FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Desarrollo de GISs basados en Componentes Dinámicos

AUTORES: Leonardo Nomdedeu y Luciano A. Benítez

DIRECTOR: Dra. Silvia Gordillo

CODIRECTOR:

CARRERA: Licenciatura en Informática

Resumen

Desde hace más de una década los Sistemas de Información Geográfica (GISs) se han convertido en una de las herramientas de trabajo más importantes para el manejo de la información vinculada con diversos niveles de agregación espacial o territorial. Estos sistemas usualmente son diseñados y construidos a medida, esto hace que el GIS o parte del mismo no pueda ser reutilizado, extendido o modificado con facilidad. En particular, existen muchas dependencias funcionales fuertemente ligadas al GIS, como por ejemplo el sistema de entrada de datos, la representación y visualización de la cartografía, las fuentes de almacenamiento de datos o las estrategias del negocio, que a la hora de evolucionar requieren cambios drásticos en la codificación del software, teniendo que reimplementar ciertos componentes, realizar una reingeniería, o hasta incluso incorporar nuevos recursos humanos especializados.
La motivación de este trabajo es definir un proceso que permita generar una arquitectura GIS flexible, que permita la incorporación o reemplazo de componentes en forma dinámica a lo largo del tiempo, de aquí surge el concepto de *Componentes Dinámicos*.

Líneas de Investigación

Sistemas de Información Geográfica
Separación temprana de concerns
Modelo de componentes y aspectos
Implementación de componentes y aspectos mediante Frameworks AOP (Aspect Oriented Programming).

Conclusiones

Los Componentes Dinámicos obtenidos mediante el proceso propuesto pueden modularizar los concerns transversales como unidades funcionales individuales evitando el fenómeno de código entremezclado. Este fenómeno genera código redundante, más difícil de entender y por sobre todas las cosas más difícil de mantener a la hora de evolucionar.
Por otro lado, estos componentes al estar desacoplados unos de otros permiten incrementar el potencial de reuso de los mismos.

Trabajos Realizados

Como caso de estudio se ha realizado un prototipo de un GIS llamado *Sistema de Asignación de Taxis*. Este GIS ha sido construido utilizando el proceso propuesto en este trabajo, comenzando con el análisis de requerimientos, separando los concerns, modelando los concerns como componentes y aspectos mediante el modelo CAM y finalmente implementando los mismos con Spring.Net.

Trabajos Futuros

Estudiar e implementar un mecanismo de mapeo automático de los concerns identificados con el modelo AORE-Multidimensional a componentes y aspectos del modelo CAM.
Estudiar en detalle los meta-concerns de los GISs para poder abstraer sus propiedades y sus relaciones, de modo de poder implementar componentes GIS reutilizables en distintos dominios.

Fecha de la presentación: Diciembre de 2008

Agradecimientos

Agradecemos a Silvia Gordillo y a Alejandra Lliteras por haber guiado y colaborado durante todo este tiempo en nuestra investigación sobre los Sistemas de Información Geográfica. Desde el año 2002 hasta el día de hoy hemos contado con el apoyo incondicional de ambas.

Agradecimientos particulares de Leonardo Nomdedeu

Agradezco a mis padres el apoyo brindado durante toda la carrera. Sin su apoyo todo hubiese sido más difícil. Dedico este trabajo a mis dos soles: mi mujer Paula y mi hija Ana.

Índice General

Indice de Figuras

Introducción

Desde hace más de una década los Sistemas de Información Geográfica (GIS) se han convertido en una de las herramientas más importantes de trabajo para los investigadores, analistas y planificadores, en todas sus actividades que tienen como materia el manejo de la información vinculada con diversos niveles de agregación espacial o territorial [Korte01].

Tradicionalmente los usuarios de los GISs han sido las instituciones públicas, pero el incremento de la demanda de esta tecnología provino de la incorporación de los sistemas de información geográfica a los sectores empresariales, como los bancos, compañías de seguros, etc. La aparición de estos nuevos usuarios ha impulsado al mercado de proveedores GIS a adaptarse a esta nueva demanda ofreciendo soluciones a medida para cada organización. Necesariamente, estos proveedores tomaron conciencia de que la calve del éxito de un GIS depende mayoritariamente de tres factores en los que se debe hacer foco a la hora de la construcción del sistema:

1. Se debe pensar y construir el GIS como una herramienta corporativa. Los GISs no pueden concebirse como parte aislada de la organización empresarial sino como un elemento integrador de la misma [Campo_Masip95].
2. Se debe conseguir que el sistema gestione información espacial de calidad, exacta y actualizada, ya que esto permitirá a las organizaciones utilizar al GIS como una herramienta fiable para la toma de decisiones [Campo_Masip95].
3. Se deben generar herramientas de software flexibles que permitan integrarse con otros GISs, y que permitan adaptarse a los cambios tecnológicos y a las nuevas reglas de negocio que evolucionan en el tiempo [Alameh01].

La realidad es que no todos los GISs han sido diseñados teniendo en cuenta estos tres factores, vulnerando principalmente el último: el factor de flexibilidad del software.
Esto se debe a que existen muchas dependencias funcionales fuertemente ligadas al GIS, como por ejemplo el sistema de entrada de datos, la representación y visualización de la cartografía, las fuentes de almacenamiento de datos o las estrategias del negocio, que a la hora de evolucionar requieren cambios drásticos en la codificación del software, teniendo que reimplementar ciertos componentes, realizar una reingeniería, o hasta incluso incorporar nuevos recursos humanos especializados.
Este hecho es un problema típico de la *evolución de todo GIS* y se ve reflejado en la dificultad que tienen los arquitectos de software para incorporar al GIS nuevas herramientas o modificar las existentes, dificultad que a nuestro entender tiene origen en la ausencia de un modelo flexible en la etapa de diseño de la arquitectura lógica del GIS.

Para afrontar este problema, en esta Tesina de Grado definiremos un proceso que permita generar una arquitectura GIS flexible, que permita la incorporación o reemplazo de

componentes en forma dinámica a lo largo del tiempo, de aquí surge el concepto de *Componentes Dinámicos*.

El proceso propuesto se aplica a cualquier metodología de desarrollo de software que incluya las etapas de Análisis de Requerimientos, Diseño de la Arquitectura y Desarrollo, por ejemplo, el Modelo en Cascada [Royce70] y el Modelo en Espiral [Boehm88]). El proceso está basado en tres fases que se deben cumplir secuencialmente (ver Figura 1). En cada una de estas fases se emplea un método, principio o modelo que permite generar una salida (output) que será a su vez la entrada (input) o punto de partida para comenzar la próxima fase.
Las fases que componen el proceso son las siguientes:

1. *Separación de Concerns*: en esta fase se deben identificar y separar los *concerns* (aspectos) del GIS. Esta tarea creemos que es necesario llevarla a cabo en las etapas más tempranas del ciclo de vida del software, en especial partiremos de la etapa de Análisis de Requerimientos. Esto nos permitirá determinar la influencia de un aspecto del GIS (concern) sobre el resto, para poder determinar cuales son aquellos aspectos propensos a evolucionar en el tiempo.
   Esta fase al ser la primera del proceso, no parte de una especificación o modelo existente (input), sino que parte de la elicitación de requerimientos y de la clasificación de los mismos en *concerns* utilizando el modelo AORE Multidimensional.
   Mediante este modelo se puede definir una especificación formal (output) de los concerns idetificados del GIS.

2. *Mapeo de Concerns a Componentes y Aspectos*: esta fase se corresponde con la etapa de Diseño de la Arquitectura, en donde se deben modelar los concerns identificados en la fase anterior como componentes de la arquitectura del GIS. Estos concerns podrán modelarse como *Componentes* o como a *Aspectos*, siguiendo la aproximación propuesta por el Modelo CAM (Component Aspect Model) [Pinto05]. Pero para determinar el destino final de un concern en el modelo propondremos un conjunto de reglas basadas en la naturaleza de los concerns que nos permitirán determinar si un concern debe modelarse como Componente o como Aspecto.
   El input de esta fase es la especificación de concerns generada en la fase previa y el output es un modelo de componentes y aspectos que conforman la arquitectura lógica del GIS.

3. *Implementación de Componentes*: utilizando el modelo generado en la fase previa, se deben implementar los Componentes y los Aspectos como entidades de software. Esta fase se corresponde con la etapa de Desarrollo (Construcción).
   Nuestra aproximación se basa en implementar los Componentes mediante el paradigma de Programación Orientado a Objetos (POO), mientras que los Aspectos son implementados mediante el paradigma de Programación Orientada a Aspectos (POA).
   El input de esta fase es el modelo de componentes y aspectos generado en la fase previa y el output es finalmente la aplicación GIS.

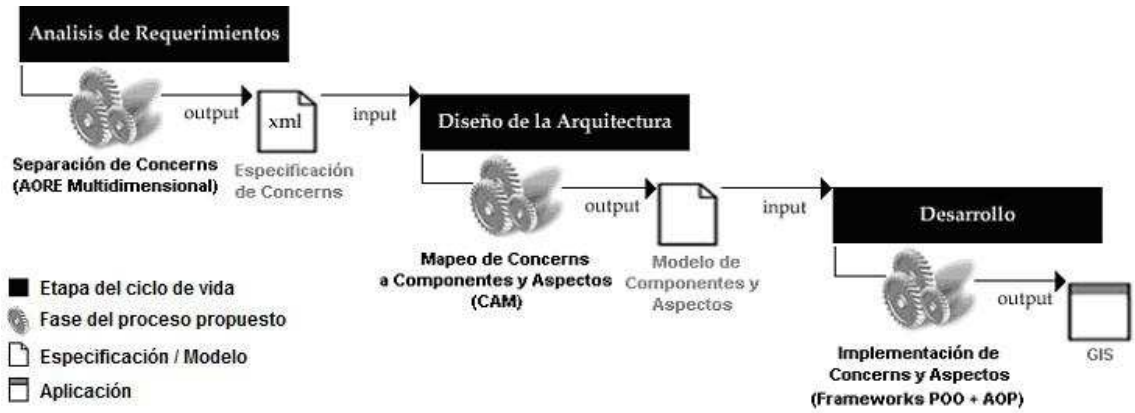


Figura 1. Proceso de construcción de una arquitectura GIS basada en Componentes Dinámicos

Tomando estas tres fases como los pilares de este trabajo, hemos organizado el documento en 5 capítulos.
El primer Capítulo pretende introducir al lector a los Sistemas de Información Geográfica, describiendo las características principales que presentan los mismos, sus componentes, su historia y las aplicaciones en el mundo real.

El foco de este capítulo está puesto principalmente en las arquitecturas de software empleadas en los GISs, describiendo desde las arquitecturas más rudimentarias (monolíticas) hasta las más modernas (orientadas a servicios), en donde los componentes se agrupan en capas lógicas de acuerdo a la funcionalidad que ofrecen los mismos y en donde esta funcionalidad puede ser consumida por otras aplicaciones a través de servicios publicados en la red.
Al final de este capítulo se propone al lector construir la arquitectura de las aplicaciones GIS desde una perspectiva orientada a servicios, aunque también se le plantea un desafío bastante complejo, que consiste en identificar claramente los componentes de las capas lógicas de la arquitectura, y sobre todo cuales son los servicios a exponer a las aplicaciones que los consumen. Una mala decisión de diseño en la definición de los componentes de la arquitectura podría atentar contra un factor fundamental que consideramos necesario alcanzar en todo GIS: *poder generar herramientas de software flexibles que permitan integrarse con otros GISs, y que permitan adaptarse a los cambios tecnológicos y a las nuevas reglas de negocio que evolucionan en el tiempo* [Alameh01].

Para prevenir este problema, y como tema principal del Capítulo 2, se plantea realizar una clara identificación y separación de los componentes de la arquitectura del GIS que pueden llegar a evolucionar en el tiempo. Para esto recurrimos a un proceso bien definido que permite identificar y separar los aspectos o incumbencias relevantes del GIS. Este proceso es conocido como Separación de Concerns (en inglés *Separation of Concerns - SoC*).
La Separación de Concerns puede ser aplicada en distintos niveles de abstracción del proceso de construcción del GIS. Como el objetivo de este trabajo es alcanzar una arquitectura flexible basada en componentes dinámicos, debemos concentrarnos en el modelado de los componentes de la arquitectura, pero antes creemos que un buen modelo parte de una buena especificación de los requerimientos del sistema, en donde ya se haya separado claramente los concerns de GIS. Por este motivo, en el Capítulo 2 introducimos el modelo AORE Multi-Dimensional [Moreira1_05] que sugiere realizar una separación de

concerns en la etapa de Análisis de Requerimientos, generando una especificación de los concerns en donde se describen cuales son los concerns y las relaciones entre ellos.
Esta especificación servirá como punto de partida para el diseño de la arquitectura lógica, tema que aborda el Capítulo 3. En el mismo, se describe un proceso para modelar cada concern identificado como un componente o aspecto de la arquitectura del GIS basándonos en el Modelo CAM (Component Aspect Model) [Pinto05].

En el Capítulo 4, ya teniendo un modelo de Componentes y Aspectos, analizaremos algunos frameworks basados en los paradigmas de Programación Orientada a Objetos y Programación Orientada a Aspectos, que nos permitirían implementar las entidades del modelo como entidades de software. Utilizar frameworks basados en ambos paradigmas nos permitirá incorporar o modificar componentes a lo largo del ciclo de vida del GIS evitando enfrentarse con una reingeniería exhaustiva de la aplicación.

Para poner en práctica el proceso en su totalidad, y como tema central del Capítulo 5, se describirá un prototipo de una aplicación GIS llamada *TaXYs* que tiene como función principal asignar el taxi más cercano a la ubicación de un cliente. Mediante este prototipo, construido con el proceso propuesto, podremos observar la facilidad para incorporar o reemplazar componentes de software ante un eventual cambio en el negocio del GIS.

Capítulo 1. Los Sistemas de Información Geográfica

El objetivo de este capítulo es introducir al lector a los Sistemas de Información Geográfica, describiendo las características principales que presentan los mismos, sus componentes, su historia y las aplicaciones en el mundo real.
A su vez, comenzaremos a focalizarnos en el proceso propuesto en este trabajo para alcanzar una arquitectura GIS flexible, pero antes creemos necesario hacer un breve recorrido por algunas de las arquitecturas GIS empleadas a lo largo de la historia, analizando sus ventajas y sus desventajas.
A pesar de que todas las arquitecturas evaluadas varían en algún aspecto de diseño, observaremos que todas presentan un problema en común, que consiste en el acoplamiento de los componentes.
Para dar comienzo a este capítulo y poder abordar esta problemática, creemos necesario primero responder la siguiente pregunta: *¿Qué es un Sistema de Información Geográfica?*

1.1 Definición de Sistema de Información Geográfica

Definir que es un GIS no es una tarea sencilla ya que existe una gran cantidad de factores a tener en cuenta para poder formular una definición que contemple todo lo necesario para diferenciar a un GIS de otros sistemas de información. A pesar de que se hayan realizado algunos debates acerca del origen del término y de la fecha de iniciación en este campo (ver [Coppock_Rhind91]) aún existen diferentes definiciones, que surgen desde distintas perspectivas. Muchas de las definiciones existentes son relativamente generales y cubren un gran rango de sujetos y actividades. Todas las definiciones, sin embargo, tienen una característica en común, y es que el término se aplica a un sistema que manipula información geográfica. La información es considerada geográfica si es mensurable y tiene localización [Longley91].

En este trabajo, nos basaremos en la definición de que un Sistema de Información Geográfica es un sistema de información que tiene como objetivo recolectar, almacenar, manipular, analizar y proporcionar datos espacialmente referenciados a la Tierra como así también datos asociados no espaciales [Parker88] [Burrough86]. A su vez, creemos necesario destacar la importancia de los GISs como sistemas de soporte para la toma de decisiones ([Cowen88], [Parent_Church87]).

Basándonos en esta definición, describiremos a continuación las características generales de un GIS para poder abordar, con las herramientas necesarias, al diseño de las arquitecturas GIS, tema en el que subyace la problemática de esta tesis.

1.2 Conceptos generales

Un GIS gestiona información sobre el mundo como una colección de niveles temáticos (o layers de información – ver Figura 2) que pueden relacionarse por su geografía. Este concepto simple ha probado ser invaluable para resolver muchos problemas, desde rastrear vehículos de repartición, registrar detalles de aplicaciones de planificación, hasta modelar la circulación atmosférica global.

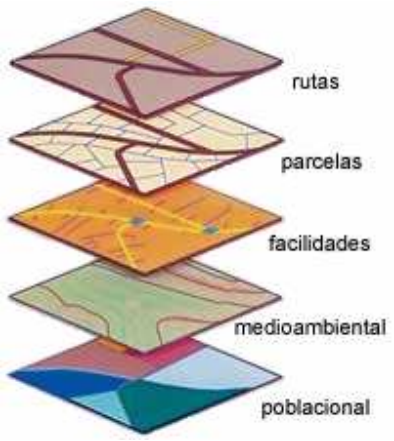


Figura 2. Niveles temáticos (o layers de información) de un GIS

En un GIS, las entidades del mundo real son representadas como una serie de entidades geográficas definidas de acuerdo a dos tipos de datos: los *elementos geográficos* (también conocidos como *datos espaciales*) son utilizados para proveer referencia a los *elementos no geográficos* conocidos como *datos no-espaciales o atributos*. Los *atributos* de una entidad geográfica pueden ser referencias geográficas explícitas, tal como latitud y longitud o una coordenada de un sistema específico, o una referencia implícita tal como domicilio, código postal, nombre de área censal, o nombre de calle. Las referencias implícitas pueden ser derivadas de referencias explícitas utilizando un proceso automatizado llamado "geocodificación." Estas referencias geográficas permiten localizar características (tales como negocios, puntos de interés, etc.) y eventos (como un terremoto) en la superficie de la tierra para el análisis. Existe una tercera clasificación de los atributos y son las referencias temporales, que se utilizan para describir los cambios ocurridos en las entidades en el transcurso del tiempo.

En cuanto a los *datos espaciales*, existen dos modelos de representación ampliamente utilizados en los GISs, el "modelo raster" y el "modelo vectorial" (ver Figura 3.).

En el modelo vectorial, la información referente a puntos, líneas y polígonos se codifica y almacena como una colección de coordenadas X,Y. La ubicación de una entidad puntual, tal como una perforación, puede describirse con un sólo punto X,Y. Las características lineales, tales como calles y ríos, pueden almacenarse como un conjunto de puntos de

coordenadas X,Y. Las características poligonales, tales como parcelas y cuencas hídricas, pueden almacenarse como un circuito cerrado de coordenadas (ver Figura 3, Modelo Vectorial). El modelo vectorial es extremadamente útil para describir características discretas, pero menos útil para describir características de variación continua, tal como el tipo de suelo o costos de accesibilidad para hospitales.
Existen diversas estructuras de datos para generar modelos vectoriales, las más importantes de acuerdo a [Bosque92] son:

- *Modelo Topológico:* en este modelo las relaciones espaciales entre entidades son explícitamente almacenadas en lo que se conoce como modelo topológico.
  Se define a la topología como el estudio matemático de las relaciones y transformaciones de configuraciones geométricas. La idea básica de este modelo radica en segmentos de línea continua que empieza y termina en la intersección con otra línea o a la curvatura en la línea.
  Entre los modelos topológicos tenemos: Modelo codificado independiente dual, Estructura Arc-nodo, Estructura relacional, Estructura gráfica de línea digital. Más información sobre estos modelos topológicos puede encontrarse en [Khalimsky87].

- *Modelo Spaghetti* (lista de coordenadas): para cada objeto espacial se registra su identificador, seguido por una lista de coordenadas de los vértices (puntos) que definen su posición en el espacio. Posee la desventaja de que el sistema almacena información sobre la localización de los elementos, pero no sobre las relaciones que existen entre los elementos; es decir se registra la geometría pero no la topología. También esta estructura de datos genera información redundante (ej. registra dos veces las coordenadas de un lado común de dos polígonos).

- *Diccionario de vértices:* un mapa se representa mediante dos archivos de datos: un archivo esta constituido por una relación de vértices, en la que constan las coordenadas X, Y, y otro archivo con los vértices que definen cada objeto.
  Esta estructura resuelve los problemas de repetición de coordenadas de los puntos, de la estructura Spaghetti; pero es muy pobre desde el punto de vista topológico.

Por otro lado el modelo raster ha evolucionado para modelar las características continuas. Una imagen raster comprende una colección de celdas de una grilla más como un mapa o una Figura escaneada (ver Figura 3, Modelo Raster). Ambos modelos para almacenar datos geográficos tienen ventajas y desventajas únicas y los GISs modernos pueden manejar ambos tipos [Di Neo02].

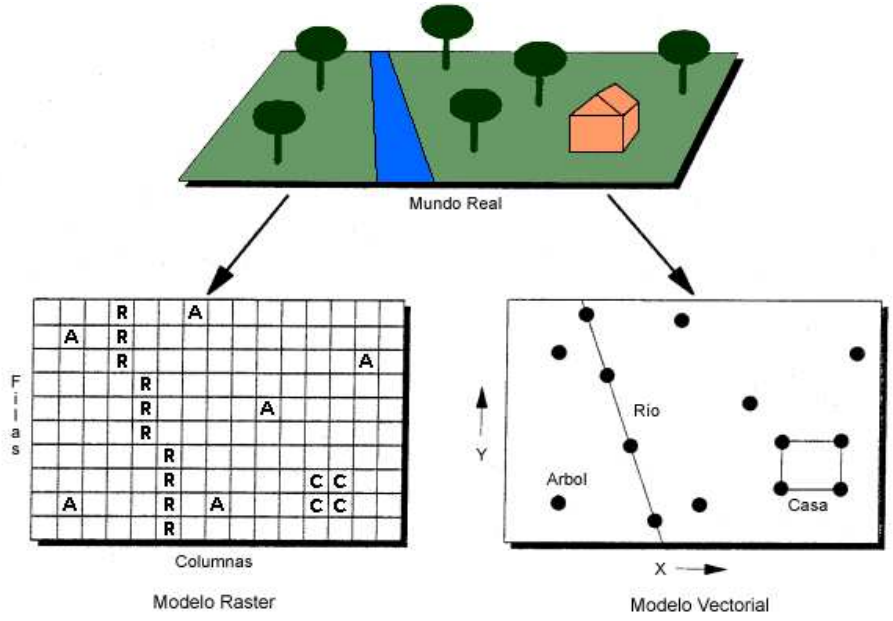


Figura 3. Representación de la información geográfica (modelo Raster y Vectorial)

Mediante estos dos modelos se pueden desarrollar algoritmos, conocidos generalmente como consultas espaciales, que permiten obtener cierto tipo de información, por ejemplo acerca de la ubicación específica de los elementos del domino, calcular distancias, adyacencias, superposición, e inclusión entre dos o más elementos del domino.

1.3 Los componentes de un GIS

Un GIS está compuesto por cinco componentes claves [MAIC04]: Personas, Datos, Hardware, Software y Procedimientos. Estos componentes necesitan estar en equilibrio para que el sistema sea exitoso. Ningún componente puede funcionar sin el otro.

Personas

Las Personas son el componente que realmente hace que un GIS funcione. Ellas conforman un equipo de trabajo donde cada individuo posee un rol específico, entre ellos se encuentran los usuarios especializados, el gestor del GIS y especialistas en informática, entre otros. Todos son responsables del mantenimiento y soporte del sistema.

Las Personas asociadas a un GIS pueden ser categorizadas en:

- Visualizadores: su única necesidad es consultar la información geográfica y no geográfica como material de referencia. Ellos constituyen la clase de usuarios más amplia.

- Usuarios Generales: son aquellos que utilizan el GIS para conducir los negocios, realizar servicios profesionales, y tomar decisiones. Esta clase de usuarios está

conformada por gestores de recursos, planificadores, geógrafos, cartógrafos, científicos, analistas, ingenieros, abogados, empresarios, etc.

- Especialistas GIS: son aquellas personas que hacen que el GIS funcione. Esta clase de usuarios está conformada por el gestor del GIS, administradores de base de datos, especialistas de la aplicación, analistas de sistemas y programadores. Ellos son responsables del mantenimiento de las bases de datos y de la provisión de soporte técnico sobre el sistema a las dos clases de usuarios previamente enunciadas.

Procedimientos

Los procedimientos hacen referencia a cómo la información debe ser ingresada al sistema, almacenada, manipulada, transformada, analizada y finalmente presentada a los usuarios. La habilidad de un GIS para realizar análisis espacial es lo que diferencia este tipo de sistema del resto de los sistemas de información

Los procesos de transformación incluyen ciertas tareas, como por ejemplo ajustar el sistema de coordenadas[1], configurar una proyección[2], corregir cualquier error de digitalización en un conjunto de datos, convertir datos entre distintos modelos (ej. de raster a vectorial y viceversa) [Carver98].

Hardware

El Hardware consiste en el equipamiento técnico necesario para que funcione el GIS, incluyendo computadoras y servidores con la suficiente potencia para ejecutar el software, suficiente memoria para almacenar grandes cantidades de datos, y dispositivos de entrada y salida como por ejemplo escáneres, digitalizadores, receptores GPS e impresoras [Carver98].

Software

Existen muchos paquetes de Software GIS diferentes en la actualidad. Algunos ejemplos de estos paquetes son ArcGis [ESRI_ArcGis], Geomedia [Intergraph_Geomedia], MapInfo [Pitney_MapInfo]. Estos paquetes deben ser capaces de incorporar datos geográficos y no geográficos, almacenar la información, manipularla, transformarla, analizarla y presentarla al usuario, pero la apariencia, los métodos, los recursos y la facilidad de uso pueden ser muy diferentes. Los paquetes de hoy en día son capaces de almacenar en una única base de datos tanto datos gráficos como descriptivos, utilizando un modelo conocido como modelo objeto-relacional. Antes de esta innovación, el modelo más utilizado era el modelo geo-relacional[3], donde los conjuntos de datos gráficos y descriptivos eran manejados por

[1] *Sistema de Coordenadas*: conjunto de valores que permiten definir unívocamente la posición de cualquier punto de un espacio geométrico respecto de un punto denominado origen. El conjunto de ejes, puntos o planos que confluyen en el origen y a partir de los cuales se calculan las coordenadas de cualquier punto constituyen lo que se denomina Sistema de Referencia [Elue08].

[2] *Proyección:* método que se utiliza para definir una correspondencia matemática entre los puntos del elipsoide y los mismos puntos transformados en un plano [Rey99].

[3] *Modelo Georelacional*: modelo de datos geográficos de ESRI© introducido en 1981 con la

separado [Sommer06].
Actualmente los paquetes modernos (como los enunciados anteriormente) incluyen un conjunto de herramientas que pueden ser configuradas para satisfacer las necesidades del usuario.

Datos

Quizá el aspecto más costoso y que lleva más tiempo en la iniciación del GIS es la creación del repositorio de datos. Existen muchos puntos a considerar antes de adquirir los datos geográficos, por ejemplo, es crucial verificar la calidad de los datos ya que los errores presentes o inconsistencias llevarán a los analistas a obtener conclusiones equivocadas. Algunos de los puntos de acuerdo a [Guptill95] son:

- Linaje: hace referencia la descripción de la fuente de donde proviene el material, y los métodos de provisión, incluyendo todas las transformaciones involucradas en la producción de los archivos digitales. Debe incluir todas las fechas de obtención del material, actualización y cambios que se realicen.

- Exactitud Posicional: hace referencia a la proximidad que existe entre las coordenadas en el sistema de una entidad respecto de la verdadera coordenada en el mundo real. La exactitud posicional incluye mediciones de la exactitud vertical y horizontal del conjunto de datos.

- Exactitud de Atributos: un atributo es un hecho acerca de una localización (también conocida como *location* en inglés), conjunto de localizaciones, o elementos sobre la superficie terrestre. Esta información usualmente incluye mediciones de distintos tipos, por ejemplo temperaturas, elevaciones o hasta incluso una etiqueta con el nombre de un lugar. Es vital que esta información sea exacta para que el GIS permita obtener un buen análisis.

- Consistencia Lógica: hace referencia a las reglas lógicas de la estructura y de los atributos de los datos espaciales, y describe la compatibilidad de un dato con el resto de los datos que conforman el conjunto de datos. Existen varias teorías y modelos matemáticos para testear la consistencia lógica, como por ejemplo las pruebas de métricas e incidencias, o pruebas de topología. Estos chequeos de consistencia deben ser puestos en práctica en las diferentes etapas de la gestión de datos espaciales.

- Completitud: es un chequeo que se utiliza para verificar si existen datos relevantes relacionados con las entidades y los atributos que están ausentes. Esto tiene relación con los errores de omisión o de reglas espaciales tales como el ancho o el área mínima que de una entidad que pueden limitar la información.

---

herramienta ArcInfo, permite relacionar en una base de datos, tanto datos espaciales con no espaciales, cada uno con su propio modelo.

1.4 Áreas de aplicación

Basándonos en la definición de que un GIS es un sistema de información que tiene como objetivo recolectar, almacenar, manipular, analizar y proporcionar datos espacialmente referenciados a la Tierra como así también datos asociados no espaciales, cualquier actividad relacionada con el espacio puede beneficiarse del trabajo de un GIS. En [Cardenas07] se describen algunas de estas actividades:

- *Agricultura*: estos GIS contribuyen a lo que hoy se conoce como Agricultura de Precisión, que permite adoptar las medidas de manejo necesarias para optimizar la producción física o económica en cada sector diferente de una determinada área o potrero del campo.
- *Estudios* ambientales: monitoreo, modelaje y manejo para evitar degradación ambiental, planificación rural, zonas de riesgo, deforestación y desertificación, calidad y cantidad de agua, plagas, calidad de aire, predicción y modelaje del clima.
- *Epidemiología y salud*: ubicación de enfermedades relacionadas con factores ambientales/o actividades humanas.
- *Forestía*: planificación, manejo y optimización de actividades de extracción de madera, reforestación.
- *Servicios de Emergencia*: optimización de rutas para servicios de ambulancia, bomberos y policía, mejor entendimiento de zonas conflictivas (crímenes).
- *Navegación*: transporte aéreo, marítimo y terrestre.
- *Estudios de Mercado*: ubicación de sitios y de grupos objetivo (targets), optimización de servicios de entrega.
- *Bienes raíces*: aspectos legales y de catastro, valor de las propiedades con delación a su ubicación, seguros.
- *Planificación local/regional*: localización, planificación y manejo de aeropuertos, muelles, carreteras y redes ferroviarias.
- *Estudios Sociales*: análisis de dinámicas demográficas.
- *Turismo*: localización y manejo de facilidades y atracciones.
- *Servicios Públicos*: ubicación, manejo y planificación de servicios de agua, drenaje, gas, electricidad, telefonía y televisión.
- *Arqueología*: estos GIS están orientados principalmente a la gestión de los recursos culturales y al desarrollo de modelos predictivos de localización de asentamientos (Kohler y Parker, 1986). A partir de estos sistemas se ampliaron los campos de aplicación principalmente hacia la reconstrucción paleoambiental y la relación de la sociedad con el medio ambiente (Allen, 1990). En la actualidad también se utilizan en las prácticas de arqueología espacial y hacia la gestión institucional del patrimonio arqueológico [Grau06].

1.5 Reseña histórica

En esta sección realizaremos una breve reseña histórica de los GISs basada en la investigación y recopilación de Klinkenberg en [Klinkenberg84].

El primer sistema formal de información geográfica fue diseñado en Canadá en el año 1962 por Roger Tomlinson. Este sistema le permitió a la Agencia de Rehabilitación y Desarrollo de Agricultura de Canadá proveer asistencia a los agricultores del país. El sistema subsiguiente fue el Inventario de Terrenos de Canadá, que involucró un masivo ejercicio de mapeo que combinaba un censado aéreo y tecnología computarizada en su versión más efímera. A partir de ese momento se comenzó un largo camino de evolución en los GISs hasta llegar a los años ochenta en donde se comenzó con la comercialización de los GISs [Urisa08].

Durante los años sesenta y setenta se comenzaron a utilizar a las computadoras para automatizar tareas que hasta el ese momento se hacían en forma manual. La mayoría de los programas desarrollados estaban dirigidos a la automatización del trabajo cartográfico; muy pocos científicos exploraron nuevos métodos para el manejo de información espacial, y se siguieron básicamente dos tendencias:

1. La producción automática de dibujos con un alto nivel de calidad gráfica.
2. La producción de información basada en el análisis espacial pero con el costo de una baja calidad gráfica.

La producción automática de gráficos se basó en la tecnología de Diseño Asistido por Computadoras (CAD). El CAD fue utilizado en la cartografía para acrecentar la productividad en la generación y actualización de mapas. El modelo de base de datos de CAD maneja la información espacial como dibujos compuestos por entidades gráficas organizadas en planos de visualización o layers (descriptos en el apartado 1.2). Cada capa contiene la información de los puntos que representan entidades específicas que se deben mostrar en pantalla. Estos conjuntos de puntos organizados por planos de visualización se guardan en un formato vectorial.
En ese entonces, las bases de datos comenzaron a incorporar funciones gráficas primitivas que se empleaban para construir nuevos conjuntos de puntos o líneas en nuevas capas para definir un símbolo interpretado por el usuario. A esta simbología luego se le adicionó una variable "inteligente" al incorporar el texto.
El desarrollo de la tecnología CAD se empleó para la manipulación de dibujos y mapas, y también para la optimización del manejo gerencial de información cartográfica.

En los años ochenta se expandió la utilización de los GIS, facilitada por una serie de factores que se detallan a continuación:

1. La generalización del uso de microcomputadoras y estaciones de trabajo en la industria.
3. La aparición y consolidación de las Bases de Datos relacionales, junto a las primeras modelizaciones de las relaciones espaciales o topología.

4. La comercialización simultánea de un gran número de herramientas CAD.

Por otro lado, la aparición del paradigma Orientado a Objetos (OO) en los GIS permitió nuevas concepciones que permitieron integrar en un objeto todo lo referido a una entidad (simbología, geometría, topología, atribución), que hasta ese momento se manipulaban por separado.
Años más tarde los GIS se comienzan a utilizar en cualquier disciplina que necesite la combinación de planos cartográficos y bases de datos como en la ingeniería civil (diseño de carreteras, represas y embalses), estudios medioambientales, estudios geológicos y geofísicos, estudios socioeconómicos y demográficos, ordenación del territorio, planificación de líneas de comunicación, prospección y explotación de minas, entre otros.

En los años noventa se intensificó el uso de los GISs, alcanzando un nivel de madurez mayor al de los sistemas de las décadas anteriores. A su vez, ocurrió un hecho que contribuyó a la expansión de estos sistemas: la utilización de los GISs en los negocios propiciada por varios factores, como la generalización en el uso de computadoras de bajo costo y de gran potencia, la enorme expansión de las comunicaciones y en especial de Internet y la Web, la aparición de los sistemas distribuidos (DCOM, CORBA) y la fuerte tendencia a la estandarización de formatos de intercambio de datos, inclusive datos geográficos.
El incremento de la popularidad de las tendencias de programación distribuida y la expansión y beneficios de los lenguajes de programación de última generación, como C++ o JAVA, permitieron la creación de nuevas formas de programación de sistemas distribuidos, de esta manera surgen los agentes móviles, que a través de la invocación de métodos remotos y la serialización de objetos logran transportar datos a través de la red. Esto significó la aparición de un nuevo paradigma para el acceso a consultas y recopilación de datos en los sistemas de información geográfica, cuyos mayores beneficios se pueden ver en la actualidad, por ejemplo en la utilización de aplicaciones móviles que consumen servicios geográficos en forma remota ofrecidos por distintas compañías.

Conocer la evolución de los GISs nos permite apreciar tanto los aspectos que los caracterizan como también adquirir una visión de conjunto sobre los mismos. El desarrollo de los GISs ha vivido un progreso continuo en el tiempo. Como cualquier otro suceso, pasó una primera fase de desarrollo muy primario con una limitación en cuanto a usuarios y usos, a periodos en los que la diversificación en sus aplicaciones vino motivada por el perfeccionamiento de la tecnología. En cada instancia de esta evolución, las aplicaciones GIS fueron diseñadas, mantenidas y extendidas sobre una estructura de software que también evolucionó con la tecnología. Esta estructura, conocida como la *Arquitectura del GIS*, será el foco de estudio del apartado siguiente. En el mismo se introducirán algunas de las arquitecturas GIS existentes, describiendo sus ventajas y desventajas, para luego poder realizar una aproximación hacia una arquitectura flexible, que se adapte a los cambios funcionales y tecnológicos que hoy en día exige el mercado.

1.6 Algunas arquitecturas GIS

La Arquitectura de un sistema de información, también denominada Arquitectura lógica,

consiste en un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software para un sistema de información [Garlan93]. En otras palabras, la arquitectura tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de componentes arquitectónicos, en donde se definen de forma adecuada sus interfaces y la comunicación ente ellos para satisfacer los requerimientos funcionales de desempeño del sistema, así como también los requerimientos no funcionales, como la confiabilidad, escalabilidad, portabilidad, y disponibilidad.
En general, la arquitectura de software nace como una herramienta de alto nivel para cubrir distintos objetivos [Vallecillo99], entre los que se destacan:

1. Comprender y manejar la estructura de las aplicaciones complejas.
2. Reutilizar dicha estructura (o partes de ella) para resolver problemas similares.
3. Planificar la evolución de la aplicación, identificando sus partes mutables e inmutables, así como los costos de los posibles cambios.
4. Analizar la corrección de la aplicación, y su grado de cumplimiento respecto a los requerimientos iniciales (prestaciones o fiabilidad).
5. Permitir el estudio de alguna propiedad específica del dominio.

Toda arquitectura de software debe describir diversos aspectos del software. Generalmente, cada uno de estos aspectos se describe de una manera más comprensible si se utilizan distintos modelos o vistas. Es importante destacar que cada uno de ellos constituye una descripción parcial de una misma arquitectura y es deseable que exista cierto solapamiento entre ellos. Esto es así porque todas las vistas deben ser coherentes entre sí dado que describen la misma cosa.
Cada paradigma de desarrollo exige diferente número y tipo de vistas o modelos para describir una arquitectura. Sin embargo, existen al menos tres vistas absolutamente fundamentales en cualquier arquitectura [Kruchten95]:

- La vista estática: describe *qué componentes* tiene la arquitectura.
- La vista funcional: describe *qué hace cada componente*.
- La vista dinámica: describe *cómo se comportan los componentes* a lo largo del tiempo y como interactúan entre sí.

Gran parte de las aplicaciones GIS han sido desarrolladas sobre el diseño previo de una arquitectura lógica, que puede ser evaluada desde las tres vistas mencionadas anteriormente. Comenzaremos describiendo los aspectos más relevantes de algunas de las arquitecturas GIS más rudimentarias (las arquitecturas monolíticas) hasta llegar a arquitecturas más sofisticadas (cliente/servidor y basadas en servicios).
El objetivo principal de este análisis es describir algunas de las arquitecturas más relevantes del mundo GIS, para poder entender la problemática respecto a la flexibilidad de las mismas y poder luego presentar en este trabajo una solución adecuada.

1.6.1 Arquitectura Monolítica

Gran parte de los GISs fueron creados para resolver una tarea específica y para que sean utilizados solo por algunos usuarios especializados. Estas aplicaciones fueron desarrolladas en forma aislada sin pensar en que podrían tener otros usos, generalmente porque se pensaba en sus comienzos del desarrollo que no era necesario.
Estas aplicaciones se diseñaron solo para abrir una fuente de información, editarlos, guardarlos y mostrarlos. Las arquitecturas de estas aplicaciones, conocidas como *arquitecturas monolíticas* [Weaver04] por residir en el mismo sistema (stand-alone), presentan una característica que las diferencia del resto de las arquitecturas: el software se estructura en grupos funcionales muy acoplados en donde los procesos de manipulación y almacenamiento de datos, como así también la interfaz de usuario están fuertemente acoplados en el código fuente.
En este tipo de arquitectura, se dificulta realizar un análisis desde la perspectiva de las tres vistas mencionadas anteriormente, ya que es muy difícil identificar y separar los componentes que la conforman. Esto se debe a que el GIS se codifica como un único subsistema, generalmente mal modularizado y con muchas ataduras de código (hardcodeo). Este tipo de arquitectura en donde no se pueden identificar los componentes funcionales toma el nombre de *arquitectura monolítica desordenada* (ver Figura 4).
La desventaja más grande de esta arquitectura es la dificultad para reutilizar el código, produciendo aplicaciones "demasiado a medida" que se adaptan a un cliente en particular, perdiendo la posibilidad de personalizarla o configurarla.

Esta característica no muy favorable hizo que se cambie la forma en que se diseña una aplicación, evolucionando entonces hacia una arquitectura un poco más organizada y alcanzando como resultado una *arquitectura monolítica ordenada* (ver Figura 5).
En esta arquitectura, el software continúa residiendo en el mismo sistema, pero se realiza una descomposición funcional en base a cinco subsistemas bien definidos; cada uno de estos subsistemas se dedica a tareas específicas y relacionadas lógicamente entre sí, esto permite ubicar más claramente cada componente dentro del sistema y comprender su papel dentro del GIS. Estos subsistemas son [Aranoff89] [Burrough86].

- Subsistema de entrada de datos: compuesto tanto por los periféricos como por los procedimientos para transformar datos espaciales de diferentes procedencias y modelos (por ejemplo, para asignar valores de altitud desde una simbolización por curvas de nivel).

- Subsistema de almacenamiento y recuperación de datos: Que organiza y mantiene la información temática y espacial.

- Subsistema de análisis de la información: Lleva a cabo funciones de análisis, estudios, estadísticas y creación de capas geográficas y temáticas con diversos propósitos y aplicando diferentes modelos.

- Subsistema de edición y mantenimiento de los datos: Además de las funciones del subsistema de análisis, es la parte administrativa del GIS en donde se llevan a cabo tareas más avanzadas como conversión de formatos, conversión de proyecciones cartográficas, normalización de los datos, publicación de mapas entre otras.

- Subsistema de visualización e interacción con los usuarios: Que muestra los resultados de las operaciones en forma de tablas, mapas, gráficos, informes, etc. en la pantalla o en otros dispositivos.

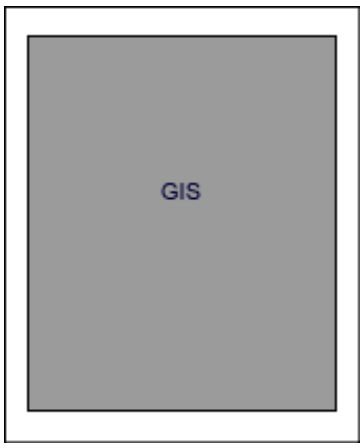


Figura 4. Arq. Monolítica desordenada

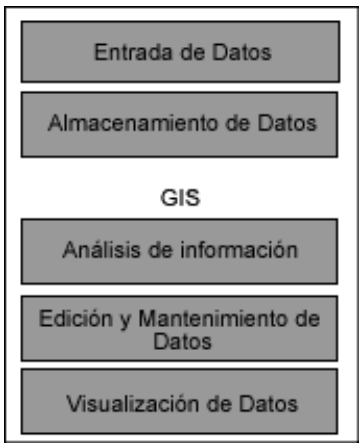


Figura 5. Arq. Monolítica ordenada

En estas arquitecturas monolíticas ordenadas se puede ver una clara distinción entre la vista estática y funcional, ya que se pueden identificar que componentes tiene la aplicación y que hace cada componente de acuerdo a la clasificación en subsistemas.

Hoy en día es muy difícil encontrar GISs basados en arquitecturas monolíticas desordenadas, siendo más populares las arquitecturas monolíticas ordenadas. De los primeros, han sobrevivido aquellos que no han necesitado evolucionar, ya sea porque el dominio de la aplicación no lo requiere o por manejar información extremadamente estática.

1.6.2 Arquitetura Cliente-Servidor

En este tipo de arquitectura la funcionalidad está dividida en tareas que se ejecutan en distintas aplicaciones [Waver04]. Por un lado, existen componentes cuyas tareas se ejecutan en una aplicación denominada *cliente*, y por otro lado existen otros componentes cuyas tareas se ejecutan en una aplicación denominada *servidor*.
Las tareas server-side son realizadas por la aplicación *servidor* que implementa servicios con una funcionalidad específica. Las aplicaciones *cliente* ejecutan las tareas client-side y a su vez delegan funcionalidades a las tareas server-side a través de una red de interconexión [Rodriguez04]. En esta arquitectura la capacidad de procesamiento puede estar repartida entre los clientes y los servidores.

La separación entre *cliente* y *servidor* es una separación lógica, donde el servidor no se ejecuta necesariamente sobre una sola computadora ni es necesariamente una sola aplicación. De hecho, se pueden implementar diferentes tipos de arquitecturas cliente-servidor, de acuerdo a la cantidad de funciones que se ejecutarán en el cliente. En un *cliente*

liviano (*thin-client) se* implementa la menor cantidad posible de funcionalidades, delegando la mayor cantidad de funciones a la aplicación *servidor*.
Por otro lado, en un *cliente grueso* (*thick client*), se trata de implementar la mayor cantidad de funcionalidades en la aplicación *cliente* [Rodriguez 04]. En este esquema, la distribución de los componentes influirá en la vista estática y funcional de la arquitectura, dependiendo de *qué* componentes existan en cada aplicación y *qué hace* cada componente.

Si evaluamos estos tipos de arquitectura desde la *vista estática (*qué componentes existen*)*, a diferencia de las arquitecturas monolíticas desordenadas se puede identificar la presencia de componentes de software que interactúan entre sí para realizar tareas específicas del GIS. En esta vista se identifican componentes de la aplicación *cliente* y componentes de la aplicación *servidor*, pero esta categorización dependerá de las necesidades del GIS, por ejemplo, en un Sistema de Navegación Vehicular, el cálculo de distancia para determinar cual es la estación de gasolina más cercana a un punto que representa las coordenadas geográficas de un automóvil, puede ser una tarea realizada por un componente de la aplicación cliente o servidor, dependiendo de la disponibilidad y distribución de la información, de la velocidad de procesamiento de las computadoras utilizadas por cada aplicación, de la velocidad de acceso a los datos, etc.

Desde la vis*ta funcional* (que hace cada componente), las arquitecturas cliente/servidor comenzaron a organizar sus componentes en capas lógicas (Layer Pattern[4]) de acuerdo a su funcionalidad. Esta decisión de diseño se debe a que la mayoría de los GIS presentan una lógica de negocio compleja y muchas reglas de negocio, las cuales varían con el tiempo, y van modificando a las actuales, y nutriéndose con otras nuevas. Para poder acompañar los cambios del negocio, se busca la manera de mantener los componentes del dominio del GIS lo mas aislado posible del resto de los componentes de la aplicación. Por este motivo, los componentes que forman parte de este dominio se agrupan en la capa del Modelo del Dominio (Domain Model Layer) [Montaldo05].
Existen arquitecturas GIS cliente-servidor en donde se realiza una separación de los componentes en tres capas lógicas de acuerdo al Layer Pattern (ver Figura 6):

- Capa de Presentación: en esta capa se agrupan todos componentes que tienen que ver con la obtención de información y presentación de datos al usuario. Se corresponde con los subsistemas de Entrada y Visualización de datos de [Aranoff89] [Burrough86].

- Capa del Modelo de Dominio: en esta capa se agrupan todos los componentes que conforman el dominio del GIS. Es recomendable modelar el domino utilizando el paradigma Orientado a Objetos, que gracias a sus características de abstracción, encapsulamiento, herencia y polimorfismo, permiten representar fácilmente las entidades del mundo real, sus estados, sus relaciones y sus comportamientos [Gamma95].

[4] *Layer Pattern*: es un patrón de diseño arquitectural que ayuda a estructurar la aplicación en subgrupos de tareas, en donde cada subgrupo está en un nivel de abstracción particular [Buschmann96].

En esta capa también se agrupan los componentes que representan las reglas y los procesos de negocio. Tomando el ejemplo anterior, para obtener la estación de gasolina más cercana a un automóvil, el proceso de negocio (simplificado) puede consistir en identificar todas las estaciones de gasolina en un radio de 500 metros tomando como centro la coordenada geográfica del automóvil, y luego determinar cual es la más cercana realizando un calculo de distancia entre las coordenadas de las estaciones y la del automóvil. Las entidades del dominio involucradas en este ejemplo serían el *Auto* y las *Estaciones de Gasolina*.
En esta capa se encontrarían los componentes correspondientes al subsistema de Análisis de Información de [Aranoff89] [Burrough86].

- Capa de Datos: en esta capa se agrupan los componentes que realizan las tareas de almacenamiento de datos tanto espaciales como no espaciales. Está compuesta por las bases de datos, archivos, etc.

Figura 6. Capas Lógicas

Aun en este esquema, la Capa del Modelo del Dominio presenta cierto acople con la Capa de Datos. Sucede que las entidades del dominio del GIS deben ser persistidas en una fuente de datos, pero estas reglas de persistencia suelen ligarse al dominio.
Este conocimiento es una cualidad que se desea evitar, ya que la manera en que los datos del modelo son persistidos o almacenados es un problema tecnológico que no debería formar parte de los problemas del dominio a resolver [Montaldo05].
Llevando esta aproximación al dominio del GIS que tomamos de ejemplo, las entidades del dominio no deberían tener conocimiento acerca de sus reglas de persistencia, es decir, debería ser transparente si son almacenadas en un archivo, o en bases de datos relacionales, orientadas a objetos o georelacionales, y si las mismas se acceden en forma local o remota.
Esto nos lleva a introducir una nueva capa entre la Capa de Domino y la Capa de Datos, ésta capa es la Capa de Persistencia. El objetivo de la Capa de Persistencia es quitar del dominio el problema asociado a este aspecto tecnológico que no debe formar parte de nuestro dominio, y agruparlo en esta nueva capa (Ver Figura 7).

Figura 7. Capa de Persistencia

Otra práctica muy común que se realiza en las arquitecturas en capas, es la de introducir una Capa de Servicios, en donde se codifiquen las reglas de negocio y la funcionalidad a exponer para la Capa de Presentación.
Al utilizar esta capa de servicios, se está separando el comportamiento puro de las entidades del dominio, del comportamiento propio de la aplicación. Para entender este concepto tomemos el ejemplo anterior, el criterio para encontrar las estaciones de gasolina más cercanas al automóvil en realidad no lo determina ni la entidad Automóvil, ni la entidad Estación de Gasolina, ya que es una regla de negocio del Sistema de Navegación Vehicular y no de estas entidades del dominio.
En un esquema de cuatro capas (Figura 7), esta lógica o bien queda en la capa de presentación o bien queda en la capa de dominio. Sin embargo, no es parte de ninguna de las dos realmente. La idea es que el modelo de dominio posea solo las reglas inherentes al mismo y pueda ser reutilizado en distintas aplicaciones. Por ejemplo, deberíamos modelar un objeto del dominio, llamémoslo *Operador Espacial*, que permita tomar como entrada una entidad X, un radio R y un tipo de entidad T, y realice un cálculo para encontrar la entidad de tipo T más cercana a la entidad X.
Modelar el objeto del dominio con este comportamiento genérico, permite reutilizar el modelo en servicios de distintas aplicaciones que tiene sus propias transacciones de negocio por ejemplo, el Sistema de Navegación Vehicular hará uso de las entidades *Auto* y *Estación de Gasolina,* y hará uso de la funcionalidad implementada por el objeto *Operador Espacial* indicando el radio de 500 metros alrededor del *Auto* y que el tipo de entidad a buscar es la *Estación de Gasolina*. Por otro lado, se puede pensar en un Sistema de Asignación de Taxis, en donde en base a una domicilio reportado se debe enviar el taxi más cercano al mismo. En esta aplicación se podría hacer uso del mismo objeto *Operador Espacial* indicándole un radio de 1500 metros alrededor de un *Domicilio Particular* y especificando que el tipo de entidad a buscar es el *Taxi*.
En la Figura 8 se puede observar un ejemplo de un modelo de objetos reducido en donde se representan algunas entidades del dominio de las aplicaciones enunciadas previamente. Se puede observar que existen tres grandes grupos de entidades geográficas (que poseen una ubicación en el espacio, más precisamente una *Coordenada*):

- *Puntos de Interés (POI):* son entidades geográficas que representan puntos en la ciudad donde se encuentran edificios o comercios que puedan ser de interés al

usuario. Generalmente la información de los POIs es provista por sistemas existentes que gestionan esta información.

- *Domicilio Particular*: representa un domicilio en particular de un usuario.
- *Automóviles*: representa a los vehículos que se mueven en la ciudad. Este tipo de entidad se diferencia del resto ya que su ubicación es dinámica, es decir, cambia a medida en que la entidad se mueve por la ciudad.

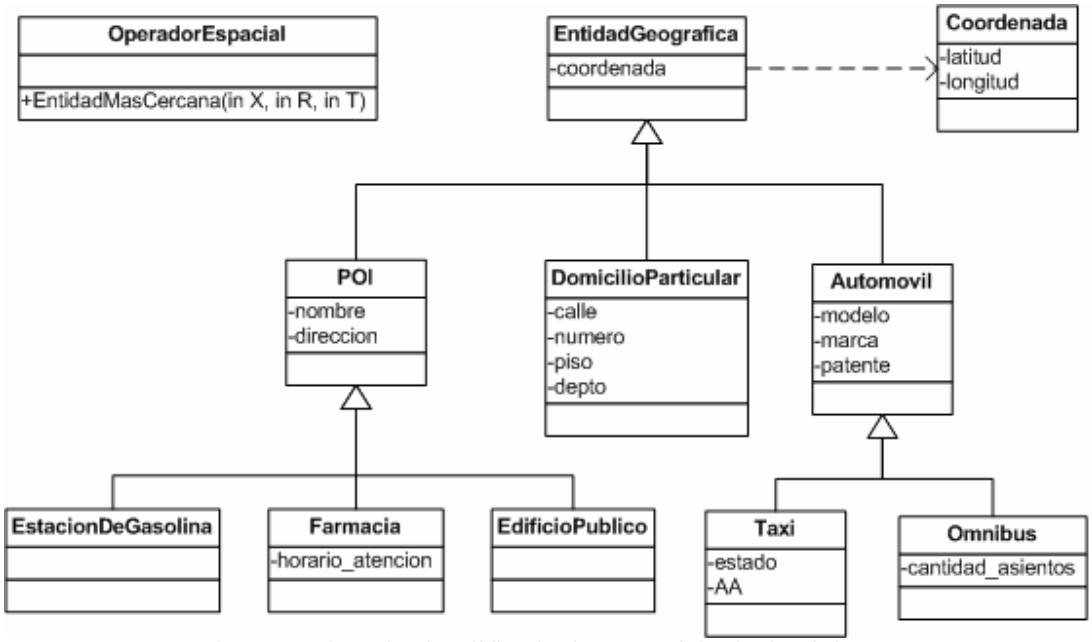


Figura 8. Ejemplo simplificado de un modelo de dominio

Otro punto a tener en cuenta en este escenario, es que si las dos aplicaciones enunciadas previamente tuvieran diferentes tipos de clientes de presentación y si ellos albergaran la lógica de la aplicación, ésta estaría distribuida en cada capa cliente, dificultando su mantenimiento. Por estos motivos se justifica el uso de una Capa de Servicios sobre el modelo de negocio, que juega el papel de fachada (patrón Facade[5]). Es decir la Capa de Servicios se encarga de exponer los servicios necesarios en la aplicación hacia la capa de presentación o hacia otros sistemas. La capa de presentación solo utiliza las funcionalidades de los servicios y expone de forma amigable y eficiente interfaces al usuario para la recolección y visualización de la información vinculada a dichos servicios.
Esta fachada conoce al modelo y en cada servicio expuesto hará uso de los objetos del dominio para la resolución del mismo. En este escenario, la capa de presentación puede estar en otro espacio físico distinto al de la capa de servicios, y juega el papel de interfaz remota. También puede suceder que los servicios sean consumidos por otras aplicaciones Figura 9.

[5] *Facade*: patrón de diseño que se utiliza para proveer una interfaz unificada sencilla que haga de intermediaria entre un cliente y una interfaz o grupo de interfaces más complejas[Gamma95].

Este diseño lógico particular, en donde los procesos de negocio se exponen como servicios, es la clave de la flexibilidad de la arquitectura, ya que permite que otras piezas de funcionalidad (incluso también implementadas como servicios) hagan uso de otros servicios de manera natural, sin importar su ubicación física. De esta forma un sistema puede evolucionar con la adición de nuevos servicios, o con la evolución independiente de los servicios existentes [Guinea07].
Esta arquitectura, conocida como "Arquitectura Orientada a Servicios" (SOA), se ha convertido en un paradigma para la integración entre sistemas libres y propietarios.

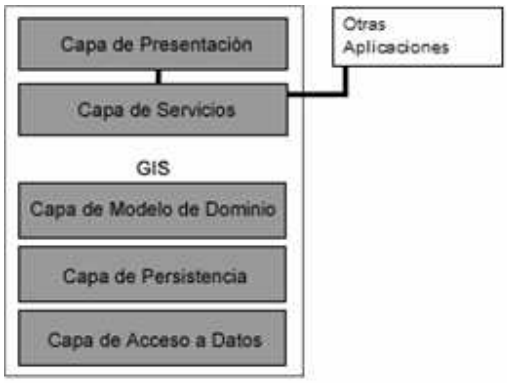


Figura 9. Capa de Servicios

1.6.3 Un problema común en las arquitecturas GIS

Si evaluamos desde la *vista dinámica* (cómo se comportan los componentes a lo largo del tiempo y como interactúan entre sí) las arquitecturas monolíticas desordenadas (organizadas en subsistemas) y las clientes servidor de n-capas, vamos a poder observar que, a pesar de que estén bien diferenciados los roles de cada componente, siguen existiendo dependencias explícitas entre los componentes de una capa/subsistema y los componentes de otras capas/subsistemas. Veámoslo con el ejemplo anterior de la arquitectura orientada a servicios; sabemos que un servicio del Sistema de Asignación de Taxis se encarga de encontrar el Taxi más cercano a la dirección de un cliente que realizó una solicitud. Este servicio codifica las reglas de negocio necesarias y utiliza objetos del dominio (*Operador Espacial*) para resolver el problema. A su vez este servicio, cada vez que encuentra un taxi debe almacenar en la base de datos la relación: *El taxi X pasa a buscar al cliente C*. Para resolver este tema de persistencia, el servicio interactuará con algún componente de la capa de Persistencia. Hasta aquí todo suena lógico, el servicio interactúa con componentes de las capas de Modelo de Dominio y de Persistencia. Las preguntas que surgen ahora son:

1. ¿Es correcto que el servicio interactué explícitamente con los componentes del Modelo?
2. ¿Es correcto que el servicio interactué explícitamente con los componentes de la Capa de Persistencia?

La respuesta a la primera pregunta es SI, ya que el servicio necesariamente debe conocer cuales son los componentes del modelo para resolver la funcionalidad que ofrece.

En cuanto a la segunda pregunta, no podemos decir lo mismo, ya que la cualidad de persistir entidades y sus relaciones en una base de datos es un aspecto de infraestructura que surge de la volatibilidad de los datos, y no forma parte de una regla del negocio. Por este motivo, para el servicio debería ser transparente la persistencia y por ende no debería existir la comunicación explicita entre el servicio y los componentes de la capa de persistencia.
Este escenario en donde un componente *A* hace referencia a otro componente *B* cuando en realidad *B* debería ser transparente para *A* sucede muy a menudo en los sistemas de información. Este hecho no hace más que establecer dependencias innecesarias entre ciertos componentes; dependencias que a la hora en que el sistema debe evolucionar, se transforman en tareas muy complejas de diseño y codificación. Cuando sucede esta situación en el que un componente depende fuertemente de otro decimos que existe un alto grado de acoplamiento en los componentes.
Para lograr independencia entre los componentes, la nueva pregunta que surge en relación al ejemplo planteado del servicio y la capa de persistencia es: ¿Como hacemos para relacionar al servicio con los componentes de la Capa de Persistencia sin establecer una dependencia explicita en el código del servicio?

Básicamente lo que nos plantea esta pregunta es como se puede reducir el grado de acoplamiento entre los componentes. Para ello primero nos concentraremos en el diseño de los componentes desde sus orígenes, es decir, desde el análisis de los requerimientos que deben cumplimentar. Creemos que comenzar el diseño de componentes desde esta etapa del ciclo de vida del software permitirá luego modularizar adecuadamente los componentes de modo de que cada uno de ellos trate en forma individual un aspecto o incumbencia del sistema, a estas incumbencias se las denomina *concerns* y es el foco principal del próximo capítulo.

Capítulo 2. Identificación y Separación de Concerns en GIS

En el capítulo anterior se describieron los conceptos principales de un GIS, haciendo foco principalmente en algunas de las arquitecturas más utilizadas a lo largo de la historia de los GISs. Se ha observado que las arquitecturas basadas en capas presentan tantas ventajas para los sistemas de información que ningún proyecto de GIS debería comenzar sin al menos evaluar seriamente la opción de enfocarlo desde esta perspectiva.

Estas arquitecturas, sin embargo, plantean un desafío complejo a los arquitectos de software, que consiste en identificar claramente los componentes de cada capa lógica, y sobre todo como será la interacción y las dependencias entre los componentes de modo de no acoplarlos cuando no es necesario hacerlo. Recordemos que una mala decisión de diseño en la definición de estos componentes de la arquitectura atenta contra uno de los tres factores que consideramos necesario alcanzar en todo GIS: *poder generar herramientas de software flexibles que permitan integrarse con otros GISs, y que permitan adaptarse a los cambios tecnológicos y a las nuevas reglas de negocio que evolucionan en el tiempo [Alameh01]*.

Para evitar este problema, y como primer paso del proceso propuesto en esta Tesis, consideramos necesario realizar una clara identificación y separación de los componentes de la arquitectura del GIS, y para esto recurriremos a un método que permita identificar y separar los aspectos o incumbencias relevantes del GIS desde las etapas más tempranas del ciclo de vida del software. Este proceso es conocido como Separación de Concerns (en inglés *Separation of Concerns*) y es el foco de análisis en este capítulo.

2.1 La Separación de Concerns

Existen diversas definiciones sobre este concepto en la comunidad de investigadores, la primera se puede atribuir a un paper del año 1974 de Edsger W. Dijkstra llamado "*En el Rol del Pensamiento Científico*" [Dijkstra74], donde el autor identifica un patrón común en el ámbito de la Ciencia de la Computación basado en que uno puede procurar estudiar en profundidad un aspecto que le incumbe en asilamiento del resto de las incumbencias. El principio básicamente consiste en encapsular características en entidades separadas con la finalidad de ubicar en un lugar particular sus cambios y tratarlas una por una en el tiempo.
Para dar un ejemplo, enfatiza que es sabido que un programa debe ser correcto, y podemos estudiarlo solamente desde ese punto de vista; también sabemos que debe ser eficiente y podemos estudiar su eficiencia desde este otro punto de vista. En otras palabras, lo que sugiere es abordar estos aspectos en forma individual pero simultáneamente. A esto lo ha llamado "separación de aspectos o incumbencias", que aunque no sea perfectamente

posible, es una técnica efectiva para ordenar eficazmente nuestros pensamientos. Focalizar la atención en un aspecto (o *concern* en inglés), no significa ignorar el resto de los aspectos, sino que desde el punto de vista de este aspecto, el otro es irrelevante.
El concepto de *concern* aparece frecuentemente asociado a incumbencia, competencia, área de interés, entre otros; para efectos de este trabajo mantendremos su sintaxis en inglés con el fin unificar términos.

Un *concern* se define como una propiedad, o punto de interés de un sistema [Filman05] [Gutierrez04]. La IEEE en [IEEE1471-00] define los *concerns* para un sistema como aquellos intereses que pertenecen al desarrollo del sistema y su operación, o demás aspectos que son críticos; o por el contrario, importantes para uno o varios *stakeholders* o participantes del proyecto de desarrollo de software. En [Laddad03] se define un *concern* como un requisito de un sistema o consideración que debe ser direccionada con el fin de satisfacer una meta del sistema.

En base a estas definiciones, *La Separación de Concerns* ("Separation of Concerns", SoC) se puede definir como la habilidad de identificar, encapsular y manipular aquellas partes del software que son relevantes para a un concepto, meta o propósito. Los Concerns son el medio principal para organizar y descomponer el software en partes más pequeñas, más entendibles y manejables [ICSE 2000].

Para otros autores [Creer08], la SoC es un principio y a la vez un proceso. El principio de la SoC se basa en que las cosas no deben ser ni más ni menos lo que deberían ser. Es decir, las cosas deben contener los atributos esenciales y comportamientos inherentes a su naturaleza, pero deben estar exentas de aquellos que no lo son.
El proceso de la SoC, por otro lado, se compone de las técnicas que determinan los *límites* de los concerns. Los límites usualmente son establecidos mediante el uso de métodos, objetos, componentes, y servicios que definen el comportamiento de la aplicación; proyectos, soluciones, capas lógicas de la aplicación, bibliotecas versionadas e instaladores de las releases del producto.

Hay que tener en cuenta, que la SoC no consiste en la división del sistema en múltiples subsistemas, sino en el establecimiento de límites (*boundaries*) basados en la esencia. Podemos concluir que el objetivo general de la SoC es establecer un sistema bien organizado, donde cada componente/módulo cumple una función significativa e intuitiva, para aumentar al máximo su capacidad y adaptarse a los cambios. Lograr la separación de concerns es un objetivo guiado más por el principio que por reglas, por esto, la SoC podría considerarse un arte más allá de la ciencia.

La SoC como principio y proceso, puede ser aplicada en las distintas fases de construcción del un sistema, independientemente del modelo de ingeniería utilizado (Cascada, Espiral, etc). En cada una de estas fases usualmente intervienen recursos humanos especializados (*stakeholders*) con un rol en particular, por ejemplo, en la fase de Análisis de Requerimientos se ven involucrados Analistas Funcionales, Analistas de Sistemas, mientras que en la fase de Diseño los involucrados son los Arquitectos de Software y Diseñadores.
Si bien el principio de la SoC está más ligado a los stakeholders que cumplen roles en las etapas de análisis y diseño, cualquier stakeholders podría organizar sus pensamientos en

base al principio de la SoC aunque algunos de los concerns identificados puedan no ser relevantes para la arquitectura del GIS. Por ejemplo, un concern para un Project Manager puede ser el *Costo* o los *Tiempos* del sistema, mientras que para un diseñador que está construyendo un modelo orientado a objetos puede ser el *Encapsulamiento de los Datos*, que logra alcanzarlo (o "implementarlo") a través de un componente de software sencillo, la Clase.
Para un Analista Funcional, este concern no es relevante, ya que la meta que éste persigue es identificar los requerimientos funcionales del sistema sin saber como serán implementados. Ejemplo de estos requerimientos funcionales en un GIS son: que el sistema pueda imprimir, persistir y visualizar datos geográficos (correspondiendo con los concerns de *Impresión*, *Persistencia* y *Visualización* respectivamente). De esta forma, podemos observar que la separación de concerns puede ser realizada en distintos niveles de abstracción de acuerdo a la fase del modelo de construcción del sistema.

Existen razones manifiestas por las cuales el tratamiento de los concerns debe ser llevado a cabo desde las primeras fases del ciclo de vida del software [Clements04]. La principal motivación de estos esfuerzos de investigación parte de la premisa de que la consideración temprana de los concerns, como temas de interés transversales a más de un stakholder, logrará un mejor entendimiento del problema y facilitará su evolución a las fases siguientes.

En este trabajo, donde perseguimos la definición de una arquitectura GIS flexible, creemos necesario analizar y separar los concerns antes de comenzar la fase de Diseño de Arquitectura, esto significa llevar a cabo esta separación desde los niveles de abstracción más altos de la construcción del sistema, en particular, nos focalizaremos en la separación de concerns en la etapa de Análisis de Requerimientos.
Separar Concerns en esta etapa del ciclo de vida del software nos permitirá generar un documento formal donde se especifiquen los concerns y sus relaciones, que podrá luego ser utilizado como herramienta para realizar el diseño de la arquitectura lógica del GIS.
A continuación, analizaremos un modelo de separación de concerns en la etapa de Análisis de Requerimientos que satisface nuestras necesidades, permitiendo obtener una especificación formal de los concerns y sus relaciones.

2.2 La Separación de Concerns en la fase de Análisis de Requerimientos (AORE)

El tratamiento temprano de los concerns que componen un sistema se ha convertido en uno de los campos que ha generado mayor interés en los últimos años. Actualmente existe una diversidad de aproximaciones que proponen la introducción de concerns en la etapa de análisis de requerimientos. Cada aproximación ha sido analizada en diferentes trabajos (por ejemplo, en [Bakker05]), y es objeto de discusiones en diferentes escenarios del ámbito tecnológico. En este trabajo, dejaremos de lado estas discusiones, y nos focalizaremos en el modelo propuesto en [Moreira1_05], [Moreira2_05] y extendido en [Ospina07], conocido como *AORE Multi-Dimensional*.

Este modelo se adapta perfectamente a las necesidades planteadas; básicamente tiene por objetivo realizar una descomposición de los requerimientos de una manera uniforme, sin importar su naturaleza funcional o no funcional.

En este modelo, los concerns mantienen encapsulados conjuntos coherentes de requerimientos tanto funcionales como no funcionales. Esta propuesta no restringe la manera en como los concerns son definidos; esto significa que el relevamiento de los requerimientos se podría realizar utilizando cualquiera de las aproximaciones existentes.
Este modelo tampoco está ideado para algún tipo de sistema de información en particular y se adapta perfectamente al dominio de los sistemas de información geográfica.
El modelo AORE Multi-Dimensional extendido consta de las siguientes tareas (Ver Figura 10):

1. Identificar y especificar concerns
2. Clasificar concerns
3. Identificar relaciones de grano grueso entre concerns
4. Especificar proyecciones de concerns usando reglas de composición.
5. Complementar relaciones de grano grueso entre concerns
6. Manejar conflictos
7. Especificar dimensiones de concerns

El cumplimiento de estas tareas, nos permitirá obtener un documento formal que servirá como punto de partida para el diseño de la arquitectura lógica.

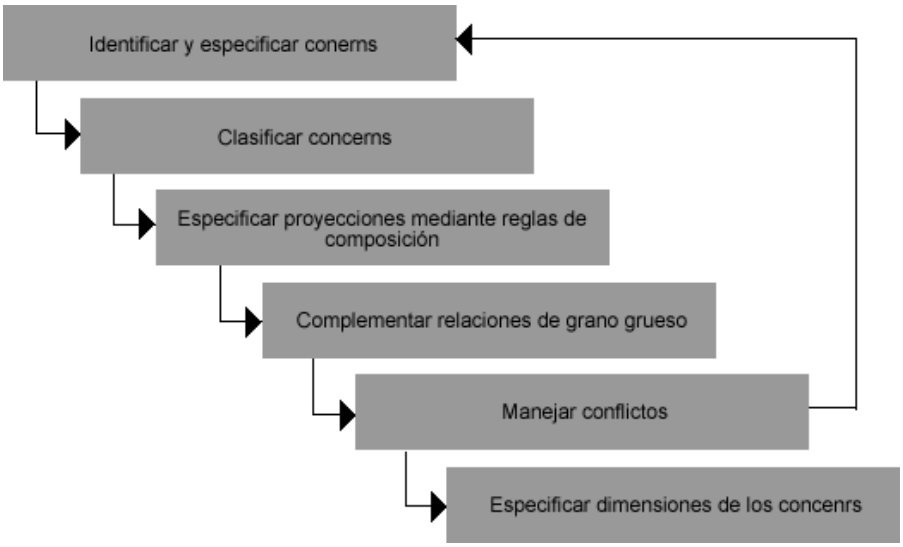


Figura 10. Modelo AORE extendido

A continuación describiremos brevemente cada una de las tareas del modelo aunque haremos hincapié en las tareas más relevantes desde el punto de vista del diseño de la arquitectura.

2.2.1 Identificando y especificando concerns

Es necesario tener en cuenta que el modelo AORE Multi-Dimensional, como así también las etapas posteriores de nuestro proceso, no definen una regla o metodología precisa para la identificación de los concerns. Esta identificación puede ser llevada a cabo a través de cualquier mecanismo de elicitación existente, como ViewPoints [Finkelstein96], Casos de Uso [Jacobson92] y Metas [Lamsweerde01].

Más allá de estos buenos mecanismos de elicitación, consideramos que la clave de una correcta identificación de concerns está en la habilidad del analista de requerimientos, esto significa que diferentes analistas con diferentes habilidades pueden identificar distintos concerns.
Por otro lado, nuestro proceso tampoco intenta identificar todos los concerns de los GISs, ya que estos presentan distintos dominios y resuelven distintas problemáticas. Por ejemplo, los concerns presentes en el Sistema de Asignación de Taxis pueden ser muy diferentes a los concerns identificados en un GIS agropecuario que permite procesar información sobre las diferentes labores sobre las parcelas de un campo (aplicación de agroquímicos, siembra, fertilización, pulverización, cosecha, etc.)
A pesar de estas diferencias de domino, existen algunos concerns que se hacen presentes en la mayoría de los GISs, como la *Ubicación, Cartografía*, *Operaciones Espaciales*, *Persistencia*, *Visualización*, etc. El análisis y especificación de los concerns propios de los GISs está fuera del alcance de esta tesis siendo parte de nuestros trabajos futuros, sin embargo, utilizaremos algunos de ellos para explicar el proceso propuesto. A estos concerns, en el marco de AORE Multi-Dimensional, se los denomina *Metaconcerns* (notar que estos ejemplos representan tanto concerns funcionales como no funcionales). Basándose en esta observación, el espacio de requerimientos se puede dividir en dos espacios separados:

- *Espacio de Meta-Concerns:* conformado por los conjuntos de concerns abstractos (funcionales y no funcionales) que se suelen repetir en la mayoría de los GISs.

- *Espacio del Sistema:* conformado por las distintas aplicaciones GIS que se desean implementar.

Cada aplicación perteneciente al Espacio del Sistema tiene ciertas características deseables que a través de entrevistas, estudios etnográficos, análisis de prácticas empresariales, etc., conducen al establecimiento de los requerimientos de la aplicación (triángulos grises en la Figura 11). A medida que los requerimientos se van derivando de las características deseables de las aplicaciones, se pueden ir abstrayendo y categorizando como concerns del Espacio de Meta-Concerns (línea punteada de la Figura 11). A estos concerns se los conoce como *concerns abstractos*, ya que los requerimientos representan abstracciones de los requerimientos que se repiten en sucesivas aplicaciones.
Los óvalos de la Figura 11 corresponden con definiciones concretas (*concerns concretos*) correspondientes al dominio específico de la aplicación, derivadas del los concerns abstractos, y pueden ser alcanzados iterativamente e incrementalmente, manipulando un conjunto pequeño de concerns en forma individual.

Es necesario tener en cuenta, que no todos los concerns del Espacio de Meta-Concerns son utilizados necesariamente durante esta categorización; en general solo serán necesarios aquellos concerns que sean relevantes al problema a resolver.
La categorización de requerimientos en concerns concretos conduce a la creación de una relación entre el Espacio de Meta-Concerns y el Espacio del Sistema (flecha gris de la Figura 11). Esto crea una unión conceptual entre la representación abstracta de los concerns en el Espacio de Meta-concerns con sus representaciones concretas del Espacio del Sistema.

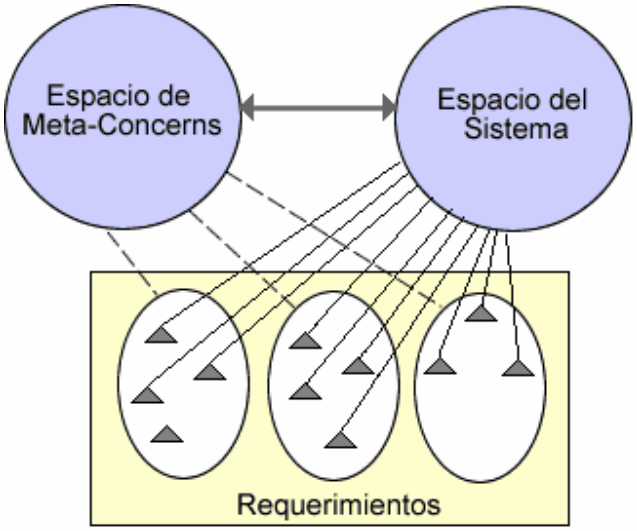


Figura 11. Espacio de Meta-Concerns

Para entender mejor el proceso de identificación y separación de concerns de AORE Multi-Dimensional, utilizaremos como caso de estudio un GIS (simplificado) previamente nombrado, el *Sistema de Asignación de Taxis* (basado en un ejemplo de [Moreira1_05]). Entre varias funcionalidades que provee este GIS, destacamos la posibilidad de recibir solicitudes de clientes que se encuentran en distintos puntos de una ciudad. Estas solicitudes se hacen a través de dispositivos móviles. Ante una solicitud, se verifican los datos personales del cliente por seguridad. Como el sistema de Asignación de Taxis no posee información acerca de los datos personales de sus clientes, la autenticación se hace mediante el número de teléfono del cliente interactuando con un sistema de una empresa de telecomunicaciones que sí dispone de los datos personales. Una vez que se identifica al cliente, el sistema debe ubicar el taxi libre más cercano al cliente. Cuando lo encuentra, notifica al taxista que debe recoger al cliente, indicándole la dirección en donde se encuentra situado el mismo. En paralelo, el sistema notifica al cliente quien es el taxista que lo recogerá, el tiempo estimado que tardará en hacerlo, y mostrará en un mapa digital la ubicación del taxista.
Cabe destacar que el cliente, puede configurar preferencias acerca de las facilidades que debe tener el taxi que debe pasar a buscarlo (ej: tipo de automóvil, aire acondicionado, taxista no fumador).

Tomando este caso de estudio podemos observar que existen varios Meta-Concerns, como por ejemplo la *Cartografía*, ya que la mayoría de los GISs suelen utilizar un conjunto de mapas digitales para representar los datos espaciales del dominio. Otro meta-concern puede

ser la *Ubicación*, ya que las entidades espaciales deben estar georeferenciadas en el marco de un sistema de referencia concreto. Otro meta-concern comprende las *Operaciones Espaciales*, ya que usualmente se deben realizar operaciones espaciales entre las entidades georeferenciadas del dominio (como adyacencia, superposición, cálculo de distancias, etc). Otros meta-concerns más fáciles de identificar, y que suelen estar presentes en la mayoría de los sistemas de información (además de en los GIS) son: *Comunicación*, *Autenticación*, *Visualización* y *Persistencia.*

Una vez identificados los meta-concerns, se deben especificar formalmente utilizando templates bien definidos basados en el estándar Extensible Markup Language (XML)[6].

En la Figura 12, se pueden ver a modo de ejemplo, la representación en XML de los meta-concerns *Cartografía*, *Operaciones Espaciales* y *Ubicación*. Estas definiciones abstractas se denotan mediante los tags <*MetaConcern*> </*MetaConcern*>. Estos tags presentan dos atributos, uno llamado *name* que representa el nombre del concern del Espacio de Meta-Concerns.

La definición también incluye una breve descripción del concern, y algunos ejemplos típicos derivados de experiencias pasadas con otros sistemas o conocimientos del dominio.

A su vez se especifican las relaciones con el resto de los meta-concerns con los que existe algún tipo de interacción o influencia. En el modelo AORE original, las relaciones se especifican en una cadena con los nombres de los meta-concerns separados por comas entre los tags <*Relationships*> </*Relationships*>. Sin embargo, nosotros hemos extendido este modelo introduciendo una estructura más formal que permita especificar de forma individual la influencia con cada meta-concern. Esta estructura consiste en especificar cada relación con el tag <*Relation*> (ver Figura 12). Esta extensión permite definir cada una de las relaciones en un nodo hijo distinto, presentando las siguientes ventajas:

1. Es más legible para la persona que está interpretando el archivo XML.
2. Facilita las búsquedas (manualmente o mediante herramientas específicas que interpretan el archivo).
3. Se pueden validar las relaciones, asegurándose que el atributo *name* del tag *Relation* corresponda con un nombre válido de un meta-concern existente (atributo *name* del tag *MetaConcern*.

```
<MetaConcern name="Cartografía">
  <Description>Representa las entidades espaciales del sistema</Description>
  <Examples> ejes viales, puntos de interes, cañerías, rutas, etc</Examples>
  <Relationships>
    <Relation name="Operaciones Espaciales"/>
  </Relationships>
</MetaConcern>

<MetaConcern name="Operaciones Espaciales">
  <Description>Conjunto de operaciones espaciales que se realizan entre entidades
georeferenciadas</Description>
  <Examples> Calculo de distancias, adyacencias, superposición de entidades,
etc</Examples>
  <Relationships>
  </Relationships>
</MetaConcern>
```

[6] *XML*: es un metalenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (W3C). Permite definir la gramática de lenguajes específicos.

```
<MetaConcern name="Ubicación">
  <Description>Representa la posiciónn geografica de una entidad en en el marco de
un sistema de referencias</Description>
  <Examples>coordenandas (latitud,longitud), (latitud, longitud, altura),
etc</Examples>
  <Relationships>
    <Relation name="Cartografía"/>
    <Relation name="Operaciones Espaciales"/>
  </Relationships>
</MetaConcern>
```

Figura 12. Ejemplos de Meta-Concerns representados con AORE Multi-Dimensional

Por otro lado, nuestro caso de estudio representa un sistema en particular del *Espacio de Sistemas*, en donde se puede identificar un conjunto de concerns concretos pertenecientes al problema planteado. Estos concerns concretos son:

- *Cliente*: representa las necesidades del cliente, como solicitar un taxi, cancelar una solicitud, especificar preferencias, etc.
- *Taxi*: representa las necesidades de un taxista, como tomar o rechazar una solicitud, etc.
- *Administrador de Taxis*: entidad del sistema responsable de decidir qué taxi puede tomar una solicitud realizada por un cliente.
- *Ciudad:* cartografía utilizada por el *Administrador de Taxis* para representar geográficamente las calles de la ciudad en donde se mueven los taxis y los clientes. Corresponde con el meta-concern *Cartografía*.
- *Ubicación:* representa la posición geográfica de los *Taxis* y los *Clientes* en el marco de un sistema de coordenadas. Corresponde con el meta-concern *Ubicación*.
- *Operador Espacial:* comprende el conjunto de operaciones espaciales que se pueden realizar entre las entidades georeferenciadas, como calcular la menor distancia entre entidades, etc. Corresponde con el meta-concern *Operaciones Espaciales.*
- *Persistencia:* comprende todas las tareas de almacenamiento y obtención de datos del sistema. Corresponde con el meta-concern *Persistencia.*
- *Autenticación:* tanto los taxistas como los clientes deben identificarse y ser usuarios válidos para poder hacer uso del sistema. Un taxi es válido si el número de teléfono se encuentra registrado en el sistema, mientras que un cliente es válido si el teléfono se corresponde con un usuario de la empresa de telecomunicaciones. Corresponde con el meta-concern *Autenticación.*
- *Visualización:* los clientes deben poder visualizar la información del taxista que lo pasará a buscar (quién es y el tiempo estimado), y el taxista debe poder visualizar información del cliente (datos personales y ubicación). Corresponde con el meta-concern *Visualización*.
- *Comunicación:* los clientes y los taxis deben poder comunicarse mediante algún protocolo de comunicación con el *Administrador de Taxis* y viceversa. A su vez este último debe comunicarse con la empresa de telecomunicaciones. Corresponde con el meta-concern *Comunicación*.

Al igual que los meta-concerns, los concerns concretos deben ser especificados en un archivo XML. Como se puede observar en la Figura 13, cada concern concreto se describe mediante el tag <*Concern*></*Concern*>, y los nodos hijos que representan los

requerimeintos del concern se describen mediante el tag <*Requirement*></*Requirement*> y se los identifica univocamente mediante el atributo *id.*
A continuación en la Figur a13 podemos observar ejemplos de algunos concerns concretos y sus requerimientos:

```
<Concern name="Cliente">
  <Requirement id="1">
    Debe poder solicitar al Administrador de Taxis un taxi a través de su dispositivo
electrónico
    <Requirement id="1.1">Debe poder obtener su ubicación geografica</Requirement>
    <Requirement id="1.2">Debe poder enviar al Administrador de Taxi la solicutud con la
ubicación, su nombre de usuario, su numero de telefono y las preferencias</Requirement>
  </Requirement>
  <Requirement id="2">
    Debe poder especificar las preferecnias del taxi
      <Requirement id="2.1">Debe poder indicar el tipo de vehículo que necesita</Requirement>
      <Requirement id="2.2">Debe poder indicar las comodidades que debe tener el
vehículo</Requirement>
      <Requirement id="2.3">Debe poder indicar si es fumador o no</Requirement>
  </Requirement>
  <Requirement id="3">
    Debe visualizar en su dispositivo cual es el taxi que lo pasará a buscar y el tiempo
estimado
  </Requirement>
  <Requirement id="4">
    Debe poder cancelar una solicitud previamente realizada
  </Requirement>
</Concern>

<Concern name="Administrador de Taxis">
  <Requirement id="1">
     Debe recibir peticiones de inicio y fin de servicio de los taxis
    <Requirement id="1.1">Debe validar el nombre de usuario y la contraseña enviada por el
taxi</Requirement>
  </Requirement>
   <Requirement id="2">
     Debe gestionar la ubicación de todos los taxis que están en servicio
     <Requirement id="2.1">Debe poder recibir la ubicación de un taxi</Requirement>
     <Requirement id="2.2">Debe almacenar la ubicación del taxi en una base de
datos</Requirement>
     <Requirement id="2.3">Debe representar gráficamente en la ciudad la ubicación del
taxi</Requirement>
  </Requirement>
  <Requirement id="3">
    Debe recibir la notificación de cuando un taxi se libera
  </Requirement>
  <Requirement id="4">
    Debe gestionar solicitudes de los clientes
<Requirement id="4.1">Debe recibir del cliente el nombre de usuario, el número de
telefono, la ubicación y las prefencias del taxi</Requirement>
  <Requirement id="4.2">Debe almacenar la solictud del cliente en una base de
datos</Requirement>
    <Requirement id="4.3">Debe enviar al cliente un mensaje de recepción exitosa de la
solicitud</Requirement>
  </Requirement>
  <Requirement id="5">
    Debe recibir cancelaciones de solicitudes de los clientes
    <Requirement id="5.1">Debe notificar al taxi que el viaje se cancela</Requirement>
  </Requirement>
  <Requirement id="6">
    Debe verificar el cliente en el sistema de la entidad externa de telecomunicaciones
    <Requirement id="6.1">Debe enviar el número de teléfono del cliente</Requirement>
```

```
    <Requirement id="6.2">Debe recibir los datos personales del cliente</Requirement>
  </Requirement>
  <Requirement id="7">
    Debe asignar un taxi a una solicitud realizada por un cliente
    <Requirement id="7.1">Debe encontrar el taxi libre más cercano a la ubicacón del cliente
que cumpla con las preferencias</Requirement>
    <Requirement id="7.2">Debe enviar al taxi la solicitud</Requirement>
    <Requirement id="7.3">Debe recibir la confirmación del taxi indicando si acepta o no la
solicitud</Requirement>
  </Requirement>
  <Requirement id="8">
    Se debe poder visualizar en un mapa digital de la ciudad la ubicación de los taxis
  </Requirement>
  <Requirement id="9">
    Se debe poder visualizar en un mapa digital de la ciudad la ubicación de los clientes que
realizaron solictudes
  </Requirement>
  <Requirement id="10">
    Debe poder almacenar el mapa de la ciudad
  </Requirement>
</Concern>

<Concern name="Autenticación">
  <Requirement id="1">
    Se debe poder autenticar un usuario a través de un número de teléfono
  </Requirement>
  <Requirement id="2">
    Se debe poder autenticar un usuario a través de un nombre de usuario y contraseña
  </Requirement>
</Concern>
```

Figura 13. Ejemplo de concerns concretos: Cliente, Administrador de Taxis y Autenticación

La información completa de todos los concerns concretos del caso de estudio se puede observar en el *Anexo II – AORE Multidimensional - Concerns Concretos*.

2.2.2 Clasificando concerns

Esta tarea está orientada a proveer la clasificación de los concerns identificados y que hacen parte de los elementos de definición del sistema requerido. Los distintos tipos de concerns propuestos por [Ospina07] pueden ser vistos en la tabla de la Figura 14, donde en la columna *Clasificación* se encuentran los distintos tipos de concerns, en la columna *Descripción* una breve reseña que describe el tipo de concerns, y finalmente hemos agregado una tercer columna con la clasificación de algunos de los concerns de nuestro caso de estudio:

Clasificación	Descripción	Concerns
Objetivo	Objetivos del sistema, los cuales pueden estar agrupados o ser vistos de manera individual como un concern.	
Negocio	Es una colección coherente de requisitos, los cuales pueden ser funcionales, de información y reglas de negocio.	Cliente, Taxi, Administrador de Taxi, Operador Espacial, Ciudad, Ubicación
Riesgos	Son concerns que representan posibles dificultades a las cuales se expone el sistema en desarrollo.	
Calidad	son concerns asociados a características no funcionales del software (“ilities”) y requisitos no funcionales del sistema que se tenga en cuestión. Estas características pueden afectar	Autenticación, Persistencia, Visualización,

	todo el sistema o sólo partes de éste.	Comunicación
Físicos	Estos concerns comprenden un conjunto amplio de elementos que no son propios del sistema, pero los cuales pueden afectarlo considerablemente en cualquiera de las etapas. Estos concerns comprenden características tales como infraestructura, plataforma (Sistema Operativo), componentes externos, entorno de desarrollo, dispositivos, equipos y frameworks.	Dispositivo Electrónico
Contexto	Este tipo de concerns comprende elementos del sistema que no son susceptibles a cambios y el sistema los debe considerar. Ejemplos de elementos que se consideran concerns de contexto son: restricciones de ley, servicios externos, sistemas existentes, modelos de referencia.	

Figura 14. Clasificación de Concerns

Clasificar los concerns nos permite saber con que clase de concern estamos tratando a la hora de leer la especificación de concerns concretos. El tipo de concern significará para algunos stakeholders un indicio importante para tomar ciertas dediciones, por ejemplo, para un arquitecto de software, el tipo de concern determinará si un concern será mapeado como un componente de software, como una decisión tecnológica, o simplemente es una decisión del negocio que no afecta la arquitectura lógica del sistema.

Como parte de la extensión al modelo AORE, introducimos el atributo type a la definición del concern concreto de modo que el tipo de concern quede reflejado en nuestra especificación de concerns. La Figura 15 muestra la incorporación de este tag:

```
<Concern name="Autenticación" type="Calidad" >
  <Requirement id="1">
    Se debe poder autenticar un usuario a través de un número de teléfono
  </Requirement>
  <Requirement id="2">
    Se debe poder autenticar un usuario a través de un nombre de usuario y contraseña
  </Requirement>
</Concern>
```

Figura 15. Ejemplo de concern cuyo tipo es Calidad

2.2.3 Identificando relaciones de grano grueso entre concerns

El próximo paso es identificar las relaciones de grano grueso entre los concerns, es decir las influencias de los concerns sobre el resto. Esta relación de "influencia" se representa mediante en una matriz, en cuyas intersecciones se señala la influencia de un concern con respecto a otro. Estas relaciones se pueden identificar analizando en detalle los requerimientos de los concerns concretos y es posible apoyarse en técnicas conocidas como el Análisis del Dominio[7] o el Procesamiento del Lenguaje Natural[8].

[7] *Análisis de Dominio:* es un paradigma funcionalista, porque intenta entender las funciones implícitas y explícitas de la información y la comunicación y de reconstruir la conducta informacional a partir de esto. Procura encontrar la base de la información en factores externos a la individualidad y subjetividad de los usuarios [Roche07].

[8] *Procesamiento del Lenguaje Natural*: es un campo de estudio de la inteligencia artificial y la lingüística computacional. Su objetivo es lograr la generación y comprensión automática de los

La forma de señalar una relación unidireccional entre dos concerns es a través de una tilde, y se interpreta leyendo la matriz de izquierda a derecha. Por ejemplo, entre el concern *Autenticación* y *Cliente* existe una relación unidireccional, ya que la autenticación influye en el comportamiento de un cliente, teniendo éste que identificarse en el sistema a través de su numero de teléfono. Por otro lado, existen relaciones bidireccionales, como por ejemplo, entre el concern *Cliente* y *Administrador de Taxis*, donde uno influye en el comportamiento del otro. Esto se puede ver cuando un cliente solicita un taxi y el administrador debe asignarle el taxi más cercano. Lo mismo sucede entre el concern *Taxi* y *Administrador de Taxis*, ya que los taxis se mueven en la ciudad y el administrador debe conocer permanentemente su ubicación para poder asignar los taxis a las solicitudes. Estas relaciones bidireccionales se representan en la matriz mediante dos tildes.

	Concern 1	Concern 2	Cliente	Taxi	Admin. de Taxis	Autenticación	Concern n
Concern 1							
Concern 2							
Cliente					✓		
Taxi					✓		
Admin. de Taxis			✓	✓			
Autenticación			✓	✓	✓		
Concern n							

Figura 16. Ejemplo de una Matriz de Relaciones de Granularidad Gruesa

La matriz de relaciones completa de todos los concerns del caso de estudio se puede observar en el *Anexo III – AORE Multidimensional - Matriz de Relaciones*.

Una vez especificadas las relaciones de grano grueso en la matriz, es posible generar una especificación de los concerns y sus relaciones de grano grueso. El esquema para especificar estos concerns es similar al de los meta-concerns, solo que cambia el tag <Meta-concern> por <Concern>. Esto se visualiza en la Figura 17.

```
<Concern name="Cliente" type="Negocio" >
  <Relationships>
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>

<Concern name="Administrador de Taxis" type="Negocio" >
  <Relationships>
    <Relation name="Cliente"/>
    <Relation name="Taxi"/>
  </Relationships>
</Concern>

<Concern name="Autenticación" type="Calidad" >
  <Relationships>
    <Relation name="Cliente"/>
    <Relation name="Taxi"/>
```

lenguajes naturales de los humanos [Manning99].

```
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>
```

Figura 17. Especificación de Concerns y sus relaciones de grano grueso

La especificación completa de todos los concerns del caso de estudio se puede observar en el *Anexo IV – AORE Multidimensional - Relaciones de Grano Grueso*.

El próximo paso consiste en especificar las relaciones de grano fino, es decir, las proyecciones de cada concern sobre el resto a nivel de requerimientos concretos. Esto se logra mediante reglas de composición. Estas reglas operan a nivel de granularidad de requerimientos y nos permiten especificar cómo un requerimiento de un concern influencia o restringe el comportamiento de uno o varios requerimientos de otros concerns.

2.2.4 Especificando las proyecciones de los concerns (relaciones de grano fino)

Habiendo estudiado el impacto de cada concern sobre el resto, ahora podemos comenzar a analizar con más detalle cada relación. Una proyección especifica la influencia de un concern dado (representado en una fila de la matriz de la Figura 16) sobre el resto de los concerns (representado en las columnas de la misma matriz). Cuando un concern afecta a otros concerns, es posible que tenga un impacto amplio en el sistema, y por este motivo puede ser clasificado como un *Concern Transversal* al sistema, más conocido como *Crosscuting Concern* en inglés.

Un crosscutting concern, evaluado desde este nivel de abstracción, indica que el concern tiene influencia en uno o más concerns del sistema, pero veremos más adelante, que en un nivel de abstracción más bajo, como lo es el diseño de la arquitectura, se convertirá en una funcionalidad que posiblemente se extienda a lo largo de múltiples componentes del sistema.

Para describir como un concern influencia o "atraviesa" a otros concerns, se utilizará una especificación basada en reglas de composición.

Como se puede observar en la Figura 16, no solo los concerns de Calidad como la *Autenticación* son crosscutting, sino que también lo pueden ser los concerns de Negocio como lo es el *Administrador de Taxis,* ya que tienen la misma naturaleza de influenciar al resto de los concerns.

Para definir las reglas de composición entre concerns también se utiliza XML. Esta representación semi-estructurada de los concerns y sus relaciones permite analizar los concerns y sus composiciones para el establecimiento de trade-offs (un trade-off se refiere a la pérdida de una cualidad de un aspecto en contraposición de ganar otras cualidades).

Estos archivos XML donde se definen las reglas de composición, pueden ser opcionalmente cumplimentados utilizando Esquemas XML (XML Schemas[9] en inglés).

Se ha escogido como lenguaje el XML, ya que mediante éste se pueden definir acciones específicas de cada concern y sus operadores de composición.

El modelo extensible ofrecido por XML, en conjunto con el rico modelo de especificación de Esquemas, hacen de ambos una elección ideal para definir las reglas de composición, ya

[9] *XML Schemas*: expresan vocabularios y permiten a las máquinas llevar a cabo reglas hechas por personas. Proveen una forma de definir la estructura, contenido y semántica de XML [W3C00].

que es imposible anticiparse a la cantidad de tipos distintos de operadores de composición y acciones que son requeridas. Como el lenguaje de Esquemas XML es extensible, ya que está basado en XML, es posible cumplimentar las restricciones de la especificación de las reglas de composición cuando se introducen nuevos operadores y acciones. A su vez, la habilidad de definir tags con significado semántico asegura la legibilidad de la especificación de los requerimientos. Por otro lado, la utilización de XML hace posible seleccionar cualquier proyección de interés utilizando consultas XPath[10] y observar su efecto acumulativo. Estas proyecciones pueden también ser visualizadas utilizando Extensible Stylesheet Language (XSL)[11]. Esto ayuda a la escalabilidad cuando existen en el sistema gran cantidad de concerns.
Para llevar a la práctica este acercamiento en forma simplificada, sólo nos acotaremos a la utilización de XML sin la utilización de esquemas.

La idea básicamente consiste en especificar un conjunto de reglas de composición por cada proyección de la matriz de relaciones. Las reglas de composición definen la relación entre los requerimientos de los concerns al nivel de granularidad fina (a diferencia de la matriz de relaciones que tiene el propósito de identificar relaciones al nivel de granularidad gruesa).
En la Figura 18 se puede observar un conjunto coherente de reglas de composición encapsuladas en tags llamados <Composition>. La Figura 18 encapsula algunas de las composiciones de los requerimientos de los concerns *Cliente*, *Administrador de Taxi* y *Autenticación*.
El concern al cual pertenece un requerimiento de la regla de composición es listado explícitamente como el atributo *concern*; esto es esencial para determinar el alcance del requerimiento. Como se explicó previamente, en las reglas de composición se está especificando la influencia de un requerimiento de un concern origen sobre otro/s requerimientos de un concern destino. Esta relación se denota mediante el tag <Requirement> (notar que aquí la semántica de este tag difiere de los tags en la definición de concerns de la Figura 15). Cada nodo debe especificar el tipo de influencia sobre el requerimiento afectado, esto en el modelo AORE se hace mediante el tag <*Constrain*>, pero por simplicidad nosotros lo hemos reducido a un atributo del tag <Requirement> llamado *constraint*. Esta simplificación, a nuestro criterio, facilita la lectura de las reglas de composición y permite expresar el mismo significado.
Este atributo *constraint,* define acciones y operadores que indican cómo los requerimientos de un concern son restringidos por los requerimientos de otros concerns. Si un requerimiento de un concern tuviese subrequerimientos, estos deberían especificarse en la restricción impuesta (*constraint)* indicando si estarán "Incluidos" o "Excluidos". Esto se logra proveyendo el valor "include" o "exclude" al atributo opcional *children* de cada nodo hijo. El valor "all" para el atributo *id* del tag <Requierement> implica que todos los subrequerimientos están restringidos.
A pesar de que las acciones y los operadores son informales, estos tienen una semántica y un significado claro para lograr la composición valida de los concerns. Esto provee a los

[10] *Lenguaje XPath*: es un lenguaje de consulta para buscar elementos de un documento XML [XPath99].
[11] *XSL*: del inglés Extensible Stylesheet Language, es una familia de recomendaciones para definir transformaciones y presentaciones de documentos XML. La especificación completa se encuentra en [XSL]

arquitectos y diseñadores un significado sistemático para interpretar la especificación de requerimientos.
Por otro lado, el modelo AORE sugiere el tag <*Outcome*> para definir el resultado de restringir los requerimientos del concern origen con un requerimiento de otro concern destino, aunque en este trabajo por simplicidad y cuestiones de legibilidad se dejará de lado, sin afectar la semántica del modelo.
Si observamos el ejemplo de la Figura 18, la primera regla de composición se interpreta de la siguiente manera: *"El requerimiento 1 del concern Cliente afecta al requerimiento 3 del concern Administrador de Taxis, afectando también a todos sus subrequerimientos*". Si buscamos por id de requerimiento en la especificación de concerns concretos, la frase podría incluso interpretarse como:
"La solicitud de un taxi por parte del cliente afecta al administrador de taxis ya que éste debe recibir la solicitud, almacenarla en la base de datos y responderle al cliente que recibió la solicitud exitosamente".

```
<Composition concern="Cliente">
   <Requirement id="1" constraint="afectan">
        <Requirement concern="Administrador de Taxis" id="3" children="include" />
  </Requirement>
   <Requirement id="2" constraint="afectan">
     <Requirement concern="Administrador de Taxis" id="7" children="include" />
   </Requirement>
   <Requirement id="4" constraint="afectan">
     <Requirement concern="Administrador de Taxis" id="5" children="include" />
   </Requirement>
</Composition>

<Composition concern="Administrador de taxi">
    <Requirement id="4.3" constraint="afectan">
    <Requirement concern="Cliente" id="1" children="include" />
  </Requirement>
  <Requirement id="5.1" constraint="afectan">
    <Requirement concern="Taxi" id="6"/>
  </Requirement>
  <Requirement id="7.2, 7.3" constraint="afectan">
    <Requirement concern="Taxi" id="4"/>
  </Requirement>
</Composition>

<Composition concern="Autenticación">
    <Requirement id="1" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="6"/>
  </Requirement>
  <Requirement id="2" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="1" children="include"/>
  </Requirement>
</Composition>
```

Figura 18. Reglas de Composición para los concerns Cliente, Administrador de Taxis y Autenticación

Las tablas de las Figuras 19 y 20 describen en su totalidad la semántica de las acciones y de los operadores de los Constraints y Outcomes respectivamente.

Acciones		
Tipo	**Significado**	**Descripción**
enforce	hace cumplir	Utilizado para imponer una condición adicional sobre un conjunto de requerimientos de un concern.
ensure	asegura	Utilizado para afirmar que una condición que debería existir para un conjunto de requerimientos realmente existe.
provide	provee	Utilizado para especificar las características adicionales que se incorporarán a un conjunto de requerimientos de un concern.
applie	aplica	Utilizado para describir reglas que aplican a un conjunto de requerimientos y pueden alterar su resultado (outcome).
exclude	excluye	Utilizado para excluir algunos de los subrequerimientos del concern, o todos si el valor del id es *all*
include	incluye	Utilizado para incluir algunos de los subrequerimientos del concern, o todos si el valor del id es *all*
affect	afecta	Utilizado para esecificar que un conjunto de requerimientos de un concern alterará el estado de otro concern

Figura 19. Acciones de los Constraint

Operadores		
Tipo	**Significado**	**Descripción**
during	durante	Describe el intervalo temporal en el que un conjunto de requerimientos es satisfecho
between	entre	Describe el intervalo temporal entre dos requerimientos que son satisfechos. El intervalo comienza cuando se satisface el primer requerimiento, y finaliza cuando el segundo comienza a ser satisfecho
on	sobre	Describe el punto temporal luego de que se satisface un conjunto de requerimientos.
for	para	Describe la existencia de características adicionales que complementarán los requerimientos del concern.
with	con	Describe la existencia de una condición que se sostendrá entre dos conjuntos de requerimientos.
in	en	Describe la existencia de una condición para un conjunto de requerimientos que se han satisfecho.
AND, OR, XOR	y, o, xor	Conjunción, Disyunción y Disyunción Exclusiva

Figura 20. Operadores de los Constraint

Un punto interesante a destacar es que no todas las combinaciones de acciones y operadores son válidas en la especificación de Constraints para un concern en particular. Esta

validación puede ser llevada a cabo a través de esquemas XML, pero por simplicidad está fuera del alcance de este trabajo.

2.2.5 Complementando relaciones de grano grueso entre concerns

Esta actividad se realiza posterior a la actividad de especificación de relaciones de grano fino, luego de haber adquirido una visión más clara de la manera en como los requisitos contribuyen o afectan a un concern. Esto permite replantear y complementar las relaciones de grano grueso definidas anteriormente en la matriz de contribución.
La actividad de definición de relaciones de grano fino implica establecer relaciones a nivel de los requerimientos que componen los concerns; en este proceso es posible encontrar que no existe una relación entre los requerimientos de los concerns, lo cual implica que la relación de grano grueso deba desaparecer. En la tabla de la Figura 21 se describen las relaciones de grano grueso refinadas que se propone en [Ospina07] como elemento de ampliación del modelo; en la tabla se tiene el nombre que identifica la relación de grano grueso refinada y una descripción semántica de la relación.

Relación de grano grueso	
Tipo	**Descripción**
Contribución	Un concern contribuye a otro cuando los requerimientos del concern origen ayudan al cumplimiento de los requerimientos del concern destino. Esta relación es vista como la relación recíproca a la afección. La contribución se da cuando el concern origen afecta de manera positiva al concern destino.
Restricción	Un concern restringe a otro cuando los requerimientos del concern origen restringen los requerimientos del concern destino.
Condición	Un concern condiciona a otro cuando los requerimientos del concern origen condicionan los requerimientos del concern destino. Se diferencia de la restricción por que las condiciones solo se aplican bajo ciertas reglas o circunstancias.
Uso	Un concern usa a otro concern cuando los requerimientos del concern origen usan requerimientos del concern destino.
Extensión	Un concern extiende a otro concern cuando los requerimientos del concern origen extienden los requerimientos del concern destino
Afección	Un concern afecta a otro concern cuando los requerimientos del concern origen afectan los requerimientos del concern destino. La afección es considerada como un aporte negativo entre los concerns. Esta relación se considera como la relación recíproca a la contribución.
Admisión	Un concern admite a otro concern cuando los requerimientos del concern origen admiten la aplicación de los requerimientos del concern destino.
Se aplica a	Un concern se aplica a otro concern cuando los requerimientos del concern origen son aplicados a los requerimientos del concern destino sin condicionarlos. Puede entenderse como la relación recíproca a la admisión.

Figura 21. Tipos de relaciones de grano grueso que intervienen el proceso de refinación

Llevada esta aproximación a nuestra especificación XML, las relaciones de grano grueso pueden ser enriquecidas con el tipo de relación propuesto por [Ospina07]. Para esto introduciremos al tag <*Relation*> un atributo llamado *type* que indique el tipo de relación.

Incorporar esta información en la especificación de concerns permite a los analistas y arquitectos de software tener una sensación más real de lo que significa cada concern y como influye en el resto de los concerns.
Notar que el atributo *type* en el tag <*Concern*> difiere del atributo *type* del tag <*Relation*>. El primero representa el tipo de concern (Negocio, Objetivo, Calidad, etc.) mientras que el segundo representa el tipo de relación (Uso, Afección, etc.)
Basándonos en el ejemplo del caso de estudio, la especificación de requerimientos enriquecida con los tipos de relación sería la que se muestra en la Figura 22:

```
<Concern name="Cliente" type="Negocio" >
  <Relationships>
    <Relation name="Administrador de Taxis" type="Uso" />
  </Relationships>
</Concern>

<Concern name="Administrador de Taxis" type="Negocio" >
  <Relationships>
    <Relation name="Cliente" type="Contribución" />
    <Relation name="Taxi" type=" Contribución" />
  </Relationships>
</Concern>

<Concern name="Autenticación" type="Calidad" >
  <Relationships>
    <Relation name="Cliente" type="Condición" />
    <Relation name="Taxi" type="Condición" />
    <Relation name="Administrador de Taxis" type="Condición" />
  </Relationships>
</Concern>
```

Figura 22. Especificación de Concerns y sus relaciones de grano grueso enriquecidas

2.2.6 Manejando conflictos entre concerns

La actividad de manejo de conflictos entre concerns, permite especificar en la matriz si un concern determinado puede contribuir positiva (+) o negativamente (-) a otros concerns. El modelo AORE Multidimensional detalla una heurística de doblamiento de la matriz de contribuciones sobre su diagonal, con el propósito de observar si existe un efecto acumulativo para las situaciones donde dos concerns se influencian entre sí.
Cunado nos referimos a un efecto acumulativo nos referimos a una contribución positiva sobre una negativa o viceversa, a está situación se la conoce como *conflicto*.
En caso de encontrar un conflicto entre los concerns, se debe negociar entre los distintos stakeholders responsables de la implementación de esos concerns, cual de ellos es el que prevalece o bien realizar una modificación en los requerimientos de los concerns para eliminar el conflicto. A cada contribución se la suele ponderar para facilitar la toma de decisiones en la resolución del conflicto.

Por simplicidad, nuestro caso de estudio no presenta conflictos entre los concerns. Mayor información acerca del manejo de conflictos se puede obtener en [Moreira1_05].

2.2.7 Identificando las dimensiones de los concerns

Ahora que hemos refinado los concerns y sus relaciones, estamos en condiciones de evaluar como estos concerns se reflejan en las diferentes elecciones de la arquitectura lógica.
La especificación de las dimensiones de un concern permite determinar la influencia del concern en las etapas posteriores de desarrollo. De acuerdo al modelo AORE Multi-Dimensional un concern puede ser representado como una característica/elemento o función del sistema (ej.: un método, un objeto o un componente), una decisión de diseño (ej.: una elección arquitectural), o un aspecto, sin embargo, el modelo no especifica cuándo ni cómo se debe realizar este mapeo. Por lo tanto, en el próximo capítulo propondremos algunas reglas basadas en nuestra experiencia previa en el desarrollo de GIS para poder determinar de una forma sistemática el mapeo de los concerns del sistema. En otras palabras, el foco principal del próximo capítulo será representar como entidades de la arquitectura lógica del GIS los concerns identificados y separados mediante el modelo AORE Multi-Dimensional.

Capítulo 3. Mapeo de Concerns a Componentes y Aspectos

En el Capítulo 2 se ha analizado un modelo en el cual se utiliza la noción de Identificación y Separación de Concerns en la fase de Análisis de Requerimientos (AORE Multi-Dimensional), que permite generar una especificación estructurada de los concerns y sus relaciones. La motivación para separar concerns en esta fase se basa en la idea de considerar a los concerns como temas de interés transversales a los distintos stakholders[12], logrando un mejor entendimiento del problema y facilitando la toma de decisiones en la etapa posterior de Diseño de la Arquitectura de Software.

Entendemos por *Arquitectura de Software* a la representación de alto nivel de la estructura de un sistema o aplicación, que describe las partes que la integran, las interacciones entre ellas, los patrones que supervisan su composición, y las restricciones a la hora de aplicar esos patrones [Vallecillo99]. En general, esta representación se realiza en términos de una colección de *componentes* y de las interacciones que tienen lugar entre ellos. De esta colección de componentes surge el concepto de Desarrollo de Software Basado en Componentes [Szyperski02] (CBSD, siglas del inglés *Component Based Software Development*), que presenta muchas ventajas desde el punto de vista de la reutilización del software. Sin embargo, desde el punto de vista de la flexibilidad y acoplamiento de los componentes, este paradigma termina siendo insuficiente para abarcar la complejidad de los sistemas distribuidos actuales [Pinto05], teniendo que introducir nuevos paradigmas, como el Desarrollo de Software Orientado a Aspectos (AOSD, del inglés *Aspect Oriented Software Development*) [AOSDnet].

Tanto CBSD y AOSD han emergido con el propósito de mejorar la modularidad y evolución de los sistemas mediante la "inyección" de entidades independientes y reutilizables. Por un lado, CBSD se basa en alcanzar una descomposición funcional precisa del sistema en componentes verdaderamente independientes, algunas veces producidos por terceras partes, listos para ser (re)utilizados en diferentes contextos. El objetivo es la reducción de tiempo de desarrollo, costos y esfuerzo, mientras se mejora la flexibilidad, fiabilidad y mantenibilidad de la aplicación final.

AOSD, por otro lado, es una disciplina prometedora que soporta la separación de concerns transversales introduciendo una nueva dimensión llamada: *aspecto*. El objetivo principal de AOSD es proveer una solución a un problema conocido como *tangling code* (en español: código entremezclado) [Kiczales97], que se refiere a la dificultad que tienen las propiedades transversales al querer evolucionar independientemente de las entidades a las que estas afectan. En este sentido los aspectos de AOSD usualmente encapsulan propiedades transverasales que son definidas generalmente como partes de varios componentes del sistema.

[12] *Stakeholders*: en la Ingeniería de Software se utiliza para referirse a quienes pueden afectar o son afectados por las actividades del proceso de desarrollo del software.

Es necesario destacar que CBSD y AOSD son dos principios complementarios [Pinto05]. En este sentido, AOSD puede ayudar a mejorar la independencia, la reusabilidad, la evolución y mantenibilidad de los componentes extrayendo los concerns transversales de los mismos y encapsulándolos en aspectos. Estos concerns transversales podrían ser gestionados por separado sin afectar la funcionalidad central del resto de los componentes del sistema.

Una aproximación que consideramos adecuada, y que integra tanto los principios de CBSD y AOSD es el modelo CAM (de las siglas en inglés *Component Aspect Model*) definido en [Pinto05]. Este modelo será el foco central de este capítulo, y nos permitirá en la fase de Diseño de la Arquitectura, representar como componentes y aspectos, aquellos concerns identificados con el modelo AORE Multi-Dimensional utilizado en la fase de Análisis de Requerimientos. Pero, para introducir CAM, creemos necesario describir primero brevemente los conceptos principales de los principios de CBSD y AOSD: los componentes y los aspectos respectivamente.

3.1 Los Componentes: conceptos generales

Existen diversos trabajos que describen el mundo de los componentes (como [Heineman01]), pero en particular nos basaremos en el trabajo de [Chambi07], el cual describe los conceptos esenciales del desarrollo de software basado en componentes (CBSD). Este trabajo comienza describiendo el mundo de los componentes desde sus inicios. Los componentes de software surgen en cierta medida de la necesidad de hacer un uso correcto de software de tal manera que sea reutilizable para la construcción de aplicaciones mediante el ensamblaje de partes ya existentes. Desde el punto de vista de la ingeniería de software, el término "*componente*" procede de las "técnicas orientadas a objetos", de los problemas de descomposición usados en "técnicas de descomposición de problemas", y de su necesidad para desarrollar sistemas abiertos[13].

Actualmente existe una gran discrepancia entre la mayoría de los ingenieros de software sobre el concepto de c*omponente de software*. La polémica suscitada entre Clemens Szyperski y Bertrand Meyer en la columna "Beyond Objects" de la revista *Software Development* [SofDev08] sobre qué es y que no es un componente, sus propiedades y su naturaleza, podemos considerarla como uno de los ejemplos más recientes y representativos sobre la confusión que suscita el propio concepto de componente.
A pesar de estas discusiones, tomaremos la definición de Szyperski: un componente es una unidad binaria de composición, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio. Las interfaces de un componente determinan tanto las operaciones que el componente implementa como las que precisa utilizar de otros componentes durante su ejecución. Los requisitos determinan las necesidades del componente en cuanto a recursos, como por

[13] *Sistemas abiertos*: Los sistemas abiertos son aquellos sistemas informáticos que proporcionan alguna combinación de interoperabilidad, portabilidad y uso de estándares abiertos [Dickinson91].

ejemplo las plataformas de ejecución que necesita para funcionar.

Un componente a su vez puede contener múltiples objetos, clases y otros componentes, es decir, la noción de componente puede variar dependiendo del nivel de detalle desde donde se lo mire, a esto se lo conoce como *Granularidad* del componente. Un componente de software puede ser desde una subrutina de una librería, hasta una clase o incluso una aplicación que pueda ser usada por otra aplicación por medio de una interfaz especificada. Un componente con granularidad gruesa se refiere a que puede estar compuesto por un conjunto de componentes, o ser una aplicación para construir otras aplicaciones o sistemas a gran escala, generalmente abiertos y distribuidos. A medida que descendemos en el nivel de detalle, se dice que un componente es de grano fino.

A lo largo de la historia, la construcción de sistemas de información basados en componentes ha involucrado la participación de componentes ad-hoc y de terceros, con lo cual, ha surgido la necesidad de unificar ciertos criterios, estandarizando la noción de componente, la forma de ensamblar los mismos, la definición de sus interfaces y la definición de un entorno de ejecución común, proporcionando de esta forma una visón clara de la arquitectura del sistema. De esta necesidad surgen los Modelos de Componentes.

3.1.1 Modelos de Componentes

Un *modelo de componentes* define la forma de sus interfaces y los mecanismos para interconectarlos entre ellos.
Actualmente existen tres grandes modelos de componentes de software:

- *Enterprise JavaBeans* (EJB), propuesto por Sun [Sun_EJB]
- *Corba Component Model* (CCM) [OMG_Corba], propuesto por el Object Management Group.
- *Component Object Model* (COM+), propuesto por Microsoft [Microsoft_COM]

El propósito de este capítulo no es entrar en detalles acerca de cada uno de estos modelos de componentes, ya que existen varios trabajos que tratan este tema (como [Kung05] o [Pickin05]) sin embargo nos parece adecuado abstraer las generalidades de los mismos para comprender los elementos básicos de un modelo de componentes. Las siguientes definiciones basadas en el trabajo de [Zheng05] son generales, y pueden ser aplicables en forma universal a cualquier modelo de componentes:

Un modelo de componentes de software debe definir:

- La *Sintaxis de los componentes*: son las reglas para la construcción y representación de los componentes que permiten definirlos físicamente. Idealmente la sintaxis debe ser parte de un lenguaje que sea utilizado para definir y construir componentes. En los modelos de componentes actuales, el lenguaje para definir componentes tiende a ser un lenguaje de programación, por ejemplo tanto en JavaBeans como en EJB, los componentes son definidos como clases de Java.

- La *Semántica de los componentes*: describe la esencia del componente y determina la función para la cual fue creado. Para esto existen ciertos elementos del componente como el nombre, la interfaz y el código, que permiten descubrir su esencia.
  El nombre del componente debe ser autodescriptivo, es decir, debe dar una idea de cual es la funcionalidad que ofrece, por ejemplo, para un componente cuya función es realizar cálculos espaciales, un nombre adecuado sería *CalculadorEspacial*, en lugar de *ComponenteX*. El código, por otro lado, implementa los servicios provistos por el componente, y no puede ser accedido o visto fuera del mismo.
  La interfaz es el único punto de acceso del componente, por ende debe proveer toda la información necesaria para utilizar el componente. En particular debe especificar los servicios requeridos por el componente para poder producir los servicios que este brinda. Estos servicios requeridos son generalmente valores de entrada para los parámetros de los servicios provistos. La interfaz de un componente entonces especifica las *dependencias* entre los servicios provistos y requeridos por el mismo.

  En los modelos de componentes actuales, los componentes tienden a ser objetos del paradigma de Programación Orientado a Objetos. Los métodos de estos objetos son los servicios provistos, y los métodos de otros objetos invocados por el componente son los servicios requeridos. Estos objetos son alojados en un entorno, que consiste en un conjunto de recursos que rodean a los componentes, y que definen las acciones que sobre ellos se solicitan. Se pueden definir al menos dos clases de entornos para los componentes: el entorno de *ejecución* y el de *diseño*. El primero de ellos es el ambiente para el que se ha construido el componente, y en donde se ejecuta. Usualmente se le da el nombre de Contenedor (*Container*). La Figura 23 muestra una imagen de un componente y su contenedor:

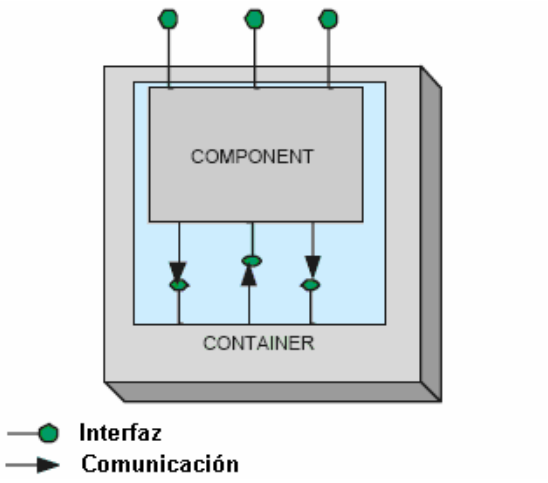


Figura 23. Un componente, sus interfaces y su contenedor

El Conteiner (bloque de color celeste) puede ser pensado como un wrapper que está encargado de tratar ciertos aspectos técnicos, como por ejemplo la creación y comunicación entre los componentes, que surgen de la tecnología utilizada. Los componentes pueden comunicarse con el Container ya que éste también posee

interfaces de comunicación. Los aspectos que pueden ser manejados por el Conteiner son conocidos como *Servicios de Infraestructura*.

Por otro lado, el entorno de diseño es un ambiente restringido, que se utiliza para localizar, configurar, especializar y probar los componentes que van a formar parte de una aplicación.

La *Composición de componentes*: la composición de componentes puede tomar lugar en diferentes etapas del ciclo de vida del software. Se identifican dos etapas principales en donde se puede realizar la composición, la etapa de *Diseño* y la etapa de *Implantación*.
En la etapa de Diseño, los componentes son diseñados y construidos, y luego son depositados generalmente en un repositorio (si existe). Los componentes en esta etapa son *stateless*, es decir, no poseen estado, ya que son sólo *templates* (como las clases) que no pueden ejecutarse. Los únicos datos que pueden contener en esta fase son constantes. Sin embargo (a diferencia de las clases), pueden formar parte de una composición de componentes. Si existe un repositorio de componentes, entonces los componentes construidos deben ser almacenados y catalogados de forma que se puedan obtener cada vez que se necesiten (más información sobre la catalogación de componentes GIS se encuentra en [Myung-Hee02].

En la etapa de Implantación, las instancias de los componentes son creadas instanciando componentes con datos iniciales, de modo que tienen estado y están listos para ser ejecutados.

3.2 Desarrollo de Software Basado en Componentes

Uno de los enfoques en los que actualmente se trabaja para el desarrollo de software constituye lo que se conoce como *Desarrollo de Software Basado en Componentes* (*Component Based Software Development*, CBSD) [Brown99], que trata de sentar las bases para el diseño y desarrollo de aplicaciones distribuidas basadas en componentes de software reutilizables. Dicha disciplina cuenta actualmente con un creciente interés, tanto desde el punto de vista académico como desde el industrial, en donde la demanda de estos temas es cada día mayor.
Esta disciplina consta de cuatro etapas:

1. La selección de componentes.
2. La adaptación de componentes.
3. El ensamblaje de los componentes al sistema.
4. La evolución del sistema.

3.2.1 La selección de componentes

La "selección de componentes" es un proceso que determina qué componentes ya desarrollados pueden ser utilizados. Existen dos fases en la selección de componentes:

- Fase de búsqueda

• Fase de evaluación.

Fase de búsqueda, se identifican las propiedades de un componente, como por ejemplo, la funcionalidad del componente (qué servicios proporciona) y otros aspectos relativos a la interfaz de un componente (como el uso de estándares), aspectos de calidad que son difíciles de aislar y aspectos no técnicos, como la cuota de mercado de un vendedor o el grado de madurez del componente dentro de la organización. La fase de búsqueda es un proceso tedioso, donde hay mucha información difícil de cuantificar, y en algunos casos, difícil de obtener.

Fase de evaluación, existen técnicas relativamente maduras para efectuar el proceso de selección. Por ejemplo ISO (International Standards Organization) describe criterios generales para la evaluación de productos [ISO/IEC-9126]. En [Poston92] se definen técnicas que tienen en cuenta las necesidades de los dominios de aplicación. Estas evaluaciones se basan en el estudio de los componentes a partir de informes, discusión con otros usuarios que han utilizado estos componentes, y el prototipado.

3.2.2 La adaptación de componentes

Debido a que los componentes son creados para recoger diferentes necesidades basadas en el contexto donde se crearon, estos tienen que ser adaptados cuando se usan en un nuevo sistema. En función del grado de accesibilidad a la estructura interna de un componente, podemos encontrar diferentes aproximaciones de adaptación [Valetto95]:

- De caja blanca, donde se permite el acceso al código fuente de un componente para que sea reescrito y pueda operar con otros componentes.
- De caja gris, donde el código fuente del componente no se puede modificar, pero proporciona su propio lenguaje de extensión o API.
- De caja negra, donde el componente sólo está disponible en modo ejecutable (binario) y no se proporciona ninguna extensión de lenguaje o API desde donde se pueda extender la funcionalidad.

3.2.3 El ensamblaje de los componentes al sistema

Para "ensamblar" los componentes en el sistema existe una infraestructura de estilos. Los más conocidos son el bus de mensajes MOM (Message-Oriented Middleware) y la tecnología ORB (Object Request Broker).

Message-Oriented Middleware (MOM): es una infraestructura cliente/servidor que mejora la interoperabilidad, portabilidad y flexibilidad de los componentes de una aplicación permitiendo que esta sea distribuida en múltiples plataformas heterogéneas, esta tecnología MOM es una tecnología asíncrona que reduce la complejidad de desarrollo al ocultar al desarrollador detalles del sistema operativo y de las interfaces de red. MOM está basada en

el uso de colas de mensajes que ofrecen almacenamiento temporal cuando el componente destino está ocupado o no está conectado. Más información sobre MOM puede ser encontrada en [Korhonen97].

Object Request Broker (ORB): es una tecnología de interconexión (conocida como middleware) que controla la comunicación y el intercambio de datos entre objetos. Los ORBs mejoran la interoperabilidad de los objetos distribuidos ya que permiten a los usuarios construir sistemas a partir de componentes de diferentes vendedores.
Los detalles de implementación del ORB generalmente no son de importancia para los desarrolladores. Estos sólo deben conocer los detalles de las interfaces de los objetos, ocultando de esta forma ciertos detalles de la comunicación entre componentes. Las operaciones que debe permitir por tanto un ORB son básicamente tres:

a. La definición de interfaces.
b. La localización y activación de servicios remotos.
c. La comunicación entre clientes y servicios.

Más información sobre ORB puede ser encontrada en [OMG93].

3.2.4 Beneficios e Inconvenientes de CBSD

El uso de esta metodología posee los siguientes beneficios [Piattini08]:

- *Reutilización del software:* nos lleva a alcanzar un mayor nivel de reutilización de software.
- *Simplifica las pruebas*: permite que las pruebas sean ejecutadas probando cada uno de los componentes antes de probar el conjunto completo de componentes ensamblados.
- *Simplifica el mantenimiento del sistema*: cuando existe un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del sistema.
- *Mayor calidad*: dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorará con el paso del tiempo. De la misma manera, el optar por comprar componentes de terceros en lugar de desarrollarlos, posee algunas ventajas:
  - *Ciclos de desarrollo más cortos*: la adición de una pieza dada de funcionalidad tomará días en lugar de meses ó años.
  - *Mejor ROI (Retorno de Inversión)*: usando correctamente esta estrategia, el retorno sobre la inversión puede ser más favorable que desarrollando los componentes uno mismo.
- *Funcionalidad mejorada*: para usar un componente que contenga una pieza de funcionalidad, solo se necesita entender su naturaleza, pero no es necesario conocer sus detalles internos.

Los inconvenientes del modelo:

- Si no existen los componentes, hay que desarrollarlos y se puede perder mucho

tiempo.

- Los componentes pueden entrar en conflicto si de estos sale una nueva versión y no esta estandarizado con lo que se ha desarrollado en la aplicación ensamblada.
- Los compromisos en los requerimientos son inevitables, por lo cual puede que el componente reutilizado no cumpla totalmente con las expectativas del cliente.
- Las actualizaciones de los componentes adquiridos no están en manos de los desarrolladores del sistema sino de sus fabricantes.
- Los componentes suelen estar fuertemente acoplados (ver Capítulo 1 apartado 1.6.3 y Capítulo 3 apartado 3.2.5).

3.2.5 Evolución del sistema

Los sistemas basados en componentes deberían ser fácilmente evolucionables y actualizables. Cuando un componente falla (por cualquier motivo) éste debe poder cambiarse por otro equivalente y con las mismas prestaciones. De igual forma, si un componente del sistema debe ser modificado, para que incluya nuevas funcionalidades o elimine algunas de ellas, esto se puede hacer sobre un nuevo componente que luego será reemplazado por el que hay que modificar. Sin embargo, este punto de vista es poco realista. La sustitución de un componente por otro suele ser una tarea tediosa y que consume mucho tiempo, ya que el nuevo componente nunca será idéntico al componente sustituido, y antes de ser incorporado en el sistema, éste debe ser perfectamente analizado de forma aislada y de forma conjunta con el resto de los componentes con los que debe ensamblar dentro del sistema.

Sustituir o modificar un componente se transforman en tareas complejas debido a que los componentes suelen estar fuertemente acoplados. Este alto grado de acoplamiento se debe a que existen concerns transversales a los componentes que deben ser implementados necesariamente como parte interna de los mismos. El código que permite implementar estos concerns suele estar entremezclado con el código que implementa la funcionalidad inherente de cada componente. Esto lleva a establecer una dependencia muy fuerte entre los componentes y los concerns, haciendo que los componentes no puedan evolucionar en forma independiente de los concens y viceversa.
Este es el principal problema que pretende atacar el paradigma de Desarrollo de Software Orientado a Aspectos, que se detalla a continuación.

3.3 Desarrollo de Software Orientado a Aspectos (AOSD)

El Desarrollo de Software Orientado a Aspectos (AOSD) es un paradigma emergente que provee abstracciones explicitas para los concerns que tienden a ser transversales a múltiples componentes de un sistema. Representar a los concerns transversales (también denominados *Aspectos* en este paradigma) como abstracciones de primera-clase, y a su vez proveer nuevas técnicas de composición para combinar aspectos y componentes, mejoran la modularidad del sistema y reduce la complejidad del mismo, sobre todo a la hora del mantenimiento y evolución de sistema [AOSDnet].

AOSD tiene sus inicios en la Fase de Desarrollo del ciclo de vida del software, y en la última década se han introducido varios lenguajes de programación orientados a aspectos (lenguajes AOP). Algunos de los lenguajes más conocidos son AspectJ, HyperJ, ComposeJ, and DemeterJ.
El objetivo de esta sección no es describir en detalle estos lenguajes AOP, sino introducir los conceptos principales del problema atacado por AOSD. Para entender esta problemática veamos dos conceptos principales desde el punto de vista de esta disciplina:

- La Descomposición Modular
- Los Concerns Transversales

3.3.1 La Descomposición Modular

El principio de Separación de Concerns para AOSD dice que cada concern debe ser desarrollado (o mapeado) preferentemente como un único módulo en el sistema. Dicho de otra manera, el problema se debe descomponer en módulos de modo que cada uno resuelva un concern del sistema. La ventaja principal de esta idea es que los concerns se encuentran localizados y por ende son más fáciles de comprender, extender, reutilizar y adaptar. Este proceso de descomposición es ilustrado en la Figura 24.

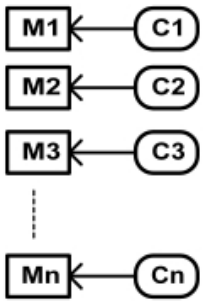


Figura 24. Mapeo de los concerns C1..Cn a los módulos M1..Mn respectivamente

El problema se descompone en *n* concerns ($C_1..C_n$), y cada uno de ellos es mapeado a un módulo individual ($M_1..M_n$). Asumimos que un módulo es una abstracción de una unidad modular en un lenguaje de diseño específico (puede ser una clase, función, proceso, etc.).

3.3.2 Concerns Transversales (*Crosscutting concerns*)

Existen concerns que pueden implementarse en módulos individuales, pero otros sin embargo, no pueden hacerlo ya que no es tan sencilla la separación, y se termina implementando un concern en varios módulos del sistema. A este fenómeno en donde un concern es implementado en varios módulos se lo denomina *crosscutting*. En otras palabras, un crosscutting concern es aquel que afecta a otros concerns (o también se dice que *atraviesa* a otros concerns). En la Figura 25, por ejemplo, el concern C_3 que debería ser implementado por el módulo C3, es también implementado y afecta por ende a los módulos M_2, M_4 y M_5. Decimos que C_3 es un crosscutting concern (concern transversal). Algunos ejemplos de concerns transversales son: login, sincronización, balance de carga, monitoreo

de operaciones, etc.

Los concerns transversales son un serio problema ya que son difíciles de comprender, reusar, extender, adaptar y mantener ya que están dispersos a través de diferentes módulos.
Una modificación sobre un concern transversal que está disperso en varios módulos implica modificar los módulos afectados.
El primer problema que debemos afrontar entonces es encontrar todos aquellos puntos en los módulos que son "atravesados" por un concern transversal. Adaptar estos concers adecuadamente es otro problema futuro.

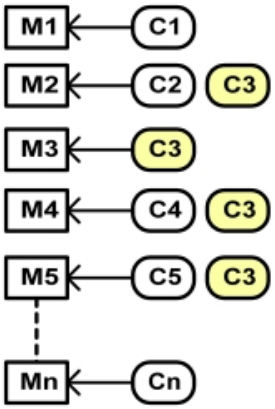


Figura 25. El Concern C3 atraviesa los módulos M2, M4 y M5

Podemos estar frente a una peor situación, si en lugar de existir un concern transversal, existiesen múltiples concerns transversales en simultaneo, como se observa en la Figura 26, en donde los concerns transversales C_3 y C_5 deben convivir en los módulos M_2, M_3 y M_4:

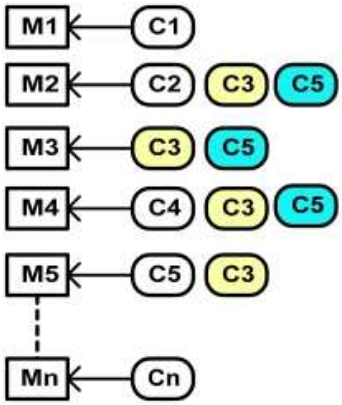


Figura 26. Múltiples concerns transversales (C3 y C5)

Esta situación en donde conviven en un mismo módulo varios concerns lleva a generar código entremezclado.

3.3.3 Concerns Entremezclados (*Tangled concerns*)

Debido a que no es una tarea fácil localizar y separar los concerns transversales, algunos módulos del sistema pueden incluir más de un concern. Diremos que estos concerns están *entremezclados* en el módulo. Por ejemplo, en la Figura 26, los concerns C_2, C_3 y C_5 están entremezclados en el modulo M_2. Notar que el concern C_2 no es transversal.
Llevémoslo a un ejemplo, donde M_2 es un módulo encargado de la asignación de los taxis, C_2 es el concern que engloba las operaciones del *Administrador de Taxis*, C_3 el concern de *Trazabilidad* y C_5 el concern de *Persistencia*. Veamos en la Figura 27 un método de M_2 escrito en C#, en donde un cliente (validado previamente) solicita un taxi para la ubicación en la que se encuentra. En éste método se puede observar código entremezclado del concern C_3 y C_5: las líneas 3, 4, 11 y 15 corresponden a C_3, y la línea 10 corresponde a C_5.

```
1. public void SolicitarTaxi(Cliente cliente)
2. {
3.     Logger.Trace("El cliente " + cliente.Nombre + "Solictó un taxi a las " +
4.     DateTime.Now();
5.     Taxi taxi = OperadorEspacial.BuscarEntidadMasCercana(typeof(Taxi),cliente);
6.     bool acepta = Taxi.AceptaViaje(cliente);
7.     if (acepta)
8.     {
9.           Taxi.TomarViaje(cliente);
10.          DaoSolicitudes.Guardar(cliente, taxi, DateTime.Now());
11.          Logger.Trace("Taxi " taxi.Codigo + "toma un viaje");
12.    }
13.    else
14.    {
15.          Logger.Trace("Taxi " taxi.Codigo + "rechaza un viaje");
16.          …
17.          …
18.    }
19. }
```

Figura 27. Ejemplo de código entremezclado

En un esquema como este, ninguno de los concerns puede evolucionar en forma independiente sin afectar a M_2. Supongamos que existe una nueva implementación del concern de Trazabilidad, en donde el trazador (*Logger*) ahora en lugar de recibir como parámetro una cadena de caracteres además recibe otro parámetro indicando la fuente en donde se debe imprimir el mensaje (un archivo, consola, base de datos, etc.). Si queremos utilizar la nueva versión del trazador, deberíamos interferir en el código de M_2 y modificar las líneas 3, 4, 11 y 15. Si replicamos esta situación en todos los módulos que son atravesados por el concern de *Trazabilidad* estamos frente a un problema de refactorización difícil de resolver.
En el ámbito de los aspectos, estas líneas podrían representar lo que se conoce como *puntos de intercepción* (*jointpoints* en inglés) si es que los concerns son modelados como aspectos.
En la Figura 28 (que representa la misma información que la Figura 26, pero los módulos han sido alineados sobre el eje vertical y los concerns sobre el eje horizontal) se pueden observar los círculos que representan los jointpoints, es decir, los lugares donde un módulo es atravesado por un concern. Los mismos pueden estar a nivel de un modulo (por ej. una clase) o estar más refinados y tratar con sub-partes del módulo (método, atributo,

operación).

En la Figura 28 se puede identificar fácilmente la naturaleza transversal de un concern, solo basta con tomar un concern y analizarlo en sentido vertical buscando múltiples jointpoints.

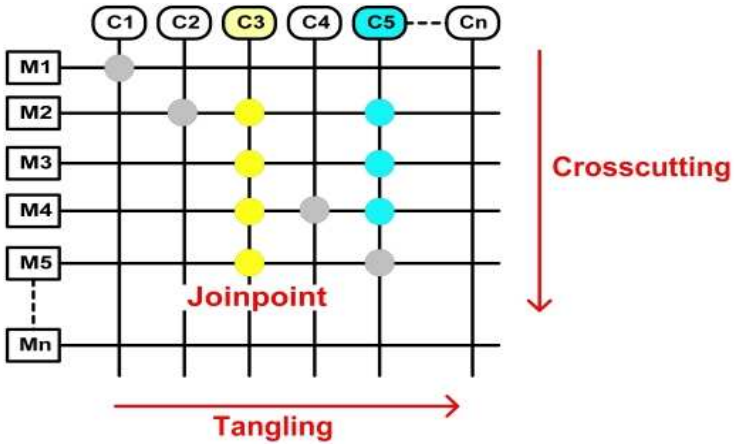


Figura 28. Concerns transversales, entremezclados y jointpoints

El entremezclado puede ser detectado en la Figura 28 si tomamos cada módulo y lo recorremos en sentido horizontal y detectamos múltiples jointpoints.

3.3.4 Descomposición de Aspectos

Es de esperar que dado un diseño de un problema, este pueda presentar concerns transversales y por consiguiente fallen los mecanismos de abstracción convencionales para tratar y modelar estos concerns.
AOSD provee abstracciones explícitas para representar a ciertos concerns transversales, también llamados *Aspectos*. De esta manera, el diseño del problema dado puede descomponerse en concerns que pueden ser localizados en módulos separados y concerns que tienden a atravesar a un conjunto de módulos.

Una definición formal de *Aspecto* muy aceptada en el ámbito de la Ingeniería de Software es: "Un aspecto es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la etapa de implementación. Un aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un aspecto de código es una unidad modular del programa que aparece en otras unidades modulares del programa" [Kiczales97]:

En la Figura 29 se puede observar un conjunto de concerns y los módulos que los implementan. Se puede observar que el concerns C3 es transversal a los módulos M2, M4 y M5 y que el concern C5 es transversal a M2, M3 y M4. Estos concerns transversales son candidatos a modelarse como Aspectos, aunque para esto es necesario evaluar ciertas características inherentes a los concerns para determinar si serán Aspectos o Componentes (más información de este tema se encuentra en el apartado 3.6 de este capítulo).

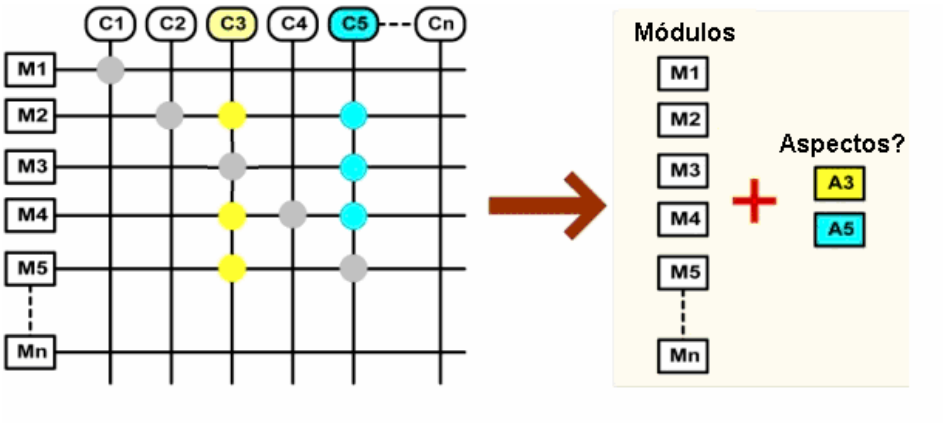


Figura 29. Separando aspectos

Para aquellos concerns transversales que serán modelados como Aspectos, es necesario especificar los puntos en los módulos que son atravesados por los mismos, para ello se utiliza lo que se conoce como *especificación de pointcuts*.
Una especificación de pointcuts es en esencia un predicado sobre el conjunto completo de jointpoints que el aspecto puede atravesar. Una especificación de pointcuts puede enumerar los jointpoints o proveer una especificación más abstracta. En un sentido abstracto, los aspectos pueden especificarse de la siguiente manera:

Aspect nombreAspecto
Especificación de pointcuts

La naturaleza transversal de un concern es en realidad localizada en la especificación de pointcuts, la cual indica cuales son los puntos que son atravesados por el aspecto, aunque no se especifique que tipo de comportamiento es requerido. Para esto último, se ha introducido el concepto de *advice*.
Un *advice* es el comportamiento que puede ser adjuntado antes, alrededor o luego de un jointpoint de la especificación de pointcuts.

Aspect nombreAspecto
Especificación de pointcuts
Advice

A su vez, existe el concepto de Weaving, que hace referencia a la tarea de aplicar un advice sobre uno o más pointcuts de modo que el advice luego pueda ser activado/ejecutado cuando el control de la aplicación encuentre los jointpoints.

Al proveer un lenguaje explícito de construcción de aspectos, la noción misma de aspecto se convierte en una abstracción de primera-clase, como lo es la clase en el paradigma orientado a objetos.
Existen varios leguajes de programación orientados a aspectos, algunos de ellos los hemos enunciado previamente (AspectJ, HyperJ, etc), y la mayoría difiere en la forma de

especificar aspectos, pointcuts y advices, sin embargo todos adoptan como base estos conceptos. Más información sobre estos lenguajes AOP se encuentra en el Capítulo 4.

3.3.5 Beneficios e Inconvenientes de AOSD

En este apartado, mencionamos beneficios e inconvenientes de AOSD.
Los principales beneficios de esta aproximación son [Kiczales97]:

- Un nivel de abstracción más alto, porque el desarrollador se puede centrar en concerns concretos y de forma aislada.
- Mayor facilidad a la hora de entender la funcionalidad de la aplicación, ya que el código de los diferentes concerns no está entremezclado.
- Mayor reusabilidad al haber un menor acoplamiento.
- Mayor mantenibilidad del código, al ser éste menos complejo, y ser independiente del resto de los componentes.
- Una mayor flexibilidad en la integración de componentes.
- Un incremento de la productividad en el desarrollo.

Los inconvenientes de esta aproximación son:

- Si se quiere aplicar el principio de AOSD sobre concerns de aplicaciones existentes Orientadas a Objetos, es posible que se requiera una refactorización de las aplicaciones.
- Como el principio de AOSD es relativamente nuevo, no se encuentran muchos casos de su uso en la industria, esto hace difícil evaluar los riesgos basándose en experiencias previas.

A su vez, existen otros inconvenientes puntuales relacionados con el diseño de aspectos en el marco de CBSD, los cuales analizaremos a continuación con más detalle con el objetivo de encontrar una solución que permita integrar ambos principios.

3.4 Component Based Software Development + Aspect Oriented Software Development

CBSD y AOSD, a pesar de que persiguen distintos objetivos, son tecnologías complementarias. AOSD puede ayudar a mejorar la independencia, reusabilidad, evolución y mantenibilidad de los componentes, obteniendo de los mismos los concerns transversales y convirtiéndolos en aspectos.
Estos concerns transversales pueden ser tratados separadamente sin afectar la funcionalidad central de los componentes. De esta manera, sería conveniente disponer de modelos que combinen CBSD y AOSD para aprovechar las ventajas de ambas disciplinas.

Recientemente, los modelos de componentes como EJB y CORBA, han evolucionado para utilizar servicios comunes fuera de la implementación de los componentes (nos referimos a

servicios comunes de la plataforma que resuelven concerns transversales). Sin embargo, aún no proveen una solución apropiada para tratar el problema del código entremezclado, ya que solo separan una cantidad limitada de servicios comunes (por ejemplo: seguridad, transacciones, persistencia).
Para incorporar el concepto de aspecto, estas plataformas deben ofrecer mecanismos adicionales para separar cualquier tipo de concern transversal.

En cuanto a las distintas aproximaciones de AOSD, muchas de ellas [AspectWerkz] [Popovici02] no tratan a los aspectos como entidades reusables e independientes del contexto ya que hardcodean el punto de enlace del aspecto con el componente. Esto impide al aspecto ser reutilizado en diferentes contextos. A su vez, lo puntos de intercepción que pueden ser interceptados por las actuales aproximaciones AOSD [Kiczales01] no son siempre adecuadas en el contexto de CBSD ya que interceptan parte del comportamiento interno de los componentes, mientras que los componentes deberían ser vistos como cajas negras y utilizarlos a través de sus interfaces.

Una solución adecuada a estos problemas basada en componentes y aspectos, que integra los beneficios de ambas disciplinas (CBSD y AOSD), es el modelo CAM (*Component an Aspects Model*) [Pinto05], el cual define un modelo de componentes y aspectos como entidades de primer orden, y a su vez un mecanismo de composición que no afecta al encapsulamiento de los componentes.
En este modelo, el código de los aspectos no incluye información acerca del lugar del componente en donde el aspecto va a actuar (pointcuts). Por el contrario, el modelo permite definir esta información en una especificación en forma separada de la definición de los componentes y los aspectos, pudiendo estos últimos ser reutilizados en diferentes contextos con distintos componentes.

3.5 El modelo CAM

Este modelo permite definir la arquitectura del sistema en función de componentes, aspectos y un mecanismo de comunicación entre ellos.
Como se puede observar en la Figura 30, se describe un diagrama basado en UML [OMG] con las entidades y relaciones principales del modelo CAM. Este modelo representa en realidad un metamodelo, que nos permite definir mediante nuevos Estereotipos UML [14]modelos para diferentes aplicaciones que utilicen la idea de componentes y aspectos.
Las entidades principales del modelo son los componentes y los aspectos (clase *Component* y clase *Aspect* respectivamente).

En la Figura 30 se ve como los componentes por un lado, interactúan entre sí enviando y recibiendo mensajes (asociaciones *sends* y *receives* con la clase *Message*) y lanzando eventos (asociación *throws* con la clase *Event*). En el marco del modelo CAM se entiende al evento, como un mensaje en el que no se proporciona información sobre el componente destino.

[14] *Estereotipos UML:* se refieren a una clasificación de alto nivel de un objeto que proporciona una idea del tipo de objeto del que se trata. Algunos estereotipos están predefinidos en el UML, otros pueden ser definidos por el usuario [OMG].

El comportamiento de los componentes se describe a través de sus interfaces. Estas interfaces describen los servicios ofrecidos y requeridos por el componente (en la Figura 30 *ProvidedInterface* y *RequieredInterface* respectivamente). También existe la interfaz de eventos (*EventInterface*) mediante la cual se describen aquellos eventos disparados por el componente.

Los aspectos por otro lado, encapsulan concerns transversales del sistema, y pueden aplicarse a distintos puntos de intercepción, como por ejemplo al envío y recepción de mensajes entre componentes, generación de eventos, y también a la creación y finalización de los componentes (asociaciones *creates* y *finalizes*).

Estos puntos de intercepción, donde los Aspectos se aplican a los Componentes, son no-invasivos ya que no interceptan puntos internos de la implementación del componente. Esta idea, si bien implica una menor expresividad con respecto al potencial ofrecido por los leguajes AOP, respeta el principio básico de encapuslamiento que sugiere CBSD, donde los componentes deben ser tratados como cajas negras teniendo solo acceso al comportamiento que el componente ofrece a través de sus interfaces.

Los Aspectos, viéndolos desde el punto de vista del encapsulameiento, proveen una interfaz llamada *EvaluatedInterface,* que describe aquellos mensajes y eventos que son posibles interceptar. Al describir a los Aspectos mediante sus interfaces se logra la 'componentización de aspectos' tratándolos como entidades COTS[15] listos para ser (re)utilizados como entidades independientes.

Un aspecto especial en el modelo CAM es el aspecto de coordinación (clase *CoordinationAspect* de la Figura 30) que encapsula un protocolo de interacción (clase *InteractionProtocol*) el cual permite coordinar la interacción entre los distintos componentes y la aplicación de los aspectos en los diferentes puntos de intercepción.
Estos puntos de intercepción, son los puntos en donde los aspectos se relacionan con los componentes; en CAM se describen estos puntos de intercepción mediante la relación de dependencia llamada *applies to*. Cabe destacar, que estas relaciones pueden aplicarse a diferentes puntos de intercepción, por ejemplo, cuando un componente envía un mensaje a otro es posible aplicar los aspectos antes y/o después del envío del mensaje (puntos de intercepción *BEFORE_SEND* y *AFTER_SEND*) y antes y/o después de la recepción del mensaje en el destino (*BEFORE_RECEIVE* y *AFTER_RECEIVE*). Todos los puntos de intercepción se pueden observar en la Figura 30 en el tipo Enumerativo llamado *AspectJointPoint*. Por simplicidad, no se describirán todos los puntos de intercepción de las relaciones *applies to*, para más información se sugiere consultar [Pinto05].

[15] *COTS*: del inglés *commercial-off-the-shelf*, se refiere a aquellas cosas que uno puede comprar manufacturadas por un tercero. La idea de los componentes COTS es ofrecer, a través de un catálogo y a un precio razonable, piezas de software listas para usar *[Oberndorf97].*

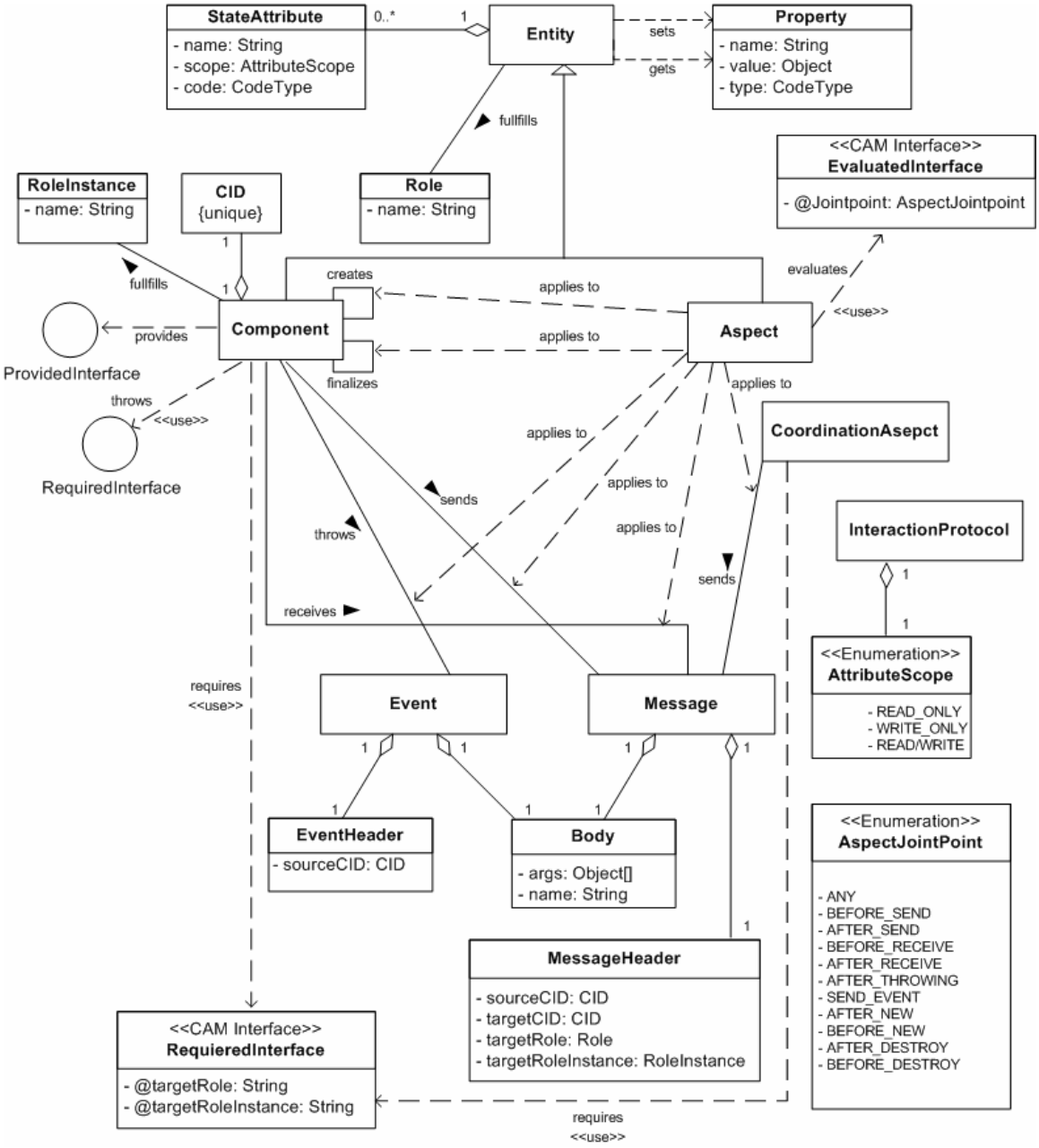


Figura 30. Modelo de Componentes y Aspectos CAM

En el modelo CAM los aspectos son tratados como un tipo "especial" de componente, y es por esto que ambas entidades comparten un conjunto de características en común. Por ejemplo, tanto los componentes como los aspectos pueden tener un conjunto de atributos de estado (clase *StateAttribute*) que representan su estado público.

Para desacoplar las interfaces de los componentes y de los aspectos de sus implementaciones finales, el modelo CAM asigna un nombre de rol único (clase *Role*) para referenciar a los componentes y a los aspectos dentro de una aplicación. Estos nombres de rol son nombres arquitectónicos, usados durante la composición e interacción de los componentes y los aspectos, reduciendo el acoplamiento entre ellos. Es decir, los

componentes y los aspectos no necesitan codificar la referencia de los componentes con los que se comunican; solamente incluyen el nombre de rol del componente destino de un mensaje. En caso de existir en una aplicación más de una instancia de un componente desempeñando el mismo rol, el nombre de instancia de rol (clase *RoleInstance*) permite diferenciarlos. Además, los componentes también pueden ser direccionados por un identificador de componente único (clase *CID*) que hace referencia a una instancia concreta de un componente, en caso de que sea necesario identificar instancias concretas.

Dado que los componentes y los aspectos normalmente tienen dependencias del contexto en el que se ejecutan; es decir, algunas veces necesitan compartir datos, la clase *Property* encapsula este tipo de dependencia de datos. Las propiedades CAM se identifican por un nombre único, su tipo y su valor actual.

3.6 Construyendo un modelo CAM basado en AORE Multidimensional

Uno de los desafíos más importantes del proceso propuesto en este trabajo es llevar los Concerns identificados mediante el modelo AORE Multidimensional a un diseño de arquitectura basado en Componentes y Aspectos del modelo CAM.

AORE por un lado, nos permite identificar cuales son los concerns transversales del GIS basándose en los requerimientos elicitados (ver Capítulo 2), pero no describe o sugiere una forma de modelar estos concerns como componentes de software (ya sean componentes, clases, aspectos, etc).
CAM por otro lado, parte de una lista de componentes y aspectos previamente identificados, es decir, asume que las tareas de separación de concerns en aspectos y componentes fueron realizadas en etapas anteriores.

El problema que surge ahora es que ninguno de los dos modelos ofrece un mecanismo detallado de reglas para poder llevar los concerns identificados a componentes de la arquitectura del GIS, ya sean aspectos o componentes, dejando al arquitecto de software la libertad para decidir que concern es un componente y que concern es un aspecto. Como se observa en la Figura 31, la Especificación de Concerns que se obtiene como output del proceso de Separación de Concerns no puede ser tomada directamente por el siguiente proceso de Mapeo de Concerns a Componentes y Aspectos, ya que no existen reglas para traducir estos concerns en componentes del la arquitectura del GIS.

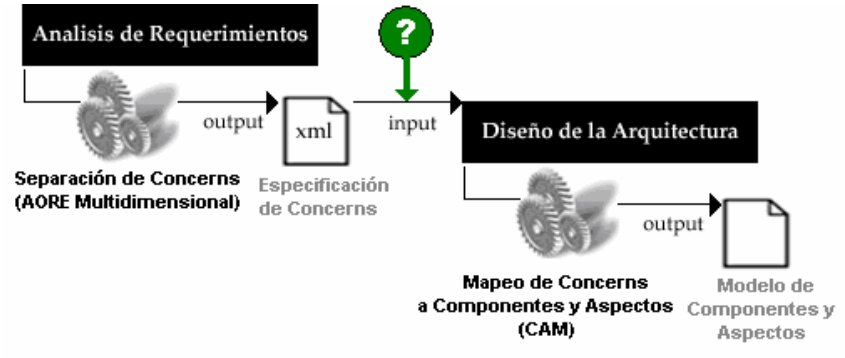


Figura 31. Integrando AORE-Multidimensional con el CAM

En la actualidad no existen trabajos que intenten integrar el modelo AORE Multidimensional con CAM, con lo cual nos pareció interesante describir en este trabajo algún mecanismo para integrar ambos modelos y poder aprovechar los beneficios que estos ofrecen.
Este mecanismo consiste en analizar el *tipo de concern* y las relaciones de la Especificación de Concerns de grano grueso generada con AORE Multidimensional (ver Capítulo 2 – Sección 2.2.3).
Basándonos en experiencias previas sobre sistemas construidos, podemos establecer las siguientes reglas:

1. Los concerns de tipo *Objetivo*, *Riesgo* y *Físico* no representan entidades del diseño de la arquitectura lógica del GIS, y no pueden representarse como aspectos o componentes. Estos concerns son tratados por diversos stakeholders en otras etapas del ciclo de vida. Por ejemplo, los *Equipos (PCs),* como concern de tipo *Físico,* no puede ser representado como un componente o un aspecto, ya que no es una incumbencia del arquitecto de software sino de otras áreas de la organización (infraestructura, planificación, etc.).

2. Los concerns de tipo *Negocio* y *Contexto* deben ser representados como componentes. Es necesario destacar que la naturaleza transversal de un concern de tipo *Negocio* no es condición suficiente para poder representarlo como un aspecto.
   Por ejemplo, si observamos la matriz de relaciones de grano grueso (Capítulo 2 - Figura 16), podemos observar que el concern *Administrador de Taxis* (de tipo *Negocio*) es transversal a los concerns *Cliente* y *Taxi*. Sin embargo, representar al *Administrador de Taxis* como un aspecto, transparente al Cliente y al Taxi, es imposible ya que estos deben tener conocimiento explícito de la entidad que les brinda servicios. Es decir, el Cliente y el Taxi como componentes, no podrían funcionar sin el Administrador de Taxis, ya que es una entidad de software clave e imprescindible del negocio.

3. Los concerns de tipo *Calidad* deben ser representados como aspectos, ya que son concerns adicionales a las reglas del negocio puro del sistema, que si bien lo afectan, deberían ser transparentes y prescindibles para el negocio. Por ejemplo, el concern de *Autenticación* afecta tanto a los concerns Cliente como al Administrador de Taxis, ya que para solicitar el servicio de un taxi es necesario identificar el cliente que realiza la solicitud. Sin embargo, la autenticación no debería ser un tema tratado por el Cliente ni por el Administrador de Taxis, ya que el objetivo primordial del Cliente es solicitar un

servicio y el objetivo primordial del Administrador de Taxis es encontrar un taxi para ese cliente. Las reglas de autenticación deberían ser transparentes para ambos concerns, por este motivo el concern Autenticación debe ser representado como un aspecto.

Basándonos en estas reglas, podemos identificar las siguientes representaciones de los concerns de nuestro caso de estudio:

Concern AORE Multidimensional	Tipo de concern	Entidad del modelo CAM
Cliente	Negocio	Componente
Taxi	Negocio	Componente
Administrador de Taxis	Negocio	Componente
Ciudad	Negocio	Componente
Ubicación	Negocio	Componente
Operador Espacial	Negocio	Componente
Persistencia	Calidad	Apecto
Autenticación	Calidad	Apecto
Visualización	Calidad	Apecto
Comunicación	Calidad	Apecto

Figura 32. Lista de componentes y aspectos

Ahora que definimos la lista de componentes y aspectos, estamos en condiciones de construir el modelo CAM.
A continuación, se puede observar en la Figura 33 un fragmento del modelo CAM del Sistema de Asignación de Taxis, en donde se puede ver la interacción entre los componentes Cliente y Administrador de Taxis, y los aspectos Autenticación, Comunicación y Persistencia.
Más específicamente, la Figura pretende demostrar como se establecen los puntos de intercepción de los aspectos en los componentes. Se puede observar que el Cliente, envía un mensaje al Administrador de Taxis solicitando un taxi (mensaje *SolicitarTaxi*). Una vez que el Cliente dispara este mensaje, el aspecto de Autenticación intercepta el llamado (en el punto *BEFORE_SEND)* y le agrega los parámetros necesarios de autenticación, en este caso el número de teléfono del cliente que realiza la solicitud. Este número servirá luego para verificar si el cliente es un usuario válido.
Luego de esta intercepción, el llamado al mensaje *SolicitarTaxi* sigue su curso y es interceptado por el aspecto de Comunicación (también en el punto *BEFORE_SEND)*, quien actúa como punto de salida y tiene la información necesaria para poder transmitir datos a través de un protocolo de comunicación adecuado (en particular Http).

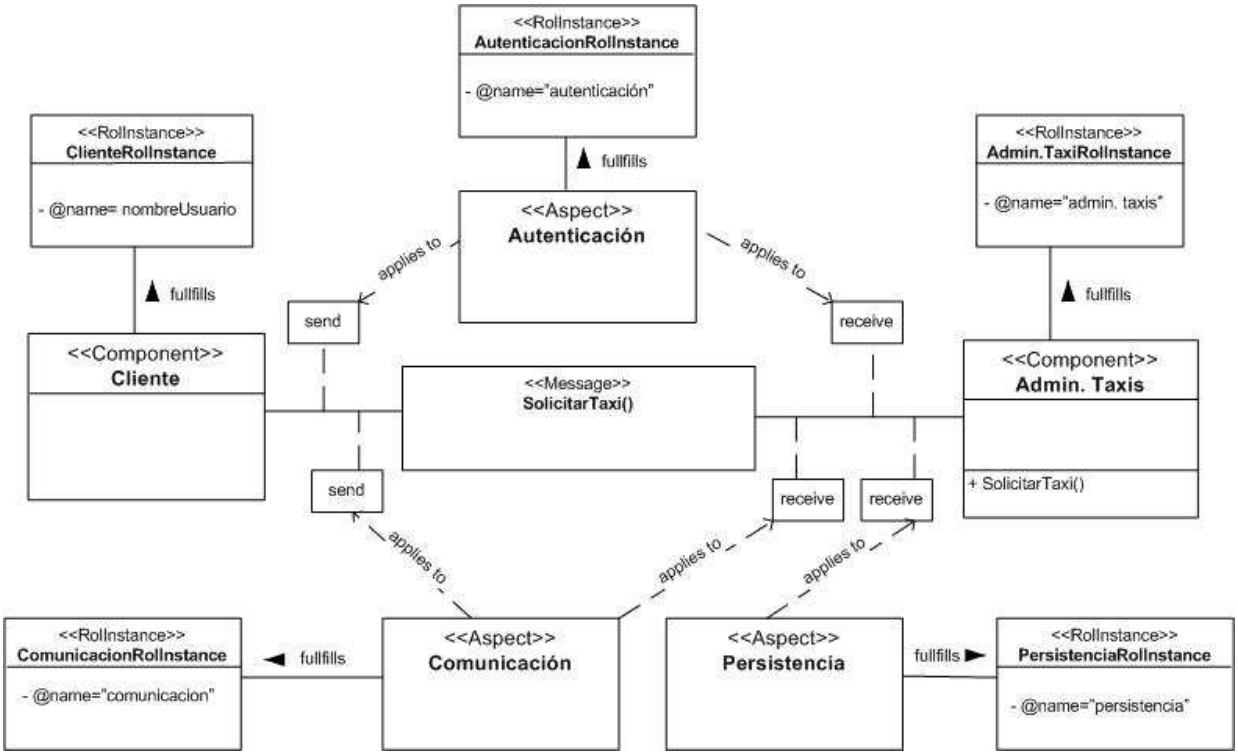


Figura 33. Fragmento del modelo CAM del Sistema de Asignación de Taxis

Por otro lado, el mensaje antes de ser recepcionado por el Administrador de Taxis, es interceptado por el aspecto de Comunicación (en el punto *BEFORE_RECEIVE)*, quien actúa como punto de entrada y posee la información necesaria para publicar a través de un protocolo de comunicación adecuado los servicios que ofrece el Administrador de Taxis. Luego de esta intercepción, el mensaje *SolicitarTaxi* sigue su curso y es interceptado nuevamente por el aspecto de Autenticación (también en el punto *BEFORE_RECEIVE)*, quien en este caso valida el número de teléfono a través de los servicios que ofrece la empresa de telecomunicaciones (que por simplicidad no se muestra en el modelo). Si el teléfono pertenece a un usuario válido, el aspecto deja seguir el curso del mensaje, que será interceptado luego por el aspecto de Persistencia, que tiene como objetivo almacenar en la base de datos la solicitud realizada por el cliente.

Una vez almacenada la solicitud, el aspecto deja seguir el curso del mensaje, y finalmente es recibido por el componente Administrador de Taxis, quien posee la lógica pura para determinar cual es el taxi que debe ser asignado al cliente. Notar que de esta manera, el Cliente en ningún momento tiene conocimiento explícito sobre las reglas de autenticación ni sobre el medio de comunicación utilizado. Por otro lado, el Administrador de Taxis tampoco tiene conocimiento explícito sobre las reglas de autenticación, comunicación y persistencia.

Modelar a los concerns como aspectos y componentes nos permite desacoplar la funcionalidad principal del negocio de la funcionalidad adicional que afecta o contribuye al mismo, permitiendo encapsular el código en unidades atómicas (componente o aspecto) que cumplen un rol determinado en el sistema y que debido al bajo nivel de acoplamiento pueden ser reutilizadas, incorporadas, reemplazadas y eliminadas del sistema a lo largo del

tiempo de manera flexible y transparente para el resto de las entidades de software del sistema. A estas entidades las denominamos *Componentes Dinámicos*.

Capítulo 4. Implementación de Componentes Dinámicos

En el capítulo 3 se ha introducido un modelo de aspectos y componentes llamado CAM, que permite modelar en la fase de Diseño de la Arquitectura los concerns identificados en la fase de Análisis de Requerimientos utilizando la aproximación de AORE-Multidimensional.
A partir de este modelo, que representa la arquitectura lógica del sistema, se debe dar comienzo a la fase de Desarrollo. En esta fase, es necesario poder implementar los componentes dinámicos modelados con CAM como entidades de software utilizando una tecnología que soporte la construcción de componentes y aspectos.

En la actualidad existen diversas tecnologías orientadas a objetos que soportan el desarrollo de componentes. Las más utilizadas en el mercado de desarrollo de software son los frameworks Java 2 Platform Enterprise Edition (J2EE) y Microsoft .NET. Sin embargo, estos carecen de mecanismos para la implementación de aspectos, teniendo que apoyarse en otros frameworks que soportan AOP.

En este capítulo, describiremos las características principales de algunos frameworks que permiten la construcción del GIS sobre la base de componentes y aspectos utilizando las tecnologías J2EE y Microsoft .Net.
Estos frameworks son CAM/DAOP, AspectJ, Aspect#, JBoss AOP, AspectWerkz y Spring. Se analizará cada uno de ellos desde la perspectiva en la que implementan los elementos principales del paradigma AOP: jointpoints, pointcuts, aspectos y mecanismos de weaving.

El objetivo principal de este capítulo por ende es orientar desde una perspectiva más técnica al lector sobre las posibilidades que existen a la hora de elegir un framework en donde se puedan implementar los componentes dinámicos del GIS. Al final del capítulo nos focalizaremos particularmente en el framework de Spring, el cual ofrece varias ventajas y a su vez provee implementaciones tanto para las tecnologías J2EE y .Net.

4.1 Analizando Frameworks AOP

Comenzaremos este capítulo analizando algunos frameworks AOP disponibles en el mercado. Entre estos frameworks describiremos CAM/DAOP [Pinto05], Aspect# y luego basándonos en [Fendt05] describiremos AspectJ, AspectWerkz, JBoss AOP y Spring.

4.1.1 CAM/DAOP

DAOP es la plataforma nativa para la construcción de componentes y aspectos del modelo CAM.
DAOP, del inglés *Dynamic Aspect-Oriented Plataform* (Plataforma Dinámica Orientada a Aspectos) permite la incorporación de aspectos sobre los componentes en tiempo de ejecución. Otra característica importante de DAOP es que las conexiones o dependencias entre los aspectos y los componentes no están hardcodeadas en la implementación de los mismos, sino que son almacenadas en la plataforma.
El mecanismo de composición de la plataforma consulta esta información en tiempo de ejecución para establecer las conexiones entre los aspectos y los componentes. Esto es muy útil porque de esta manera podemos hacer componentes y aspectos más reusables, aislados de dependencias entre ellos. A su vez esta información puede ser adaptada en tiempo de ejecución, mejorando la flexibilidad y adaptabilidad de la aplicación final. Algunos modelos de componentes, como CCM, deben describir el ensamblaje de componentes explícitamente mediante archivos en tiempo de depolyment.

Otra ventaja de DAOP, es que la información arquitectural (expresada en DAOP-ADL, lenguaje de arquitectura nativo de CAM) se carga junto a los componentes y aspectos en tiempo de ejecución. Esto es útil para adaptar y customizar el diseño de la arquitectura dinámicamente.

4.1.2 AspectJ

AspectJ, en su versión 1.2, es un lenguaje AOP para Java basado en código pre-compilado. En otras palabras, los aspectos y pointcuts son expresados en archivos complementarios a los del proyecto Java, que luego en tiempo de compilación un pre-compilador los toma para generar las entidades de software necesarias.
AspectJ centraliza la definición de aspectos y pointcuts en un solo archivo XML, sin embargo se tiene el costo adicional de introducir extensiones del lenguaje Java en el proceso de compilación.
Para configurar los aspectos del sistema, se debe especificar al compilador una lista de aspectos en un archivo con extensión (.lst). De esta forma, el proceso de weaving toma el archivo y aplica los aspectos en tiempo de compilación. Cabe aclarar que no hay forma de habilitar o deshabilitar un aspecto luego en tiempo de ejecución.
La característica más atrayente de AspectJ es que provee un plug-in para integrarse al entorno de desarrollo (IDE) Eclipse. Este plug-in incluye las siguientes características:

- Detecta automáticamente errores de sintaxis y de gramática en tiempo de codificación.
- Provee ventanas donde se pueden ver las secciones de código que son afectadas por los advices.
- Resalta con colores todos los jointpoints afectados.

4.1.3 Aspect#: Castle Project

Aspect# (leido Aspect Sharp) es un framework AOP para .Net. Se basa en la utilización del patrón de diseño DynamicProxy [16] y ofrece un lenguaje integrado para declarar y configurar aspectos. El proceso de weaving de Aspect# se realiza en tiempo de ejecución, y lo que hace es aplicar un Proxy a la clase interceptada. Provee weaving dinámico proxeando la clase interceptada.
Es compatible con AopAlliance, este es un proyecto open-source que tiene como objetivo ofrecer una solución a muchos problemas que existen en las ciertas tecnologías, como EJB. AOP Alliance pretende facilitar y estandarizar el uso de AOP para enriquecer las aplicaciones J2EE. También tiene como objetivo asegurar la interoperabilidad entre las distintas implementaciones AOP para Java.

Aspect# comenzó a formar parte del proyecto Castle [CastleProject] en junio de 2005. Castle es un proyecto open source para la plataforma .Net que tiene como objetivo simplificar el desarrollo de aplicaciones web y Enterprise.

4.1.4 AspectWerkz

AspectWerkz, en su versión 2, es un framework también para Java, pero difiere significantemente de AspectJ en la forma de proveer AOP, ya que no utiliza un pre-compilador, sino que utiliza *annotations* [17] *o un archivo XML (aop.xm*) para especificar los pointcuts. La forma que tiene de conectar los aspectos y los componentes es a través de la manipulación de bytecodes de la Java Virtual Machine[18](JVM), tanto en tiempo de compilación como en tiempo de carga de la aplicación.
Como los aspectos se aplican mediante la JVM, estos pueden ser clases Java, donde los advices son implementados mediante métodos de estas clases.
En cuanto al proceso de weaving, AspectWerkz soporta tanto weaving en tiempo de compilación como de ejecución. Notar que cuando se habla de weaving en tiempo de ejecución nos referimos a habilitar o deshabilitar aspectos. Nuevamente, esta tarea la realiza a nivel de JVM .

En cuanto a la integración con los entornos de desarrollo, AspectWerkz provee un plug-in para NetBeans 3.6 (para soportar el weaving); también provee un plug-in para Eclipse que soporta las siguientes características:

- Soporte para anotaciones de estilo java-doc para Java 1.4 AOP.
- Logging/Tracing de información específica de AspectWerkz .

---

[16] *DynamicProxy*: es un patrón de diseño que permite generar proxies livianos dinámicamente en tiempo de ejecución (proxies dinámicos) para una o más interfaces o incluso clases concretas. Más información en [CastleProject].

[17] Annotations: es metadata que puede ser agregada al código fuente Java. Las annotations Java son reflectivas y son embebidas en los archivos .class generados por el compilador. Más información en [Annotations]

[18] *Java Virtual Machine*: en español Máquina Virtual Java, es un programa nativo capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (Java bytecodes), el cual es generado por el compilador del lenguaje Java.

- Resaltado de jointpoints en el editor con la posibilidad de saltar al código en donde se aplica el advice.

En resumen, AspectWerkz es un poderoso framework AOP. Una desventaja de este framework es la falta de chequeos de errores durante la declaración de pointcuts, cosa que sí se tiene en cuenta en AspectJ.
Cabe destacar que en la actualidad AspectJ y AspectWerkz están integrando sus tecnologías. Una de las primeras pruebas de esta integración es una versión llamada AspectJ 5 que provee anotaciones AOP. Más información sobre AspectJ 5 se puede encontrar en [Eclipse_AJ5].

4.1.5 Spring AOP

El framework Spring AOP, en su versión 2.5 para Java y en su versión 1.2 para .Net, es un framework que implementa AOP basándose en la utilización proxies dinámicos (utilizando las librerías de *reflexión* sobre clases tanto para Java como para .Net).

Si bien la implementación de esta aproximación es más compleja que las otras aproximaciones, posee una gran ventaja: la forma de proveer AOP es mediante código Java 100% (lo mismo sucede para la versión en .Net., pero con código C# o Visual Basic .Net). En otras palabras, no existen dependencias entre el código de los aspectos y un pre-compilador o el entorno de ejecución correspondiente (JVM o CLR) y no se debe extender el lenguaje.

En cuanto a la especificación de aspectos y pointcuts, Spring AOP utiliza un archivo XML (springconig.xml) sin tener que especificar esta información en el código fuente. Tanto los pointcuts como los advices se implementan mediante el framework de Inversión de Control (IoC) [19]de Spring. Esto es una ventaja para aquellos que estén familiarizados con este framework.

En cuanto al proceso de weaving, Spring realiza el weaving en tiempo de ejecución.

Spring permite aplicar aspectos sobre atributos, interfaces, excepciones y métodos (antes y luego de su invocación). A diferencia de AspectJ no permite interceptar código dentro del bloque de un método. Esto es visto por muchos desarrolladores como una desventaja, ya que no se permiten aplicar aspectos en otros niveles de la clase, sin embargo, viéndolo desde el punto de vista de CBSD, no es una desventaja sino una forma de no violar el encapsulamiento de los componentes, principio básico que también sostiene el modelo CAM.

Una desventaja de Spring es que las clases que deseamos aspectizar deben implementar una

[19] *Inversión de Control (IoC):* sinónimo de Inyección de Dependencias, es un concepto junto a unas técnicas de programación en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales, en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones [Fowler04].

interfaz que sea soportada por los proxies dinámicos.

4.1.6 JBoss AOP

JBoss AOP es un framework para Java similar a AspectWekz. Soporta la definición de pointcuts, advices y aspectos mediante anotaciones y archivos XML (jboss-aop.xml).
En la aproximación basada en anotaciones, se puede utilizar Java 1.4 o Java 1.5. Un aspecto JBoss puede ser una clase Java pero debe tener un constructor vacío. JBoss AOP también soporta una primitiva llamada scope asociada a los aspectos e interceptores para especificar el ciclo de vida de los mismos. Se puede crear una instancia de un aspecto a nivel de la máquina virtual, por clase o por jointpoint (más información de la primitiva Scope puede encontrarse en [JBoss_Aop]).

JBoss AOP, como otros frameworks, soporta inversión de control y el proceso de weaving es en tiempo de ejecución.

En cuanto a la integración con IDEs, provee un plug-in para Eclipse que permite:
- Definir interceptores
- Generar automáticamente el archivo jboss-aop.xml
- Resaltar automáticamente los métodos o campos a los cuales se les aplicó un advine.
- Visualizar en una ventana llamada ASpect Manager todos los pointcuts.

4.2 Spring como framework AOP candidato

Llegada la hora de escoger un framework AOP para implementar los componentes dinámicos, es necesario tener en cuenta los principios elementales del proceso que se describió a lo largo de este trabajo, de modo de escoger el framework que mejor se adapte al proceso y a las necesidades de la organización.

Los principios más decisivos para escoger un framework, son los establecidos en la fase anterior de Diseño de la Arquitectura, en nuestro caso serán los establecidos por el modelo CAM, y son los siguientes:

1. Los aspectos se aplican a las interfaces de los componentes para no violar el encapsulamiento y respetar los principios de CBSD.

2. La especificación de los pointcuts no se realiza sobre el código fuente de los componentes, de modo de poder reutilizar los componentes y los aspectos en otras aplicaciones.

3. A su vez, tener desacoplados los aspectos de los componentes, nos permite componerlos y descomponerlo en cualquier estado de la aplicación (ya sea codificación, compilación o ejecución).

Creemos que el framework que mejor se adapta a estos principio es Spring, ya que los aspectos son aplicados sobre los métodos de las clases (respetando el principio 1), la

especificación de aspectos y pointcuts se realiza en archivos XML y no en el código fuente (respetando el principio 2) y a su vez el proceso de weaving es en tiempo de ejecución mediante Inversión de Control (respetando el principio 3).

A su vez, Spring posee otras ventajas que hacen que sea un buen candidato para implementar los componentes dinámicos:

- Es un framework open-source.
- Existen implementaciones para las dos tecnologías de desarrollo más utilizadas en la actualidad: Java y .Net.
- Posee una comunidad de seguidores muy amplia que actúa como soporte ante los problemas e inquietudes que suelen surgir en la fase de Desarrollo [Spring_Community].

4.2.1 Implementando componentes dinámicos en Spring

Es necesario destacar que el proceso de implementación de componentes dinámicos de CAM a componentes dinámicos de Spring no es directo. Existen algunos elementos del modelo CAM que no pueden ser traducidos directamente de acuerdo a la definición de aspectos propuesta por Spring. Por ejemplo, en CAM es necesario que cada componente o aspecto tenga un CID (identificador unívoco). Esto en Spring se representa mediante el atributo *id* de cada objeto.

Por otro lado, en CAM cada componente o aspecto debe tener asociado un *Rol* para desacoplar las dependencias entre componentes y aspectos, de modo que la interacción entre los mismos se lleve a cabo a través de los roles en lugar de los componentes mismos.

A continuación veremos como se establecen las dependencias en Spring sin la necesidad de utilizar roles.

4.2.2 Estableciendo dependencias entre componentes dinámicos

En Spring esta tarea se resuelve definiendo una interfaz para cada componente dinámico, de modo que estos en el código hagan referencia a las interfaces y no a las instancias de los componentes. Dicho de otro modo, en tiempo de compilación los componentes dinámicos no hacen referencia explícita a instancias concretas de otros componentes dinámicos sino a las interfaces de los mismos. Luego en tiempo de ejecución se "inyectan" las instancias concretas de los componentes dinámicos de acuerdo la composición que se define en los archivos XML de Spring.

Por ejemplo, tomemos el caso de estudio del *Sistema de Asignación de Taxis*, hemos visto que el componente *AdministradorDeTaxis* requiere la colaboración del componente *OperadorEspacial* para poder determinar cual es el taxi más cercano a la ubicación de un cliente que ha realizado una solicitud. Veamos una implementación típica en C# (simplificada) de la clase AdministradorDeTaxis (Figura 34).

```
1. public class AdministradorDeTaxis
2. {
3.     private OperadorEspacial operadorEspacial;
4.
5.     public OperadorEspacial Operador
6.     {
7.           get{ return operadorEspacial;}
8.           set{ operadorEspacial = value;}
9.     }
10.
11.    public AdministradorDeTaxis()
12.    {
13.          this.operadorEspacial = new OperadorEspacial();
14.    }
15.


16.    public void SolicitarTaxi(Cliente cliente)
17.    {
18.          Taxi taxi =Operador.EntidadMasCercana(cliente.Ubicacion,radio,typeof(Taxi));
19           bool acepta = Taxi.AceptaViaje(cliente);
20.          if (acepta)
21.                Taxi.TomarViaje(cliente);
22.          else
23.          {
24.                …
25.          }
26.    }
27. }
```

Figura 34. Dependencias entre componentes

Podemos ver que en esta implementación la referencia al componente OperadorEspacial es explícita, es decir, la dependencia está hardcodeada en distintas partes del código (color azul, líneas 3, 5 y 13).
Para desacoplar ambos componentes, lo que sugiere Spring es que el componente AdministradorDeTaxis haga referencia a una interfaz que implemente los métodos del Operador Espacial en lugar de hacer referencia al componente propiamente dicho. Para esto, definiremos la siguiente interfaz visualizada en la Figura 35.

```
1. public interface IOperadorEspacial
2. {
3.     public EntidadMasCercana(Ubicación ubicacion, decimal radio, Type tipo);
4.     ...
5.     ...
6. }
```

Figura 35. Interfaz del componente Operador Espacial

Que luego debe implementar la clase Operador Espacia, como se ve en la Figura 36:

```
1. public class OperadorEspacial: IOperadorEspacial
2. {
3.
4.     public OperadorEspacial()
5.     {
6.
7.     }
```

```
8.
9.     public EntidadMasCercana (Ubicación ubicacion, decimal radio, Type tipo)
10.    {
11.           ...
12.           ...
13.    }
14. }
```

Figura 36. Componente Operador Espacial

Entonces, ahora el código de la clase AdministradorDeTaxis sería el que se muestra en la Figura 37.

```
1. public class AdministradorDeTaxis
2. {
3.     private IOperadorEspacial operadorEspacial;
4.
5.     public IOperadorEspacial Operador
6.     {
7.            get{ return operadorEspacial;}
8.            set{ operadorEspacial = value;}
9.     }
10.
11.    public AdministradorDeTaxis()
12.    {
13.           …
14.    }
15.
16.    public void SolicitarTaxi(Cliente cliente)
17.    {
18.           Taxi taxi =Operador.EntidadMasCercana(cliente.Ubicacion,radio,typeof(Taxi));
19            bool acepta = Taxi.AceptaViaje(cliente);
20.           if (acepta)
21.                  Taxi.TomarViaje(cliente);
22.           else
23.           {
24.                  ...
25.           }
26.    }
```

Figura 37. Componente AdministradorDeTaxis haciendo referencia a la interfaz IOperadorEspacial

Notar que ya no se hardcodea el componente específico del tipo OperadorEspacial, sino que hace referencia a cualquier componente que implemente la interfaz IOperadorEspacial (línea 3 de la Figura 37).

La pregunta que surge ahora es: ¿Cómo y en qué momento se especifica cual es el componente concreto que se desea referenciar?

Spring propone como patrón la *Inyección de Dependencias* [Fowler04] que radica en resolver las dependencias de cada componente (objetos en este caso) generando los mismos cuando se ejecuta la aplicación e inyectándolos en los demás componentes que los necesiten a través de métodos *set* o bien a través del constructor de la clase. En el ejemplo previo de la Figura 37, de las líneas 5 a la 9 se define lo que se conoce en C# como *Propiedad* (Property), que no es más que una forma simplificada de implementar getters y setters, para luego poder realizar la inyección con Spring.

La forma de especificar la inyección de componentes es a través de un archivo XML. Siguiendo el ejemplo del Operador Espacial, la inyección (en Spring.Net) se haría de la siguiente forma (Figura 38):

```
<!—Definición del componente OperadorEspacial -->
<object id="operadorWGS84" type="OperadorEspacial">
</object>
<!—Definición del componente Administrador de Taxis -->
<object id="administradorDeTaxis" type="AdministradorDeTaxis">
    <property name="Operador" ref="operadorWGS84"/>
</object>
```

Figura 38. Inyección del componente Operador Espacial

Podemos observar que en la definición del componente *administradorDeTaxis* se está inyectando en la propiedad *Operador* (setter) la instancia concreta del componente cuyo id es *operadorWGS84*. Notar que al inyectar la instancia del Operador Espacial a través de un setter se evita tener que hardcodear la creación del objeto en el constructor de la clase AdministradorDeTaxis.

Puede suceder que exista otra implementación del Operador Espacial con otras reglas y otras fórmas de realizar los cáluclos, con lo cual podríamos tener:

```
<!—Definición del componente OperadorEspacial que opera con WGS84-->
<object id="operadorWGS84" type="OperadorEspacial">
</object>

<!—Definición del componente OperadorEspacial que opera con UTM-->
<object id="operadorUTM" type="OperadorEspacialUTM">
</object>

<!—Definición del componente Administrador de Taxis -->
<object id="administradorDeTaxis" type="AdministradorDeTaxis">
    <property name="Operador" ref="operadorUTM"/>
</object>
```

Figura 39. Inyección de nuevos componentes

De esta forma, en tiempo de ejecución se pueden componer y descomponer objetos sin modificar el código fuente de la lógica de negocio de la aplicación, logrando un alto grado de desacomplamiento, que es justamente uno de los principios que persigue el modelo CAM mediante la utilización de Roles.

4.2.3 Definiendo aspectos

Hemos visto como se definen y como se componen los componentes en Spring, A continuación vamos a describir un ejemplo de implementación de un aspecto. Lo que haremos será tomar el fragmento del modelo CAM de la Figura 33 (Capítulo 3), en especial tomaremos un aspecto sencillo, el aspecto de *Persistencia.*

Podemos observar en la Figura 33, que el Cliente envía un mensaje llamado *SolicitarTaxi()* al Administrador de Taxis para solicitar un taxi. Para el Cliente es transparente la ubicación física del Administrador de Taxis (si es un componente local o remoto) ya que de esto se encarga el aspecto de *Comunicación,* y también es transparente el mecanismo y los parámetros de autenticación, ya que de esto se encarga el aspecto de *Autenticación*.
Ahora bien, el Cliente y el Administrador de Taxis tampoco necesitan saber si se guardan los datos de una solicitud, menos aun cómo y dónde se almacenan. Por este motivo, lo que haremos es definir un aspecto de *Persistencia,* que intercepte el mensaje SolicitarTaxi luego de ser recibido y procesado por el Administrador de Taxis, y que almacene esta información en una base de datos de manera transparente al Administrador de Taxis.

Como se expresó en el apartado 4.1.5, para construir este aspecto en Spring necesitamos:

a. Definir el proxy dinámico de la clase a aspectizar
b. Definir los pointcuts y el advice

4.2.3.1 Definiendo el Proxy Dinámico

Hemos visto previamente que en Spring no se requiere una extensión del lenguaje para definir aspectos, así que solo es necesario definir una clase que actúe como proxy dinámico del objeto que queremos aspectizar.
A continuación en la Figura 40, se muestra como se define usando Spring un Proxy dinámico para el componente con id *administradorDeTaxis* perteneciente a la clase del modelo *AdministradorDeTaxis* de la Figura 37. Esta clase es la que tiene implementado el menaje SolicitarTaxi().

```
  <object id="aspectoPersistencia"
type="Spring.Aop.Framework.AutoProxy.ObjectNameAutoProxyCreator,
Spring.Aop">
    <property name="ObjectNames">
      <list>
        <value>administradorDeTaxis</value>
      </list>
    </property>
    <property name="InterceptorNames">
      <list>
        <value>PersistenciaAdvisor</value>
      </list>
    </property>
  </object>
```

Figura 40. Definiendo el aspecto mediante un proxy dinámico de Spring

Notar que en la Figura 40 se define un objeto con id *aspectoPersistencia* que es del tipo *ObjectNameAutoProxyCreator*. Esta clase viene incorporada en las librerías AOP de Spring y permite definir aspectos sobre objetos mediante el nombre/identificador de los mismos. Para especificar los objetos aspectizados se debe agregar el nombre o identificador del objeto bajo la propiedad *ObjectNames*.
Además de especificar en el proxy cuales son los objetos aspectizados, es necesario

especificar cual será la clase que actuará como *Advisor*, es decir aquella clase que definirá un advice y los pointcuts. Esta clase se especifica bajo la propiedad *InterceptorNames.*
En el ejemplo de la Figura 40 se puede observar que el advisor es una clase llamada *PersistenciaAdivsor*, ésta contiene la definición de los pointcuts y los advice, tema que se detalla a continuación.

4.2.3.2 Definiendo los pointcuts y el advice

Existen diversas formas de especificar los pointcuts en Spring. Uno de las más utilizadas es la que se basa en expresiones regulares [20] sobre el nombre de los métodos. A continuación se puede observar un ejemplo en donde se define un Advisor del tipo *RegularExpressionMethodPointcutAdvisor*, clase que pertenece a la librería de AOP de Spring, que permite especificar bajo la propiedad *patterns* los patrones a analizar sobre los métodos de las clases aspectizadas.
En el ejemplo se especifica el texto "*SolicitarTaxi*", que se corresponde con el nombre completo del método que se quiere interceptar. Si quisieramos especificar más de un pointcut para el mismo advice podriamos especificar cada pointcut nombre por nombre, sin embargo esto se puede hacer de alguna manera más genérica que abarque a más de un método, por ejemplo, si especificaramos el texto *"Solicitar.*"* (solicitar punto asterisco), estaríamos interceptando todos los métodos que comienzan con el texto "*Solicitar*" sin importar el nombre completo del método. Esto significa que si existe un método llamado *SolicitarDatosDeCalle* también estaría siendo interceptado por el aspecto de Persistencia. En este sentido hay que ser muy cuidadoso con los patrones a la hora de especificar pointcuts para no obtener efectos colaterales.

```
  <object id="PersistenciaAdvisor"
type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor,
Spring.Aop">
    <property name="advice">
      <ref local="persistenciaAdvice"/>
    </property>
    <property name="patterns">
      <list>
        <value>SolicitarTaxi</value>
      </list>
    </property>
  </object>

<object id="persistenciaAdvice" type="PersistenciaAdvice ">
</object>
```

Figura 41. Definiendo los pointcuts y el advice

Notar que en la Figura 41 bajo la propiedad *advice* se especifica el objeto que contiene el comportamiento del advice, en este caso es el objeto con id *persistenciaAdvice* que pertenece a la clase *PersistenciaAdvice*.

[20] *Expresiones regulares*: son expresiones que describen un conjunto de cadenas de caracteres sin enumerar sus elementos.

A continuación vamos a codificar esta clase con el comportamiento necesario para poder almacenar la solicitud del cliente luego de que se ejecute el método SolicitarTaxi(). En este caso particular, la clase *PersistenciaAdvice* almacena la solicitud en una base de datos local, guardando el nombre de usuario, la dirección de donde efectuó la solicitud y la fecha y hora de solicitud.
Para que esta clase sea interpretada por Spring como un Advice que se aplica luego de ejecutar un método, debe implementar la interfaz *IAfterReturningAdvice* que obliga a definir un método llamado *AfterReturning* con el siguiente encabezado:

```
public void AfterReturning(object returnValue, System.Reflection.MethodInfo
method, object[] args, object target)
```

donde:

`returnValue`: es el objeto de retorno del método interceptado, en este caso el método interceptado es SolicitarTaxi(), que retorna un objeto de la clase *Dirección* cuando se recibe exitosamente la solicitud de un cliente. Este objeto contiene los datos de la dirección que representa la ubicación del cliente.

`method`: es un objeto de la clase *System.Reflection.MethodInfo*, que representa el método interceptado.

`args`: es una colección de objetos que representa los argumentos del método interceptado. En el caso de ejemplo, esta colección tendría 3 argumentos: el nombre de usuario, las preferencias y la ubicación (por ejemplo un punto expresado en latitud y longitud) del cliente que realizó la solicitud.

`target`: es el objeto al cual se le envío el mensaje. En el caso de ejemplo, el mensaje Solicitar Taxi() fue enviado al componente AdministradorDeTaxis (por lo tanto *target* será la instancia de este componente).

En el ejemplo de la Figura 42, se puede observar una forma simplificada de almacenar los datos de una solicitud de manera transparente, aunque es necesario destacar que en la práctica existen otros patrones más efectivos para la persistencia de datos, como lo es el patrón Data Access Object [21] y a su vez existen otros factores como la transaccionabilidad y la concurrencia que por cuestiones de simplicidad no han sido tenidos en cuenta en este trabajo a los fines de demostrar como se implementa un aspecto en Spring. Más información sobre concurrencia y transaccionabilidad se puede encontrar en [Spring_Community].

```
public class PersistenciaAdvice : IAfterReturningAdvice
{

  public void AfterReturning(object returnValue, System.Reflection.MethodInfo
method, object[] args, object target)
  {
```

[21] *Data Acces Object (DAO)*: es un patrón de diseño que permite abstraer y encapsular todos los accesos a una fuente de datos. A su vez gestiona la conexión con la fuente de datos para obtener y almacenar datos [Sun07].

```
      string conString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Base.mdb";
      OleDbConnection connection = new OleDbConnection(conString);
      connection.Open();

      Direccion dir = returnValue as Direccion;

      //Agergámos los parámetros del query
      OleDbCommand comando = new OleDbCommand("INSERT INTO Solicitud (nombre_usuario,
calle, numero, piso, depto, fecha_hora) Values (@nombre_usuario, @calle, @numero, @piso,
@depto, @fecha_hora)",connection);
      comando.Parameters.Add("nombre_usuario", OleDbType.VarChar);
      comando.Parameters.Add("calle", OleDbType.VarChar);
      comando.Parameters.Add("numero", OleDbType.VarChar);
      comando.Parameters.Add("piso", OleDbType.VarChar);
      comando.Parameters.Add("depto", OleDbType.VarChar);
      comando.Parameters.Add("fecha_hora", OleDbType.Date);

      //Cargamos los parámetros
      comando.Parameters["nombre_usuario"].Value = args[0] as string;
      comando.Parameters["calle"].Value = dir.Calle;
      comando.Parameters["numero"].Value = dir.Numero;
      comando.Parameters["piso"].Value = dir.Piso;
      comando.Parameters["depto"].Value = dir.Depto;
      comando.Parameters["fecha_hora"].Value = DateTime.Now;

      //Ejecutamos el comando
      comando.ExecuteNonQuery();
      connection.Close();
   }
}
```

Figura 42. Definiendo la clase que actúa como advice

En este capítulo hemos visto desde una perspectiva más técnica como se pueden implementar los componentes dinámicos en un framework AOP cubriendo de esta manera la última fase del proceso propuesto. Cabe destacar que éste proceso no especifica la utilización de un framework AOP en particular, sino que se puede emplear cualquier framework AOP que permita implementar componentes dinámicos en el marco del modelo CAM, respetando los principios de modularización y encapsulamiento.

Trabajos Realizados

Para poner en práctica el proceso en su totalidad, se desarrolló un prototipo de la aplicación GIS tomada como caso de estudio a lo largo de este trabajo. Esta aplicación denominada *Sistema de Asignación de Taxis* tiene como función principal asignar el taxi más cercano a la ubicación de un cliente.

Motivación

La idea de construir un prototipo surge de la necesidad de llevar a la práctica el proceso propuesto en esta tesis para analizar principalmente la viabilidad del mismo y experimentar la evolución de algunos de sus componentes evaluando el impacto desde el punto de vista del re-diseño o re-codificación de los componentes.

Pudimos comprobar que los componentes dinámicos generados mediante este proceso nos proporcionaron en su conjunto una arquitectura GIS flexible que puede evolucionar fácilmente ante los cambios tecnológicos o reglas de negocio propuestas.

En particular, tomamos el concern OperadorEspacial, quien posee entre otras funciones, la función de obtener la entidad más cercana a una coordenada geográfica. Este operador espacial toma la coordenada (latitud y longitud) y realiza una búsqueda de la entidad más cercana realizando un cálculo de distancia entre estas entidades también georeferenciadas mediante coordenadas geográficas.
Lo que hemos hecho fue reemplazar el componente OperadorEspacial por otro de similar funcionalidad denominado OperadorEspacialAltura que tiene la característica de realizar el cálculo de distancia entre entidades teniendo en cuenta la latitud, longitud y a su vez la altura de las entidades involucradas en el cálculo (para poder realizar este cálculo la pre-condición es que los datos de las entidades geográficas dispongan del dato de la altura).
Pudimos confirmar que el reemplazo de este componente no tuvo impacto en el resto de los componentes debido a su bajo nivel de acoplamiento.

Por otro lado, simulamos un cambio en las políticas de seguridad de la aplicación. La nueva política se basa en no utilizar un mecanismo de validación de clientes, por lo tanto deshabilitamos el concern de Autenticación (implementado como un Aspecto) verificando que este cambio no tuvo impacto en el resto de los componentes.

Proceso de desarrollo

Para construir la aplicación comenzamos en la fase de Análisis de Requerimientos identificando y separando concerns mediante el modelo AORE-Multidimensional.
Con este modelo hemos generado las siguientes especificaciones:

- **Metaconcerns**: aquellos concerns que suelen estar presentes en la mayoría de los GIS (ver Anexo 1. Modelo AORE-Multidimensional – MetaConcerns).
- **Concerns Concretos**: cada uno de los concerns identificados con sus requerimientos (ver Anexo 2. Modelo AORE-Multidimensional – Concerns Concretos).
- **Matriz de contribución**: representa la influencia de un concern sobre el resto de los concerns (ver Anexo 3. Modelo AORE-Multidimensional – Matriz de Contribución).
- **Relaciones de Grano Grueso**: representa las relaciones de los concerns reflejada en la matriz de contribución. En esta especificación definimos a su vez el tipo de concern y los tipos de relaciones existentes (ver Anexo 4. Modelo AORE-Multidimensional – Relaciones de Grano Grueso).
- **Reglas de Composición**: representa las relaciones entre los concerns pero a nivel de cada requerimiento (ver Anexo 5. Modelo AORE-Multidimensional – Reglas de Composición).

Una vez establecidos los concerns y sus relaciones, pasamos a la fase de Diseño de Arquitectura en donde modelamos a los concerns mediante el modelo CAM como componentes y aspectos (ver Anexo 6. Modelo CAM – Modelo de Componentes y Aspectos).

Con la arquitectura lógica completa, implementamos cada uno de los componentes en C#, utilizando el Framework .Net versión 2.0., en conjunto del Framework Spring.Net versión 1.2 que nos ha provisto las herramientas AOP.

Este prototipo es una arquitectura cliente-servidor en capas, y está compuesto por 3 aplicaciones:

- *Cliente*: es una aplicación móvil para Smartphones. Para implementarla utilizamos un emulador provisto por Visual Studio 2005 (ver Figura 43).
- *Taxi*: es una aplicación móvil para PocketPCs. Para implementarla también utilizamos un emulador provisto por Visual Studio 2005 (ver Figura 44).
- *Administrador de Taxis*: es una aplicación servidor que provee servicios web que son consumidos por las aplicaciones Cliente y Taxi. Los componentes de esta aplicación que realizan cálculos espaciales utilizan las librerías GIS de Geomedia Professional 5.2. Esta última herramienta también la utilizamos para monitorear el estado de los clientes y los taxis desde una vista panorámica (ver Figura 45).

Figura 43. Aplicación móvil Cliente

Figura 44. Aplicación móvil Taxi

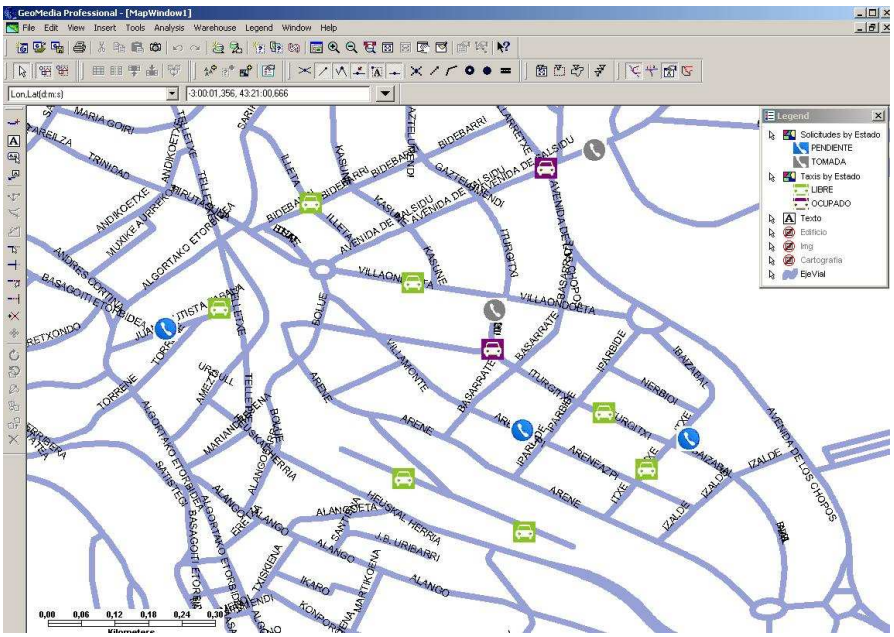


Figura 45. Vista panorámica en Geomedia Professional 5.2

La Figura 43 muestra la aplicación que utilizan los Clientes. Desde esta aplicación se pueden realizar las solicitudes de los viajes. Esta aplicación es capaz de obtener mediante un dispositivo GPS incorporado al teléfono móvil, la ubicación en la ciudad en donde se encuentra el cliente y en donde un Taxi deberá pasar a recogerlo.

La Figura 44 muestra la aplicación que utilizan los Taxis. A diferencia de la primera, es una aplicación que se ejecuta en un dispositivo PocketPC[22]. Desde esta aplicación se pueden recibir las solicitudes de los viajes realizadas por los clientes, y mediante un mapa digital se pueden observar las ubicaciones precisas de los mismos. El dispositivo PocketPC también tiene incorporado un dispositivo GPS que permite obtener la ubicación geográfica de cada Taxi. La aplicación cada 5 segundos reporta la ubicación al Administrador de Taxis para que éste pueda tener actualizada la ubicación de todos los taxis de la ciudad, de modo que al recibir una nueva solicitud, el Administrador la toma y evalúa cual es el taxi más cercano a la ubicación del cliente.
El Administrador de Taxi es una aplicación servidor que se ejecuta en Internet Information Server[23] 5 y toda la funcionalidad que ofrece puede ser accedida mediante servicios web[24].

En la Figura 45 se puede observar un mapa digital con distintos Layers de información. Entre ellos están los layers de Calles de la Ciudad, Taxis de la ciudad y Clientes que han realizado solicitudes. Mediante este mapa se puede observar el estado actual de todas las entidades del dominio que son actualizadas por el Administrador de Taxis.

---

22 *PocketPC*: es una pequeña computadora diseñada para ocupar el mínimo espacio y ser fácilmente transportable. Ejecuta el sistema operativo Windows CE de Microsoft entre otros, el cual le proporciona capacidades similares a las PCs de escritorio.

23 *Internet Information Server (IIS):* es una serie de servicios para los sistemas operativos de la familia Windows que actúan en conjunto como servidor de Internet o Intranet, es decir que en las computadoras que tienen estos servicios instalados se pueden publicar aplicaciones web.

24 *Servicio web*: en inglés *web service*, es un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes de computadoras como Internet.

Conclusión y Trabajos Futuros

Se ha observado que las arquitecturas basadas en capas presentan muchas ventajas para los sistemas de información y en especial para los GISs, que además de involucrar los componentes típicos de cualquier sistema de información (interfaz de usuario, servicios, persistencia, etc.) involucran componentes relacionados con el tratamiento de datos espaciales (sistemas de coordenadas, cartografía, etc.).

Estas arquitecturas, sin embargo, plantean un desafío complejo a los arquitectos de software, que consiste en identificar claramente los componentes de cada capa lógica, y sobre todo como será la interacción entre estos componentes.
Esta interacción define las dependencias entre los componentes pero no el grado de acoplamiento entre los mismos. En realidad, el grado de acoplamiento está determinado por el diseño de los componentes y la forma en la que se implementan.

Hemos visto en el capítulo 3 que uno de los principales inconvenientes de los componentes subyace en la mala modularización sumada a un alto grado de acoplamiento entre los mismos, impidiendo que éstos evolucionen en forma individual en el tiempo. Por este motivo, en este trabajo se ha propuesto un proceso para alcanzar una buena modularización y reducir al máximo el grado de acoplamiento entre los componentes del GIS, obteniendo de esta manera una arquitectura GIS flexible que permite fácilmente la modificación o sustitución de los componentes cuando la aplicación requiere evolucionar en el tiempo.

Para lograr este objetivo, involucramos en el proceso a las fases de Análisis de Requerimientos, Diseño de la Arquitectura y Desarrollo.

La clave de este proceso se basa en la separación temprana de concerns en la fase de Análisis de Requerimiento. Hemos visto que el modelo AORE-Multidimensional es una buena aproximación para separar y especificar los concerns a partir de los requerimientos del GIS. Este modelo nos ha permitido generar una especificación formal de los concerns, que luego ha sido utilizada como punto de partida en la fase de Diseño de la Arquitectura.
En esta fase hemos utilizado el modelo CAM, que ofrece una clara forma de representar los concerns y combina los principales beneficios de los paradigmas CBSD y AOSD. Mediante este modelo hemos podido representar los concerns en función de dos entidades de primera clase: los *componentes* y los *aspectos*. Estas entidades, que hemos denominado *Componentes Dinámicos*, nos proveen muchas ventajas. En primer lugar, mediante los Componentes Dinámicos se pueden modularizar los concerns transversales como unidades funcionales individuales evitando el fenómeno de código entremezclado. Este fenómeno genera código redundante, más difícil de entender y por sobre todas las cosas más difícil de mantener a la hora de evolucionar.

A su vez, los Componentes Dinámicos facilitan la tarea de desarrollo a los programadores, ya que solo se deben concentrar en la lógica pura que implementa cada componente.
Por otro lado, estos componentes al estar desacoplados unos de otros permiten incrementar el potencial de reuso de los mismos.

Por último, en la fase de Desarrollo sugerimos al lector interesado en la implementación de Componentes Dinámicos a utilizar algún framework AOP que respete los principios fundamentales de CAM y que permita la composición de Componentes Dinámicos en tiempo de ejecución para alcanzar un alto grado de flexibilidad y dinamismo.

A partir de la investigación que ha requerido este trabajo surgieron algunas ideas para continuar trabajando alrededor del proceso propuesto. Entre estas ideas se encuentran:

- *Mecanismo de mapeo automático*: se puede estudiar e implementar un mecanismo de mapeo automático de los concerns identificados con el modelo AORE-Multidimensional a componentes y aspectos del modelo CAM.
Si bien en este trabajo se estableció una regla informal de mapeo basándose en experiencias previas, creemos que es posible abstraer reglas concretas de mapeo analizando los tipos de concerns y sobre todo los tipos de relaciones entre los mismos.

- *Meta-concerns de GIS*: a partir del concepto de meta-concern introducido por AORE Multi-Dimensional ha surgido la idea de investigar en detalle y modelar los concerns que suelen estar presentes en la mayoría de los GIS. Descubir un meta-concern de GIS implica poder abstraer sus propiedades y sus relaciones, para luego poder implementar componentes GIS genéricos que sean reutilizables en distintos dominios.

Hemos visto que el proceso propuesto en este trabajo ofrece muchas ventajas y a su vez es un buen punto de partida para futuros trabajos en lo que respecta al modelado y mapeo de concerns de GIS. A nuestro criterio, resulta ser una muy buena aproximación a tener en cuenta a la hora de comenzar el desarrollo de un proyecto GIS que sea propenso a evolucionar en el tiempo.

Anexo I. Modelo AORE-Multidimensional – MetaConcerns

```
<?xml version="1.0" ?>
<MetaConcern name="Cartografía">
  <Description>Representa las entidades espaciales del sistema</Description>
  <Examples> ejes viales, puntos de interes, caÃ±erÃ-as, manzanas, rutas, etc</Examples>
  <Relationships>
    <Relation name="Operador Espacial"/>
  </Relationships>
</MetaConcern>

<MetaConcern name="Operador Espacial">
  <Description>Realiza las operaciones espaciales entre entidades
georeferenciadas</Description>
  <Examples> Calculo de distancias, superficies, adyacencias, superposición de entidades,
etc</Examples>
  <Relationships>
  </Relationships>
</MetaConcern>

<MetaConcern name="Ubicación">
  <Description>Representa la posición geografica de una entidad en en el marco de un
sistema de coordenadas</Description>
  <Examples>coordenandas (latitud,longitud), (latitud, longitud, altura), etc</Examples>
  <Relationships>
    <Relation name="Cartografía"/>
    <Relation name="Operador Espacial"/>
  </Relationships>
</MetaConcern>

<MetaConcern name="Comunicación">
  <Description>Especifica el mecanismo de comunicación entre los sistemas</Description>
  <Examples>servicios web, colas de mensajes, etc</Examples>
  <Relationships>
  </Relationships>
</MetaConcern>

<MetaConcern name="Autenticación" >
  <Description>Es el proceso de verificacón de la identidad de un usuario del
sistema</Description>
  <Examples>nombre de usuario y contraseña, certificados digitales, etc</Examples>
  <Relationships>
  </Relationships>
</MetaConcern>

<MetaConcern name="Persistencia">
  <Description></Description>
  <Examples></Examples>
  <Relationships>
  </Relationships>
</MetaConcern>


<MetaConcern name="Visualización" >
  <Description></Description>
  <Examples></Examples>
  <Relationships>
  </Relationships>
 </MetaConcern>
```

Anexo II. Modelo AORE-Multidimensional – Concerns concretos

```
<?xml version="1.0" ?>
<Concern name="Cliente">
  <Requirement id="1">
    Debe poder solicitar al Administrador de Taxis un taxi a través de su dispositivo
electrónico
    <Requirement id="1.1">Debe poder obtener su ubicación geografica</Requirement>
    <Requirement id="1.2">Debe poder enviar al Administrador de Taxi la solicutud con la
ubicación, su nombre de usuario, su numero de telefono y las preferencias</Requirement>
  </Requirement>
  <Requirement id="2">
    Debe poder especificar las preferecnias del taxi
      <Requirement id="2.1">Debe poder indicar el tipo de vehículo que
necesita</Requirement>
      <Requirement id="2.2">Debe poder indicar las comodidades que debe tener el
vehículo</Requirement>
      <Requirement id="2.3">Debe poder indicar si es fumador o no</Requirement>
  </Requirement>
  <Requirement id="3">
    Debe visualizar en su dispositivo cual es el taxi que lo pasará a buscar y el tiempo
estimado
  </Requirement>
  <Requirement id="4">
    Debe poder cancelar una solicitud previamente realizada
  </Requirement>
</Concern>

<Concern name="Taxi">
  <Requirement id="1">
    Debe indicar al Administrador de Taxis si está o no en servicio especificando un
nombre de usuario y contraseña
  </Requirement>
  <Requirement id="2">
    Debe indicar su ubicación al Administrador de Taxis cada 10 segundos
  </Requirement>
  <Requirement id="3">
    Debe indicar al Administrador de Taxis si esta libre
  </Requirement>
  <Requirement id="4">
    Debe aceptar o rechazar una solicitud de taxi que es indicada por el Administrador de
Taxi
  </Requirement>
  <Requirement id="5">
    Debe visualizar en su dispositivo electronico datos del cliente que realiza la
solicitud
      <Requirement id="5.1">Debe poder visualizar los datos personales del
cliente</Requirement>
      <Requirement id="5.2">Debe poder visualizar en la ciudad la ubicación del
cliente</Requirement>
  </Requirement>
  <Requirement id="6">
    Debe recibir notificaciones de cancelación de viajes
  </Requirement>
</Concern>

<Concern name="Administrador de Taxis">
  <Requirement id="1">
     Debe recibir peticiones de inicio y fin de servicio de los taxis
    <Requirement id="1.1">Debe validar el nombre de usuario y la contraseña enviada por
el taxi</Requirement>
```

```
  </Requirement>
   <Requirement id="2">
     Debe gestionar la ubicación de todos los taxis que están en servicio
     <Requirement id="2.1">Debe poder recibir la ubicación de un taxi</Requirement>
     <Requirement id="2.2">Debe almacenar la ubicación del taxi en una base de
datos</Requirement>
     <Requirement id="2.3">Debe representar graficamente en la ciudad la ubicación del
taxi</Requirement>
  </Requirement>
  <Requirement id="3">
    Debe recibir la notificación de cuando un taxi se libera
  </Requirement>
  <Requirement id="4">
    Debe gestionar solicitudes de los clientes
    <Requirement id="4.1">Debe recibir del cliente el nombre de usuario, el número de
telefono, la ubicación y las prefencias del taxi</Requirement>
    <Requirement id="4.2">Debe almacenar la solictud del cliente en una base de
datos</Requirement>
    <Requirement id="4.3">Debe enviar al cliente un mensaje de recepción exitosa de la
solicitud</Requirement>
  </Requirement>
  <Requirement id="5">
    Debe recibir cancelaciones de solicitudes de los clientes
    <Requirement id="5.1">Debe notificar al taxi que el viaje se cancela</Requirement>
  </Requirement>
  <Requirement id="6">
    Debe autenticar el cliente en el sistema de la entidad externa de telecomunicaciones
    <Requirement id="6.1">Debe enviar el número de telefono del cliente</Requirement>
    <Requirement id="6.2">Debe recibir los datos personales del cliente</Requirement>
  </Requirement>
  <Requirement id="7">
    Debe asignar un taxi a una solicitud realizada por un cliente
    <Requirement id="7.1">Debe encontrar el taxi libre más cercano a la ubicacón del
cliente que cumpla con las preferencias</Requirement>
    <Requirement id="7.2">Debe enviar al taxi la solicitud</Requirement>
    <Requirement id="7.3">Debe recibir la confirmación del taxi indicando si acepta o no
la solicitud</Requirement>
  </Requirement>
  <Requirement id="8">
    Se debe poder visualizar en un mapa digital de la ciudad la ubicación de los taxis
  </Requirement>
  <Requirement id="9">
    Se debe poder visualizar en un mapa digital de la ciudad la ubicación de los clientes
que realizaron solictudes
  </Requirement>
</Concern>

<Concern name="Ciudad">
  <Requirement id="1">
    La Ciudad debe ser un mapa digital representado por un conjunto de ejes viales
relacionados topologicamente y georeferenciados
    <Requirement id="1.1">Los ejes viales deben contar con la información del nombre de
la calle y su altura</Requirement>
  </Requirement>
  <Requirement id="2">
    La Ciudad está compuesta por capas (layers) de puntos donde se representan las
entidades del dominio
  </Requirement>
</Concern>

<Concern name="Ubicación">
  <Requirement id="1">
    Para georeferenciar entidades del dominio se deben utilizar coordenadas geográficas
(latitud, longitud) en el sistema de referencia WGS84.
```

```
  </Requirement>
</Concern>

<Concern name="Operador Espacial">
  <Requirement id="1">
    Debe poder realizar calculos espaciales entre entidades espaciales de la ciudad
  </Requirement>
</Concern>

<Concern name="Comunicación">
  <Requirement id="1">
    El Administrador de Taxis debe poder comunicarse con la Entidad Externa a través de
internet(Servicios Web)
  </Requirement>
  <Requirement id="2">
    Un Taxi debe poder comunicarse con el Administrador de Taxis a través de internet
movil (3G)
  </Requirement>
  <Requirement id="3">
    Un Cliente debe poder comunicarse con el Administrador de Taxis a través de internet
movil (3G)
  </Requirement>
</Concern>

<Concern name="Persistencia">
  <Requirement id="1">
    Se debe poder persistir en una base de datos la información acerca de los taxis
existentes
  </Requirement>
  <Requirement id="2">
    Se debe poder persistir en una base de datos las solicitues de los clientes
  </Requirement>
</Concern>

<Concern name="Visualización">
  <Requirement id="1">
    Se debe proveer una herramienta de visualización de la ciudad
    <Requirement id="1.1">Se debe poder hacer zoom in </Requirement>
    <Requirement id="1.2">Se debe poder hacer zoom out </Requirement>
    <Requirement id="1.3">Se debe poder hacer panning </Requirement>
    <Requirement id="1.4">Se debe poder visualizar información de la entidades de la
ciudad</Requirement>
  </Requirement>
  <Requirement id="2">
    Se debe poder visualizar una imagen de un sector de la ciudad
  </Requirement>
  <Requirement id="3">
    Se debe poder visualizar texto informativo acerca de entidades del dominio
  </Requirement>
</Concern>

<Concern name="Autenticación">
  <Requirement id="1">
    Se debe poder autenticar un usuario a través de un numero de telefono
  </Requirement>
  <Requirement id="2">
    Se debe poder autenticar un usuario a través de un nombre de usuario y contraseña
  </Requirement>
</Concern>
```

Anexo III. Modelo AORE-Multidimensional – Matriz de Contribución

Concerns	Cliente	Taxi	Administrador de Taxis	Ciudad	Ubicación	Operador Espacial	Comunicación	Persistencia	Visualización	Autenticación
Cliente			✓							
Taxi			✓							
Administrador de Taxis	✓	✓								
Ciudad			✓			✓				
Ubicación	✓	✓		✓		✓				
Operador Espacial			✓							
Comunicación	✓	✓	✓							
Persistencia			✓							
Visualización	✓	✓	✓							
Autenticación	✓	✓	✓							

Anexo IV. Modelo AORE-Multidimensional – Relaciones de Grano Grueso

```
<?xml version="1.0" ?>
<Concern name="Cliente">
  <Relationships>
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>

<Concern name="Taxi">
  <Relationships>
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>

<Concern name="Administrador de Taxis">
  <Relationships>
    <Relation name="Cliente"/>
    <Relation name="Taxi"/>
  </Relationships>
</Concern>

<Concern name="Ciudad">
  <Relationships>
    <Relation name="Operador Espacial"/>
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>

<Concern name="Ubicación">
  <Relationships>
    <Relation name="Cliente"/>
    <Relation name="Taxi"/>
    <Relation name="Operador Espacial"/>
    <Relation name="Ciudad"/>
  </Relationships>
</Concern>

<Concern name="Operador Espacial">
  <Relationships>
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>

<Concern name="Comunicación">
  <Relationships>
    <Relation name="Cliente"/>
    <Relation name="Taxi"/>
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>

<Concern name="Persistentcia">
  <Relationships>
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>

<Concern name="Visualización">
  <Relationships>
```

```
    <Relation name="Cliente"/>
    <Relation name="Taxi"/>
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>

<Concern name="Autenticación">
  <Relationships>
    <Relation name="Cliente"/>
    <Relation name="Taxi"/>
    <Relation name="Administrador de Taxis"/>
  </Relationships>
</Concern>
```

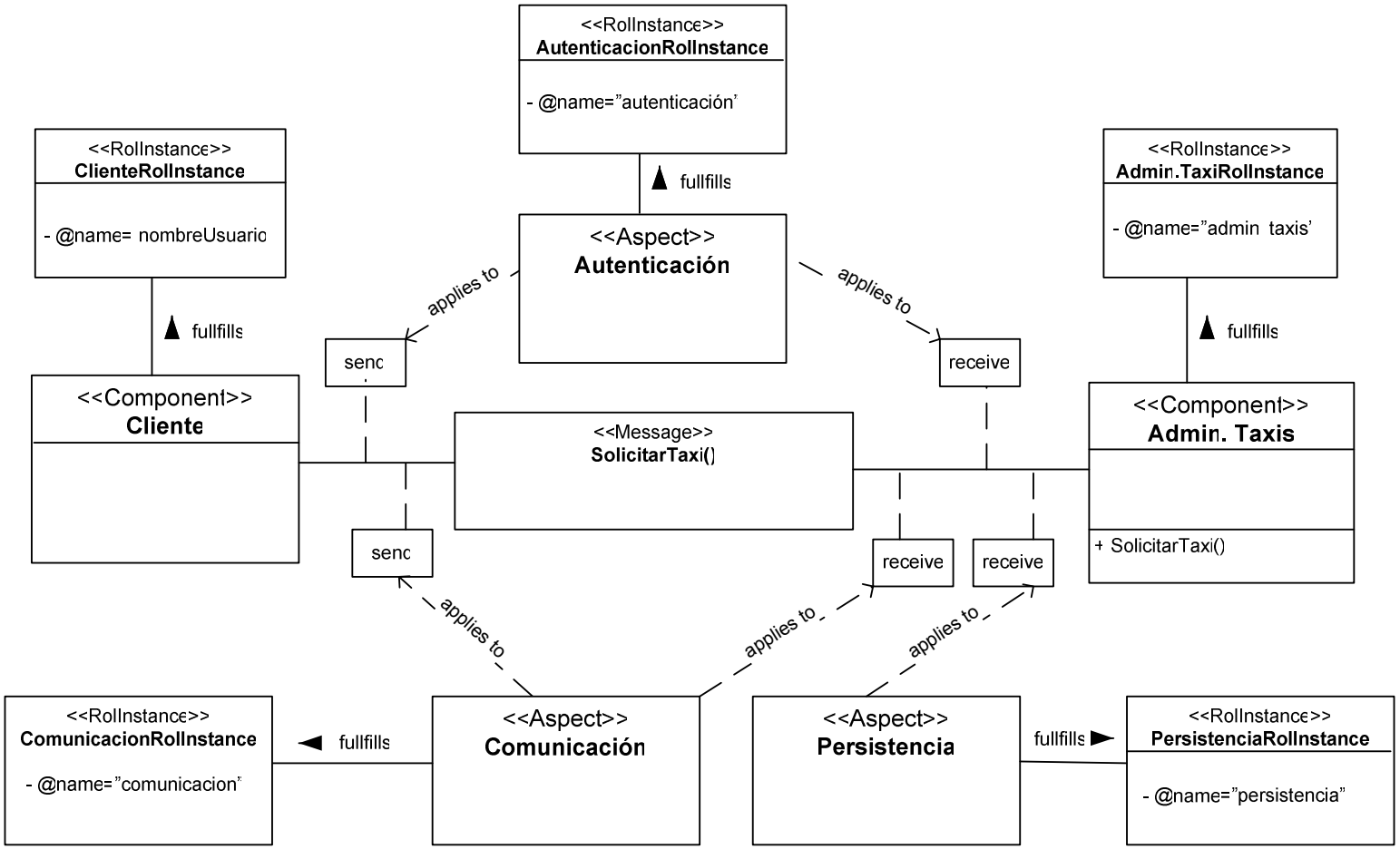
Anexo V. Modelo AORE-Multidimensional – Reglas de Composición

```
<?xml version="1.0" ?>
 <Composition concern="Cliente">
   Los requerimientos del concern Cliente afectan/tiene influencia/actuan sobre los
siguientes requerimientos del concern Administrador de Taxis
   <Requirement id="1" constraint="afectan">
        <Requirement concern="Administrador de Taxis" id="4" children="include" />
  </Requirement>
   <Requirement id="2" constraint="afectan">
     <Requirement concern="Administrador de Taxis" id="7" children="include" />
   </Requirement>
   <Requirement id="4" constraint="afectan">
     <Requirement concern="Administrador de Taxis" id="5" children="include" />
   </Requirement>
</Composition>

<Composition concern="Taxi">
  Los requerimientos del concern Taxi afectan/tiene influencia/actuan sobre los
siguientes requerimientos del concern Administrador de Taxis
  <Requirement id="1" constraint="afectan">
    <Requirement concern="Administrador de Taxis" id="1" children="include" />
  </Requirement>
  <Requirement id="2" constraint="afectan">
    <Requirement concern="Administrador de Taxis" id="2" children="include" />
  </Requirement>
  <Requirement id="3" constraint="afectan">
    <Requirement concern="Administrador de Taxis" id="3"/>
  </Requirement>
  <Requirement id="4" constraint="afectan">
    <Requirement concern="Administrador de Taxis" id="7"/>
  </Requirement>
</Composition>

<Composition concern="Administrador de taxi">
  Los requerimientos del concern Administrador de taxi afectan/tiene influencia/actuan
sobre los siguientes requerimientos
  <Requirement id="4.3" constraint="afectan">
    <Requirement concern="Cliente" id="1" children="include" />
  </Requirement>
  <Requirement id="5.1" constraint="afectan">
    <Requirement concern="Taxi" id="6"/>
  </Requirement>
  <Requirement id="7.2, 7.3" constraint="afectan">
    <Requirement concern="Taxi" id="4"/>
  </Requirement>
</Composition>

<Composition concern="Ciudad">
  Los requerimientos del concern Ciudad son requeridos para resolver los siguientes
requerimientos del concern X
  <Requirement id="all" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="8, 9" />
    <Requirement concern="Operador Espacial" id="1" />
  </Requirement>
</Composition>

<Composition concern="Ubicación">
  Los requerimientos del concern Ubicación son utilizados para resolver los siguientes
requerimientos del concern X
```

```
  <Requirement id="all" constraint="utilizado">
    <Requirement concern="Cliente" id="1.1, 2" />
    <Requirement concern="Taxi" id="2"/>
    <Requirement concern="Ciudad" id="1" children="exclude"/>
    <Requirement concern="Ciudad" id="2"/>
    <Requirement concern="Operador Espacial" id="all"/>
  </Requirement>
</Composition>

<Composition concern="Operador Espacial">
  Los requerimientos del concern Operador Espacial afectan a los siguientes
requerimientos del concern X
  <Requirement id="all" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="7.1"/>
  </Requirement>
</Composition>

<Composition concern="Comunicación">
  Los requerimientos del concern Comunicación son requeridos para resolver los siguientes
requerimientos del concern X
  <Requirement id="1" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="6" children="include" />
  </Requirement>
  <Requirement id="3" constraint="requerido">
    <Requirement concern="Cliente" id="1.2, 4" />
  </Requirement>
  <Requirement id="2" constraint="requerido">
    <Requirement concern="Taxi" id="1,2,3,4"/>
  </Requirement>
</Composition>

<Composition concern="Persistencia">
  Los requerimientos del concern Persistencia afectan a los siguientes requerimientos del
concern X
  <Requirement id="1" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="2.2"/>
  </Requirement>
  <Requirement id="2" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="4.2"/>
  </Requirement>
</Composition>

<Composition concern="Visualización">
  Los requerimientos del concern Persistencia afectan a los siguientes requerimientos del
concern X
  <Requirement id="1" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="8, 9"/>
  </Requirement>
  <Requirement id="2" constraint="requerido">
    <Requirement concern="Taxi" id="5.2"/>
  </Requirement>
  <Requirement id="3" constraint="requerido">
    <Requirement concern="Cliente" id="3"/>
  </Requirement>
</Composition>

<Composition concern="Autenticación">
  Los requerimientos del concern Autenticación afectan a los siguientes requerimientos
del concern X
  <Requirement id="1" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="6"/>
  </Requirement>
  <Requirement id="2" constraint="requerido">
    <Requirement concern="Administrador de Taxis" id="1" children="include"/>
```

```
    </Requirement>
  </Composition>
```

Anexo VI. Modelo CAM – Modelo de Componentes y Aspectos



Referencias

[Alameh01] Nadine Alameh, "*Infrastructures for Distributing Geographic Information Services*", URISA, Octubre 2001.

[Annotations] Sun Microsystems, "*JDK 5.0 Developer's Guide: Annotations*", 2007.

[AOSDnet] Aspect-oriented software development. Website: http://aosd.net.

[Aranoff89] Aranoff, S., *"Geographic information systems: a management perspective"*, WDL Publications, Ottawa, 1989.

[AspectWerkz] AspectWerkz, web page, http://aspectwerkz.codehaus.org/.

[Bakker05] BAKKER, Jethro, TEKINERDOGAN, Bedir, AKSIT, Mehmet. *"Characterization of Early Aspects Approaches"*. Presented at Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with AOSD Conference, 2005.

[Boehm88] Boehm B., "*A Spiral Model of Software Development and Enhancement*", IEEE 21(5):61-72, Mayo 1988.

[Bosque92] Bosque Sendra, Joaquín, "Sistemas de Información Geográfica", Ediciones RIALP, 1992.

[Brown99] Brown, A.W., "*Large-Scale, Component-Based Development*", Prentice Hall PTR, 2000.

[Burrough86] Burrough P.A., "*Principles of geographical information systems for land resource assessment"*, Clarendon Press, Oxford, U.K, 1986.

[Campo_Masip95] Yolanda Letón Campo, Ramón Masip, *"GIS for Business", Madrid, Febrero 1995*.

[Cardenas07] Oscar Cardenas Hernandez, *"Ingeniería Ecológica e Impacto Ambiental",* Universidad de Guadalajara, 2007.

[Carver 98] Carver, S. , Heywood, I., Cornelius, S., "*An Introduction to Geographical Information Systems*", Longman, 1998.

[CastleProject] Castle Stronghold, "Castle Project", URL: http://www.castleproject.org/

[Chambi07] Aruquipa Chambi Marcelo G., Márquez Granado Edwin P., "Desarrollo de Software Basado en Componentes", 2007.

[Clements04] TEKINERDOGAN, Bedir, MOREIRA, Ana, ARAÚJO, Joao, CLEMENTS, Paul. "Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design" Workshop Proceedings, University of Twente, TR-CTIT-04-44, 119 pp, October 2004.

[Coppock_Rhind91] Coppock, J. T. and Rhind, D. W., *"The history of GIS In Geographical Information Systems"*, Vol. 1 (Eds, Maguire, D. J., Goodchild, M. F. and Rhind, D. W.) Longman Scientific and Technical, Harlow, 1991.

[Cown88] Cowen, D., "*GIS versus CAD versus DBMS: what are the differences?",* Photogrammetric Engineering and Remote Sensing*, 1988.*

[Creer08] Derek Creer, "*The Art of Separating Concerns*", Ctrl+Shift+B, Enero 2008.

[Di Neo02] Néstor Di Neo, "*Sistemas de Información Geográfica*", Universidad Nacional de Rosario, Argentina, 2002.

[Dickinson91] Ian Dickinson, "*Open Systems Strategy from IBM*", 1991.

[Dijkstra74] Edsger W. Dijkstra, *"On the role of scientific thought"*, 1974.

[Eclipse_AJ5] The Eclipse Foundation, "*AspectJ: crosscutting objects for better modularity*", URL: http://www.eclipse.org/aspectj/

[Elue08] Enciclopedia Libre Universal en Español, 2008.

[ESRI_ArgGis] ESRI, "ArcGIS", URL: www.esri.com/arcgis.
[Fendt05] Patrick Fendt, "*AOP Technology Update, A framework comparison and internals tour*", SYS-CON Maganizine, agosto 2005.
[Filman05] Filman R., Elrad, T., Clarke, S. y Aksit, M., *"Aspect-Oriented Software Development"*, Addison Wesley, Boston. 2005.
[Finkelstein96] A. Finkelstein and I. Sommerville, *"The Viewpoints FAQ"* BCS/IEE Software Engineering Journal, 1996.
[Fowler04] Martin Fowler, "*Inversion of Control Containers and the Dependency Injection pattern*", junio 2004.
[Gamma95] Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides. "*Design Patterns: Elements of Reusable Object Oriented Software"*. Addison-Wesley, 1995.
[Garlan93] David Garlan, Mary Shaw, "*An Introduction to Software Architecture", Advances in Software Engineering and Knowledge Engineering, Volume I,* World Scientific Publishing Company, 1993.
[Grau06] Ignacio Grau, "*La aplicación de los SIG en la arqueología del paisaje*", Grau Mira, I. ,2006.
[Guinea07] A. Guinea de Salas, S. Abellán, "*Arquitectura SOA para la integración entre software libre y software propietario en entornos mixtos*", I Jornadas de Software Libre, SIGTE, 2007.
[Guptill95] Guptill, S. C., Morrison J. L. (eds.), *"Elements of spatial data quality",* Elsevier Science Ltd., Oxford, 1995.
[Gutierrez04] Gutiérrez J., Villadiego D., Escalona M. y Mejías M., *"Aplicación de la Programación Orientada a Aspectos en el Diseño e Implementación",* 2004.
[Heineman01] George Heineman, Bill Councill, "*Component-Based Software Engineering: Putting the Pieces Together*", Addison-Wesley, pp. 33-48, junio 2001.
[ICSE 2000] Workshop on Multi-Dimensional Separation of Concerns, International Conference on Software Engineering, ICSE 2000. Disponible en: www.research.ibm.com/hyperspace/workshops/icse2000
[IEEE1471-00] IEEE, *"Recommended Practice for Architectural Description of Software-Intensive Systems"*, IEEE Std. 1471-2000.
[Intergraph_Geomedia] Intergraph, "*Geomedia Professional*", URL: www.intergraph.com
[ISO/IEC-9126] ISO/IEC-9126 Information Technology, "*Software Product Evaluation - Quality Characteristics and Guidelines for their Use*", International Standard ISO/IEC 9126, 1991.
[Khalimsky 87] E. Khalimsky, "*Topological structures in computer science!*, J. Appl. Math. Simulation, 1, 1987, 25-40.
[Jacobson92] I. Jacobson, "*Object-Oriented Software Engineering - a Use Case Driven Approach*", Addison-Wesley,1992.
[JBoss_Aop] Red Hat, "*JBoss AOP*", URL:http://www.jboss.org/jbossaop
[Kiczales01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J.and Griswold,W. G., "*An overview of AspectJ*", in Proc.of ECOOP'01, Springer-Verlag, 2001.
[Kiczales97] Kiczales, G. et al. "*Aspect-oriented programming*", ECOOP'97, Finland, Lecture Notes in Computer Science, 1241, pp. 220–242. Springer-Verlag, 1997.
[Klinkenberg84] Brian Klinkenberg, "*Applied GIS: Using your knowledge*", 1984.
[Korhonen97] Markku Korhonen, "*Message Oriented Middleware (MOM)*", 1997.
[Korte01] George B. Korte, P.E., *"The GIS Book",* 5ta edición, OnWord Press, 2001.

[Kruchten95] Kruchten, Philippe, "*Architectural Blueprints--The 4+1 View Model of Software Architecture*". IEEE Software, Institute of Electrical and Electronics Engineers. Noviembre 1995, pp. 42-50.
[Kung05] Kung-Kiu Lau y Zheng Wang, "*A survey of software component models*", 1995.
[Laddad03] Laddad R., *"AspectJ In Action"*, Practical Aspect-Oriented Programming. Manning Publications, 2003.
[Lamsweerde01] A. Lamsweerde, *"Goal-Oriented Requirements Engineering: A Guided Tour"*, 5th International Symposium on Requirements Engineering, 2001, IEEE Computer Society Press, pp. 249-261.
[Longley91] Paul A. Longley, Michael F. Goodchild, David J. Maguire, Ph.D., David W. Rhind "*Geographic Information Systems, First Edition*", 1991.
[MAIC04] Mine Action Information Center, *"Spatial Information Clearinghouse for Humanitarian Mine Action"*, James Madison University, 2004.
[Manning99] Christopher D. Manning, Hinrich Schutze, "*Foundations of Statistical Natural Language Processing"*, MIT Press , 1999, ISBN 978-0262133609, p. xxxi.
[Microsoft_COM] Microsoft, "*COM+ (Component Services) 1.5*", http://msdn.microsoft.com.
[Montaldo05] Diego Fernando Montaldo - Ing. Guillermo Pantaleo, *"Patrones de Diseño de Arquitecturas de Software Enterprise"*, UBA, 2005.
[Moreira1_05] Moreira, Ana, RASHID, Awais, ARAUJO, Joao. "*Multi-dimensional Separation of Concerns in Requirements Engineering*". 2005. The 13th International Conference on Requirements Engineering (RE'05), August 29, September 2, 2005, Paris, France. IEEE Computer Society.
[Moreira2_05] 2. MOREIRA, Ana, RASHID, Awais, ARAUJO, Joao. "A Concern-Oriented Requirements Engineering Model", in Conference on Automated Information Software Engineering (CAISE), Lecture Notes in Computer Science 3520, Berlin: Springer, 2005, pp. 293-308
[Myung-Hee02] Myung-Hee Jo, Yun-Won Jo, "*A study on the construction of GIS component repository System*", Asian Conference on Remote Sensing, 2002.
[Oberndorf97] Tricia Oberndorf, "*COTS and Open Systems--An Overview*", SEI, 1997.
[OMG] Object Management Group, web page: http://www.uml.org/.
[OMG93] Object Management Group, "*The Common Object Request Broker: Architecture and Specification* ", Revision 1.1/Omg Documento Nro 91.12.1, 1993.
[OMG_Corba] Object Managment Group, "*Corba Component Model v4.0*", www.omg.org, 2006.
[Ospina07] Carlos Andrés Ospina, Carlos Andrés Parra, Luis Fernando Londoño, Raquel Anaya, *"Extensión al modelo de Separación Multi-Dimensional de Concerns en Ingeniería de Requisitos",* IDEA's, 2007.
[Parent_Church87] Parent, P. and Church, R., *"Evolution of Geographic Information Systems as Decision Making Tools",* Proceedings in GIS '87, American Society for Photogrammetry and Remote Sensing (ASPRS) y American Congress on Surveying and Mapping (ACSM), Falls Church, Virginia, 1987.
[Parker88] Parker H.D. The Unique Qualities of a Geographic Information-System, comentario en "*Photogrammetric Engineering And Remote Sensing*", 1998.
[Piattini08] Piattini Mario, García Javier, "*Fábrica de Software: experiencias, tecnología y organización*", Rama, Alfaomega, 2008.
[Pickin05] Simon Pickin, "*Introducción al modelo de componentes de CORBA: motivación*

y visión general", Departamento de Ingeniería Telemática Universidad Carlos III de Madrid, 2005.
[Pinto05] Mónica Pinto, Lidia Fuentes and José María Troya*, "A Dynamic Component andAspect-Oriented Platform"*, The Computer Journal, 2005.
[Pitney_MapInfo] Pitney Bowes, "*MapInfo*", URL: http://www.mapinfo.com/
[Popovici02] Popovici, A., Gross, T. and Alonso, G. "Dynamic *weaving for aspect-oriented programming*",Proc. First Int. Conference on AOSD, ACM Press, 2002.
[Poston92] Poston, R. y Sexton, M., "Evaluating and Selecting Testing Tools", IEEE Software, 1992.
[Rey99] Jorge Franco Rey, "*Nociones de Topografía, Geodesia y Cartografía*", Universidad de Extremadura, 1999.
[Roche07] Rodríguez Roche S., "*El análisis de dominio en la ciencia de la información*", Acimed 2007;15(6).
[Rodriguez 04] Miguel Ángel Rodríguez Luaces, "*A Genirc Architecture for Geographic Information Systems*", Universidad de La Coruña, 2004.
[Royce70] Royce, Winston, "*Managing the Development of Large Software Systems*", Proceedings of IEEE WESCON 26, 1970.
[SofDev08] Bertrand Meyer y Clemens Szyperski, "*Beyond Objects*", Software Development magazine, 1998—2000.
[Sommer06] S. Sommer, T. Wade , "*A to Z GIS: An Illustrated Dictionary of Geographic Information Systems*", 2006.
[Spring_Community] Spring Source Community, URL: http://forum.springframework.org
[Sun07] Core J2EE Patterns, "*Data Access Objects*", Sun Microsystems Inc., agosto 2007.
[Sun_EJB] Sun, "*Enterprise JavaBeans: EJB Core Contracts and Requirements*", www.sun.com, 2006.
[Szyperski02] Szyperski, C., "*Component Software. Beyond Object-Oriented Programming"*, (2nd edn), Addison-Wesley, 2002.
[Urisa08] The Urban and Regional Information Systems Association (URISA), "*GIS Hall of Fame*", 2008.
[Valetto95] Valetto, G. y Kaiser, G., *"Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments"*, 7th IEEE International Workshop on CASE, IEEE Computer Society Press, 1995.
[Vallecillo99] Antonio Vallecillo, Raúl Monge, "*Desarrollo de Aplicaciones basado en Componentes y Frameworks*", 1999.
[W3C00] C. M. Sperberg-McQueen y Henry Thompson, W3C, "*XML Schema*", 2000.
[Weaver04] James L. Weaver, Kevin Mukhar, and Jim Crume, "*Beginning J2EE 1.4 From Novice to Professional*", Apress, ISBN: 1590593413, 2004.
[XPath99] W3C, "*XML Path Language (XPath)*", Version 1.0, Recommendation 16 Noviembre de 1999.
[Zheng05] Kung-Kiu Lau y Zheng Wang, "*A taxonomy of software component models*", In *Proc. 31st Euromicro Conference*, IEEE Computer Society Press, University of Manchester, 2005.