

4.1.2 Instance adaptation

In the second part of the referred comparative case study [Rashid, 2002], the instance adaptation problem is addressed providing a detailed comparative description of this adaptation process in those four systems. Those systems were also partially described, as well as the proposed one, although in greater detail, in Chapters 2 and 3. Considering the existing system descriptions, here just the positioning of the proposed approach in relation to those studied systems is discussed.

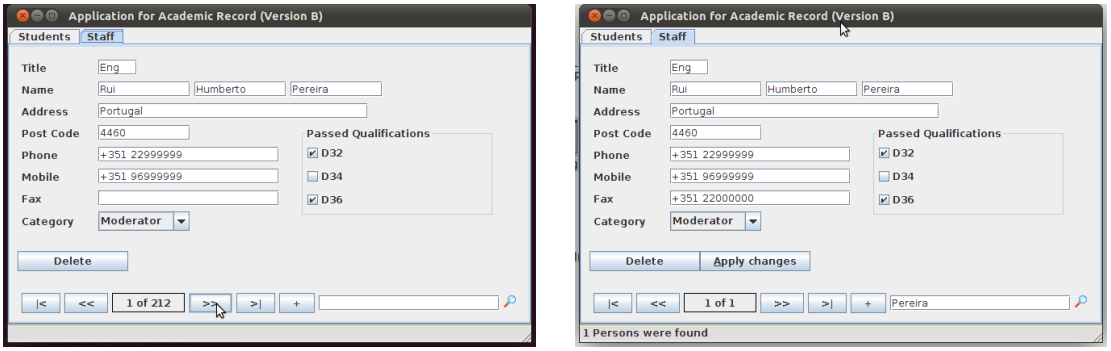
SADES system, in relation to Orion, Encore and TSE stands out by the customization and flexibility capability of the instance adaptation mechanisms. The mechanisms' aspect-orientation enables that customization and flexibility by dynamically applying the most adequate technique. Those advantages were presented in Section 2.4.1. Here SADES and AOF4OOP framework are totally equivalent since both systems follow the same approach.

The proposed meta-model does not condition the chosen instance adaptation strategy. On the contrary, it provides a simple multi-versioned schema approach, which allows a broad spectrum of instance adaptation strategies. The separation of the instance adaptation concern from classes, using aspect-oriented programming *aspects*, provides the database management system with exchangeable and dynamic woven mechanisms.

4.2 Programmer's productivity

This section will present another comparative case study, which intended to assess the framework's added value of its underlying object repository, a db4o object-oriented database. The db4o is a reference system in object-oriented databases that provides object persistence with a considerable level of transparency and orthogonality. Thus, it was considered as the most compatible system with the AOF4OOP framework and chosen to be used in this system comparison.

In order to obtain some conclusive results, the same redesign scenario used in the previous section was considered to perform another comparative case study. In this study, two distinct programmers were invited to adapt two simple applica-



(a) Navigating through records (b) Searching and editing

Figure 4.4: Application's user interface

tions that use the data model represented in Figure 4.1, before their redesign. The former application stores its data directly in a db4o database, while the second one uses the AOF4OOP framework to manage its persistence. Both databases were populated with real data, about 3500 `Student` objects and 20 objects of each 5 subclasses of `Staff` class. Since both applications share a common schema (class structure), its code, which defines the class structure, was implemented as a common library. Figures 4.4a and 4.4b depict two screenshots of the application's user interface when the staff's tab is selected. The application of these screenshots is in version `"B"`.

The case study started with the adaptation of that common library to the new class structure version. The time used by each programmer to conclude this first task was registered. The obtained time value was considered the same in both applications (we considered the average time of both). Since this task is exactly the same in both applications, this strategy guaranties a fare measure avoiding time variations. Notice that the second application would benefit from the knowledge acquired during the first application, solving the same problem.

After that, both programmers start the adaptation of the db4o application. Their first challenge lies in a db4o database limitation, which does not transparently support the following two schema changes:

- Inserting classes into an inheritance hierarchy;
- Removing classes from inheritance hierarchy.

Both class transformation types occur in this study, so an additional application is required to adapt object instances (`Student` and all `Staff` subclasses) from the old class version to the new one. The db4o authors' suggested process is the following one:

- Create the new hierarchy with different names, preferably in a new package;
- Copy all values from the old classes to the new classes;
- Redirect all references from existing objects to the new classes.

This eager stop-the-world model application sequentially scans the entire database, copies, converts and stores data on objects that pertain to a different class name. Notice that in this scenario a semantic and structural transformation in class `Person`'s name occurs: in version `"A"` first name and surname; while in version `"B"` first, last and middle name. In the third phase, the conversion application uses a db4o special API to rename all new classes to their original name. Figure 4.5 illustrates that renaming process.

```
configuration.common()
    .objectClass("rhp.aof4oop.cs.datamodel.StudentOld")
    .rename("rhp.aof4oop.cs.datamodel.Student");
```

Figure 4.5: db4o renaming API

Both case study applications have some code that depends on the class structure, which requires programmer intervention. In version `"A"`, `Tutor` class (and the remaining `Staff` subclasses) has a `surname` member that does not exist in the new version `"B"`, as well as the new member `middleName` in version `"B"`. These structural variations require a simple modification in both applications' source code (in particular in the GUI presented in Figures 4.4a and 4.4b). Once again, in order to obtain a fair comparison, we considered that programmers' knowledge benefits the second application. Thus, the same time was considered in both applications (we consider the average time of both).

Contrasting with db4o case study application, the AOF4OOP based application does not require any additionally helper application, saving programmer working time. As discussed earlier, our framework dealt transparently with inheritance hierarchy modifications types, these being the case. However, the semantic

consistency cannot be transparently dealt with in the framework, which requires programmer intervention. As previously referred, an UBMO meta-object is required due the need for a user-defined conversion function to convert the `Person` class's `surname` attribute, in former version `"A"`, to version `"B"`, which has distinct attributes called `lastName` and `middleName`. Figure 4.6 presents the required pointcut/advice construct. As discussed previously in Section 3.3.2.6, this code is called within a function (`doConversion()`) that takes an old-class object (`oldObj` parameter) and a newly allocated new-class object (`newObj` parameter). Since the UBMO meta-object's option `applyDefault` is marked as true, that new-class reference was instantiated and submitted to the default adaptation routine before being delivered to the user-defined conversion.

```
<ubmo name="ConvStaff$A_to_Staff$B"
      matchSuperClassName="rhp.aof4oop.cs.datamodel.Staff"
      matchClassName="rhp.aof4oop.cs.datamodel.*"
      matchOldClassVersion="A"
      matchCurrentClassVersion="B">
  <conversion applyDefault="true" outputClassName="rhp.aof4oop.cs.datamodel.Staff">
    <sourceCode>
      String middlename=.....;
      newObj.setLastName(oldObj.getSurname());
      newObj.setMiddleName(middlename);
      return newObj;
    </sourceCode>
  </conversion>
</ubmo>
```

Figure 4.6: Pointcut/advice construct for conversion of `Staff`'s subclasses from version `"A"` to `"B"`

Due to performance, this code is applied only to `Staff`'s subclasses, since `Student` objects can be adapted by the default mechanism with less overhead. Besides, this aspect-oriented conversion mechanism is applied once to `Staff` class, avoiding individual UBMO meta-objects in each subclass. Thus, only one UBMO meta-object is required with the best possible performance. This pointcut/advice construct (as an UBMO meta-object) is the only programmer's intervention, besides class structure updates and minor adjustments in the application (GUI issues). This UBMO meta-object provides update application compatibility. However, an additional pointcut/advice construct is also required to provide backward application compatibility. This second one must be applied to super class `Person` for the same reasons.

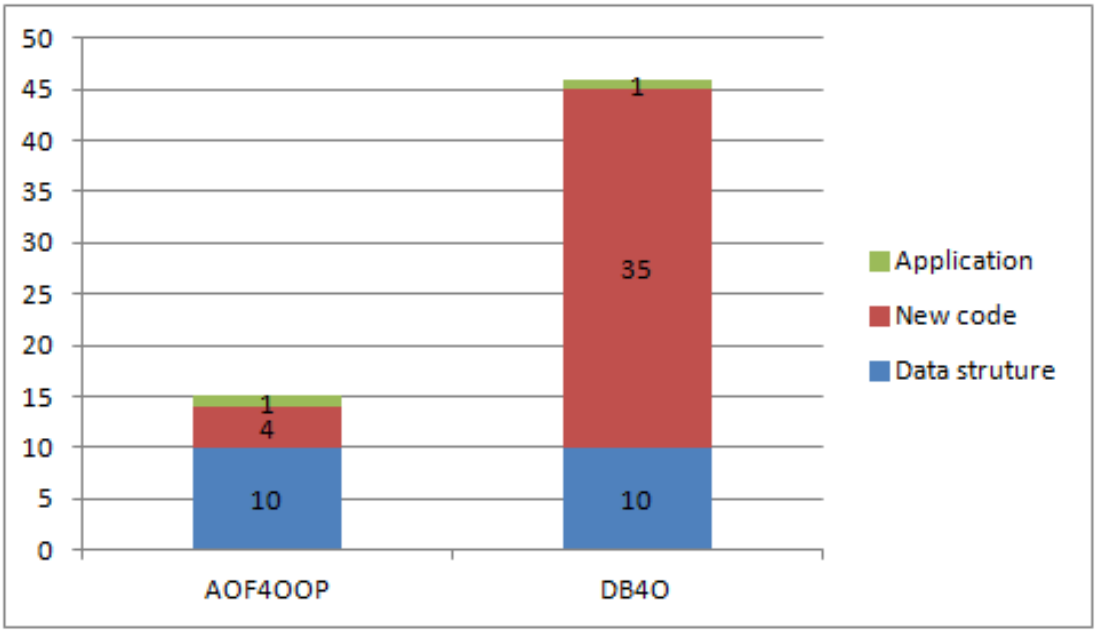


Figure 4.7: Effort comparison (work minutes)

The graph in Figure 4.7 depicts the benefits enabled by the framework for programmer's work in this specific scenario. This graph reveals measurable benefits at productivity level for the presented scenario. The db4o application requires three interventions at the application level. Two of them are common with AOF4OOP based application while the third is the development of an additional application, which converts all objects (`Student` and `Staff` subclasses). The average time required by those two programmers to develop this application was 35 minutes. In this time value, the time for the execution of the helper application in order to convert objects is included. On the other hand, the AOF4OOP based application does not require any additional development. Only a pointcut/advice construct was required, implying a lower working effort of 4 minutes. Both common interventions, on data structures and applications adjustments, required 10 and 1 minute, respectively.

However, there are two other additional advantages that must be considered:

- In order to perform the required schema updates, db4o must follow an eager stop-the-world approach, while the proposed framework does not; and

- The db4o does not allow the previous application to continue being operational, while AOF4OOP does just require an additional pointcut/advice construct.

Regarding the presented scenario, these experimental results have demonstrated the framework's advantages in terms of programmer productivity over a reference system in the state-of-the-art of object-oriented databases.

4.3 Meta-model and prototype robustness

Previous tests have demonstrated the meta-model and framework advantages, now our focus will be on the system's robustness to manage data structures of more complex objects.

The OO7 benchmark was initially developed by Carey *et al.* [1993] to test many different aspects of object-oriented database systems. This benchmark intends to model a typical CAD/CAM/CASE application. This application data model is simulated using the OO7 Benchmark's Design Library, which is based on a structure of classes. In this structure, classes have three types of relationships: one-one, one-many and many-many. Additionally, these relationships have an intricate structure with some circular references, which also provides the means for database robustness testing.

The OO7 database size is highly configurable, while being structured in three main database types: small, medium and large [Carey *et al.*, 1993]. The OO7 benchmark also defines a complete set of tests based on traversal, query and update operations.

4.3.1 Developed benchmark

None of those three original OO7 operation types are focused in schema evolution. Thus, we designed new tests that intend to fill that gap. The new two tests are based on a database Traversal #1 and Traversal #2 [Carey *et al.*, 1993]. In each test phase some schema updates were introduced, which put application's objects according to another schema version

The developed benchmark is a testing environment based on two components: the application developed over the AOF4OOP framework; and a common library that implements all data classes, the OO7 Design Library. In future work, this common library can be reused by another system/framework in order to perform additional comparative performance benchmarks. Besides these two main components, we also developed an automated tool (based on Apache Ant[1]) that cyclically repeats the OO7 benchmark and collects data into a text file.

The designed tests also intend to assess the framework's added overhead due to the existence of distinct class versions in the database, in the following five cases:

a) Without schema variations (reference measure);

b) Add and drop class fields;

c) Add and drop an interface to a class;

d) Insert and remove a class in class inheritance hierarchy;

e) Backdate/update application compatibility (with all mixed cases: b, c, d, as well as semantic changes).

f) Complex class updates (more than one of the previous cases combined)

In order to perform these types of tests, in four schema versions, the six following scenarios were considered:

S1 - Initial schema (class structure in version `"A"`);

S2 - In schema version `"B"`, a new `Z` field integer type is added to some of the `AtomicPart` objects in the database. The `Z` field is initialized as `Z:=X+Y`. The running application knows schema `"B"`;

S3 - In schema version `"C"`, in addition to `"B"`, a dummy class was introduced between the `Assembly` and `BaseAssembly` classes, and also a dummy interface to `ComplexAssembly`. The running application knows schema `"C"`;

[1] http://ant.apache.org/

S4 - In schema version `"D"`, in addition to `"C"`, `Y` field is removed. The running application knows schema `"D"`;

S5 - The test continues to run an application that only knows the initial schema version `"A"`.

S6 - The test of scenario S5 is repeated. The application only knows the initial schema version `"A"`.

Before these six scenarios are possible, it was necessary to populate the database with objects in version `"A"` of the schema. After the database was populated, the six tests are performed sequentially on the same database. In Traversal #2 tests, some objects are updated. This leaves the database populated with objects in distinct class versions. Thus, in each scenario, applications are faced with objects in distinct class versions, performing the two types of tests: Traversal #1 and Traversal #2

We consider the fifth scenario (S5) as the most interesting one, since applications that only know the initial schema version `"A"`, when the `Y` field still exists, run transparently, emulating that class version. Notice that in the database some objects, which do not have this field, are in version `"D"`.

As to the database, we chosen a small OO7 database composed by 7 levels, 500 `CompositPart`, 729 `BaseAssembly`, 364 `ComplexAssembly`, 10.000 `AtomicPart` and 30.000 `Connection`.

4.3.2 Testing robustness and performance

Table 4.1 presents all obtained results in those six scenarios for the two types of tests: Traversal #1 and #2. Besides these six scenarios, the database creation is also represented in Table 4.1. For each operation, two types of measures defined in the OO7 benchmark [Carey *et al.*, 1993] were done: Cold - first running when all system caches are empty; and Hot - when data has already been loaded into the system's cache and the conversion code (embedded in UBMO meta-objects) is already woven. Average times of these operations, as well as minimums and maximums are presented. Additionally, the average consumed time on direct mapping (header Avg. DM) and used-defined mappings (header Avg. UDM) are

Table 4.1: Cold and Hot database traversal #1 and #2 (milliseconds)

Scn	Sch.V	Operation	Minimum	Average	Maximum	Avg.DM	Avg.UDM
		CreateDB	24.386	25.694	26.643	2.052	0
S1	A	T#1 Cold	10.927	11.090	11.199	2.217	0
		T#1 Hot	2.381	2.431	2.488	0	0
		T#2 Cold	12.865	13.072	13.348	2.588	0
		T#2 Hot	3.594	3.716	3.848	302	0
S2	B	T#1 Cold	158.491	160.845	163.196	2.594	151.405
		T#1 Hot	2.317	2.400	2.439	0	0
		T#2 Cold	251.416	261.244	271.629	2.770	154.962
		T#2 Hot	84.216	86.164	89.094	458	0
S3	C	T#1 Cold	145.464	146.980	149.660	2.750	137.459
		T#1 Hot	2.246	2.366	2.375	0	0
		T#2 Cold	148.378	151.268	168.179	3.025	141.583
		T#2 Hot	3.293	3.423	3.616	297	0
S4	D	T#1 Cold	145.559	147.208	150.061	2.705	137.556
		T#1 Hot	2.324	2.422	2.451	0	0
		T#2 Cold	240.098	256.778	276.618	2.653	140.564
		T#2 Hot	82.968	95.472	109.439	565	0
S5	A	T#1 Cold	19.711	20.089	20.590	2.486	8.882
		T#1 Hot	2.337	2.381	2.406	0	0
		T#2 Cold	21.919	22.468	22.982	2.991	9.040
		T#2 Hot	3.426	3.843	4.127	328	0
S6	A	T#1 Cold	10.948	11.161	11.339	2.227	0
		T#1 Hot	2.307	2.369	2.433	0	0
		T#2 Cold	12.826	13.094	13.456	2.594	0
		T#2 Hot	3.467	3.643	3.760	288	0

also presented in the table. All measured times are in milliseconds. The table's columns from left to right indicate: scenario; application's schema version; type of operation; minimum, average and maximum consumed time in the operation; average times consumed in direct mappings and user-defined mappings.

Some disturbance introduced by other operating system processes was observed in each execution. For this reason, all presented values are the result of an average of nine consecutive runs, where the lowest and highest values were discarded. The presented values were collected by our test unit that automated the entire process.

In all six scenarios, when compared to Cold operations, Hot operations required much less time because objects and meta-objects were already in cache. In the case of Traversal #1 operations, 100% of the objects were obtained from the framework's cache. Since Traversal #1 Hot operations only read data which is already in cache, theoretically, this operation should take exactly the same time in all six scenarios. Unfortunately, that was not observed due to the disturbances originated by the operating system's normal functioning.

Traversal #2 operation updates objects, contrasting with Traversal #1, which only reads data. Thus, in S2, S3, S4 and S5, after these operations took place, some objects were updated to its application's class version. This fact justifies the time consumed by the framework on user-defined mappings. In all other operations and scenarios this value is zero.

Comparing the consumed times in S2, S3 and S4 with S5, we can observe that they are very different in these two scenario groups (Cold and Hot). In all four cases, user-defined conversion is required in order to ensure the backdate application compatibility. Hence, in S2, S3, S4 and S5, the Cold operations require much more time. However, in S2, S3 and S4, almost all objects are in version `"A"` of the schema, thus consuming more time than S5. Notice that in S5 just the objects that were updated are in other versions other than version `"A"`, thus, consuming less time.

In test Traversal #2 of S5, the application updates objects again to version `"A"` of the schema. Thus, in S6, the application meets the exact same database state of S1. Thus, theoretically, one would expect the exact same results as the ones obtained in S1.

As to the time consumed in direct mappings, we observed that this value is zero only for Traversal #1 operations in Hot case. In the remain cases, the consumed time is the overhead that our framework introduces in order to map application classes in the database classes. As discussed in Section 3.3.2.5, objects must be renamed (converted) to another class name in order to accommodate multiple versions of the same class in the database.

This experiment allows us to confirm both meta-model and prototype robustness, through read (traversal #1) and write (traversal #2) operations upon a complex data structure based on the OO7 Design Library. After each traversal

#2 operation, which updates objects in the database, we observed that the entire database maintains its consistency. In each case, an additional traversal #1 operation was done, counting all objects and confirming database consistency.

In addition to our main goal of testing the framework's robustness, we also intended to find where the main performance bottlenecks were. These results will be very important for our future work in order to improve our framework.

4.3.3 Application compatibility

In S2, S3 and S4 backward compatibility was observed. On the other hand, in S5, an application that knows an earlier schema version (`"A"`) is faced with objects in future versions (`"B"`, `"C"` and `"D"`). In this scenario, forward compatibility is observed.

In the test discussed above, in each of the six scenarios, the database remains populated with objects in distinct class versions. This test allowed us to check compatibility under scenarios of an application when faced with multiple schema versions different than its own, simultaneously. In our tests, all the four application versions were capable of running transparently and obliviously in regards to the schema version in the database.

Chapter 5

Real world application

In order to test our framework in a real scenario, a real world application was developed. This proof of concept application uses data obtained from the online OpenStreetMap[1] geographical database. This online tool allows you to export a user-defined area based on its coordinates. The OSM export files, in XML format[2], contain all data related with that geographical area. Each one contains the coordinates of the boundaries and it is structured as a set of objects such as *Nodes*, *Ways* and *Relations*.

The *Node* is one of the core elements in the OpenStreetMap data model. It consists of a single geospatial point using a latitude and longitude format. *Nodes* can be used to define standalone point features such as shops or bus stops. These elements can also be used to define the pathway.

Ways are used to represent linear features like footpaths, roads, railway lines and power lines. In OSM files, *Ways* contain references (using <`nd`> nodes) to its *Nodes*. In our geographical application data model, *Ways* and imported areas share their common *Node* objects.

Geographical areas do not have a specific data type. They are simply a closed *Way* where the first *Node* is the same as the last. In this case, *Ways* are used to represent building outlines, parks or any other solid polygon.

A *Relation* is another core data element. It consists of one or more tags, as well as an ordered list of one or more nodes and/or ways as members, which are

[1] http://www.openstreetmap.org
[2] http://wiki.openstreetmap.org/wiki/OSM_XML

used to define logical or geographic relationships between other elements.

Tags are key-value pairs of strings optionally attached to each geographic element in OpenStreetMap. These tags describe the feature they are attached to, and can be any pair of strings.

5.1 User interface

Figures 5.1, 5.2, 5.3 and 5.4 depict four screenshots of the application's user interface, as follows: the first shows the University Campus of Orense; the second, choosing an OSM file to be imported to the local database; the third, three overlapping geographical areas and, the fourth figure, the user interface when selecting points of interest. This user interface is organized into three panes: a menu on top, object browser on the left and maps on the right. The object browser enables the selective hiding of map objects, such as points of interest and roads. The map and these objects are presented on the right side of the interface area.

The application's user interface was implemented using JMapViewer library[1]. This graphical component implements all map visualization objects.

5.2 Data model

The application's data model follows the same structure as the OSM files, presented in Figure 5.5. An `Area` object corresponds to an imported area using an OSM file. First, the file is imported to an `OSMFile` object. Then, its contents are copied to an `Area` object. Only `Area` objects are made persistent. Thus, each `Area` contains a set of `Node`, `Way` and `Relation` as arrays. Since a `Node` object can pertain to more than one `Area`, `Way` or `Relation`, they are shared, occupying a unique slot in the database, while having a unique object identity.

[1] http://wiki.openstreetmap.org/wiki/JMapViewer

Figure 5.1: User interface - University Campus of Orense

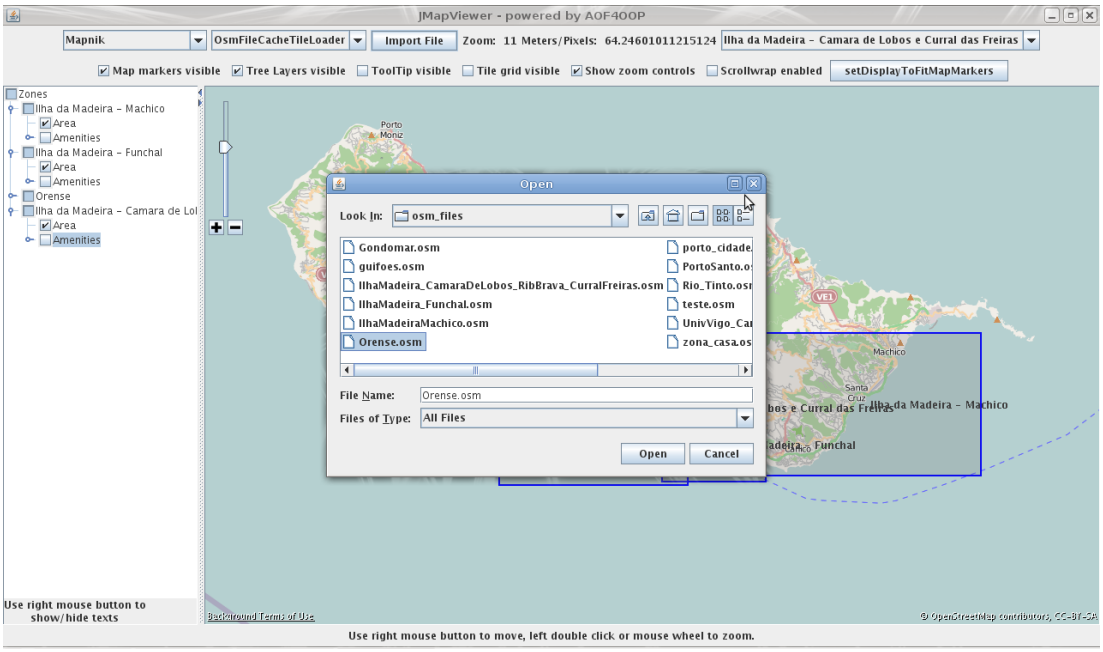


Figure 5.2: User interface - Importing an OSM file

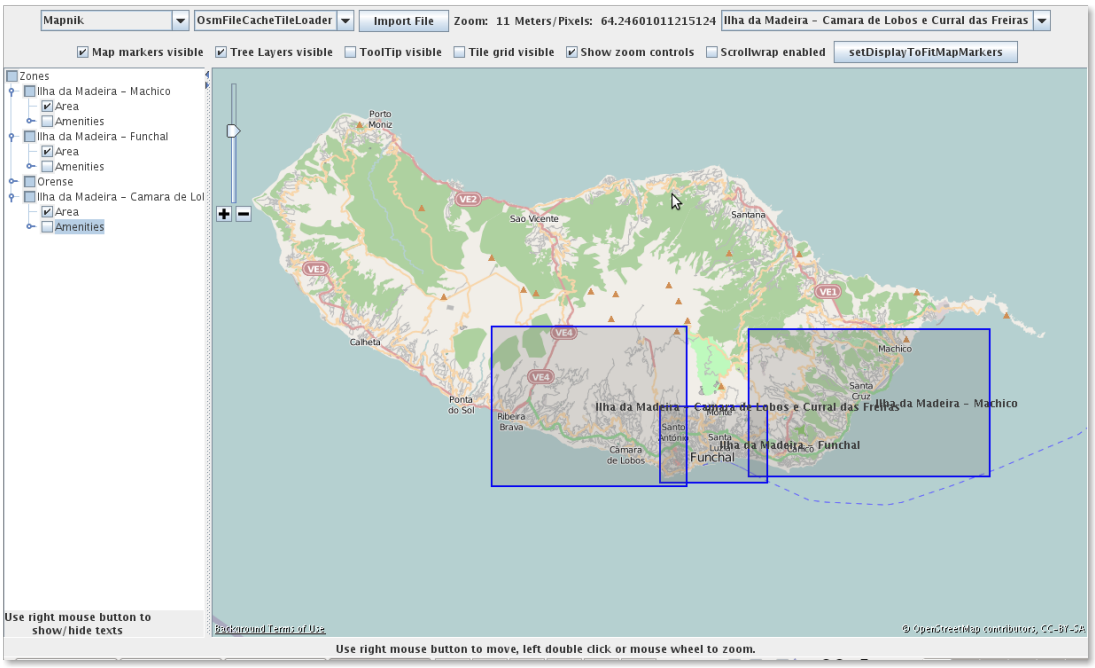


Figure 5.3: User interface - Viewing three overlapping geographical areas

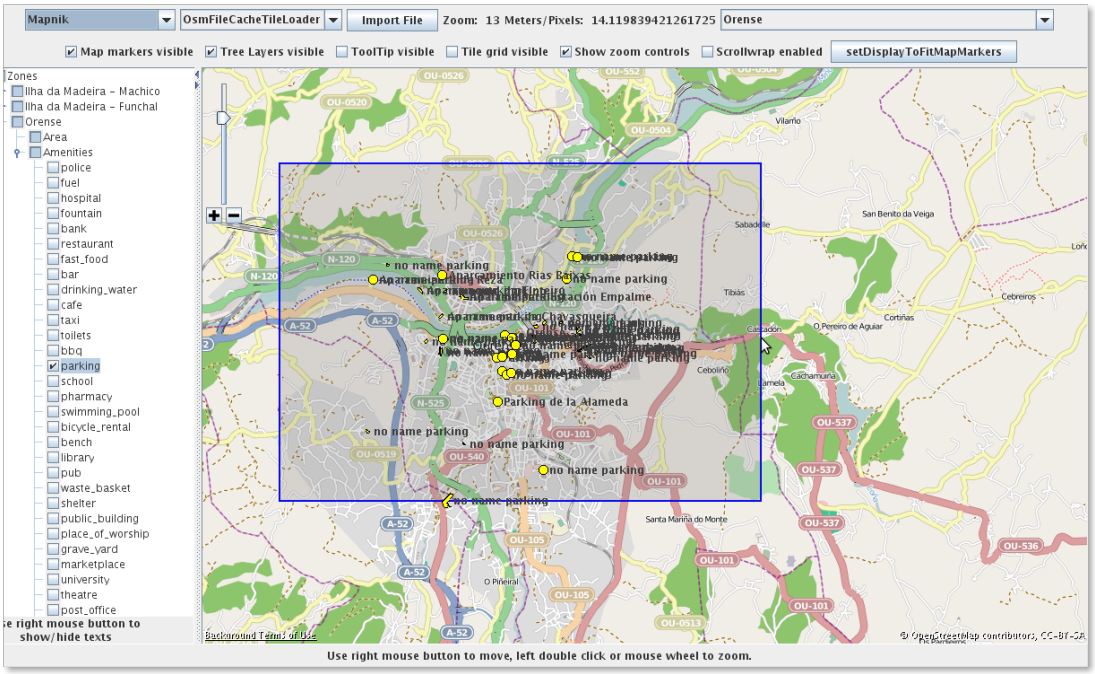


Figure 5.4: User interface - Selecting points of interest

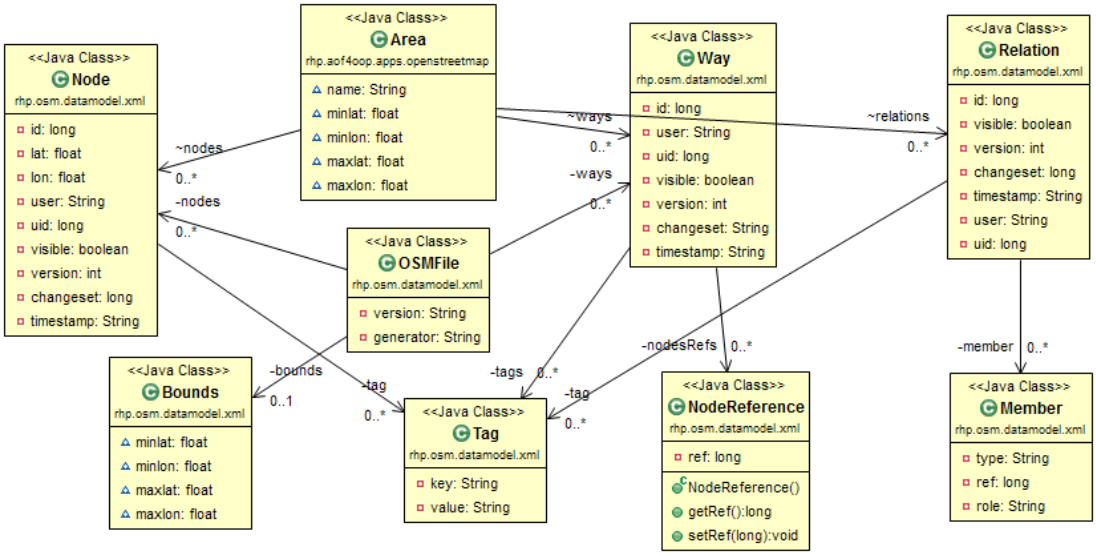
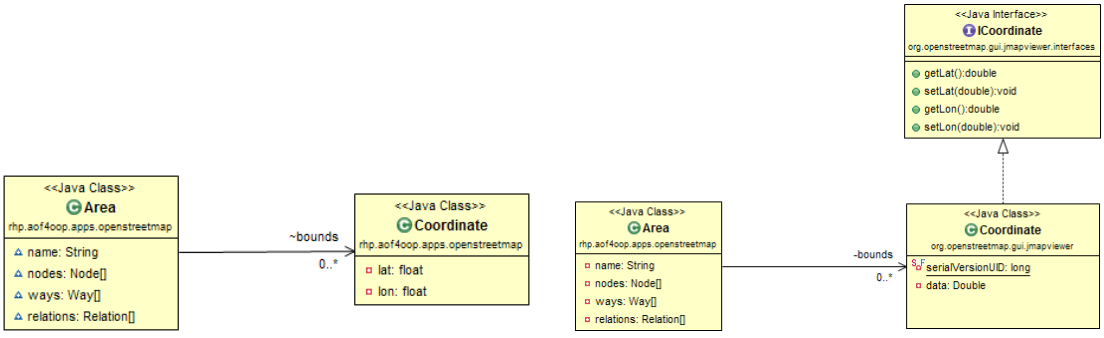


Figure 5.5: Application's data model with class `Area` in version `"A"`



(a) Class `Area` in version `"B"`

(b) Class `Area` in version `"C"`

Figure 5.6: Application's data model

5.3 Providing the application with persistence

During the initial phase of the application's development, we just imported map areas to memory without any concern regarding data persistence. Map areas were imported from an OSM file and then put in a `Hashtable` collection. Thus, during the application's execution time, all the imported data remained available. However, all data is lost in any further application's execution after its restart. The following Java code listing illustrates that first stage of development.

```
//Non persistent application
Hashtable<String,Area> areas=new Hashtable<String,Area>();
Area importedArea=importFile(name,file.getAbsolutePath());
areas.put(importedArea.calcKey(),importedArea);
```

The next listing presents an example of how our framework alleviates the programmer's effort in order to place persistence concern in applications.

```
//Persistent application
CPersistentRoot psRoot=new CPersistentRoot();
Area importedArea=importFile(name,file.getAbsolutePath());
psRoot.setRootObject(importedArea.calcKey(),importedArea);
```

The orthogonal persistence mechanism has made the process very simple. The framework's API provides a collection of persistent named root objects, following the same approach as the `Hashtable` collection. The `Area` objects, and its related objects, are made transitively persistent after being placed in that data structure. After that, any update made to these objects, and any other that is reachable, are transparently reflected on the database.

5.4 Evolving the application

In the first stage of the application, each imported `Area` is a rectangle in the map. In the application's current state, the data model has evolved enabling map areas to be merged. When the user imports an area in the map, it can be merged if it overlaps other existing areas. Thus, a map area is now defined as an irregular polygon. Figure 5.6a presents the data model after the schema update. Modified classes are identified using version `"B"`.

These application refactorings have implied a schema evolution in the class `Area`. From one version of the class `Area` to another, there are semantic changes that cannot be handled autonomously by the default instance adaptation aspect. In order to turn these schema updates transparent to applications, a database evolution *aspect* was defined at the framework level, by means of our pointcut/advice constructs. Note that the framework already has an instance adaptation default mechanism. Using those user definitions, it is possible to expand the system by adding behaviours.

Regarding the schema, an incremental evolution takes place. At runtime, when version `"B"` of class `Area` is first loaded, the framework autonomously detects a new version, producing its new CVMO meta-object. From that moment on, the class version is available for the entire environment without human intervention.

In version `"B"` of the `Area`, programmers find that JMapViewer library already has a specific class and interface to represent map coordinates. Thus, a new version of `Area` is created in order to avoid the duplication of classes. In this `"C"` version of the `Area`, the coordinates are represented as `org.openstreetmap.gui.jmapviewer.Coordinate` instead of our class to represent map coordinates `rhp.aof4oop.apps.openstreetmap.Coordinate`. Figure 5.6b illustrates the data model after this second schema update.

5.5 Schema evolution handling

Although our framework enables bidirectional application compatibility, an `Area` defined as a polygon in versions `"B"` and `"C"` must be represented as a rectangle in `"A"`. Thus, to enable that compatibility, the conversion procedure should produce a rectangle that includes the entire polygon or, the major rectangle should fit inside the polygon.

Emulated `Area` objects, in version `"A"`, contain the rectangle edges coordinates, which are calculated from the polygon area edges in versions `"B"` and `"C"`. Figure 5.7 presents the pointcut/advice construct that contains the user-defined code required for that emulation. Notice that `org.openstreetmap.gui.jmapviewer.Coordinate` and `rhp.aof4oop.apps.openstreetmap.Coordinate`

have exactly the same interface to applications (same getters and setters). Thus, we use the | (Or) operator in `matchOldClassVersion` parameter in order to intercept both cases reusing the same pointcut/advice construct. As discussed in Section 3.3.2.6, at runtime, the framework's instance adaptation aspect is extended with the user-defined source code provided in the pointcut/advice construct. The framework's new behaviour is woven, by means of the database weaver, which adds a new class to the framework that implements the `doConversion(oldObj, oldObjParent, newObj, newObjParent)` method. This method intercepts the framework's instance adaptation default mechanism when the target object is being converted.

When converting `"A"` to `"B"`, an array of coordinates that define the four edges of the polygon (rectangle) is emulated. This interesting case is presented in Figure 5.8. The member `bounds` do not exist in version `"A"` of the `Area`. After being emulated, this array is marked as reachable but not persistent. That is to say, it does not have its own LOID. Conversion case `"A"` to `"C"` is identical to the previous one.

Figure 5.9 presents the pointcut/advice construct that handles object emulation in version `"C"` that derive from version `"B"` in the database. Although member `bounds` exist in both class versions, the types are not the same. Thus, a conversion definition is required. From `"C"` to `"B"`, the process is similar.

Throughout the three stages of the application's schema, we created objects in each class version. In the end we had three applications sharing a unique database populated with objects in these distinct three versions: `"A"`, `"B"` and `"C"`. The three applications were capable of consuming and updating data, despite the state of the version.

```
<ubmo name="AreaBounds$BorC_to_A"
      matchOldClassVersion="B | C"
      matchCurrentClassVersion="A"
      matchClassName="rhp.aof4oop.apps.openstreetmap.Area">
 <conversion applyDefault="true" outputClassName="rhp.aof4oop.apps.openstreetmap.Area">
  <sourceCode>
      newObj.setMaxlat((float)oldObj.getBounds()[0].getLat());
      newObj.setMinlat((float)oldObj.getBounds()[2].getLat());
      newObj.setMaxlon((float)oldObj.getBounds()[1].getLon());
      newObj.setMinlon((float)oldObj.getBounds()[3].getLon());
      return newObj;
    </sourceCode>
 </conversion>
</ubmo>
```

Figure 5.7: Conversion from B or C to A

```
<ubmo name="AreaBounds$A_to_B"
      matchOldClassVersion="A"
      matchCurrentClassVersion="B"
      matchClassName="rhp.aof4oop.apps.openstreetmap.Area">
  <conversion applyDefault="true" outputClassName="rhp.aof4oop.apps.openstreetmap.Area">
   <sourceCode>
     rhp.aof4oop.apps.openstreetmap.Coordinate[] bounds=new rhp.aof4oop.apps.openstreetmap.Coordinate[4];
     bounds[0]=new rhp.aof4oop.apps.openstreetmap.Coordinate(oldObj.getMaxlat(),oldObj.getMinlon());
     bounds[1]=new rhp.aof4oop.apps.openstreetmap.Coordinate(oldObj.getMaxlat(),oldObj.getMaxlon());
     bounds[2]=new rhp.aof4oop.apps.openstreetmap.Coordinate(oldObj.getMinlat(),oldObj.getMaxlon());
     bounds[3]=new rhp.aof4oop.apps.openstreetmap.Coordinate(oldObj.getMinlat(),oldObj.getMinlon());
     newObj.setBounds(bounds);
     return newObj;
   </sourceCode>
 </conversion>
</ubmo>
```

Figure 5.8: Conversion from A to B

```
<ubmo name="AreaBounds$B_to_C"
      matchOldClassVersion="B"
      matchCurrentClassVersion="C"
      matchClassName="rhp.aof4oop.apps.openstreetmap.Area">
 <conversion applyDefault="true" outputClassName="rhp.aof4oop.apps.openstreetmap.Area">
   <sourceCode>
    org.openstreetmap.gui.jmapviewer.Coordinate[] bounds=
                    new org.openstreetmap.gui.jmapviewer.Coordinate[oldObj.getBounds().length];
    int n=0;
    for(rhp.aof4oop.apps.openstreetmap.Coordinate c_old:oldObj.getBounds())
    {
       bounds[n]=new org.openstreetmap.gui.jmapviewer.Coordinate(c_old.getLat(),c_old.getLon());
       n++;
     }
     newObj.setBounds(bounds);
     return newObj;
   </sourceCode>
  </conversion>
</ubmo>
```

Figure 5.9: Conversion from B to C

Chapter 6

Conclusions and future work

This PhD thesis addresses the problems that arise to programmers and database administrators when the schema of an application must evolve, as well as all the problems related with data persistence. The schema evolution is a very common scenario, which monopolizes significant work effort from IT professionals. Additionally, it also works as a barrier to the applications' life cycle, in many times resulting in the so-called legacy applications because other applications may also share the same database.

In order to address the aforementioned problems, we present a meta-model to support both data and metadata on object-oriented databases under a multi-versioned schema. Our meta-model was designed in order to enable the development of aspect-oriented frameworks for orthogonal persistence. The main goal of such aspect-oriented frameworks is transparency and obliviousness as persistence and database evolution mechanisms are provided to the applications.

6.1 Summary of results

In this section we will recover the formulated hypothesis outlined in Section 1.2 and explain how the research carried out has made its way to provide answers to those questions.

1. *Does the orthogonal persistence paradigm offer special conditions to implement transparent and oblivious mechanisms of persistence, which include*

database evolution, concurrency and error recovering?

2. *Which is the most adjusted meta-model to support such mechanisms?*
3. *How can aspect-oriented programming techniques be applied in order to develop systems with such transparency and obliviousness?*
4. *How could that transparency and obliviousness improve the work of programmers and database administrators?*

Formulate Hypothesis 1 and 2:

Earlier research has proved that aspect-orientation of the persistence concern has customization and flexibility benefits.

Our hypothesis is that obliviousness and transparency, in those mechanisms, are also possible to achieve in the context of orthogonal persistent systems by applying aspect-oriented programming techniques. Furthermore, its practical implementation is possible as a reusable framework.

If such hypothesis is confirmed, IT professionals' work can benefit in terms of quality and productivity.

Orthogonal persistent programming systems, due to their specific characteristics, provide special conditions which are not observed in non-orthogonal systems. Two of these characteristics were considered as key to achieve database evolution mechanism transparency.

The first characteristic is the intrinsic transparency as this type of system naturally provides applications with persistence mechanisms. In this category of systems, the embedded schema in the applications' source code, being exactly the same as the one that resides in database, enables the incremental propagation of the schema from applications to the database.

The second characteristic is the close-coupled interaction between applications and the database management system. In the AOF4OOP framework, these two subsystems share the same running environment, enabling a seamless access to the entire system. Thus, the database works like as an extension of the application memory heap. Given this rapport between the two subsystems,

the aspect-oriented programming techniques were considered well suited to implement persistence mechanisms and address the database evolution problem. Additionally, the use of those techniques avoided changes to the Java Virtual Machine.

During our research work, application schema enrichment has demonstrated to be a powerful strategy to improve our goals in terms of the database evolution mechanism transparency. This enrichment empowers the schema version self description, enabling the default conversion mechanisms to autonomously solve, in many cases, the instance adaptation problem. Additionally, this enrichment of the application's schema also enables the reinforcement of the object's attribute integrity constraints following Meyer [1992]'s principles of Design by Contract. Our pointcut/advice constructs enable users to extend the framework's default instance adaptation *aspect* when it is unable of autonomously deal with instance adaptation.

Regarding the schema evolution, it is done once by the programmer while he/she develops the new application version. That is to say, programmers are free of update the applications' schemas, because each update transparently and incrementally produces a new schema version in the database. This incremental approach contrasts with non-orthogonal systems, where both programmer and database administrator must intervene. Given all the considerations aforementioned, our class versioning-based approach enables existing applications, developed prior to a new schema version, to continue running oblivious to the existence of new versions.

The gathered experimental results presented in Chapter 4 and Chapter 5, when compared with others pertaining to similar systems, have showed the benefits of the meta-model's simplicity, the system's orthogonality, schema enrichment and our pointcut/advice constructs. This fourfold combination enables an incremental approach of schema updates propagation from the application into the database, as well as semi-transparent instance adaptation mechanisms. Such paradigm, combined with aspect-oriented techniques, has showed its advantages in order to add flexibility and obliviousness to these mechanisms.

Transactions and failures are part of the data persistence concern. Thus, our framework enables users to choose their concurrency control approach. Users can

apply the so-called "open" approaches [Blackburn & Zigman, 1999], as well the framework's transactional model discussed in Section 3.5.2. Both mechanisms can coexist in the same application.

The OO7 database was used for the robustness tests. The data model of this database contains intricate object relationships with some circular references, which is a good robustness test instrument. The observed experimental results have demonstrated the framework's and meta-model's capability to support such level of relationship complexity. Our proof-of-concept application discussed in Chapter 5 reinforces this conclusion.

The Java parametric classes implementation limitation, based on type erasure, discussed in Section 2.1.2, does not provide means for a correct object introspection at run-time. This problem compromises two of the Atkinson's three principles because parametric classes normally are not correctly stored in a database, hence breaking orthogonality type and, consequently, reachability. At the current development stage of our framework, the classes of this category have their typing parameters managed and saved into database.

6.2 Main outcomes

We proposed a meta-model and approach which enables the development of reusable and transparent aspect-oriented frameworks. In order to test their feasibility, we developed a prototype. This prototype is a totally reusable framework, extensible by following an aspect-oriented paradigm. It can provide applications with class evolution, instance adaptation, transactional mechanisms and constraint checking in an orthogonal environment, which is completely novel compared to other research works.

Regarding schema updates (class evolution inside the database), applications supported on our prototype are oblivious even without requiring its stop. Only in semantic or structural updates does our framework need to be extended. In such cases, the programmers' tasks require less effort than in those other studied systems. Programmers can extend the prototype using our pointcut/advice constructs, which are completely novel compared to other research works. Our

prototype, besides the flexibility that other systems also provide, overcomes such systems in terms of transparency mechanism.

The prototype's transactional mechanisms enable programmers to write transactional code sentences while keeping him/herself abstracted from the database's technical issues. These transactional mechanisms are orthogonal, *i.e.* they are applied orthogonally to persistent and non-persistent data.

6.3 Discussion and future work

We now discuss issues related to our approach. Its benefits for programmers, as well as its limitations, will be addressed in this section.

6.3.1 Meta-model analysis

Our meta-model was inspired in the one proposed by Rashid & Sawyer [1999]. As discussed in Section 4.1, theoretically, our meta-model leads to better results in terms of complexity. In that Section we reused an earlier case study that compares other meta-models including the one from Rashid & Sawyer [1999]. The results for that specific database evolution scenario of the case study showed that our meta-model requires less meta-objects, as well as less modified meta-objects.

Thus, we believe that our meta-model is better adapted for supporting persistence services, in terms of data overhead, since it require less meta-objects to represent applications' data, as well as updates to its schemas. Besides, the overall system performance also can benefit due to the meta-model's simplicity. However, the intensive use of reflection may also affect performance, a cost that may not be compensated. This issue, as well as the prototype's performance, should be subject of future research work.

6.3.2 Flexibility and customization

Rashid & Leidenfrost [2004, 2006] and Kusspuswami *et al.* [2007] acknowledged flexibility as a major benefit of aspect-oriented programming techniques when

addressing the database evolution problem. We totally agree with these authors. As discussed in Section 4.1.2, our approach achieves the same results in terms of flexibility. However, since our prototype enables programmers to extend the framework by means of our pointcut/advice constructs, we argue that our approach goes further, overcoming their results in terms of how flexibility is enabled to users. The next two sections continue this discussion.

6.3.3 Obliviousness and transparency

The presented framework has enabled a simple and quick implementation of the persistence concern in our geographical application. We argue that this proof-of-concept application is oblivious to that concern. As discussed in Section 5.3, the application initially was developed without any implementation of the persistence concern. Applying a minimal change in its source code, we provide the application's objects with persistence *aspect*. All remaining code of the application is kept unchanged, remaining oblivious to the persistence concern.

This framework's *aspect* is non-invasive, totally decoupled from applications, being a *spectator* [Przybylek, 2013], which does not violate the specified behaviour of base modules. Despite providing applications with additional behaviour, if turned off, they continue working oblivious to that lack of persistence. Additionally, the orthogonal persistence paradigm followed by our framework still enables transparent data access while promoting programming quality.

Regarding schema evolution, an incremental approach was adopted, freeing the programmer of having to intervene at the database level, avoiding the use of schema evolution primitives. The instance adaptation concern was addressed with our pointcut/advice constructs. These types of expressions enable the quantification of join points that need user-defined conversion code. This user-defined code provides the framework with knowledge regarding the semantics of the update applied to the application's schema. Thus, the framework and application are kept oblivious to instance adaptation, while the schema evolution is addressed transparently by the framework.

Thus, our framework (AOF4OOP) contrasts with AspOEv Rashid & Leidenfrost [2004, 2006], which is just focused on the flexibility of the applied strate-

gies. The AspOEv system clearly provides a powerful environment that enables a dynamic adoption of strategies of schema evolution and instance adaptation However, it has the following drawbacks:

- In the AspOEv system, programmers must be aware of low level database evolution details - The *Reflective Handlers* Rashid & Leidenfrost [2006], that handle mismatch exceptions, ensure database consistency but require programmers to have technical knowledge. This contrasts with the simplicity and expressiveness of our pointcut/advice constructs.

- The AOF4OOP framework does not require an additional language - Applications that rely on our prototype are written in their own programming language without requiring any additional one. Furthermore, the code that ensures database consistency, the conversion functions, is also written in the same base program language.

- The AOF4OOP framework clearly separates the application code and evolution code - AOF4OOP based applications only know a unique data schema, while being oblivious to any other schema versions. Meanwhile, Vejal programming language-based applications are aware of several versions of a same class. Vejal-based applications require maintenance every time a new class version is created.

Like AspOEv, our approach also enable the same flexibility and customization, in terms of applied strategies.

6.3.4 Expressiveness of conversion code

Regarding our pointcut/advice constructs, inside the body's *advice* (the user-defined conversion function), programmers can use local and non-local information pertaining to the target object being converted. We argue that our approach empowers the richness and expressiveness of user-defined conversion functions. However, we consider that non-local data is still somewhat limited, since it only allows access until the immediate level of the parent object. Solving this limita-

tion rises complex issues in orthogonal systems. We consider this an interesting issue which should be the subject of further research.

6.3.5 Avoiding helper programs

In this PhD thesis, we used two evolution scenarios (design correction in Section 4.2 and our geographical application in Section 5) to compare the support provided by our framework with the one in other systems, such as db4o [Paterson *et al.*, 2006], Versant [Corporation, 2010] and ObjectDB [Ltd, 2011]. In these two examples of schema evolution scenarios, if applications are supported by these systems, helper programs are required. This contrasts with our framework that just requires definitions based on our pointcut/advice constructs. The development of these helper programs penalize the work in terms of productivity because they are additional tasks for programmers. Additionally, our framework still enables bidirectional application compatibility, thus previous versions of the application continue to work.

6.3.6 Keeping Java Virtual Machine (JVM) unchanged

Aspect-oriented programming languages support the modular definition of crosscutting concerns through a join point model. The seamless integration of application and database environments, provided by the orthogonal persistence paradigm, gave access to all the system's joint points that need to be advised. Thus, in our system, the pointcut/advice mechanism provided by AspectJ enabled the modularization of the application's persistence and database evolution concerns. The AspectJ weaver enabled the injection of persistence behaviour into the application. Additionally, our database weaver had enabled the injection of new evolution behaviour into the framework. During the development of our framework, the main technical issues were overcome, hence keeping the JVM unchanged. However, the solution to some of the framework's limitations may involve changes in the JVM.

6.3.7 Limitations of the framework

Type Orthogonality - Our prototype enables object persistence in its multiple class versions. In order to store instances of a class in these versions, a class renaming strategy was adopted. Such solution, besides the impact in the system performance, rises implementation issues due to JVM restrictions.

Note that classes pertaining to the standard Java library can't be manipulated through reflection. Thus, in the current version of our implementation, important classes such as `ArrayList` are limited in terms of versioning. As workaround, in our prototype these classes are treated as non-versionable objects.

Performance - During the tests discussed in Section 4.3.2, as well as in our geographical application, we observed performance degradation when the framework is faced with objects in different versions than what is expected by the running application. Notice that our framework follows a lazy approach on instance adaptation. Although most of this performance degradation is unavoidable and inherent to the multi-version schema approach, many optimizations yet may be introduced in order to improve the overall system performance.

Some of the current prototype's limitations arise from the usage of an object-oriented database as an object repository. The db4o database provides high level mechanisms of interaction in a single version schema, not well adapted to all requirements of our framework. Although this technological option had facilitated and accelerated the development work, the prototype's performance is seriously compromised due to that decision.

Theoretically, a new specialized object store for our framework and meta-model would be able to enhance its overall performance and orthogonality, avoiding some reflective operations. The development of such object store is subject of future work. Considering the aspect-oriented system architectural approach, the existing storing *aspect*, discussed in Section 3.3.5.1, will facilitate such development. This *aspect* wraps all physical accesses to the data and metadata. Thus, only few system modifications will be required in order to use that new object store.

Concurrency and error recover - In our framework, each attribute of a persistent object is always updated within a single database transaction. Using this approach, database consistency is always ensured. If any error occurs, a framework's exception is thrown and the entire transaction, involving objects and meta-objects, is reverted.

Additionally, to this basic support for concurrency and error recover, our prototype still provides an API to delimit transactions in a multithreaded environment. However, our current prototype is based on a single JVM environment, not covering the distributed systems' requirements over several concurrent JVM. Thus, our current results are applicable to specific types of systems such as embedded systems, smart phone applications or others which are supported by a single JVM. In order to overcome this limitation, our framework requires additional object cache synchronization mechanisms across several JVM. These issues should be the subject of additional research work, originating a new line of research.

Database garbage collector - The database garbage collector is another component that has a significant impact on the systems' performance. The current implementation of a garbage collector in our prototype uses a very simple algorithm to release object references that lose their reachability. As workaround, this mechanism is invoked explicitly by the user. Notice that this option has a minor impact on performance. It is just a question of storage space. On the other hand, calling it automatically would introduce intolerable waiting states in the system's functioning. This part of the system should be subjected to enhancements and be fully integrated with that specially designed object store subsystem, which was previously mentioned.

Full implementation of the meta-model - Although the meta-model in which the prototype is based on considers the existence of Aspect Meta-Objects (AspMO), this type of meta-objects is not fundamental to our research work. Notice that our work is focused on database evolution, concurrency, error recovering and persistence in general. Besides, this category of meta-objects was discussed in earlier related research [Rashid & Pulvermueller, 2000; Rashid & Sawyer, 1999].

Thus, in the current development stage of our prototype, these meta-objects have not yet been implemented. For now, they are just part of our meta-model since its implementation is seen as work to be carried out in the future. The internal structure and representation of these meta-objects in the database should follow a similar approach to the UBMO meta-objects, hence enabling the reuse of the framework's weaver. As discussed in Section 3.2, these meta-objects represent *aspects* as a special class of objects, as well as the application's entity objects, which can be persistent, making the database aspect-oriented [Rashid & Pulvermueller, 2000; Rashid & Sawyer, 1999].

Graphical tools - Our database classloader module allows transparent access from the Java Virtual Machine and run-time weaver to any class version, using its name variation based on the renamed classes. It would also be interesting to develop an integration plug-in for the development tools (like Eclipse or Netbeans), enabling the inclusion of those class versions resident in the database as a special classpath. Currently, users must write the pointcut/advice constructs and conversion code in a text or XML editor. For them, this represents an additional cognitive overhead. An IDE integration will facilitate the task of writing these constructs.

6.4 Published material on the subject of this PhD thesis

The following has already been published since the beginning this research.

- **An aspect-oriented framework for orthogonal persistence** (*Pereira, Rui Humberto and Perez-Schofield, J.B.G.*, 2010).

  This paper presents the first version of AOP4OOP protytype, which provides applications with persistence services. This communication was presented at the 5th Iberian Conference on Information Systems and Technologies (CISTI). Language: English.

- **Orthogonal persistence in Java supported by Aspect-Oriented Programming and Reflection** (*Pereira, Rui Humberto and Perez-Schofield, J.B.G.*, 2011).

  This work addresses the persistence of parametric classes in orthogonal environments and how the AOF4OOF framework deals with the type erasure based on the Java approach. This communication was presented at the 6th Iberian Conference on Information Systems and Technologies (CISTI). Language: English.

- **Database Evolution on an Orthogonal Persistent Programming System - A Semi-Transparent Approach** (*Pereira, Rui Humberto and Perez-Schofield, J.B.G.*, 2012).

  In our third presence at the CISTI conference, the first research results on database evolution, as well as the second version of AOF4OOF framework were presented. This communication was presented at the 7th Iberian Conference on Information Systems and Technologies (CISTI). Language: English.

- **Towards a Flexible and Transparent Database Evolution** (*Pereira, Rui Humberto and Perez-Schofield, J.B.G.*, 2014). Book: Rocha, A.M. Correia, F..B. Tan & K..A. Stroetmann, eds., New Perspectives in Information Systems and Technologies, Volume 2, vol. 276 of Advances in Intelligent Systems and Computing, 23-33, Springer International Publishing. DOI: 10.1007/978-3-319-05948-8_3

  Additionally, a communication was presented at the 2014 World Conference on Information Systems and Technologies (WorldCIST 2014). Language: English.

- **Evolution of the application and database with aspects** (*Pereira, Rui Humberto and Perez-Schofield, J.B.G.*, 2014). DOI:10.5220/0004966903080313. This communication was presented at the 16th International Conference on Enterprise Information Systems (ICEIS). Language: English.

The following manuscript has been was revised based on the reviewers' first comments and submitted.

- **Modularizing application and database evolution - An aspect-oriented framework for orthogonal persistence**

  It is an article sent to the "Journal of Software: Practice and Experience" a prestigious international journal. The meta-model, framework and experimental results are deeply discussed in the article. Language: English.

References

Advani, D., Hassoun, Y. & Counsell, S. (2006). Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, 1713–1720, ACM, New York, NY, USA. iv, 189

Agesen, O., Freund, S.N. & Mitchell, J.C. (1997). Adding type parameterization to the java language. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, 49–65, ACM, New York, NY, USA. 14

Akit, M., Bergmans, L. & Vural, S. (1992). An object-oriented language-database integration model: The composition-filters approach. In O. Madsen, ed., *ECOOP 92 European Conference on Object-Oriented Programming*, vol. 615 of *Lecture Notes in Computer Science*, 372–395, Springer Berlin Heidelberg. 18

Al-Mansari, M. & Hanenberg, S. (2006). Path expression pointcuts: Abstracting over non-local object relationships in aspect-oriented languages. In R. Hirschfeld, A. Polze & R. Kowalczyk, eds., *NODe/GSEM*, vol. 88 of *LNI*, 81–96, GI. 30

Al-Mansari, M., Hanenberg, S. & Unland, R. (2007). Orthogonal persistence and aop: a balancing act. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '07, ACM, New York, NY, USA. xxv, 27, 29, 30

Alagic, S. & Nguyen, T. (2001). Parametric polymorphism and orthogonal persistence. In *Proceedings of the International Symposium on Objects and Databases*, 32–46, Springer-Verlag, London, UK, UK. 14, 17

Alagić, S. & Royer, M. (2008). Genericity in java: persistent and database systems implications. *The VLDB Journal*, **17**, 847–878. 16, 17, 77

Aldrich, J. (2004). Open modules: A proposal for modular reasoning in aspect-oriented programming. In *In Workshop on foundations of aspect-oriented languages*, 7–18. 22, 23

Atkinson, M. (2000). Persistence and java - a balancing act. *Proceedings of Objects and Databases, International Symposium at ECOOP 2000. Sophia Antipolis, France, June 2000. Published as Lecture Notes in Computer Science, (Dittrich, KR et al Eds). Volume No. 1944.*, 1–31. 3, 9, 12

Atkinson, M. & Morrison, R. (1995). Orthogonally persistent object systems. *The VLDB Journal*, **4**, 319–402. iv, v, 3, 9, 11, 57, 61, 190

Atkinson, M.P. & Jordan, M.J. (1999). Issues raised by three years of developing pjama: An orthogonally persistent platform for java. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, 1–30, Springer-Verlag, London, UK, UK. 10, 12, 14, 109, 111

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, P.W. & Morrison, R. (1983). An approach to persistent programming. *The Computer Journal*, **26**, 360–365. 9, 14

Atkinson, M.P., Jordan, M.J., Daynès, L. & Spence, S. (1996). Design issues for persistent java: A type-safe, object-oriented, orthogonally persistent system. In *POS*, 33–47. 109, 111

Balzer, S. (2005). Contracted persistent object programming. In *PhD Workshop, ECOOP 2005*. 12

Banerjee, J., Kim, H.J., Kim, W. & Korth, H.F. (1986). Schema evolution in object-oriented persistent databases. In H.F. Korth, ed., *XP / 7.52 Work-*

shop on Database Theory, University of Texas at Austin, TX, USA, August 13-15, 1986. 34, 118

Banerjee, J., Chou, H.T., Garza, J.F., Kim, W., Woelk, D., Ballou, N. & Kim, H.J. (1987a). Data model issues for object-oriented applications. *ACM Trans. Inf. Syst.*, **5**, 3–26. 33, 34, 36

Banerjee, J., Kim, W., Kim, H.J. & Korth, H.F. (1987b). Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, 311–322, ACM, New York, NY, USA. 34, 35, 36, 42, 43, 75

Bank, J.A., Myers, A.C. & Liskov, B. (1997). Parameterized types for java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, 132–145, ACM, New York, NY, USA. 14

Benatallah, B. (1999). A unified framework for supporting dynamic schema evolution in object databases. In *Proceedings of the 18th International Conference on Conceptual Modeling*, ER '99, 16–30, Springer-Verlag, London, UK. 34, 45, 46

Bjornerstedt, A. & Britts, S. (1988). Avance: an object management system. *SIGPLAN Not.*, **23**, 206–221. 33

Blackburn, S. & Zigman, J.N. (1999). Concurrency - the fly in the ointment? In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3): Advances in Persistent Object Systems*, 250–258, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 10, 11, 108, 109, 146

Bloch, J. (2004). Jsr 175: A metadata facility for the java programming language. `http://jcp.org/en/jsr/detail?id=175`. 69

Bodden, E., Tanter, E. & Inostroza, M. (2014). Join point interfaces for safe and flexible decoupling of aspects. *ACM Trans. Softw. Eng. Methodol.*, **23**, 7:1–7:41. 22

Boyapati, C., Liskov, B., Shrira, L., Moh, C.H. & Richman, S. (2003). Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, **38**, 403–417. 50

Bracha, G., Odersky, M., Stoutamire, D. & Wadler, P. (1998). Making the future safe for the past: adding genericity to the java programming language. *SIGPLAN Not.*, **33**, 183–200. 14, 15

Cabana, B., Alagić, S. & Faulkner, J. (2004). Parametric polymorphism for java: is there any hope in sight? *SIGPLAN Not.*, **39**, 22–31. 16, 17, 99

Campbell, R.H. & Habermann, A.N. (1974). The specification of process synchronization by path expressions. In *Operating Systems, Proceedings of an International Symposium*, 89–102, Springer-Verlag, London, UK, UK. 30

Carey, M.J., DeWitt, D.J. & Naughton, J.F. (1993). The 007 benchmark. *SIGMOD Rec.*, **22**, 12–21. xi, 112, 127, 129, 195

Castagna, G. (1995). Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst.*, **17**, 431–447. 14

Castor, F., Cacho, N., Figueiredo, E., Garcia, A., Rubira, C.M.F., de Amorim, J.S. & da Silva, H.O. (2009). On the modularization and reuse of exception handling with aspects. *Software: Practice and Experience*, **39**, 1377–1417. 31

Chiba, S. (1998). Javassist – a reflection-based programming wizard for java. In *Proceedings of the Workshop on Reflective Programming in C++ at the 13th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, Vancouver, Canada, http://www.csg.is.titech.ac.jp/∼chiba/oopsla98/proc/chiba.pdf. 85, 114

Chrysanthis, P.K. & Ramamritham, K. (1990). Acta: A framework for specifying and reasoning about transaction structure and behavior. In *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, 194–203. 31

Clamen, S.M. (1992). Type evolution and instance adaptation. Tech. rep., Pittsburgh, PA, USA. vi, 33, 42, 45, 61, 62, 63, 77, 192

Clifton, C. & Leavens, G.T. (2002). Observers and assistants: A proposal for modular aspect-oriented reasoning. In *In FOAL Workshop*. 23

Clifton, C. & Leavens, G.T. (2003). Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *IN: SPLAT*. 23, 103

Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Morrison, R. (1994). Using persistence technology to control schema evolution. In *Proceedings of the 1994 ACM Symposium on Applied Computing*, SAC '94, 441–446, ACM, New York, NY, USA. 54

Cook, W.R. & Rai, S. (2005). Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 97–106, ACM, New York, NY, USA. 12, 82

Cook, W.R. & Rosenberger, C. (2005). Native queries for persistent objects, a design white paper. 12, 82

Cooper, T. & Wise, M. (1996). Critique of orthogonal persistence. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, IWOOOS '96, 122–, IEEE Computer Society, Washington, DC, USA. 11

Corporation, V. (2010). Versant object database fundamentals manual. http://developer.versant.com/developer/resources/objectdatabase/documentation/VODFundamentals.pdf, online: accessed 2-April-2014. xii, 150, 196

Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindstrom, A., Rosenberg, J. & Vaughan, F. (1994). Grasshopper: an orthogonally persistent operating system. *Comput. Syst.*, **7**, 289–312. 3, 14

Dearle, A., Hulse, D. & Farkas, A. (1996). Persistent operating system support for java. In *IN PROCEEDINGS OF THE FIRST INTERNATIONAL WORKSHOP ON PERSISTENCE AND JAVA*. 14

Dearle, A., Kirby, G.N.C. & Morrison, R. (2010). Orthogonal persistence revisited. *CoRR*, **abs/1006.3448**. iv, v, 3, 9, 190

Dijkstra, E.W. (1976). *A Discipline of Programming*. Prentice-Hall. 18

Dmitriev, M. (1998). The first experience of class evolution support in pjama. In *THE THIRD INTERNATIONAL WORKSHOP ON PERSISTENCE AND JAVA*. 39, 47

Dmitriev, M. & Atkinson, M. (1999). Evolutionary data conversion in the pjama persistent language. In *In 1st ECOOP Workshop on ObjectOriented Databases*, 25–36. 39, 43, 47

Dmitriev, M. & Hamilton, C. (2001). Towards scalable and recoverable object evolution for the pjama persistent platform. In *Proceedings of the International Symposium on Objects and Databases*, 61–70, Springer-Verlag, London, UK. 39

Fabry, J. (2005). *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*. Ph.D. thesis, Vrije Universiteit Brussel. v, 108, 190

Fabry, J., ric Tanter & DHondt, T. (2008). Kala: Kernel aspect language for advanced transactions. *Science of Computer Programming*, **71**, 165 – 180. 31

Fayad, M. & Schmidt, D.C. (1997). Object-oriented application frameworks. *Commun. ACM*, **40**, 32–38. 25

Ferrandina, F., Meyer, T. & Zicari, R. (1994). Implementing lazy database updates for an object database system. In *Proceedings of the 20th*

International Conference on Very Large Data Bases, VLDB '94, 261–272, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 43, 48, 49

Ferrandina, F., Meyer, T., Zicari, R., Ferran, G. & Madec, J. (1995). Schema and database evolution in the o2 object database system. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, 170–181, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 33, 34, 36, 46

Filman, R.E. (2001). What is aspect-oriented programming, revisited. Tech. rep. 21

Filman, R.E. & Friedman, D.P. (2000). Aspect-oriented programming is quantification and obliviousness. Tech. rep. 20, 21, 71

Gabriel, R.P., Jr., G.L.S., Steimann, F., Waldo, J., Kiczales, G. & Sullivan, K.J. (2006a). Aspects and/versus modularity the grand debate. In P.L. Tarr & W.R. Cook, eds., *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, 935–936, ACM. 22

Gabriel, R.P., Steele, G.L., Jr., Steimann, F., Waldo, J., Kiczales, G. & Sullivan, K. (2006b). Aspects and/versus modularity the grand debate. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, 935–936, ACM, New York, NY, USA. 21

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2005). *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional. 14, 15, 16, 18, 67, 69, 81, 83

Gray, J. & Reuter, A. (1992). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. 109

Gudmundson, S. & Kiczales, G. (2001). Addressing practical software development issues in aspectj with a pointcut interface. In *In Advanced Separation of Concerns*. 23

Hanenberg, S. (2000). Multi-design application frameworks. 26

Harrison, W. & Ossher, H. (1993). Subject-oriented programming: a critique of pure objects. *SIGPLAN Not.*, **28**, 411–428. 18

Hoffman, K. & Eugster, P. (2013). Trading obliviousness for modularity with cooperative aspect-oriented programming. *ACM Trans. Softw. Eng. Methodol.*, **22**, 22:1–22:46. 22, 23

Hohenstein, U. (2005). Using aspect-orientation to add persistency to applications. In *BTW*, 235–244. 27

Inc., J. (2009). Jboss aop - user guide - the case for aspects - v2.0.0. http://docs.jboss.org/jbossaop/docs/2.0.0.GA/docs//aspect-framework/userguide/en/pdf/jbossaop_userguide.pdf. 24

Jie, S. (2010). Applying aop in the evolution of object-oriented database. In *Computer and Information Application (ICCIA), 2010 International Conference on*, 64–66. 51

Johnson, R.E. (1997). Components, frameworks, patterns. *SIGSOFT Softw. Eng. Notes*, **22**, 10–17. 26

Kersten, M. & Murphy, G.C. (1999). Atlas: A case study in building a web-based learning environment using aspect-oriented programming. *SIGPLAN Not.*, **34**, 340–352. 20

Kiczales, G. & Mezini, M. (2005). Aspect-oriented programming and modular reasoning. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, 49–58. 22, 23, 29

Kiczales, G. & Rivieres, J.D. (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA. 18

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M. & Irwin, J. (1997). Aspect-oriented programming. In M. Aksit & S. Matsuoka, eds., *ECOOP'97 Object-Oriented Programming*, vol. 1241 of *Lecture Notes in Computer Science*, 220–242, Springer Berlin Heidelberg. iv, 18, 22, 103, 190

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W.G. (2001a). An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, 327–353, Springer-Verlag, London, UK, UK. 20, 24, 92

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W.G. (2001b). An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, 327–353, Springer-Verlag, London, UK, UK. 22

Kienzle, J. (2001). *Open Multithreaded Transactions - A Transaction Model for Concurrent Object-Oriented Programming*. Ph.D. thesis, École Polytechnique Fédérale de Lausanne, Lausanne,Switzerland. 32

Kienzle, J. & Guerraoui, R. (2002). Aop: Does it make sense? the case of concurrency and failures. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, 37–61, Springer-Verlag, London, UK, UK. v, 30, 31, 106, 108, 190

Kienzle, J., Duala-Ekoko, E. & Gélineau, S. (2009). Transactions on aspect-oriented software development v. chap. AspectOptima: A Case Study on Aspect Dependencies and Interactions, 187–234, Springer-Verlag, Berlin, Heidelberg. v, 32, 108, 190

Kim, W. & Chou, H.T. (1988). Versions of schema for object-oriented databases. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, 148–159, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 33

Kirby, G.N.C., Connor, R.C.H., Morrison, R. & Stemple, D. (1996). Using reflection to support type-safe evolution in persistent systems. Tech. rep., University of St Andrews. 54

Kusspuswami, S., Palanivel, K. & Amouda, V. (2007). Applying aspect-oriented approach for instance adaptation for object-oriented databases. In *Proceedings of the 15th International Conference on Advanced Computing and Communications*, 35–40, IEEE Computer Society, Washington, DC, USA. iv, xiii, 48, 51, 55, 56, 147, 190, 197

Lautemann, S.E. (1997). Schema versions in object-oriented database systems. In R.W. Topor & K. Tanaka, eds., *DASFAA*, vol. 6 of *Advanced Database Research and Development Series*, 323–332, World Scientific. 33

Leavens, G.T. & Clifton, C. (2007). Multiple concerns in aspect-oriented language design: A language engineering approach to balancing benefits, with examples. In *Proceedings of the 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies*, SPLAT '07, ACM, New York, NY, USA. xii, 21, 22, 196

Lee, S.W., Ahn, J.H. & Kim, H.J. (2006). A schema version model for complex objects in object-oriented databases. *J. Syst. Archit.*, **52**, 563–577. 34

Lerner, B.S. & Habermann, A.N. (1990). Beyond schema evolution to database reorganization. *SIGPLAN Not.*, **25**, 67–76. 33, 35, 46

Liskov, B., Adya, A., Castro, M., Ghemawat, S., Gruber, R., Maheshwari, U., Myers, A.C., Day, M. & Shrira, L. (1996). Safe and efficient sharing of persistent objects in thor. *SIGMOD Rec.*, **25**, 318–329. 3, 14

Loomis, M.E.S. (1992). Object versioning. *J. Object Oriented Program.*, **4**, 40–43. 52

Lopes, C.V. (1997). *D: A Language Framework for Distributed Programming.*. PhD in computer science, College of Computer Science, Northeastern University. 19

Lorenz, D.H. (1998). Visitor beans: An aspect-oriented pattern. In *Workshop ion on Object-Oriented Technology*, ECOOP '98, 431–432, Springer-Verlag, London, UK, UK. 19

Ltd, O.S. (2011). Objectdb 2.3 developer's guide. http://www.objectdb.com, online: accessed 2-April-2014. xii, xxv, 3, 37, 39, 150, 196

Marquez, A., Blackburn, S.M., Mercer, G. & Zigman, J. (2000). Implementing orthogonally persistent java. In *In Proceedings of the Workshop on Persistent Object Systems (POS*, 218–232. 3, 10, 14

Martin, R.C. (1994). Oo design quality metrics - an analysis of dependencies. http://docs.jboss.org/jbossaop/docs/2.0.0.GA/docs//aspect-framework/userguide/en/pdf/jbossaop_userguide.pdf, available at: http://groups.google.com/group/comp.lang.c++/msg/b9c929f55c52e4a2? 25

Martin, R.C. (1996). The Dependency Inversion Principle. *The C++ Report*, available at: http://www.objectmentor.com/base.asp?id=40. xxv, 25, 26

Meyer, B. (1992). Applying "design by contract". *Computer*, **25**, 40–51. 69, 105, 145

Mezini, M. & Lieberherr, K. (1998). Adaptive plug-and-play components for evolutionary software development. *SIGPLAN Not.*, **33**, 97–116. 18

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, **17**, 348 – 375. 97

Monk, S. & Sommerville, I. (1992). A model for versioning of classes in object-oriented databases. In P. Gray & R. Lucas, eds., *Advanced Database Systems*, vol. 618 of *Lecture Notes in Computer Science*, 42–58, Springer Berlin / Heidelberg. 65

Monk, S. & Sommerville, I. (1993). Schema evolution in oodbs using class versioning. *SIGMOD Rec.*, **22**, 16–22. 33, 36, 44

Nettles, S. (1993). Implementing orthogonal persistence: A simple optimization using replicating collection. In *In Proceedings of the Third International Workshop on Object Orientation and Operating Systems*, 177–181. 10

Nettles, S. & O'Toole, J. (1996). A rollback technique for implementation orthogonal persistence. In *POS*, 120–127. 10

Neward, T. (2006). Interoperability happens - the vietnam of computer science. http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science. 9

Odersky, M. & Wadler, P. (1997). Pizza into java: translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, 146–159, ACM, New York, NY, USA. 16

Odersky, M., Runne, E. & Wadler, P. (2000). Two ways to bake your pizza - translating parameterised types into java. In *Selected Papers from the International Seminar on Generic Programming*, 114–132, Springer-Verlag, London, UK, UK. 14, 16

Ortin, F., Lopez, B. & Perez-Schofield, J.B.G. (2004). Separating adaptable persistence attributes through computational reflection. *IEEE Softw.*, **21**, 41–49. 27

Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, **15**, 1053–1058. 23

Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers, Pragmatic Bookshelf, 1st edn. 115

Paterson, J., Edlich, S., Hrning, H. & Hrning, R. (2006). *The Definitive Guide to db4o*. Apress, Berkely, CA, USA. xii, 3, 37, 58, 115, 150, 196

Pereira, R.H. & Perez-Schofield, J. (2010). An aspect-oriented framework for orthogonal persistence. In *Information Systems and Technologies (CISTI), 2010 5th Iberian Conference*, 1 –6. 13, 57, 78

Pereira, R.H. & Perez-Schofield, J.B.G. (2011). Orthogonal persistence in java supported by aspect-oriented programming and reflection. In *Information Systems and Technologies (CISTI), 2011 6th Iberian Conference*, 1 –6. 13, 57, 78, 99, 102

Pereira, R.H. & Perez-Schofield, J.B.G. (2012). Database evolution on an orthogonal persistent programming system - a semi-transparent approach. In *Information Systems and Technologies (CISTI), 2012 7th Iberian Conference*, 1 –6. 13, 57, 78

Pereira, R.H. & Perez-Schofield, J.G. (2014a). Evolution of the application and database with aspects. In S. Hammoudi, L.A. Maciaszek & J. Cordeiro, eds., *ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems, Volume 1, Lisbon, Portugal, 27-30 April, 2014*, 308–313, SciTePress. 13, 57, 78

Pereira, R.H. & Perez-Schofield, J.G. (2014b). Towards a flexible and transparent database evolution. In l. Rocha, A.M. Correia, F..B. Tan & K..A. Stroetmann, eds., *New Perspectives in Information Systems and Technologies, Volume 2*, vol. 276 of *Advances in Intelligent Systems and Computing*, 23–33, Springer International Publishing. 13, 57, 72, 78

Perez-Schofield, J.B.G., Rosell, E.G., Cooper, T.B. & Cota, M.P. (2002). Managing schema evolution in a container-based persistent system. *Softw. Pract. Exper.*, **32**, 1395–1410. 47

Perez-Schofield, J.B.G., Garca Rosell, E., Ortn Soler, F. & Prez Cota, M. (2008). Visual zero: A persistent and interactive object-oriented programming environment. *J. Vis. Lang. Comput.*, **19**, 380–398. 3, 14

Piccioni, M., Oriol, M. & Meyer, B. (2011). Schema evolution for persistent object-oriented software: Model, empirical study, and automated support. *CoRR*, **abs/1103.0711**. iv, 189

Popovici, A., Gross, T. & Alonso, G. (2002). Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on*

Aspect-oriented software development, AOSD '02, 141–147, ACM, New York, NY, USA. 20

Przybylek, A. (2011). Systems evolution and software reuse in object-oriented programming and aspect-oriented programming. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns*, TOOLS'11, 163–178, Springer-Verlag, Berlin, Heidelberg. 24

Przybylek, A. (2013). Quasi-controlled experimentations on the impact of aop on software comprehensibility. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, 253–262, IEEE Computer Society, Washington, DC, USA. xii, 22, 23, 148, 196

Ra, Y.G. & Rundensteiner, E.A. (1997). A transparent schema-evolution system based on object-oriented view technology. *IEEE Trans. on Knowl. and Data Eng.*, **9**, 600–624. 34, 118

Radenski, A., Furlong, J. & Zanev, V. (2008). The java 5 generics compromise orthogonality to keep compatibility. *J. Syst. Softw.*, **81**, 2069–2078. 16, 17

Ramos, R.A., Camargo, V., Penteado, R. & Masiero, P.C. (2004). Reuso da implementação orientada a aspectos do padrão de projeto camada de persistência. *The Fourth Latin American Conference on Pattern Languages of Programming-SugarLoafPLoP, Fortaleza-CE*. 27

Rashid, A. (1998). Sades - a semi-autonomous database evolution system. In *Workshop ion on Object-Oriented Technology*, ECOOP '98, 24–25, Springer-Verlag, London, UK. 5, 48, 51, 118

Rashid, A. (2001). On to aspect persistence. In G. Butler & S. Jarzabek, eds., *Generative and Component-Based Software Engineering*, vol. 2177 of *Lecture Notes in Computer Science*, 26–36, Springer Berlin Heidelberg. 5, 27, 29

Rashid, A. (2002). Aspect-oriented schema evolution in object databases: A comparative case study. In *Workshop on Unanticipated Software Evolution*. xxvi, 5, 117, 118, 119, 120, 122

Rashid, A. & Chitchyan, R. (2003). Persistence as an aspect. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, 120–129, ACM, New York, NY, USA. 5, 27, 28, 29, 30

Rashid, A. & Leidenfrost, N.A. (2004). Supporting flexible object database evolution with aspects. In G. Karsai & E. Visser, eds., *GPCE*, vol. 3286 of *Lecture Notes in Computer Science*, 75–94, Springer. iv, x, 5, 51, 52, 53, 72, 147, 148, 190, 194

Rashid, A. & Leidenfrost, N.A. (2006). Vejal: An aspect language for versioned type evolution in object databases. Workshop on Linking Aspect Technology and Evolution (held in conjunction with AOSD). iv, xiii, 5, 51, 52, 54, 147, 148, 149, 190, 197

Rashid, A. & Moreira, A. (2006). Domain models are not aspect free. In O. Nierstrasz, J. Whittle, D. Harel & G. Reggio, eds., *Model Driven Engineering Languages and Systems*, vol. 4199 of *Lecture Notes in Computer Science*, 155–169, Springer Berlin Heidelberg. 21, 22

Rashid, A. & Pulvermueller, E. (2000). From object-oriented to aspect-oriented databases. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*, DEXA '00, 125–134, Springer-Verlag, London, UK. 5, 62, 152, 153

Rashid, A. & Sawyer, P. (1999). Dynamic relationships in object oriented databases: A uniform approach. In T. Bench-Capon, G. Soda & A. Tjoa, eds., *Database and Expert Systems Applications*, vol. 1677 of *Lecture Notes in Computer Science*, 816–816, Springer Berlin / Heidelberg. vi, x, 5, 52, 63, 65, 147, 152, 153, 191, 195

Rashid, A. & Sawyer, P. (2001). Aspect-orientation and database systems: an effective customisation approach. *IEE Proceedings - Software*, **148**, 156–164. iv, 5, 53, 190

Rashid, A. & Sawyer, P. (2005). A database evolution taxonomy for object-oriented databases. *Journal of Software Maintenance and Evolution: Research and Practice*, **17**, 93–141. iv, 5, 35, 36, 51, 75, 190

Rashid, A., Moreira, A. & Tekinerdogan, B. (2004). Special issue of early aspects: Aspect-oriented requirements engineering and architecture design. *IEE Proceedings: Software*, **151**, 153–156. 22

Roddick, J.F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, **37**, 383–393. 34

Skarra, A.H. & Zdonik, S.B. (1986). The management of changing types in an object-oriented database. *SIGPLAN Not.*, **21**, 483–495. 42, 44, 118

Soares, S. & Borba, P. (2007). Towards reusable and modular aspect-oriented concurrency control. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, 1293–1294, ACM, New York, NY, USA. 27, 30

Soares, S., Borba, P. & Laureano, E. (2006). Distribution and persistence as aspects. *Software: Practice and Experience*, **36**, 711–759. 27, 29, 30

Solorzano, J.H. & Alagić, S. (1998). Parametric polymorphism for java: a reflective solution. *SIGPLAN Not.*, **33**, 216–225. 14, 17

Song, Y., Peng, X., Xing, Z. & Zhao, W. (2012). Automatic adaptation of software applications to database evolution by graph differencing and aop-based dynamic patching. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, 111–118. 51

Steimann, F. (2004). Aspects are technical, and they are few. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS04)*, Berlin, Germany. 21, 22

Steimann, F. (2005). Domain models are aspect free. In *MODELS/UML 2005 (SPRINGER*, 171–185, Springer. 21, 29, 54

Steimann, F. (2006). The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, 481–497, ACM, New York, NY, USA. 21

Störzer, M. & Koppen, C. (2004). Pcdiff: Attacking the fragile pointcut problem, abstract. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany. 24, 29

Tekinerdogan, B., Moreira, A., ao Araújo, J. & Clements, P. (2004). Early aspects: aspect-oriented requirements engineering and architecture design. 22

Tresch, M. & Scholl, M.H. (1992). Meta object management and its application to database evolution. In *In Proc. 11th Int'l Conf. Entity-Relationship Approach*, 299–321, Springer. 36

Vieira de Camargo, V. (2006). *Crosscutting framweorks: definitions, classifications, architecture and using in a software development process*. PhD in computer science, Instituto de Cincias Matemticas e de Computao - USP, Brasil. 26, 27

Zdonik, S.B. (1987). Object-oriented type evolution. In F. Bancilhon & P. Buneman, eds., *Advances in Database Programming Languages, Papers from DBPL-1, September 1987, Roscoff, France*, 277–288, ACM Press / Addison-Wesley. 44

Zicari, R. (1991). A framework for schema updates in an object-oriented database system. In *Data Engineering, 1991. Proceedings. Seventh International Conference on*, 2 –13. 1, 35, 42

Zigman, J. & Blackburn, S.M. (1998). Java finalize method, orthogonal persistence and transactions. In *3 rd International Workshop on Persistence and Java (PJW3*. 10

Appendix A - AOF4OOP Annotation Reference Guide

In this appendix the annotations supported by the AOF4OOP framework are presented. These annotations are organized in three groups, as follows:

Class Annotations

`@Aof4oopVersionAlias` - This annotation has as its main goal to make the class version identification as humanly readable as possible. Although the framework is capable of transparently calculating a class version identifier, this annotation can force the calculus process in order for it to become an alias.
Required arguments:
`alias`: class version identifier alias
Optional arguments:
None
Example:

```
@Aof4oopVersionAlias(alias = "A")
```

Attributes Initialization

`@Aof4oopDefault` - In general, objects' attributes are initialized with a fixed default value. When values are not explicitly assigned to attributes in the class constructor, the numerical ones are initialized with zero while strings or references

to complex structures, with null values. This behaviour is not always the desirable one, thus, this annotation enables the explicit definition of how an attribute must be initialized during the conversion process (at emulation or physical conversion). Notice that these definitions are superposed to the ones in the class's constructor.
Required arguments:
`value`: attribute value
Optional arguments:
None
Example:

```
@Aof4oopDefault(value = "Student")
```

Attribute Constraints
Integrity constraint definitions may change from one class version to another. Using this annotation, the framework ensures that attributes always have values that pertain to their domain of values in the class version. These constraints are applied in the application scope, during normal application functioning, as well as during the database evolution process, reinforcing data consistency.

`@Aof4oopConstraintNotNull` - Throws the `EIntegrityFault` framework's exception when the attribute value is null.
Required arguments:
`message`: exception message given when the constraint is violated
Optional arguments:
None
Example:

```
@Aof4oopConstraintNotNull(message = "Invalid name")
```

`@Aof4oopConstraintCheck` - Throws the `EIntegrityFault` framework's exception when the attribute value is out of the allowed domain. Currently, the framework just allows this type of constraint in numerical attributes.
Required arguments:
`message`: exception message given when the constraint is violated

`expression`: specification of the allowed domain values

Optional arguments:

None

Example:

```
@Aof4oopConstraintCheck(message = "Invalid age", expression=">0")
```

Appendix B - Pointcut/Advice Construct reference Guide

In this appendix we present a reference guide to the pointcut/advice constructs and UBMO meta-objects. These meta-objects are stored into the UBMO meta-objects' space in the system's database, which is based on an XML file. This XML repository has <aof4oop> as top level root node. Each one of its child nodes <ubmo> define a UBMO meta-object. The <ubmo> nodes are composed by two data parts: matching and action. That is to say, in the UBMO meta-objects' space is stored the system's instance adaptation *aspect* (the entire space), that is composed by its *advices* (the UBMO meta-objects).

In the following listing we present an example of this XML based database, which includes its Document Type Definition (DTD). This DTD ensures the UBMO meta-objects' consistency in order to avoid run-time errors.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE AOF4OOP [
  <!ATTLIST AOF4OOP version CDATA #REQUIRED>               <!-- version of UBMO database -->
  <!ELEMENT AOF4OOP (ubmo+)>                               <!-- many ubmo elements are allowed -->
  <!ATTLIST ubmo name CDATA #REQUIRED>                     <!-- unique token name used in ubmo -->
  <!ATTLIST ubmo matchOldClassVersion CDATA #IMPLIED>      <!-- optional token for matching Old Class Version of the target object -->
  <!ATTLIST ubmo matchCurrentClassVersion CDATA #IMPLIED>  <!-- optional token for matching target's current Class Version -->
  <!ATTLIST ubmo matchClassName CDATA #IMPLIED>            <!-- optional token for matching target's Class Name -->
  <!ATTLIST ubmo matchSuperClassName CDATA #IMPLIED>       <!-- optional token for matching target's Super Class Name -->
  <!ATTLIST ubmo matchParentMember CDATA #IMPLIED>         <!-- optional token for matching target's parent member name -->
  <!ATTLIST ubmo matchParentClassName CDATA #IMPLIED>      <!-- optional token for matching target's parent class name -->
  <!ATTLIST ubmo matchParentClassVersion CDATA #IMPLIED>   <!-- optional token for matching target's parent class version -->
  <!ATTLIST ubmo matchOldParentClassVersion CDATA #IMPLIED> <!-- optional token for matching old parent class name of the target object -->

  <!ELEMENT ubmo  (conversion)>
```

```
  <!ELEMENT conversion (sourceCode)>
  <!ELEMENT sourceCode (#PCDATA)>                          <!-- plain source code in a programming language -->
  <!ATTLIST conversion outputClassName CDATA #REQUIRED>    <!-- conversion function's output class name -->
  <!ATTLIST conversion applyDefault (true|false) "true">   <!-- Should apply default conversion -->
  <!ATTLIST conversion loadingDepth (1|2|3|4) "1">         <!-- Depth of loading when activating target object from DB-->
  <!ATTLIST conversion loadingExcludeMembers CDATA #IMPLIED><!-- Members to be excludes for loading-->
]>
<AOF4OOP version="1.0">
    <ubmo name="ConvTutor$A_to_V"
         matchOldClassVersion="A"
         matchCurrentClassVersion="V"
         matchClassName="rhp.aof4oop.cs.datamodel.Tutor">
         <conversion applyDefault="false" outputClassName="rhp.aof4oop.cs.datamodel.Tutor">
           <sourceCode>
             <![CDATA[
                ....
             ]]>
           </sourceCode>
         </conversion>
   </ubmo>
   <ubmo>
       ....
   </ubmo>
<AOF4OOP>
```

Base definitions

`name` - This is a mandatory parameter which enables the identification of the pointcut and *advice*. Since it is used in the generation of the name of the woven class, which implements the *advice*, this name is very useful for debugging purposes. Thus, in the case of any runtime error, by knowing this name the programmer can understand the error's localization.

Matching

`matchClassName` - Defines the class canonical name of the advised classes. The emulated/converted (target) objects whose classes match this parameter are advised. This parameter allows the * wildcard. If the class name is followed by `[]`, the target object must be an array of this class.

`matchSuperClassName` - Defines the super class canonical name of the advised

classes in the application's class version. Matching through a super class is meant to reduce the number of user-definitions. All target objects that share this common super class are advised. This parameter allows the * wildcard.

`matchOldClassVersion` - Specifies the class version of the persistent object, at the database, that is converted to the application's class version. This parameter is required when the target class already has more than one version. In these cases, we must ensure a correct source class version. This parameter allows the * wildcard and `or` operator.

`matchCurrentClassVersion` - Defines the class version identifier in the running application. This parameter is required when the target class already has more than one version. In these cases, we must ensure a correct target application's class version. This parameter allows the * wildcard and `or` operator.

`matchParentClassName` - This contextual parameter enables advising objects, which, at runtime, are pointed from an object of a certain class. That is to say, it enables the identification of the object's parent. This parameter allows the * wildcard.

`matchParentMember` - This is another contextual parameter that enables advising objects, which, at runtime, are pointed from a certain object attribute. That is to say, it enables the identification of the attribute in the object's parent that contains its reference.

Action

`outputClassName` - This parameter is optional. It is required when the `matchSuperClassName` parameter is used. In such cases, the output class is ambiguous because a superclass may have several subclasses. This parameter specifies the type (class in the application) of the returned value. It is very useful when we intend to apply a unique construct to several classes that share a common superclass.

`applyDefault` - This boolean option only accepts a `true` or `false` value. If `true`, the default instance adaptation is applied before the user-defined instance adaptation. Thus, if just few differences exist between the two class versions, the user-defined conversion code can be minimized.

`sourceCode` - Contains plain Java source code that performs the conversion. As reserved words it has `oldObj`, `newObj`, `oldObjParent` and `newObjParent`, which are respectively references to the object to be converted, a pre-instantiated object in the current class version and their object parents. This Java source code must end with a return statement, which delivers the reference of the converted object. That is to say, this is the body of a Java function which receives these four arguments and returns the already adapted object (`oldObj`) to its current class version. The return type may be explicitly defined using the `outputClassName` parameter.

`loadingDepth` - During the conversion process, the objects (in the previous class version) that feed conversion functions are loaded from the database. This option enables the definition of the depth of this loading. Limiting the depth, unnecessary loadings are avoided. Notice that the framework follows the orthogonal principle of reachability, that enables access to unlimited depth in object hierarchy. This option, as well the next one, are concerned with performance issues. In future versions of the framework, this immediate loading will be replaced by another performed on demand (lazily), as required by the user-defined conversion code.

`loadingExcludeMembers` - This enables the exclusion of certain object attributes to be loaded. Notice that, some objects may have complex data structures, thus, its full loading should be avoided.

Appendix C - API Reference Guide

In this appendix, the current version of the framework's Application Programming Interface (API) is presented. Besides the access to the entire database, this API provides programmers with a set of operations to: manage transactions, querying, run-time reflection of parametric classes, get system's statistics and some utilities.

Persistence services

```
public <T> T getRootObject()
```

In the AOF4OOP framework, persistent objects are accessible by reachability. This method returns the reference to the database's default *root object*, which provides references to other reachable objects. It returns a generic type in order to enable type inference. The *root object* can be a single object or an array of objects.

```
public <T> T getRootObject(String rootName)
```

In order to enable multiple *root objects*, this method provides named references to the entry points in the object structures in the database. Invoking this method with a `null` name, the same object reference as `getRootObject()` is returned.

```
public Hashtable<String,Object> getRootObjects()
```

In some circumstances, programmers could need the knowledge of all existent *root objects*. This operation returns a `Hashtable<String,Object>` containing all of them and their associated names.

```
public synchronized void setRootObject(Object rootObject)
```

This method sets an object, given as argument, as the default *root object*. If the default *root object* is already defined, them it is replaced by this new one. The old *root object*, and all the objects only accessible through it, if are not reachable from other persistent objects, will be considered as deleted and collected by the garbage collector.

This method is marked as synchronized because it modifies the framework's internal shared data structures, in order to support multithreaded applications. As discussed in Section 3.3.4.1 and Section 3.5, the method synchronization is effectively done in its implementation as an aspect. The framework's persistence aspect is thread-safe.

```
public synchronized void setRootObject(String rootName,Object rootObject)
```

This operation has the same purpose as the previous one. However, it enables the definition of arbitrary named *root objects*. When the `rootName` argument is `null`, the default root is considered.

Transactions

```
public static long beginTransaction()
```

By calling this operation the autcommit mode is turned off. While autocommit mode is off, updates on persistent and non-persistent objects are recorded into a transaction log. This operation returns the identifier of the started trans-

action.
If a transaction is already in progress, the identifier of that previously started transaction is returned. In these cases, the nested transaction has no practical effect.

```
public static boolean commitTransaction()
```

This finishes the current transaction and turns on the autocommit mode. If called within the scope of a nested transaction, this operation returns false and the autocommit mode continues off. Only when within the main transaction does it return true and the entire transaction is committed.

Programmers must be aware of this transactional control approach. The use of a `try/catch/finally` control structure is strongly recommended.

```
public static boolean rollBackTransaction()
```

This operation rolls back all transaction. If called within the scope of a nested transaction, this operation returns false and no rollback is done. As in the commit operation, only when within the main transaction does it return true, and the entire transaction is rolled backed.

Programmers must be aware of this transactional control approach. The use of a `try/catch/finally` control structure is strongly recommended.

Querying services

```
public <T> List<T> query(IQuery query)
```

In order to easily obtain *ad hoc* object references, the framework provides this operation, which returns a `List` of references that meet the predicates in the argument given as a query object. This query object must implement the `IQuery` interface.

```
public boolean createView(String viewName,IQuery query)
```

The queries can be saved enabling their later usage through a view mechanism. This operation saves the query associated to a unique name.

```
public boolean dropView(String viewName)
```

When a view is not needed any more, it can be dropped using this operation.

```
public <T> List<T> view(String viewName)
```

A previously saved query as a view can be executed using this operation. It returns a `List` with references, as a `query` operation does.

Parametric class reflection

```
public String[] findRuntimeTypeParameters(Object obj)
```

By calling this operation, the run-time information of parameterized objects is obtained. It returns an array of strings which are all class type parameters used in the instantiation of the object given as an argument. The order in the array is the same as the order the parameters appear in the constructor statement.

Statistics and utilities

In orthogonal persistent systems, applications do not manage persistence. However, our API expose some of the framework's internal operations to the applications, such as the objects' identifiers. These mechanisms may be very useful for debugging purposes.

```
public void printStats()
```

Statistics regarding the object and meta-object caches as well as the overall system's state data is printed to the screen.

```
public void gc()
```

Runs the database's garbage collector.

```
public void gc(boolean verbose)
```

Runs the database's garbage collector. The `verbose` argument controls the operation verbosity, enabling additional information regarding the collected objects and meta-objects to be printed on the screen.

```
public long getLOID(Object object)
```

Gets the logical identifier (LOID) of an application object. If the given object reference is not a persistent object, this operation returns 0 (zero).

```
public boolean isPersistent(Object object)
```

Enables applications to find out if an object is persistent or not. If the given object as an argument is persistent, it returns true.

```
public boolean isCached(Object object)
```

Enables applications to find out if an object is cached. All persistent objects are put inside the system's cache. However, objects can be non-persistent and be cached. This happens when objects lose their reachability. That is to say, they continue to be cached although they are not reachable from any other persistent object. If the given object as an argument is cached, it returns true.

```
public boolean isReachable(Object object)
```

Enables applications to find out if an object is reachable. For applications, a reachable object also is persistent. However, in the framework's scope, first objects are detected as being reachable and then are made persistent. If the given

object as argument is reachable, it returns true. The main goal of this operation is for the internal use of the framework.

Appendix D - Extended abstract

Applications are subject of a continuous evolution process, having a profound impact on their underlining data model, hence requiring frequent updates in the applications' class structure and database structure as well. The database evolution problem, in object-oriented databases, has two parts each, a distinct layer: (1) schema evolution - at database metadata layer; and (2) instance adaptation - at data layer. This twofold problem, schema evolution and instance adaptation, usually known as database evolution, as well as the concurrency and error recover problems are addressed in this thesis with a novel meta-model and its aspect-oriented implementation.

The object-oriented database evolution problem has been intensively studied in the last decades, but still remains an error-prone and time-consuming undertaking for programmers and database administrators.

Modern object-oriented databases provide features that help programmers deal with the object persistence, as well as all related problems, such as database evolution, concurrency and error handling. In most systems there are transparent mechanisms to address the aforementioned problems. These mechanisms alleviate the programmers' workload by keeping the data and application layers synchronized. However, some type of updates such as class movements in the inheritance hierarchy, attribute renaming and semantic changes in attributes content, require a programmer's intervention in order to convert the data from the old structure to the new one. Thus, although these modern systems provide such transparent mechanisms, the most frequent types of class structure updates identified in earlier works [Advani *et al.*, 2006; Piccioni *et al.*, 2011] do not have support, requiring helper programs that sequentially read all objects, convert its content

and then store it again under the new class version. Additionally, in the case of a class that is moved within its inheritance hierarchy, these systems also require a temporary class, with a distinct name, which must be renamed at the end of the conversion process. Consequently, real world applications require substantial effort from programmers in order to evolve the database. This database evolution problem and all other problems related with persistence motivated us to find solutions to improve the programmer's work in order to be more productive and less prone to errors.

Earlier research works have demonstrated that Aspect-Oriented Programming (AOP) techniques [Kiczales *et al.*, 1997] enable the development of flexible and pluggable systems. As far as we aware, the main research work carried out in the field of object-oriented database evolution, applying AOP techniques, points to solutions that partially and, in some cases, totally solve the database evolution problem. The research work carried out by Rashid & Leidenfrost [2004, 2006]; Rashid & Sawyer [2001, 2005] has made an important contribution to the improvement of the flexibility and customization features of existing evolution techniques. The authors of this research have proposed an approach that allows the combination of all isolated techniques into one unique, powerful and integrated solution, applying AOP techniques. These earlier works showed that aspect-orientation of the database evolution enables high level of flexibility and customization. Nonetheless, neither this, nor any other work [Kusspuswami *et al.*, 2007], are focused on the orthogonal persistence paradigm [Atkinson & Morrison, 1995; Dearle *et al.*, 2010]. Regarding transaction management and failure control, Kienzle & Guerraoui [2002]; Kienzle *et al.* [2009] and Fabry [2005] showed that just syntactic obliviousness between the main program base and *aspect* is achievable.

This thesis addresses the database evolution problem, in particular the context of orthogonal object-oriented persistent systems in both its parts: schema evolution and instance adaptation. These systems, due to their intrinsic characteristics, have motivated our research work. We have found that some of these characteristics are distinctive and are not observed in non-orthogonal systems, hence enabling a novel approach in dealing with the aforementioned problem. The orthogonal persistence [Atkinson & Morrison, 1995; Dearle *et al.*, 2010], the-

oretically, provides the solution to problems in terms of transparency and native querying mechanisms that improve work productivity and quality. Additionally, this paradigm of data persistence also provides conditions for other approaches for database evolution, since its characteristics enable an incremental evolution of the application's schema and seamless application/database integration. Our contribution is focused on identifying the advantages and disadvantages of orthogonal persistence for that purpose. We argue that these characteristics provide special conditions for the modularization of persistence and database evolution concerns when applying AOP techniques. Thus, our research work intends to demonstrate the benefits of this combination: orthogonal persistence paradigm and AOP techniques. We also intend to propose a new aspect-oriented approach to modularize both aspects of the database evolution: technical and applications' domain. Thus, in this thesis we present our meta-model and its prototype, which explore the benefits of this combination.

Our meta-model and framework follow an aspect-oriented approach focused on the object-oriented orthogonal persistent context. It is characterized by its simplicity in order to achieve efficient and transparent database evolution mechanisms. Our meta-model was inspired by Rashid & Sawyer [1999]'s meta-model; however, it defines a much more limited set of meta-classes. These meta-classes represent the applications' classes, the relationships, and all the metadata required to emulate the applications' objects in every existing class versions. Important issues such as data consistency, integrity and object identity are taken into account in our meta-model proposal. In order to achieve these goals, a set of entities were defined: objects, meta-objects and meta-classes. In the meta-model there are six meta-classes: (1) CVMO - Class Version Meta-Object - Each CVMO supports the metadata of a class in a specific version; (2) RMO - Root Meta-Object - Each points to a persistent root object; (3) IMO - Instance Meta-object - This meta-object represents a logical object instance of applications, *i.e.* the application's object instances regardless their physical version in database; (4) AMO - Attribute Meta-Object - This meta-object represents a relationship between two objects, or an object to an array; (5) UBMO - Update/Backdate Meta-Objects - UBMO meta-objects support our pointcut/advice constructs in order to represent explicit mappings among class versions; (6) AspMO - Aspect

Meta-Objects - This kind of meta-object enables aspect storing. That is to say, it makes aspects persistent in the database. Besides these meta-objects, there are still data objects, which are the applications' objects stored in the database according to its class versions. The meta-objects form the metadata layer in the database. Notice that meta-objects are instances of meta-classes. On the other hand, objects form the data layer. Each one of them supports an application's data object (*facet* [Clamen, 1992]) according to a specific class version. Regarding object identity, each application's object instance has a Logical Object Identifier (LOID), while data objects and meta-objects in database have Object Identifiers (OID). We argue that our meta-model approach reduces the number of affected meta-objects (metadata) each time the application's class structure is updated.

Our meta-model supports multiple versions of a class structure by applying a class versioning strategy. Thus, enabling bidirectional application compatibility among versions of each class structure. That is to say, the database structure can be updated because earlier applications continue to work, as well as later applications that only known the updated class structure. The specific characteristics of orthogonal persistent systems, as well as a strategy of enrichment with metadata within the application's source code, complete the inception of the meta-model and have motivated our research work.

Our approach focuses on supporting programmers with persistence and database evolution services in a programming perspective rather than a database perspective. Thus, allowing him/her to only concentrate on the application's logic and class structure definition within the application scope.

In order to test the feasibility of our approach, we developed a prototype. Our prototype is a framework that mediates the interaction between applications and database, providing them with orthogonal persistence mechanisms. Applications rely on our framework in order to be provided with transparent mechanisms of data persistence. These mechanisms are introduced into applications as an *aspect* in the aspect-oriented sense. Objects do not need to extend any super class, implement interfaces or contain a particular annotation. The parametric type classes are also correctly handled by our framework. However, classes that belong to programming environment must be handled as not versionable due to restrictions imposed by the Java Virtual Machine (JVM).

Regarding the concurrency support, the framework provides the applications with a multithreaded environment which supports database transactions and error recover. Additionally, the framework provides programmers with an API for controlling concurrency. This API enables the delimiting of database transactions, as well as commit and rollback operations, within a programming perspective.

The system's architecture is aspect-oriented, which enables the modularization of some crosscutting concerns regarding applications and framework. The internal framework's functioning is aspect-oriented as well as the way applications are provided with the persistence aspect. This architecture enables an easy replacement of the framework's evolution mechanisms. Furthermore, this architecture also enables framework reusability due to its extensibility. A main module, a set of aspects, database and preprocessor compose the framework.

Our aspect-oriented framework enables the modularization of the applications' persistence concern as well as the database's evolution. Programmers can perform updates in the structure of the applications' classes, because our framework will create a new version of these structures in the database. Our schema evolution approach is incremental, *i.e.* new versions are always added to the metadata layer in the database. This incremental approach was enabled by the orthogonal persistence paradigm. This frees the programmer from having to intervene at the database level, avoiding the use of schema evolution primitives.

Regarding the instance adaptation problem, it is, in many cases, handled autonomously by the framework through its default mechanism based on direct mapping. Only when structural or semantic variations occur are default mechanisms unable to perform the adaptation of the instances autonomously. In these cases, programmers must write conversion code as aspect-oriented modules which extend the framework's default mechanisms. We develop a new kind of XML based pointcut/advice constructs to write these aspect-oriented modules. During the normal functioning of the applications, our framework adapts the objects loaded from the database using these mechanisms. This lazy approach of adaptation provides applications with instances of objects according to the structures of its internal classes.

Instance adaptation is a crosscutting concern of the classes that are the subject

of evolution. The domain model aspects of the instance adaptation concern were addressed with our pointcut/advice constructs, while the technical aspects use the framework's base mechanisms. Using our XML based pointcut/advice constructs, the framework's instance adaptation mechanism is extended, hence keeping the framework also oblivious to this problem.

These expressions are structured into two parts: trigger conditions and action. At runtime an action is triggered when the object instance being loaded from the database and the one required by the application satisfy the conditions of a construct. The action information is used to extend the framework's instance adaptation aspect with the required conversion behaviour. This user-defined code provides the framework with knowledge regarding the semantics of the update applied to the applications' class structures. These user-definitions establish an explicit mapping between two instances of an object in distinct class versions: the one in the database and the expected one by the application. We term this as User-Defined Mapping.

Although these modules have dependencies on the applications' source code, their interface with the framework is well-defined. Furthermore, all interfaces referenced in its implementation are statically checked by the database weaver. Thus, the framework and application are kept syntactically oblivious from instance adaptation problem, while the framework addresses class evolution transparently.

These constructs are written in XML language in order to enable easier editing using a graphical tool or, simply, a text editor. Each pointcut/advice construct is supported at the meta-object layer as an UBMO meta-object.

In our implementation, the action (the *advice*'s body) is written in Java programming language, which is embedded into an XML node. The conditions to trigger the action are specified through a group of matching parameters represented as attributes of a XML node. We also note that our approach does not require another programming language like Vegal [Rashid & Leidenfrost, 2004]. Programmers write conversion code in the same programming language used within the application's source code. Furthermore, inside the *advice*'s body (the user-defined conversion function) programmers can use local and non-local information pertaining to the target object being converted. We argue that our

approach empowers the richness and expressiveness of those conversion functions.

The potential developing gains provided by the prototype were benchmarked in a case study. Regarding our case study, the results confirm that mechanisms' transparency has positive repercussions on the programmer productivity, simplifying the entire evolution process at application and database levels.

The meta-model itself also was benchmarked in terms of complexity and agility. Although our meta-model was inspired in the one proposed by Rashid & Sawyer [1999], theoretically, it leads to better results in terms of complexity. We reused an earlier case study that compares four meta-models including the one of Rashid & Sawyer [1999]. We extended this case study by comparing our meta-model. Compared with the other three meta-models, the authors' meta-model requires less meta-object modifications in each schema evolution step. Our results, for that specific scenario of the case study, showed that our meta-model requires even less meta-objects, as well as less modified meta-objects. Thus, we believe that our meta-model is better adapted to support persistence services, in terms of data overhead, since it requires less meta-objects for representing applications' data, as well as updates to its schemas. Besides, the overall system performance also can be beneficiated due to meta-model's simplicity. However, our meta-model requires the intensive use of reflection affecting the system's performance, a cost that may not being compensated. The prototype's performance should be subject of future research work.

Another kind of test was done in order to validate prototype and meta-model robustness. In order to perform these tests, we used an OO7 [Carey *et al.*, 1993] small size database due to its data model complexity. Since the developed prototype offers some features that were not observed in other known systems, performance benchmarks were not possible. However, the developed benchmark is now available to perform future performance comparisons with equivalent systems.

Additionally, we developed a proof-of-concept application that allowed us to test our approach in a real world scenario. This proof of concept application uses data obtained from the online OpenStreetMap (http://www.openstreetmap.org) geographical database. The OpenStreetMap online geographical database allows a user-defined area to be exported through its coordinates. The OSM export files, in XML format (http://wiki.openstreetmap.org/wiki/OSM_XML), contains all

the data related with that geographical area. Each one contains the coordinates of the boundaries and it is structured as a set of objects such as *Nodes*, *Ways* and *Relations*. Our geographical application enables users to import those OSM files into the local database. Users can browse these locally stored geographical areas, hence having access to all the information (their points of interest).

This geographical application was initially developed without any implementation of the persistence concern. Applying a minimal change in its source code, we provide the application's objects with a persistence aspect. All remaining code of the application is kept unchanged, being oblivious to persistence concern and decoupled from the framework. Moreover, the persistence aspect does not violate the specified behaviour of the base modules, which makes it a *spectator* [Leavens & Clifton, 2007; Przybylek, 2013]. Despite providing applications with additional behaviour, if turned off they continue working oblivious to that lack of persistence.

This real world experience using our framework showed that applications remain oblivious regarding persistence and database evolution. In this case study, our framework showed to be a useful tool for programmers and database administrators.

The proposed approach aims to provide programmers with more transparent mechanisms for database evolution support, when compared with other systems [Corporation, 2010; Ltd, 2011; Paterson *et al.*, 2006] such as the ones referred to previously. In these system, programmers must write helper programs that perform data migration when structural or semantic variations occur. This contrasts with our framework where pointcut/advice constructs can be added into the system in order to extend its base mechanisms.

In earlier research, persistence and database evolution were effectively modularized. However, we argue that our approach overcomes the earlier approaches regarding the following issues:

- Enable programmers to be semi-oblivious regarding database evolution, since just semantic updates in the applications' schemas require the programmers' attention. Programmers can modify persistent data structures in the applications' source code, because the framework will produce a new database schema version. No primitives for schema evolution are required.

- Enable applications to be oblivious regarding its data persistence and evolution. Applications that rely on our framework continue to work, unchanged, even after their persistent data structures are updated. In Rashid *et al.*'s approach [Rashid & Leidenfrost, 2006], applications must be written in a special programming language capable of acknowledging all existing class versions.

- In regards to Rashid *et al.*'s meta-model, our approach to support the system's meta-objects layer is simpler and, thus, is well adapted to support higher levels of complexity in multi-version schemas.

- The approach proposed by Kusspuswami *et al.* [2007], implements *role aspects* as update/backdate *aspects*. Although this approach is, in its base, similar to the one proposed in this thesis, it does not offer a systematic interface to declare and implement aspects. Programmers must implement those aspects using AspectJ or another aspect-oriented programming language.

Performance issues and the single JVM instance are the major limitations of the framework.

Due to object version emulation, the adaptation to the applications' class versions introduces a significant delay (if the objects are not yet in cache). Although most of this performance degradation is unavoidable and inherent to the class versioning approach, many optimizations may yet be introduced in order to improve the overall system performance. We argue that the flexibility and customization enabled by the aspect-orientation of the database evolution provides means for other adaptation approaches. Such flexibility and customization provides the necessary means to apply the adjusted strategies to each case.

In terms of performance, some of the current prototype's limitations derive from the use of an object-oriented database as an object repository. This database provides high-level interaction mechanisms in a single version schema, and therefore it is not well adapted to our framework's requirements. Theoretically, a new specialized object store for our framework and meta-model would be able to enhance its overall performance and orthogonality, avoiding many additional

operations. We will consider this specialized object repository as a future improvement to be made to our framework.

The single JVM instance limitation arises from the orthogonal persistence paradigm where objects' persistence state depends on its reachability. Two instantiated objects in two distinct JVM do not share memory. Hence, if one of these objects is updated, the other one does not get synchronized. Memory synchronization and locking across several JVM are still unresolved issues that we intend to address in our future work.

Appendix E - Resumen

Las aplicaciones sufren un proceso de evolución continuo, con profundas influencias en el modelo de datos relacional, lo cual exige actualizaciones frecuentes tanto de la estructura de clases de la aplicación como de la estructura de la base de datos.

En esta tesis doctoral se aborda este doble problema, evolución de esquema y adaptación de las instancias, también conocido como evolución de la base de datos, así como los problemas de concurrencia y de recuperación de errores, para los que se propone un nuevo metamodelo y su implementación orientada a aspectos.

Las actuales bases de datos orientadas a objetos contienen funcionalidad que ayuda a los programadores a lidiar con la persistencia de objetos, así como con otros problemas relacionados, tales como la evolución de la base de datos, la concurrencia y el control de errores. Aunque la mayoría de los sistemas cuentan con mecanismos transparentes para atacar estos problemas, la evolución de la base de datos exige un gran nivel de intervención humana, que consume una buena parte del esfuerzo de los programadores y administradores de bases de datos.

Algunas investigaciones previas han demostrado que las técnicas de programación orientada a aspectos (POA) permiten el desarrollo de sistemas reutilizables y flexibles. En esos trabajos, se abordaban la evolución del esquema y la adaptación de instancias como problemas de la gestión de la base de datos. Sin embargo, ninguno de ellos se ocupaba de los sistemas de persistencia ortogonal. En la presente investigación se sostiene que las técnicas de POA son apropiadas para enfrentar estos problemas en los sistemas de persistencia ortogonal. Con

respecto a la concurrencia y a la recuperación de errores, la investigación previa solo muestra la posibilidad de transparencia sintática entre el programa de base y aspectos.

Nuestro metamodelo y su respectivo *framework* siguen un abordaje orientado a aspectos, centrado en el contexto de persistencia ortogonal orientado a objetos. Este metamodelo se caracteriza por su simplicidad para lograr mecanismos eficientes y transparentes de evolución de bases de datos. Nuestro metamodelo soporta múltiples versiones de una estructura de clases a través de la aplicación de una estrategia de versionado, lo cual permite la compatibilidad bidireccional de la aplicación entre diferentes versiones de cada clase. En otras palabras, la estructura de la base de datos puede actualizarse dado que las versiones anteriores de la aplicación continúan funcionando, del mismo modo que las aplicaciones posteriores reconocen la estructura actualizada. El diseño del metamodelo se completa con las características específicas de los sistemas de persistencia ortogonal, así como con una estrategia de enriquecimiento de metadatos en el código fuente de la aplicación.

Con respecto a la viabilidad de esta propuesta, se desarrolló un prototipo que consiste en un *framework* mediador de la interacción entre las aplicaciones y la base de datos através de mecanismos de persistencia ortogonal. Estos mecanismos son introducidos en las aplicaciones como aspectos (es decir, en un sentido orientado a aspectos): los objetos no requieren extensiones de súper clases o implementación de interfaces ni contienen anotaciones especiales. Además, el *framework* también maneja correctamente las clases de tipo paramétrico. Sin embargo, las clases pertenecientes a entornos de programación deben ser consideradas no versionables, debido a las restricciones impuestas por la *Java Virtual Machine*. En lo que se refiere a la concurrencia, el *framework* dota a las aplicaciones de un entorno *multithreading* que soporta las transacciones y la recuperación de errores de las bases de datos.

Este *framework* oculta tanto el problema de la evolución de la base de datos, como sus detalles de persistencia, a las aplicaciones que usan dicha base de datos. Los programadores pueden actualizar la estructura de clases de las aplicaciones, ya que el *framework* produce una nueva versión en la capa de metadatos de la base de datos. Usando construcciones XML de puntos de corte, se extiende el

mecanismo de adaptación de instancias, lo que hace que el *framework* también se mantenga ajeno a este problema.

Se cotejaron los beneficios potenciales del prototipo. Durante nuestro estudio de caso, los resultados confirmaron que la transparencia de los mecanismos tienen efectos positivos en la productividad de los programadores, al simplificar todo el proceso de evolución tanto a nivel de la aplicación como de la base de datos. El metamodelo mismo fue también cotejado con respecto a su complejidad y agilidad. Comparado con otros metamodelos, éste requiere menos modificaciones de metaobjetos en cada paso de la evolución del esquema. También se realizaron otras pruebas para validar la robustez del prototipo y del metamodelo. Para llevar a cabo estas pruebas, se usó una pequeña base de datos 007, debido a la gran complejidad de su modelo de datos. Toda vez que el prototipo desarrollado contiene algunos rasgos que no se observan en otros sistemas conocidos, no fue posible comparar su rendimiento directamente. Sin embargo, sí se han realizado pruebas de rendimiento, que están disponibles para futuras comparaciones de la solución con otros sistemas equivalentes.

Para evaluar nuestro abordaje en un escenario real, se desarrolló una aplicación de prueba de concepto. Esta aplicación no contaba con mecanismos de persistencia, los que fueron añadidos utilizando el *framework* y realizando pequeñas alteraciones en el código fuente de la aplicación. También se evaluó la aplicación en un escenario de evolución de esquema. Estas experiencias reales mostraron que las aplicaciones continúan funcionando ajenas a la persistencia y evolución de la base de datos.

En este estudio de caso, el *framework* resultó una herramienta útil para programadores y administradores de bases de datos. Por otra parte, las limitaciones principales del *framework* se refieren a cuestiones de rendimiento y a la restricción del uso simultáneo de varias *Java Virtual Machine*.

Palabras Clave: Aspect-Oriented Programming, Schema Evolution, Instance Adaptation, Database Evolution