Universidad Tecnológica Nacional
Facultad Regional Tucumán

Tesis de Maestría

Estudio y definición de reglas de validación para modelos basados en el enfoque organizacional de Sistemas Multiagentes

Autor:
Esp. Ing. Pedro B. Araujo

Director:
Dr. Sebastián A. Rodríguez

Esta tesis cumple con los requisitos
para el grado de Magister en Ingeniería en Sistemas de Información

en

Grupo de Investigación en Tecnologías Avanzadas Informáticas
Universidad Tecnológica Nacional - Facultad Regional Tucumán

3 de marzo de 2016

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL TUCUMÁN

Abstract

Magister en Ingeniería en Sistemas de Información

Estudio y definición de reglas de validación para modelos basados en el enfoque organizacional de Sistemas Multiagentes

por Esp. Ing. Pedro B. Araujo

La Ingeniería de Software Orientada a Agentes alcanzó, en estos últimos tiempos, un notable nivel de madurez. Sin embargo, aún carece de determinadas técnicas que son muy comunes dentro del paradigma orientado a objetos, y que permiten generar modelos correctos que facilitan la comprensión y adopción del paradigma, como así también su implementación. Los patrones de diseño, antipatrones, micropatrones, la adopción de estándares; son algunos de los elementos existentes en la actualidad y que, en muchas casos, capitalizan experiencias pasadas. Es por ello que es necesario adaptar o redefinir –debido a las diferencias de los conceptos subyacentes entre paradigmas- estas técnicas al paradigma de los Sistemas Multiagentes.

En esta Tesis de Maestría se propone el desafío de estudiar y definir reglas que permitan detectar, de manera temprana, diseños pobres o malos en modelos organizacionales de Sistemas Multiagentes. Dos enfoques son explorados: El primero, es la validación sintáctica que permite eliminar de los modelos: las ambigüedades, inconsistencias, etc. llevando a que los artefactos generados por el proceso de desarrollo sean conceptualmente válidos y correctos. El segundo enfoque, es la detección de diseños pobres o malos –denominados smells- que, si no son abordados adecuadamente, pueden tener consecuencias en futuros desarrollos del sistema.

Por último, y para validar las propuestas realizadas, se han analizado dos proyectos a las cuales se le aplicaron las reglas definidas. A su vez, se hacen las explicaciones pertinentes para cada caso.

Agradecimientos

Quiero expresar mi profundo agradecimiento, en primer lugar, a toda mi familia: Padres, hermanos y sobrinos, que sin su entendimiento y apoyo esta tesis no hubiera sido posible. Estos siempre me han alentado en cada uno de mis aventuras acompañándome por igual, en mis logros como en las caídas.

A mi Director, Dr. Sebastián Rodríguez, por sus consejos, su experiencia, apoyo constantes.

A los amigos y compañeros que pasaron por mi vida, aunque a muchos de ellos veo poco me hicieron llegar sus buenos deseos.

Por último, y no por ello menos importantes, a mis compañeros de Grupo de Investigación en Tecnologías Informáticas Avanzadas (GITIA): Diego, Nicolás, Jorge, Adrían, Ana, Matías, Alan, Dr. Will, Dr. Gotay, por todas las horas de trabajo compartidas como así también las alegrías vividas.

Índice general

Índice de figuras

Índice de cuadros

Índice de reglas

Abreviaciones

ACL	**A**gent **C**ommunications **L**anguage
AUML	**A**gent **U**nified **M**odeling **L**anguage
AGR	**A**gent **G**roup **R**ole
AOTDD	**A**gent **O**riented **T**est **D**riven **D**evelopment
BBN	**B**ayesian **B**eliefs **N**etworks
CASE	**C**omputer **A**ided **S**oftware **E**ngineering
CI	**C**apacity **I**dentification
CRIO	**C**apacity **R**ole **I**nteraction **O**rganization
DML	**D**omain **S**pecific **L**anguage
DRD	**D**iagrama de **R**equerimientos del **D**ominio
DOD	**D**iagrama de **O**ntología del **D**ominio
DSL	**D**omain **S**pecific **L**anguage
ECL	**E**psilon **C**omparison **L**anguage
EGL	**E**psilon **G**eneration **L**anguage
EMF	**E**clipse **M**odeling **F**ramework
EML	**E**psilon **M**erging **L**anguage
EMP	**E**clipse **M**odeling **P**roject
EOL	**E**psilon **O**bject **L**anguage
ETL	**E**psilon **T**ransfomation **L**anguage
EVL	**E**psilon **V**alidation **L**anguage
EWL	**E**psilon **W**izard **L**anguage
FIPA	**F**oundation for **I**ntelligent **P**hysical **A**gents
GMF	**G**raphical **M**odeling **F**ramework
GEF	**G**raphical **E**diting **F**ramework
GUI	**G**raphical **U**ser **I**nterface

HFO	**H**ighly **F**ragmented **O**rganization
IDE	**I**ntegrated **D**evelopment **E**nviroment
IDK	**I**ngenias **D**evelopment **K**it
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IRI	**I**nteraction and **R**ole **I**dentification
ISOA	**I**ngeniería de **S**oftware **O**rientada a **A**gentes
KQML	**K**nowledge **Q**uery and **M**anipulation **L**anguage
MACODO	**M**iddleware **A**rchitecture for **CO**ntext-driven **D**ynamic agent **O**rganizations
MDA	**M**odel **D**riven **A**rchitecture
MDD	**M**odel **D**riven **D**evelopment
MVC	**M**odel **V**iew **C**ontroller
MOF	**M**eta **O**bject **F**acilities
ODS	**O**rganization **D**esign **S**mells
OCMAS	**O**rganizational-**C**entered **M**ultiagent **S**ystems
OCL	**O**bject-**C**onstraint **L**anguage
ODML	**O**rganizational-**D**esign **M**odeling **L**anguage
OI	**O**rganization-**I**dentification
OMG	**O**bject **M**anagement **G**roup
OMNI	**O**bject **M**odel for **N**ormative **I**nstitutions
OperA	**O**rganizations per **A**agents
OZS	**O**bject-**Z** **S**tatechart
O-MaSE	**O**rganization-based **M**ulti-agent **S**oftware **E**ngineering
PBI	**P**roducto **B**ruto **I**nterno
POD	**P**roblem **O**ntology **D**iagram
POM	**P**roject **O**bject **M**anagement
PDT	**P**rometheus **D**esign **T**ool
PTK	**P**ASSI **T**ool**K**it
PASSI	**P**rocess for **A**gent **S**ocieties **S**pecification and **I**mplementation
QA	**Q**uality **A**ssurance
RCP	**R**ich **C**lient **P**latform
RCA	**R**ich **C**lient **A**pplication
RIO	**R**ol **I**nteraction **O**rganization
RUP	**R**ational **U**nified **P**rocess

SPEM	**S**oftware **P**rocess **E**ngineering **M**etamodel
SMA	**S**istemas **M**ulti-**a**gentes
T-Tool	**T**ropos-**T**ools
UML	**U**nified **M**odeling **L**anguage
XML	e**X**tensible **M**arkup **L**anguage

A mi familia, a mis amigos y a Dios.

Capítulo 1

Introducción

De acuerdo a la definición dada por Cossentino en [2] "Diseñar un Sistemas Multiagentes no es una tarea sencilla: crear agentes, ambientes, normas, organizaciones y hacer que estos cooperen de manera que resuelvan una tarea colectiva es tanto un arte como una ciencia". De hecho, las características subyacentes de los Sistemas Multiagentes (SMA de ahora en más), tales como la autonomía, reactividad, apertura, interacción entre agentes y la proactividad; dan paso a la ausencia de un control centralizado en el sistema. Este problema ha dado nacimiento a un prolífico campo denominado Ingeniería de Software Orientada a Agentes (ISOA). Existen muchas contribuciones del ISOA al análisis y diseño de los SMA y sus problemas. Entre estas contribuciones, uno puede citar las definiciones de los conceptos de los SMA, metodologías y herramientas de software que proveen soporte para el análisis y diseño (denominadas herramientas CASE en la ingeniería de software clásica). La idea de estas herramientas es asistir a los analistas/diseñadores proveyéndoles de un conjunto de servicios automáticos o semiautomáticos durante las fases de análisis y/o diseño.

Esta tesis está inspirada en trabajos que tuvieron lugar décadas atrás en el paradigma orientado a objetos y que facilitan la gestión del desarrollo y sobre todo la calidad del proyecto. Diferentes técnicas y conceptos han sido propuestos para abordar estos desafíos. Entre ellos podemos encontrar: patrones de diseño, micro-patrones, estándares de modelado y codificación, buenas prácticas y finalmente los *code-smells*. Todas estas técnicas son ampliamente utilizadas en el paradigma orientado a objetos; si embargo, si uno está interesado en agentes y sistemas multiagentes, dichas técnicas y sus herramientas no pueden ser utilizadas directamente dado que los SMA están construidos sobre la base de abstracciones y tecnologías diferentes. Así, consideramos que es imperativo refinar o redefinir muchas de estas técnicas para el desarrollo de SMA.

Tanto las metodologías como las herramientas proveen una serie de guías y consejos respecto a como producir modelos; sin embargo, estas no alcanzan a cubrir una variedad de situaciones. Así, el punto de partida de este trabajo está basado en el análisis de las instancias de un modelo organizacional en busca de diseños pobres o defectuosos. La contribución de este trabajo es doble: el primero son las validaciones sintácticas, estas permiten verificar que las instancias de un metamodelo sean conceptualmente válidas con respecto a las restricciones propias del metamodelo como con las definidas en las actividades impuestas por la metodología. Sin embargo, y aún contando con el diagrama sintácticamente correcto, el analista/diseñador no puede estar seguro de su calidad inherente, por ejemplo, en términos de coherencia, robustez, eficiencia, modularidad, extensibilidad, etc. Es por ello que el segundo objetivo es analizar dichas instancias para definir -con la ayuda de algún lenguaje que enriquezca el modelo- reglas que identifiquen situaciones que potencialmente puedan conducir a smells organizacionales. Consideramos que detectar dichos problemas en etapas tempranas del ciclo de vida de desarrollo de un sistema tienen un impacto significativo en el ahorro de costo y tiempo. A su vez, se genera un modelo potencialmente más fácil de entender y extender, aumentando de esta manera su grado de mantenibilidad y extensibilidad.

1.1. Objetivos del Trabajo

El objetivo general de esta tesis consiste en el estudio, definición y aplicación de reglas de validación para la detección temprana de "problemas" en los modelos de sistemas multiagentes basados en el enfoque organizacional. Dos aproximaciones son abordadas: la sintáctica y la semántica. La validación de estas aproximaciones garantizará que los desarrolladores siempre cuenten con artefactos del modelo completos, consistentes, carentes de ambigüedad y correctos. Es por ello que contar con un mecanismo de validación automatizada nos permitirá responder rápidamente a los errores detectados.

A continuación se describirá brevemente los objetivos específicos de la tesis:

- A partir del estudio de numerosos modelos se deben extraer los problemas comunes de diseño que estos presentan, identificando las características del mismo en relación al dominio en el que se desarrollan.
- Determinar el lenguaje que será utilizado tanto para la validación sintáctica como para la detección de los "smells" (término que será explicado más adelante). Una vez determinado que lenguaje será utilizado se deben expresar en reglas como detectar dichos problemas.

- Evaluar si el lenguaje adoptado es posible de integrar, desarrollando una extensión, en la herramienta de desarrollo Janeiro Studio CASE Tool [3]. El plugin deberá permitir en lo posible, la creación, modificación y eliminación como así también la importación y exportación de reglas de validación

1.2. Principales contribuciones

En el marco de desarrollo de esta Tesis de Maestría, se trabajó principalmente sobre el área de Calidad de modelos organizacionales para Sistemas Multiagentes. Esta tesis representa una continuación de diversos trabajos que se vienen desarrollando dentro de Grupo de Investigación en Tecnologías Informáticas Avanzadas (GITIA) y que tuvo su inicio con el desarrollo de una herramienta que brinda soporte para modelos basados en el enfoque organizacional CRIO.

Paralelamente a estos desarrollos, se ha trabajado en el área del calidad, más precisamente en la detección de anomalías en los modelos. La primera propuesta es un conjunto de reglas de validación sintácticas que verifica que las instancias de un metamodelo cumplan con las especificaciones propias del metamodelo y las especificaciones definidas en las actividades de la metodología. Los resultados de este desarrollo han sido comunicados en la revista indexada Ciencia y Tecnología de la Universidad de Palermo [4]. Una vez realizadas las validaciones sintácticas, el siguiente paso es ejecutar las reglas cuyo objetivo es poner de manifiesto los diseños pobres en los modelos organizacionales y que pueden impactar negativamente en las etapas posteriores. Para este objetivo se ha realizado una comunicación en el COIN@IJCAI 2015 [5].

1.3. Casos de Estudio

Con el objetivo de validar y mejorar las propuestas realizadas, las mismas han sido aplicadas a dos casos de estudio reales. Estas aplicaciones han sido muy relevantes gracias a la devoluciones que permitieron refinar las propuestas vertidas en esta tesis.

El primer caso de estudio, está relacionado con el análisis de un modelo organizacional realizado para un proyecto homologado que se desarrolla dentro del GITIA. El mismo modela una de las actividades más importante de la provincia de Tucumán y que es la zafra cañera. Más precisamente, se abordan los temas de optimización en la distribución y transporte de la caña de azúcar desde el campo a los distintos ingenios de la provincia en busca de un beneficio global.

El segundo proyecto fue extraído de la literatura agentes y representa un modelo de un simulador de Smart-Grid. El objetivo es administrar de manera más eficiente la red eléctrica. Dentro de los Smart-Grid están contenidos los MicroGrid, que hace referencia a una parte de la red que puede auto-abastecerse y gestionarse separada de la red principal. De hecho, estas redes, y si es que así lo requieren, pueden o conectarse a la red principal para abastecer o proveerles de energía. El modelo permite tener en cuenta diversos tipos de dispositivos que pueden existir en las redes eléctricas.

1.4. Plan del Documento

El presente documento está estructurado de la siguiente manera:

- El **Capítulo 2** presenta una breve introducción a la teoría de Agentes Inteligentes y de los Sistemas Multiagentes, haciendo foco en la sistemas centrados en los conceptos organizacionales. A su vez, se da introducción al proceso de desarrollo ASPECS haciendo hincapié en la primera fase del mismo. Se describe el metamodelo CRIO y sus principales características. Por último, se destaca la herramienta de desarrollo Janeiro Studio que tiene como núcleo principal una adaptación informática del metamodelo elegido y que será la base de nuestras propuestas.
- El **Capítulo 3** describe una introducción a la Calidad de Software, más particularmente a la rama de verificación y validación. En la misma, resaltamos la importancia de los Code/Design Smells que representan desviaciones por parte de los desarrolladores de los estándares de programación/codificación adoptadas por un organización. Además, se realiza una revisión bibliográfica de la literatura dentro del paradigma orientado a objeto y agentes.
- En el **Capítulo 4** se describe la primera contribución de la tesis junto a los elementos desarrollados como soporte. En el mismo se propone un conjunto de reglas de validación sintáctica basadas en un lenguaje particular. Estas reglas permiten comprobar, automáticamente, que los modelos sean conceptualmente válidos. En el mismo capítulo se justifica la adopción de Epsilon Validation Language (EVL) por sobre otro lenguaje de expresividad detallando su integración en Janeiro Studio. El soporte informático es un plugin desarrollado que permite la definición, almacenamiento y ejecución de dichas reglas.
- El **Capítulo 5** presenta la contribución principal de la tesis, denominadas “smells” de diseño organizacional. Son un conjunto de reglas semánticas definidas en EVL que tiene por objetivo detectar automáticamente -y en etapas tempranas del

modelado- los errores más frecuentes cometidos por los diseñadores. Para cada smell se realiza un descripción detallada de su propósito, los pasos metodológicos de como identificarlo en el modelo; y finalmente, la regla EVL con su respectiva explicación y limitaciones.

- En el **Capítulo 6** se validaron las propuestas realizadas usando diversos proyectos que se están desarrollando dentro del marco de GITIA. En el mismo, se da introducción al proyecto de optimización del transporte de la zafra cañera de la provincia de Tucumán. El segundo, es el de la modelización de un Smart-Grid para la administración inteligente de los recursos energéticos que pueden estar disponibles en una ciudad o localidad.

- Por último, en el **Capítulo 7** se realiza una recapitulación de las principales contribuciones de la tesis, resaltando los aspectos principales de la misma y que constituyen el punto de partida para el desarrollo de futuros trabajos.

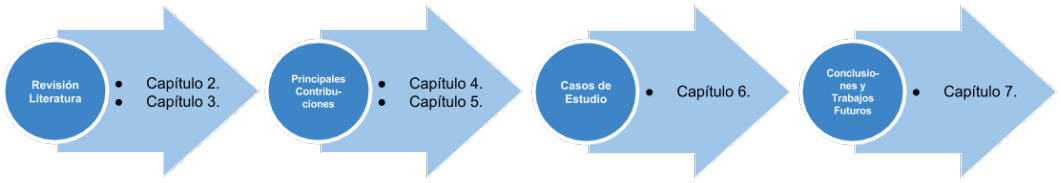

FIGURA 1.1: Contenidos de la Tesis.

Capítulo 2

Agentes, Sistemas Multiagentes y Teoría Organizacional

2.1. Agentes y SMA

En los 80's la Teoría Agentes y de los Sistemas Multiagentes emergieron como unas de las tecnologías más interesantes para abordar sistemas complejos, distribuidos y abiertos. En este sentido los sistemas basados en agentes han demostrado exitosamente su potencial abordando una amplia variedad de problemas que va desde la robótica, resolución de problemas distribuidos, modelado y simulación, sólo por mencionar algunas áreas [6].

El creciente interés por esta novedosa tecnología se debe, entre otros, al concepto de agente como entidad autónoma. Dentro de la literatura de agencia existen diferentes definiciones de lo que es un agente, una de las más conocidas y aceptadas se debe a Wooldridge [7] que establece:

"Un agente es una sistema computacional que está situado en algún ambiente y que es capaz de realizar acciones autónomas en el ambiente de manera de alcanzar los objetivos que le fueron delegados"

En cambio, Weiss [8] realiza la siguiente definición:

"Un agente es una entidad física o virtual con un alto grado de autonomía, independiente, capaz de cooperar, competir, comunicarse, actuar flexiblemente y ejercer control sobre su comportamiento dentro del marco de sus objetivos. Estas entidades evolucionan dentro de un ambiente al cual son capaces de percibir y modificar."

En este sentido, IBM Agent Group define a los agentes como:

"Los agentes inteligentes son entidades de software que llevan a cabo un conjunto de operaciones por cuenta de un usuario o programa, con un cierto grado de independencia o autonomía, y al hacerlo, emplean algún conocimiento o representación de los deseos o metas de los usuarios."

A pesar de lo prolífico del campo y el nivel de madurez que se ha alcanzado, no existe una definición universalmente aceptada de lo que significa un agente. De hecho, comparando las definiciones dadas es posible identificar la autonomía como la característica principal que diferencia a los agentes de otros paradigmas como por ejemplo: algoritmos genéticos, redes bayesianas, objetos, etc. Sin embargo, Wooldridge estima que es posible identificar un agente a partir de -al menos- cuatro propiedades básicas [6]:

- **Autonomía**. El agente toma decisiones basado en su propio estado, sin intervención directa de humanos u otros agentes. En otras palabras, nadie puede decirle lo que tiene que hacer; en este sentido, se le puede solicitar/pedir o influenciar para que realice algo por nosotros. Además, tiene control sobre su estado interno y sobre sus propias acciones.
- **Reactividad**. Los agentes son capaces de percibir su ambiente y responder en un periodo de tiempo adecuado frente a cambios producidos en el ambiente. El domino de la aplicación nos indicará el tiempo y dependerá de donde se aplica.
- **Proactividad**. Los agentes son capaces de exhibir un comportamiento dirigido por metas, tomando la iniciativa. Así, el agente es capaz de generar los pasos intermedios para alcanzar los objetivos planteados reconociendo oportunidades.
- **Capacidad Social**. Los agentes son capaces de interactuar con otros agentes mediante algún mecanismo de comunicación. Esta interacción no es más que el intercambio directo o indirecto de información. El primero hace referencia a un intercambio sin intermediarios; en cambio, los mecanismos indirectos generalmente usan el ambiente para comunicarse, tal es el caso de las hormigas que se comunican depositando feromonas.

Los atributos enunciados son posiblemente los de mayor consenso dentro de la comunidad científica. Sin embargo, estos no son los únicos dado que existen otros atributos asociados a la Teoría de Agentes que según el dominio considerado cobran relevancia. Tales atributos son: movilidad, adaptabilidad, versatilidad, confiabilidad, robustez, persistencia, benevolencia (altruismo), etc.

En la Figura 2.1 se puede observar donde se sitúa la tecnología agentes respecto a otros paradigmas clásicos. En la misma se pueden deducir dos cuestiones: La primera, *Nivel*

de Abstracción, ubica al paradigma en un nivel mayor de abstracción que el resto de las tecnologías tales como objetos, procedural y ensamblador. Aun así, esto no significa que la teoría de agencia apareció para reemplazar a las otras, sino que permite abordar problemas con ciertas características que con el resto de los parasigmas, el esfuerzo requerido para implementar dichos sistemas resultarían muy altos.

La segunda cuestión, *Tiempo*, nos indica que la teoría de agencia representa una tecnología relativamente nueva frente a tecnologías como objetos o procedural que han alcanzado un notable nivel de madurez en cuanto a metodologías, metamodelos y aspectos referidos a la calidad de software.

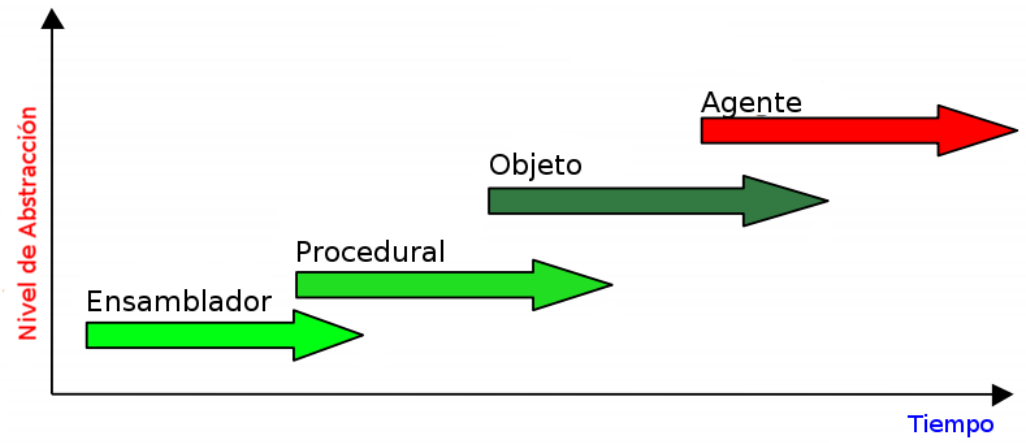

FIGURA 2.1: Niveles de abstracción según los diferentes paradigmas

Este paradigma, desde sus inicios, evolucionó desde un enfoque mono-agente, compuesto de un agente y con el foco puesto en su arquitectura interna, al de un sistema multiagentes donde el foco ahora está puesto en la interacción entre dichos agentes. Así, un sistema multiagentes está caracterizado por múltiples agentes inteligentes que interactúan entre sí para alcanzar objetivos que pueden ser compartidos o no entre ellos. Decimos que la realización de la tarea global, u objetivo que deben alcanzar, reposa sobre el conjunto de los agentes y no en uno en particular. Esto se debe a que la solución emerge como producto de las interacciones que tienen lugar en un ambiente. Además es posible definir comportamientos simples en los agentes pero que en grupos, y como consecuencia de las interacciones, estos son capaces de exhibir comportamientos mucho más complejos. Con los SMA se amplió el campo de investigación incluyendo tópicos como: comunicación, coordinación y organización.

2.2. Teoría Organizacional

Diferentes enfoques han sido propuestos para los sistemas multiagentes, entre ellos, el enfoque organizacional ha ganado especial importancia gracias a su forma más natural de conceptualizar sistemas. Inspirado en la metáfora social, es usada tanto en metodologías (GAIA [9], MESSAGE [10], ASPECS [1]) como metamodelos (AGR [11], MOCA [12], CRIO [13]). Este novedoso enfoque es una evolución en la forma de modelar, que parte de una visión donde el sistema estaba centrado principalmente en el agente y sus aspectos individuales, hacia una visión en donde el sistema es considerado como una organización en la cual los agentes forman grupos y jerarquías, además de seguir reglas y comportamientos específicos. Conceptos como "Organización", "Grupo","Comunidad", "Roles", "Protocolos" son términos comunes en este tipo de teoría.

El enfoque antes mencionado permite descomponer los problemas en partes más pequeñas además de proveer el contexto de interacción entre los agentes de cada una de estas unidades. Ferber [11] define que dos niveles son posibles: *organizacional* y *agente*. El nivel organizacional o social (donde se resuelve el "que") es posible observar los aspectos estructurales y dinámicos de una organización SMA. Este nivel describe las relaciones y los patrones de actividad que deberían ocurrir en el nivel de agentes. Por otro lado, el nivel de agentes (el "como" se va a resolver) describe el comportamiento propio del agente detallando su arquitectura interna, estados mentales, creencias, deseos, intenciones, metas y si es reactivo o intencional.

Adoptar el enfoque organizacional permite al diseñador tratar los problemas a través de dos estrategias posibles: la descomposición vertical y la horizontal. La descomposición vertical permite que el comportamiento que es representativo de la organización sea fragmentado en un conjunto de sub-organizaciones refinando diversos aspectos de la misma. En cambio, la descomposición horizontal modela las interacciones entre las entidades presentes en un mismo nivel de abstracción, las cuales son necesarias para alcanzar los objetivos requeridos [11].

Como indica Ferber en [11], el enfoque organizacional realiza una importante contribución a la Ingeniería de Software Orientada a Agentes en los siguientes puntos:

- **Heterogeneidad en los lenguajes.** Cada grupo es considerado como un espacio de interacción donde es posible encontrar recursos específicos para la comunicación tales como KQML[1] o ACL[2] sin modificar la arquitectura de todo el sistema.

[1] Knowledge Query and Manipulation Language
[2] Agent Communication Language

- **Modularidad.** Las organizaciones pueden ser vistas como módulos que proveen una descripción, a partir de la cual se puede obtener un comportamiento particular de los miembros. Se emplea el enfoque para definir reglas de visibilidad claras que ayudan en el diseño de un sistema multiagentes.
- **Múltiples arquitecturas.** El enfoque organizacional no hace ninguna suposición sobre la arquitectura interna de los agentes, dejando su especificación abierta a un considerable número de modelos e implementaciones diferentes.
- **Seguridad de las aplicaciones.** Si todos los agentes se comunican sin ningún tipo de control externo, esto puede llevar a problemas de seguridad. Ahora, si permitimos que cada grupo controle el acceso a los roles definidos dentro del mismo, se puede alcanzar un nivel de seguridad sin la necesidad de recurrir a un control global centralizado.

Considerando a las organizaciones como moldes que pueden ser usados para definir una solución a un problema, creemos que el enfoque organizacional fomenta modelos reusables tanto en un mismo proyecto como en desarrollos futuros. Se ha demostrado repetidamente que la organización de un sistema tiene un impacto significativo en el rendimiento a corto y largo plazo, y depende de las características de la población de agentes, escenario de metas y el ambiente que lo rodea.

El diseño organizacional empleado en un sistema agentes puede tener un impacto significativo con efectos cuantitativos en sus características de performance. En [14] se realizó un estudio de los paradigmas organizacionales más usados en los SMA. Esto incluye: jerarquías, holarquías, coaliciones, equipos, congregación, sociedad, federación, mercado, matricial y compuestos. La conclusión alcanzada es que ningún enfoque organizacional es necesariamente mejor que el resto para cualquier situación. La selección hecha por el diseñador va a estar determinada a partir de las necesidades impuestas por los objetivos del sistema, los recursos que estén al alcance y el ambiente en el que los participantes van a existir. En otras palabras, un enfoque organizacional que pueda ser descripto como un enfoque para todos los dominios aún no debe ser desarrollado (al menos de acuerdo al relevamiento realizado por la literatura especializada). Un SMA puede ser estáticamente organizado (en tiempo de diseño) usando cualquiera de los presentados en la tabla, eligiendo uno en particular o realizando combinaciones de los mismos. Cabe mencionar que el intercambio dinámico desde un estilo de organización a otro es posible pero conlleva un costo significativo de implementación. Este último caso se denomina reorganización dinámica, la cual es actualmente un área de investigación muy activa dentro de la disciplina de los SMA.

CUADRO 2.1: Paradigmas Organizacionales: Ventajas y Desventajas.

Paradigma	Características Claves	Beneficios	Inconvenientes
Jerárquicas	Descomposición	Mapeo a dominios comunes, buen manejo de escalas.	Potencialmente frágil, puede conducir a cuellos de botellas o atrasos.
Holarquía	Descomposición con autonomía	Explota la autonomía de las unidades funcionales.	Debe organizar holones, falta de rendimiento predecible.
Coalición	Dinámico, dirigido por metas	Aprovecha la fuerza en número.	Beneficios a corto plazo pueden no ser mayores que los costos de construcción de la organización.
Equipo	Cohesión a nivel de grupo.	Abordar grandes problemas, centrado en tareas.	Incrementa la comunicación.
Congregación	Larga duración; dirigido por utilidad.	Facilita la detección de agentes.	El ingreso puede ser excesivamente restrictivo.
Sociedad	Sistema abierto	Servicios públicos, convenciones bien definidas.	Potencialmente compleja, los agentes pueden requerir capacidades adicionales relacionadas a la sociedad.
Federación	Agentes intermedios	Búsqueda de parejas, brokering, servicios de traducción, facilita un conjunto dinámico de agente.	Los intermediarios se convierten en cuellos de botella.
Mercado	Competencia a través de precios	Bueno en la asignación, aumento de la utilidad a través de la centralización, aumento de la equidad a través de licitación.	Posibilidad de colusión, de comportamiento malicioso, complejidad de la decisión de asignación puede ser alta
Matricial	Múltiples administradores	Compartición de recursos, múltiples agentes influenciados	Posibilidad de conflictos, necesidad de una mayor sofisticación en los agente.
Compuesto	Organizaciones Concurrentes	Aprovecha los beneficios de varios estilos de organizacionales	Aumento de la sofisticación, inconvenientes en varios estilos organizacionales

2.3. El proceso de desarrollo ASPECS

Una metodología es una secuencia sistematizada de actividades para la obtención de un producto. Brindan un marco de trabajo para estructurar, planificar y controlar el proceso de desarrollo de software indicando los recursos y los actores involucrados en dicha actividad.

A lo largo de los últimos 40 años, el desarrollo de metodologías fue y es en la actualidad uno de los tópicos más activos en la campo Ingeniería de Software en general. Sobre todo en el paradigma Orientado a Objetos que ha alcanzado un gran nivel de madurez debido, entre otras razones, a los años de experiencia acumulados tanto en el campo académico como industrial (RUP es la metodología de facto en la actualidad). A su vez, la teoría agentes representa una tecnología relativamente nueva, útil para abordar sistemas complejos por lo que las metodologías propuestas son también recientes pero diversas y de gran crecimiento. Esta diversidad es probablemente atribuida a que no existen definiciones acabadas de los diversos conceptos que conforman la teoría y a la

enorme cantidad de dominios donde el paradigma es aplicado. Dentro de la literatura organizacional existen un gran número de metodologías destacándose entre ellas GAIA [15], TROPOS [16], O-MaSE [17], Ingenieras [18], PASSI [19], Prometheus [20], entre otros.

Para el desarrollo de esta tesis se adoptó la metodología ASPECS [1] junto al ecosistema creado para el mismo. ASPECS fue pensado fundamentalmente para modelar sistemas complejos. Es un proceso de ingeniería de software para el análisis y diseño de soluciones multiagentes jerárquicos comenzando desde los análisis de requerimientos, el diseño, la generación de código hasta la implementación del sistema en una plataforma específica.

Como la mayoría de las metodologías; ASPECS está compuesta de 3 fases, a saber: (i) *Requerimientos del Sistema*, permite definir los requerimientos del sistema e identificar las organizaciones; (ii) *Sociedad de Agencia*, define la comunicación entre roles, agentes y la arquitectura de holones[3]; y por último, (iii) *Implementación y Despliegue*, que implementa una solución usando conceptos dependientes de una plataforma específica y desplegando la solución en dicha plataforma.

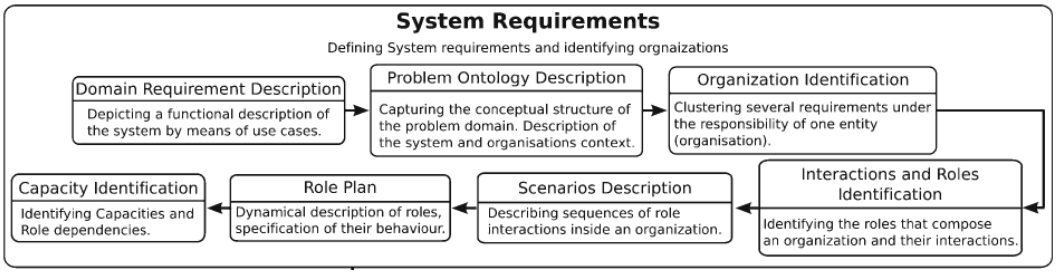

FIGURA 2.2: Fase *Requerimientos del Sistema* y las actividades que la componen (Fragmento extraído de [1])

Las primeras actividades del proceso son las más relevantes para el actual trabajo: Inicialmente los objetivos de la aplicación son identificados y descriptos en términos de *Casos de Uso* (la primera actividad de ASPECS se denomina *Descripción de Requerimientos del Dominio*), para que después toda la información acerca del problema y su contexto sea conceptualizada en el *Diagrama de Ontología del Problema*.

Dicha ontología debe proveer las primeras definiciones del contexto de aplicación y el vocabulario específico del dominio. El objetivo es profundizar el entendimiento del problema y completar el análisis de requerimientos con la introducción de conceptos que componen el dominio del problema y sus relaciones.

La tercera actividad definida en la primera fase del proceso es *Identificación de la Organización* (Organization Identification, OI). Tiene por objetivo relacionar cada requerimiento a un comportamiento global plasmado en una organización. El contexto que

[3]Estructura autosimilar que permite describir agentes que están compuestos por otros agentes.

provee la organización es definido de acuerdo a la odontología del problema generado en la actividad anterior.

Identificación de Roles e Interacciones (Interaction and Role Identification, IRI): tiene como objetivo descomponer el comportamiento global asignado a la organización en comportamientos de menor tamaño que interactúan entre sí. Cada uno de estos fragmentos serán representados por un rol que colaboran en el cumplimiento de los objetivos de la organización a la cual pertenece.

Finalizada la actividad descripta para IRI, se procede a realizar las tareas relacionadas a la *Descripción de los Escenarios* (Scenarios Description). El objetivo de la misma es describir la secuencia de interacción entre los roles involucrados en cada escenario. Es posible asignar una organización y un conjunto de comportamientos interactuantes (adoptadas por los roles involucrados) a cada requerimiento. Los escenarios son dibujados con la forma de diagramas de secuencia de UML y los roles participantes son diagramados como objetos-roles.

Como se ha mencionado anteriormente, un rol representa un comportamiento que contribuye a alcanzar los objetivos de la organización a la cual pertenece. La actividad *Role Plan* tiene por objetivo concebir, para cada rol, un plan que le permita alcanzar los objetivos de la parte de los requerimientos organizacionales que le han sido delegados. En este contexto, el rol describe como una meta (goal) puede ser alcanzada; es una descripción de como combinar interacciones, eventos externos y tareas del role, de manera de cumplir con los requerimientos. Un *Rol Task* es la especificación de un comportamiento parametrizado en la forma de una secuencia coordinada de unidades subordinadas. La primera actividad es detallar las responsabilidades asignadas. Se debe realizar un Diagrama de Actividad donde cada *swinlane* representa a cada rol.

La última actividad de la fase en cuestión es la de *Identificación de las Capacidades* (Capacity Identification, CI). El principal objetivo es la definición de comportamientos de roles genéricos identificando cuales son las competencias necesarias que debe tener el agente para poder tomarlo ("know-how"). A su vez, permite distinguirlo de todos los comportamientos de los cuales puede depender de las propiedades internas y los datos de la entidad el cual jugará el rol. Es útil decir que la capacidad es una descripción de lo que una organización (y por lo tanto uno de sus roles componentes) es capaz de hacer sin ningún tipo de especificaciones de como realizarlo (se pueden adoptar diferentes estrategias). La realización de la capacidad -o implementación de una- es pertinente del dominio de agencia.

La metodología ASPECS es, en la actualidad, considerada como una de las más completas del enfoque organizacional [21]. En la siguiente sección se realizará una presentación más detallada del metamodelo CRIO y sus principales características.

2.4. Metamodelo Organizacional CRIO

Desde el comienzo de la ciencia de la computación, la necesidad de una gestión adecuada de los conceptos relacionados con el dominio de la aplicación en desarrollo se incrementa con la complejidad del sistema. Un enfoque prometedor es el armado de un metamodelo específico.

Un metamodelo es una formalización de los conceptos que se van a utilizar para construir modelos. En otras palabras, especifica una sintaxis abstracta que define un conjunto de conceptos, su significado, los atributos que lo conforman y cómo estos se relacionan entre sí, como así también las reglas que surgen del metamodelo y sirven para combinar conceptos que permiten construir un modelo parcial o completo [22].

Cuadro 2.2: Metamodelos organizacionales y sus propósitos.

Metamodelo	Descripción
AGR [12]	*(Agent-Group-Role)* Es un metamodelo muy conciso y minimalista que junto a un conjunto de notaciosn y una metodología ayudan a construir un SMA.
AUML [23]	*(Agent Unified Modeling Language)*. Es una extensión de UML para modelar SMA incluyendo los conceptos de recursos, ambientes, unidades de organización y servicios. No es normativo.
MOISE+ [24]	Es un metamodelo que posee tres elementos dimensionales caracterizados por la dimensión estructural, funcional y normativa.
ISLANDER [25]	Provee un framework formal para instituciones y ha probado ser bien adaptado para modelar aplicaciones prácticas. Ve a las instituciones basadas en agentes como un sistema dialógico donde todas las interacciones dentro de la institución son composición de múltiples actividades dialógicas.
OperA [26]	*(Organizations per Agents)* Combina las especificaciones de las estructuras organizacionales, requerimientos y objetivos permitiendo a los participantes actuar libremente sobre sus propias capacidades y demandas. Es un metamodelo que combina los modelos organizacionales, sociales y de interacciones.
ODML [27]	*(Organizational Design Modeling Language)* Es un metamodelo minimalista que provee los elementos para modelar y evaluar los aspectos estructurales de las organizaciones.
OMNI [28]	*(Organizational Model for Normative Institutions)* Ofrece dimensiones normativas, organizacional y ontológica que describe diferentes caracterizaciones del ambiente. Usa los frameworks de OperA y HarmonIA para la implementación de un SMA.
MACODO [29]	*(Middleware Architecture for COntext-driven Dynamic agent Organizations)* Es un modelo organizacional para organizaciones de agentes dinámicos dirigido por contexto. El modelo define las abstracciones que soporta el desarrollo de aplicaciones para describir reorganizaciones dinámicas.

Dentro ASPECS -descripta en parte en la subsección 2.3- el metamodelo CRIO [13](Figura 2.3) es la base fundamental de dicha metodología. Es un metamodelo basado en los conceptos organizacionales que deriva de la integración y extensión de dos metamodelos previos. El primero es el metamodelo Role-Interaction-Organization (RIO) [30], que fue pensado para el modelado organizacional de sistemas multiagentes. El segundo es un

framework para el modelado holónico de sistemas [31]. El metamodelo re-define muchos de los conceptos previamente definidos e introduce el concepto de capacidad [32].

Además, CRIO sigue los principios del enfoque Desarrollo Dirigido por Modelos (Model Driven Development, MDD). Este tipo de método es ampliamente utilizado en la industria del software, situación que se evidenció con la adopción en 2003 por parte de la Object Management Group (OMG)[4] de los estándares de Arquitectura Dirigida por Modelos (Model Driven Architecture, MDA) sumado al creciente número de herramientas basadas en los principios detrás de este enfoque. El enfoque MDD pone al modelo en el corazón del proceso de diseño de software. Algunos de sus principios son: (i) la posibilidad de especificar el sistema objetivo independientemente de la plataforma de implementación. (ii) especificar la plataforma de implementación y determinar una plataforma específica para el sistema; y finalmente (iii) transformar la especificación del sistema en una especificación compatible con la plataforma seleccionada.

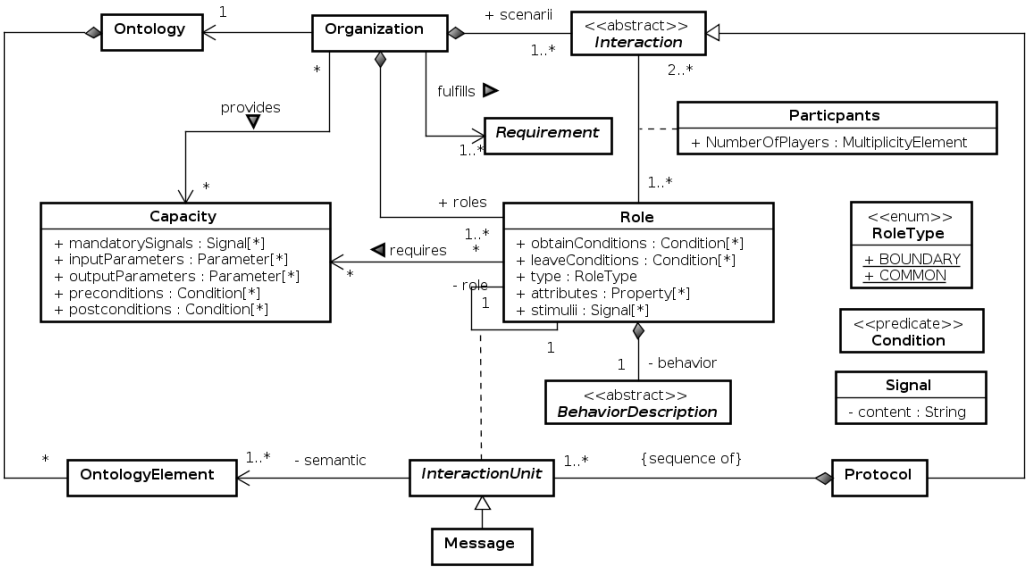

FIGURA 2.3: Metamodelo CRIO (Teórico).

El nombre que adopta el metamodelo CRIO resulta del acrónimo formado por sus cuatro conceptos principales:

Una *Capacidad* es la descripción de un know-how/servicio. En otras palabras, es la especificación de una transformación de una parte de un sistema o de su ambiente. Es una abstracción de alto nivel que promueve la reusabilidad y modularidad y en este sentido puede ser considerado como un componente básico de diseño. Además, el concepto de capacidad permite la definición de un rol sin hacer ninguna suposición de la arquitectura interna de este.

[4] http://www.omg.org

Un *Rol* es definido como un comportamiento esperado (un conjunto de tareas del rol ordenados por un plan) y un conjunto de derechos y obligaciones dentro del contexto de la organización. El objetivo de cada rol es contribuir en alcanzar los requerimientos de la organización dentro del cual está definido.

Una *Interacción* es una secuencia de eventos intercambiados entre roles (una especificación de alguna ocurrencia que puede potencialmente disparar efectos en el sistema) o entre roles y entidades fuera del sistema.

Una *Organización* se define como una colección de roles y sus interacciones dentro de un determinado contexto. Dicho contexto consiste en Conocimiento Compartido (Shared Knowledge) y reglas/normas sociales, "social feelings", etc. y es definido de acuerdo con la ontología. El objetivo de una organización es cumplir con los requerimientos establecidos. Puede ser vista como una herramienta para descomponer un sistema y es estructurada como un agregado de diferentes particiones disjuntas, siendo que cada organización puede estar compuesta por múltiples roles y que a su vez puede descomponerse en suborganizaciones.

Por último, el metamodelo posee una base formal denominada OZS. Este lenguaje formal propuesto por [33] está compuesto de dos formalismos. Uno de ellos es Object-Z [34] el cual extiende Z, y el otro son los "Statecharts" de Harel [35]. El primero sirve para describir las estructuras de datos y funciones mientras que el segundo captura los aspectos de comportamientos y reactivos. La combinación de estos elementos es ideal para modelar sistemas multiagentes, dado que permite la especificación formal del modelo y la verificación de las propiedades definidas para los conceptos.

2.5. Herramienta de modelado

Posiblemente los elementos más importantes que promueven la adopción del enfoque organizacional en las áreas académicas e industriales son la asistencia en el modelado y validación de modelos, entre otros. En este contexto, las herramientas del tipo Computer Aided Software Engineering (CASE de ahora en más) han cobrado notoriedad. Una CASE es un conjunto de herramientas informáticas que asisten al diseñador en algunas de las actividades relacionadas con el desarrollo de un sistema (requerimientos, análisis, diseño, codificación y pruebas). Estas herramientas permiten transmitir rápidamente cuáles son las intenciones de los desarrolladores mediante una combinación de notaciones gráficas y/o textuales que representa algún lenguaje específico compartido por el equipo de desarrollo. Además, incrementan la productividad de dichos equipos reduciendo tiempo y costos a su vez que mejora la calidad del software entregado [36].

En el paradigma de los SMA existe un número importante de herramientas de desarrollo. La lista está comprendida por aproximadamente 24 aplicaciones comerciales y 40 proyectos académicos [37]. Estos números reflejan la constante evolución de la teoría agentes. Muchas veces el propósito de estos desarrollos es reforzar las propuestas realizadas por los grupos de investigación, ya sea de una metodología en particular, elementos de la teoría de agencia, arquitecturas o algún lenguaje agentes. Dentro de los SMA centrados en el enfoque organizacional -denominado OCMAS[5] por Ferber [11]- se destacan herramientas tales como agentTool III [38], GAIA4E [39] o PDT [40], por mencionar sólo algunos casos. Ahora bien, para la metodología ASPECS en general, y el metamodelo CRIO en particular, existe una herramienta para el modelado y la validación de modelos denominada Janeiro Studio [3]. La misma es una herramienta CASE multiplataforma de libre distribución, open-source, completamente desarrollada en JAVA que facilita el modelado de sistemas multiagentes basado en el mencionado enfoque.

Para la construcción de Janeiro Studio fue necesario la utilización de una serie de frameworks tales como Rich Client Platform (RCP) [41], Eclipse Modeling Framework (EMF) [42], Graphical Modeling Framework[6]. Todos ellos forman parte de una *suite* perteneciente a la Eclipse Foundation y son ampliamente reconocidos y utilizados por la comunidad para el desarrollo de aplicaciones para distintos dominios.

Con el objetivo de que Janeiro Studio provea un soporte adecuado para la fase *Requerimientos del Sistema* fue necesario la definición e implementación de tres *Ecores* diferentes. Rapidamente, un Ecore permite la definición de un metamodelo de datos estructurados simplificando la programación requerida para la implementación de algún lenguaje. El primero, representado en la Figura 6.4, contiene los elementos necesarios para construir el *Diagrama de Requerimientos del Dominio*. El segundo, ilustrado por la Figura 6.5, permite describir los conceptos y las relaciones entre ellos usando el *Diagrama de Ontología del Problema*, por último, en la Figura 2.6 se ilustra el modelo EMF de CRIO. Dado que el metamodelo teórico es demasiado abstracto para ser implementado directamente fue necesario agregar ciertos elementos que hicieron posible su adaptación informática. En otras palabras, la transformación no es una mapeo uno a uno entre el metamodelo teórico y el implementado por la herramienta. Esto llevó al surgimiento de tres diagramas: *Diagrama Organizacional*, *Diagrama de Comportamiento* y el *Diagrama de Interacción*. Con estos últimos diagramas es posible para el diseñador cubrir tanto los aspectos estructurales -diagrama organizacional- como los dinámicos -comportamiento del rol e interacción entre roles-, respectivamente.

[5] Organizational Centered MultiAgent Systems
[6] Graphical Modeling Framework, www.eclipse.org/gmf/

Más información acerca de los distintos elementos alrededor del desarrollo de Janeiro Studio consulte el Apéndice A.

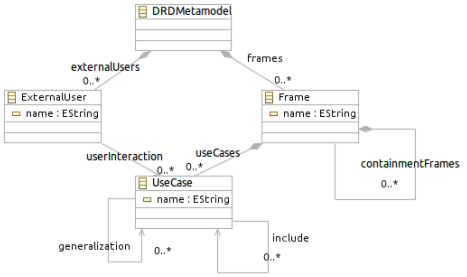

FIGURA 2.4: Metamodelo para el Diagrama de Requerimientos del Dominio

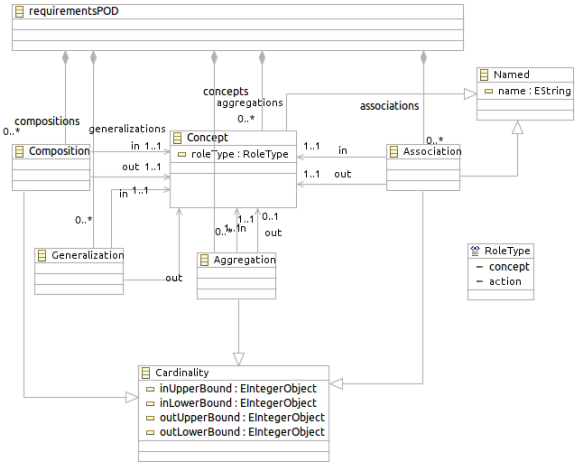

FIGURA 2.5: Metamodelo para el Diagrama de Ontologías del Dominio

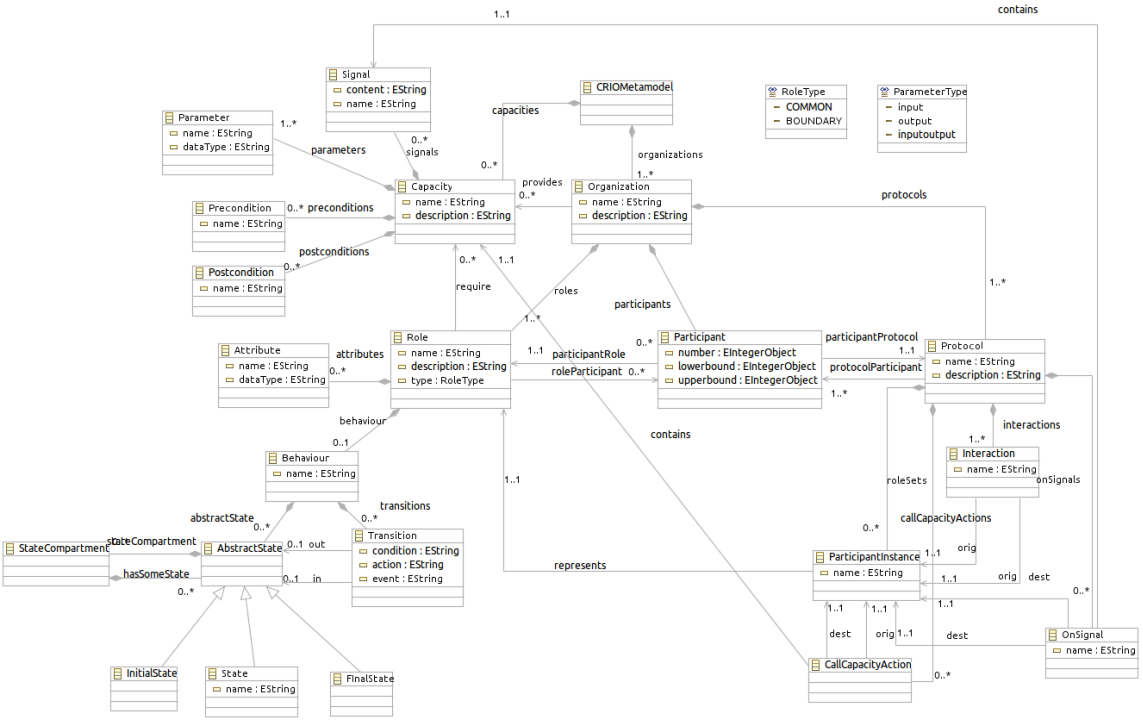

FIGURA 2.6: Adaptación del Metamodelo CRIO usando EMF.

Cuadro 2.3: Ecores y sus diagramas

Metamodelo	Diagramas	Descripción
Requerimientos	*Diagrama de Requerimientos*	Con diagramas similares a los usados en los Casos de Uso de UML, es posible documentar las expectativas y necesidades de los interesados. Estos diagramas permiten capturar los requerimientos funcionales y no funcionales, utilizando un lenguaje específico del dominio provisto por los usuarios.
Ontología	*Diagrama de Ontologías*	Permite identificar, modelado como un diagrama de clases, los conceptos, predicados y acciones relevantes del dominio. La idea principal es conceptualizar los requerimientos descriptos en el Diagrama de Requerimientos del Dominio o en cualquier documento que describa el sistema a desarrollarse.
Dominio del Problema	*Diagrama Organizacional*	Permite representar gráficamente la estructura organizacional. La idea detrás de este diagrama es modelar la organización que represente el comportamiento global. Esta representación es realizada a través de un conjunto de roles presentes en la organización y de la interacción entre ellos. Las capacidades asociadas a los roles definen el comportamiento que es requerido por el agente para que pueda tomar un rol. Los conceptos usados en este diagrama son: *Organización*, *Rol*, *Capacidad*, *Protocolo* entre otros.
	Diagrama de Interacción	Describe la secuencia de intercambio de información entre los roles que tienen lugar en un protocolo. Los conceptos más relevantes en este diagrama son: *Protocolo*, *Rol*, *Interacción*, *CallCapacity*, *Señales*, *Atributos*, etc.
	Diagrama de Comportamiento	Describe el comportamiento de un rol (la dinámica de la instancia). En otras palabras, permite representar los estados y transiciones posibles en un rol. Este diagrama es un perfil del diagrama de transición-estados de UML.

2.6. Conclusiones

En el presente capítulo se introdujo la noción de agente y sistemas multiagentes. A su vez, dentro de la teoría de los SMA, se destaca el creciente interés del enfoque organizacional como una nueva forma de abstraer y modelar sistemas complejos. La ventajas de este novedoso enfoque radica; por lado en que es posible proveer una solución evitando los modelos centrados en la arquitectura interna de los agentes tal como sucede en los enfoques tradicionales. Y por otro, capitalizar los beneficios que puede aportar a la ingeniería de software orientada a agentes, tales como la abstracción, modularidad, seguridad, reusabilidad, independencia de los lenguajes, etc.

También se ha realizado una introducción de la metodología ASPECS, considerada como una de las más completas en la actualidad [21]. Inmediatamente, se realiza una descripción de la primera fase y su metamodelo fundamental, CRIO.

Por último, se introduce el prototipo de la herramienta Janeiro Studio que fue desarrollada con el objetivo de que los diseñadores cuenten con una herramienta CASE para la creación de modelos basados en la notación definida en ASPECS. En la actualidad

Janeiro se encuentra en sus primeras etapas de desarrollo solo cubriendo la primera fase de la metodología. Para cubrir dicha fase fue necesario la implementación de cinco diagramas: DRD, POD, Organizacional, Comportamiento e Interacción entre roles.

Capítulo 3

Introducción a la Calidad de Software

En la actualidad los sistemas de software tienen una alta penetración en la vida de las personas. Esperamos que realicen más tareas por nosotros delegándoles un número creciente de responsabilidades. Es por ello que la calidad de estos sistemas es un tópico que ha ganado importancia en los últimos tiempos. Esto se debe a que los sistemas potencialmente pueden contener una serie de defectos que resultan molestos y/o costosos en términos económicos. Ahora bien, si el sistema considerado resulta ser crítico, tales como los sistemas de navegación de los aviones, control de tráfico aéreo, operadores de plantas nucleares, sistemas de soporte de vida para pacientes en estado crítico, donde un mal funcionamiento puede tener efectos negativos que conlleve incluso a la pérdida de vidas humanas.

Un defecto en cualquier parte del sistema puede traer consecuencias. Mientras que parece improbable que un error en una parte "poco importante" de un sistema puede ser desastroso, en general son la fuente más común de problemas. A medida que los sistemas se van haciendo más rápidos, complejos, y a su vez, ganando mayor autonomía, las fallan catastróficas son cada vez más probables y perjudiciales.

En el diseño de sistemas complejos, los problemas que se presentan a menudo han sido estudiados, revisados y probados en detalle. Como resultado del análisis, la causa más común de los problemas del software son los descuidos o errores involuntarios. También existen errores típicos más simples que son introducidos individualmente por los desarrolladores/ingenieros de software. No obstante, la mayoría de estos errores son detectados en tiempo de compilación y/o testing, y en ciertas ocasiones -debido a que son los mismos programadores los que inyectan defectos- un gran número ellos logran escapar a estos procesos y lamentablemente surgirán durante el uso del producto. El problema

principal es que a menudo los desarrolladores confunden lo simple con fácil. Sienten que sus errores son triviales y que además serán fáciles de encontrar. En ciertas ocasiones se detectan que tales errores son omisiones de puntuaciones, mal nombrado de parámetros, condiciones declaradas incorrectamente, por mencionar solo algunas. Sin embargo, son los causantes de una gran proporción de los problemas en donde la industria del software invierte mucho dinero en el desarrollo de técnicas para su búsqueda y erradicación. Mientras que la mayoría de los problemas triviales tendrán triviales consecuencias, otros pueden causar problemas impredecibles y posiblemente dañinos. Aun así, siempre que se identifique un defecto se deberá evaluar su severidad y el costo de solucionarlo.

Todos estos inconvenientes generaron las condiciones que propiciaron el surgimiento de una disciplina del área de la ingeniería de software denominada *Aseguramiento de Calidad*. Dicha disciplina representa el proceso de verificar cuando un producto de software alcanza la calidad requerida, acordada -en muchos casos- con el cliente. Denominada por la mayoría de las empresas como QA o Quality Assurance; permite elaborar actividades sistemáticas que se necesitan para lograr calidad en el producto y que son aplicables a los artefactos generados a partir del uso de la metodología que se haya adoptado [43]. Resulta claro que esta planificación debe hacerse antes que se inicie el desarrollo del software. El QA puede estar compuesto de las siguientes actividades: evaluaciones en las etapas del desarrollo, auditorias y revisiones, estándares que se aplicarían al proyecto (estudio y selección), mecanismos de medición (métricas), métodos y herramientas de análisis, diseño, programación y prueba, documentación y control de software. Sin duda, este panorama, reviste una gran importancia en el proceso de elaboración del software ya que permite proponer y desarrollar las herramientas necesarias para garantizar la calidad. Por eso estas actividades deben hacerse en la etapa inicial del proceso. Una correcta elección de las actividades para un proyecto determinado, propiciarán que ciertos atributos y características tengan una influencia significativa en la calidad del producto de software.

El concepto de calidad es complejo y todavía objeto de debates, producto de estas discusiones surgieron diversas definiciones. En resumen, la calidad es la aptitud de un producto o servicio para satisfacer las necesidades del usuario. Es el grado en que un software cumple con los requerimientos especificados ya sean funcionales o no funcionales. Alcanzar la calidad deseada y/o establecida es definir cual es el grado en el cuál el sistema de software posee una combinación deseada de dichos atributos [44]. A su vez, el QA es, en este contexto, el proceso de verificar cuando un producto de software alcanza la calidad requerida. No se puede separar la calidad de un producto o servicio de las evaluaciones de las personas que lo estarán usando en algún tiempo y lugar, debido a que la calidad está ligada a un proceso subjetivo.

A comienzos de los '90 el Instituto de Ingenieros Eléctricos y Electrónicos (Institute of Electrical and Electronics Engineers, IEEE) estableció una definición como un intento de estandarizar el significado de calidad de software. La misma dice: "Es el grado en que el software posee una combinación deseada de atributos, claramente definida; si no es así, la estimación de la calidad se deja a la intuición. La calidad de software de un sistema es equivalente a definir una lista de atributos requeridos para ese sistema. Para medir dichos atributos se debe identificar un conjunto apropiado de métricas de software".

Durante aproximadamente una década, la comunidad científica realizó aportes significativos en el campo de la calidad de software. Gran parte de esta comunidad concuerda que la calidad aparece cada vez más como una necesidad estratégica dentro de la organización, pues la calidad permite generar un producto que pueda competir con mayores posibilidades de éxito. En este sentido, la identificación de los atributos que más representan a la calidad para un sistema en un determinado contexto es el eje central del campo en cuestión. Un atributo es una propiedad de un producto que cuando es asociada con la calidad termina conformando aquellos elementos que son tomados en consideración por parte del cliente para aceptar o rechazar un producto. En este sentido, Bass [45] establece -a grandes rasgos- una clasificación de los atributos de calidad divididos en dos categorías: (i) Observables vía ejecución: aquellos atributos que se determinan del comportamiento del sistema en tiempo de ejecución. (ii) No observables vía ejecución: son aquellos atributos que se establecen durante el desarrollo del sistema. Además, los atributos de calidad deben ser medidos para realizar una comparación. Es por esto que la Organización Internacional de Normalización (International Organization for Standardization, ISO) liberó el estándar ISO 9126:2001 con el objetivo de identificar cuáles son los atributos más relevante, unificando todo lo relacionado con la notación y la diversidad de modelos de calidad existentes hasta ese momento. Dicho estándar identifica seis características de calidad (Figura 3.1): *Funcionalidad*, *Mantenibilidad*, *Eficiencia*, *Usabilidad*, *Fiabilidad* y *Portabilidad*.

La *Funcionalidad* es la capacidad del software de cumplir y proveer las funciones para satisfacer las necesidades explícitas e implícitas cuando es utilizado en condiciones específicas. Las sub-características de la Funcionalidad son: Idoneidad, Exactitud, Interoperabilidad y Seguridad.

La *Confiabilidad* es el conjunto de atributos que se refiere a la capacidad del software de mantener su nivel de rendimiento bajo ciertas condiciones especificadas durante un período tiempo, también especificado. Las sub-características de la *Confiabilidad* son: Madurez, Tolerancia a fallas y Facilidad de Recuperación.

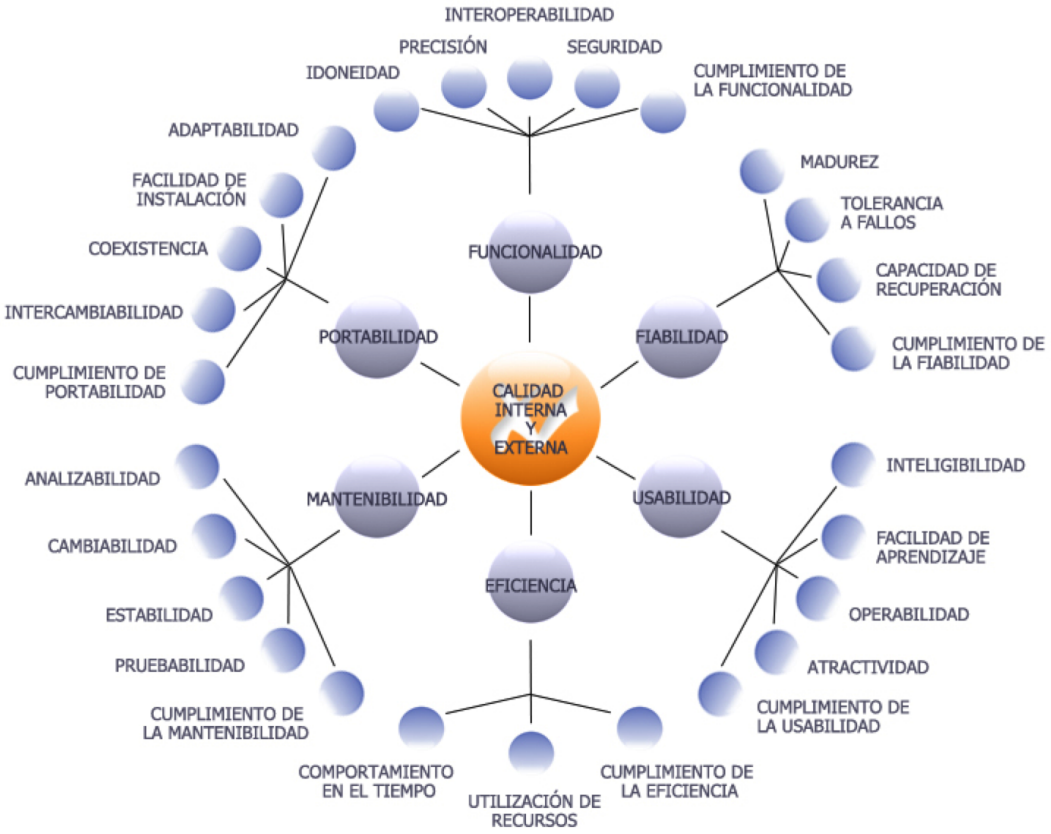

FIGURA 3.1: Estrella de la calidad de software (ISO 9126:2001).

La *Facilidad de Uso* es el conjunto de atributos que se refieren al esfuerzo necesario para usarlo y sobre la valoración individual de tal uso por parte de los usuarios. Las sub-características de la *Facilidad de Uso* son: Comprensibilidad, Facilidad de Aprendizaje, Operatividad, Adaptabilidad al Usuario y Atractivo al Usuario.

La *Eficiencia* es el conjunto de atributos que se refieren a las relaciones entre el nivel de rendimiento del software y la cantidad de recursos utilizados bajo condiciones predefinidas. Las sub-características de la *Eficiencia* son: Comportamiento en el Tiempo y Utilización de los Recursos.

La *Facilidad de Mantenimiento* es la habilidad para identificar y corregir un defecto dentro de un componente de software. En otras palabras, son los atributos que se refieren al esfuerzo necesario para hacer modificaciones específicas (costo de introducir cambios). Las sub-características de la *Facilidad de Mantenimiento* son: Facilidad de análisis, Facilidad de Cambio, Estabilidad y Facilidad de Prueba (testability).

Finalmente, la *Portabilidad* se refiere al conjunto de atributos que están relacionados con la habilidad del software para ser transferido desde un entorno a otro. Las sub-características de la Portabilidad son: Adaptabilidad, Facilidad de Instalación, Coexistencia y Facilidad de Reemplazo.

La calidad de los programas depende de la calidad de los componentes sobre los cuales son construidos. Así, para producir una aplicación de alta calidad, todos los ingenieros de software que desarrollan una o más partes del sistema deben hacer su trabajo con profesionalismo. Esto significa que todos los involucrados en el desarrollo deben personalmente comprometerse con la calidad.

Dado que la calidad de software impacta en los costos de desarrollo, entregas planificadas y satisfacción del usuario; diversas técnicas y/o herramientas fueron propuestas para garantizar que un software cumpla con los requerimientos de calidad establecidos. La mayoría capitalizan experiencias pasadas haciendo uso intensivo del concepto de reúso como los patrones de diseño, micro-patrones y anti-patrones; aunque existen otras como las métricas, buenas prácticas, estándares de codificación, etc.

3.1. Validación y Verificación

La verificación y validación de software son actividades dentro del campo del QA que tienen por objetivo asegurar que el sistema es desarrollado de acuerdo a un proceso de desarrollo y que además cumple con las necesidades de los usuarios. La validación se clasifica en dos tipos: estática y dinámica. Las validaciones estáticas chequean la correctitud del producto de software sin haber ejecutado el sistema o prototipo; mientras que la validación dinámica es realizada durante la ejecución del sistema o prototipo para verificar si el comportamiento que exhibe es el especificado. Por ejemplo, el testing de software es una forma de validación dinámica.

3.1.1. Las dos perspectiva del Aseguramiento de la Calidad (QA)

Para un mejor entendimiento, la verificación está relacionado con el proceso de producir un producto, tratando de dar respuesta a la clásica pregunta: *¿Se está construyendo el producto de una manera correcta?*. La pregunta incluye dos aspectos: i) el proceso correcto y ii) el correcto seguimiento del proceso. Como condición mínima, el *proceso correcto* requiere que los artefactos de un nivel más bajo satisfagan los requerimientos establecidos en los artefactos ubicados en niveles de mayor abstracción. A diferencia de la verificación, la cual está relacionada con la correctitud del proceso, la validación está interesada en la correctitud del producto. La pregunta que trata de responder la validación es: *¿Estamos construyendo el producto correcto?*. Necesitamos tanto la verificación como la validación debido a que cualquiera de ellas en solitario no es suficiente. Por ejemplo, una implementación puede satisfacer la especificación pero la especificación

en sí puede ser incorrecta. Sin embargo, la implementación puede satisfacer la especificación y la especificación ser también correcta pero el código puede ser difícil de entender, probar y mantener. Por lo tanto, un proceso de desarrollo de software debería incluir ambas actividades: verificación y validación. Además, las mismas deben estar presentes en todas las etapas del ciclo de vida de desarrollo y ser partes fundamentales del mismo.

En la actualidad, la verificación y validación son actividades ineludibles e importantes debido a que el software está siendo usado en todos los sectores de la sociedad. Como se ha mencionado con anterioridad, los sistemas en la actualidad son extremadamente grandes, complejos y capaces de procesar millones de transacciones por día en los sectores financieros, ventas, manufactureros, transporte, telecomunicaciones, entre otros. Muchas aplicaciones de software son sistemas embebidos, sistemas de tiempo real o sistemas de misión crítica. Las fallas son financieramente muy costosos, y no aceptables políticamente debido a que pueden incurrir en el retiro de un producto del mercado, daños a la propiedad, daños al cuerpo humano o incluso la pérdida de vidas humanas. Por lo tanto, la verificación, validación y testeo del software se han convertido en un tópico candente tanto en la investigación académica como industrial.

3.1.2. La importancia de adoptar un estándar en una organización

Otro aspecto importante de la calidad del código es la conformación de los estándares de codificación. Los estándares de codificación aseguran que todos en la compañía puedan entender -y trabajar con- el código generado por otros integrantes. Si el estándar no es alcanzado, o sea si el código no es escrito y organizado de acuerdo con los lineamientos de programación, será mucho más difícil para un equipo grande de programadores desarrollar, integrar y mantener un pieza particular de software. Esto cobra importancia en un ambiente donde los desarrolladores están geográficamente distribuidos como es por ejemplo: los proyectos de desarrollo open-source.

Desafortunadamente, cumplir con los estándares de codificación no es una tarea fácil de alcanzar en la práctica. Todos los desarrolladores involucrados en el proyecto debe conocer y apreciar los lineamientos lo suficiente como para construir un software de acuerdo a los mismos. La experiencia muestra que solo publicando un conjunto de lineamientos de programación es suficiente. Si los desarrolladores no entienden las ideas detrás de una regla particular, o se sienten restringidos por ella, o tan solo no cree que esa regla pueda ser útil, estos serán más propensos a ignorar esa regla durante el desarrollo. En otros casos, un conjunto de guías puede ser demasiado extenso que es fácil pasar por alto algunas de ellas durante el desarrollo. A su vez, cuando el proyecto está bajo restricciones de tiempo, estos efectos suelen incrementarse y notarse aún más.

En resumen, la calidad del código global se puede mejorar asegurando que el mismo se ajusta a los estándares de codificación. Este proceso es soportado por herramientas automáticas que verifican el grado de cumplimiento. Ahora bien, si estas herramientas permitieran la definición adicional de elementos para la detección automática de las desviaciones de los estándares adoptados sería posible identificar dichos errores y enmendarlos rápidamente. A continuación se describirá una técnica para la detección de los elementos que degradan la calidad del sistema.

3.2. Importancia de los Code Smells y su Evolución

El término *Code Smells* fue acuñado por Fowler y Beck [46]. Los autores presentaron una lista conformada por 22 *codes smells* con descripciones sobre como identificarlos. Esta novedosa técnica rápidamente fue acepta por la industria del software que además fue objeto de diversos estudios en los años que le siguieron.

Un *code smells* -según la definición propuesta por los autores- es un indicador de una estructura pobre de codificación que no cumple con los estándares impuestos por una organización y que puede tener un impacto negativo en el futuro, particularmente en la etapa de mantenimiento. Entre las múltiples desventajas de los smells, la más importante es que su presencia genera un ambiente de desarrollo lento e inefectivo provocando que se requiera cada vez más esfuerzos para agregar nuevas funcionalidades o en la realización de tareas de mantenimiento correctivo (aumento considerable de los costos). Además, cada smells está asociado a una estrategia de refactoreo (pasos necesarios para erradicar el *smell*), por lo que un conjunto de smells puede ser indicador de la necesidad de iniciar un proceso de refactoring. Refactorizar significa introducir modificaciones en la estructura interna del sistema sin alterar el comportamiento que exhibe haciendo que sea más fácil de entender y extender.

God Class y *Feature Envy* son dos clásicos ejemplos sobre *smells* definidos por Fowler. El término *God Class* se refiere a aquellas clases que tienden a centralizar la inteligencia del sistema. Una instancia de dicho smell es una clase que realiza la mayor parte del trabajo delegando pequeñas tareas a un conjunto triviales de clases. Tiende a presentar un bloque de código enorme con un alto grado de responsabilidad dentro del sistema, el cual puede hacer que sea difícil de entender y mantener. Por otro lado, el término *Feature Envy* indica la presencia de un método en una clase que para realizar su tarea usa principalmente datos y métodos de otra clase.

Es importante mencionar que los autores realizaron un catálogo -sin hacer ninguna categorización- de smells de bajo nivel con descripciones informales, ciertamente abiertos

a interpretaciones dejando a criterio y experiencia de los desarrolladores sobre cuando estamos en presencia de algún smell. Por citar un ejemplo, para el smell *Long Parameter List* en ninguna parte de su especificación establece cual es el número adecuado de parámetros que permita determinar cuando estamos en presencia de dicho smell; quedando a criterio de los desarrolladores tales restricciones. Misma suerte corren todos los smells declarados.

Para un mejor entendimiento de los smells se han realizado esfuerzos por proponer una clasificación. Muchos de estos trabajos están basados en la naturaleza de los mismos: propiedades, estructura, léxico, relaciones entre ellos y/o dominio como así también su cobertura (intra o inter clases). Del conjunto de trabajos, dos son los más significativos. El primero de ellos fue propuesto por Wake [47] en donde distingue a los code smells que ocurren dentro de las clases como aquellos que pueden presentarse entre clases. A partir de esta clasificación se realizan otras distinciones: smells mesurables, smells relacionados a la duplicación de código, smells debido a la lógica condicional, entre otros.

En la Tabla 3.1 se muestra la segunda taxonomía más relevante -y posiblemente la más utilizada- propuesta por Mantyla [48]. En la misma, los smells originales de Fowler son agrupados en seis categorías, quedando fuera sólo dos que no encajan en ninguna de las categorías descriptas (*Library Class* y *Comments*). Para el armado de la tabla y las categorías, el autor lleva adelante un estudio empírico sobre la existencia de una correlación entre smells que permite determinar cómo estos están conectados entre sí. Un trabajo similar fue realizado por Fontana [49] pero buscando relaciones directas o indirectas entre ellos. En el mismo trabajo afirma que identificado un smell existe la posibilidad de que otro esté presente.

Cuadro 3.1: Taxonomía de los Code Smells

Categoría	Descripción	Code Smells
Bloaters	*Representa código que creció de tal manera que no puede ser manejado efectivamente.*	*Long Method, Large Class, Primitive Obsession, Long Parameter List y Data Clumps.*
Objet-Orientation Abusers	*Relacionada con casos donde la solución no explota completamente las posibilidades de diseño OO.*	*Switch Statements, Tamporary Field, Refused Bequest, Alternatives Classes with Different Interfaces y Parrallel Inheritance Hierarchies.*
Change Preventers	*Se refiere a estructuras del código que entorpecen considerablemente las modificaciones del software.*	*Divergent Change y Shotgun Surgery.*
Dispensables	*Representan algo innecesario que debe ser removido del código.*	*Lazy Class, Data Class, Duplicate Code y Speculative Generality.*
Encapsulators	*Representan smells que son aparentemente opuesto, esto significa que la disminución o erradicación de un smell provocará el aumento o aparición de otro.*	*Message Chain y Middle Man.*
Couplers	*Smells que representan un alto grado de acoplamiento, lo cual es una violación a los principios del diseño OO.*	*Feature Envy y Inappropriate Intimacy.*

Existe en la actualidad dos técnicas principales que pueden ser usadas para detectar *code smells*:

- La primera propuesta hecha por Fagan [50], la verificación del código es realizada manualmente. El nombre que recibe es *inspección de código* y tiene asociada ciertas dificultades tales como: propenso a errores, consumidor de tiempo, no repetible y no escalable. Un trabajo similar fue desarrollado por Travassos [51], pero introduciendo un proceso basado en técnicas de lectura e inspección manual para identificar smells. Sobre dicho proceso no hubo ningún intento de automatizarlo, por lo que no es fácilmente escalable en grandes sistemas. También, el proceso sólo cubre la detección manual de smells, no su especificación.
- La segunda técnica está basada en herramientas semiautomáticas o automáticas, y son las más utilizadas en la actualidad. Tal como indica Emden [52], este proceso es delegado para que sea realizado por un sistema informático; sin embargo, no está exento de errores dado que la lista de reglas que debe ser aplicado al sistema de interés nunca es completa. Cada nuevo proyecto requiere una nueva lista de reglas. Además, la confección de la lista de *code smells* es un proceso subjetivo basado en la experiencia y opiniones de los desarrolladores involucrados, sobre todo aquellos de mayor experiencia y/o líderes de proyecto.

La literatura sobre smells es muy rica y amplia por lo que en el resto del capítulo se citarán a los que se han considerado como los trabajos más relevantes. Entre ellos se propusieron estudios de los smells, lenguajes para describir/especificar smells, métodos para la detección, técnicas de detección usando métricas u heurísticas, entre otros.

3.2.1. Code Smells y su impacto en la mantenibilidad

Diversos estudios fueron realizados acerca del impacto que tienen los code smells sobre el mantenimiento, más precisamente sobre el esfuerzo necesario en dicha etapa. Por otro lado, existen trabajos sobre métodos y/o herramientas para la detección automática como así también estudios relacionado al proceso de refactoring.

Algunos smells han recibido especial atención y hasta fueron estudiados de forma aislada del resto. Por ejemplo, *Duplicated Smell* es posiblemente uno de los más estudiados. Las conclusiones van desde un aumento en el índice de confiabilidad de los módulos que poseen dicho smell llegando incluso a ser considerado por algunos autores una situación beneficiosa [53]. En este sentido, es hasta recomendable que no todos los *Duplicated Smells* deban ser sometidos a un proceso de refactoring [54]. Casi las mismas conclusiones

las obtuvo Kim [55] al encontrar que solo el 36 % de los *Duplicated Smells* necesitan ser erradicados.

En cuanto a la estimación del esfuerzo necesario para llevar a cabo una modificación, al menos el 50 % de los métodos que presentan el smell requieren un esfuerzo mayor que aquellos que no [56]. La erradicación de este smells debe hacerse -como todos- después de realizar una evaluación de los riesgos y costos, dado que introducir cambios inconsistentes para su erradicación puede conducir a más fallas que el promedio previo existente en el código [57]. Esta conclusión surgió a partir de la observación de sistemas hechos en los lenguajes de programación Java y C#. Sin embargo, no se ha observado el mismo fenómeno en sistemas heredados realizados en COBOL.

El segundo smell más estudiado es *God Class*. Se han evaluado diversos sistemas concluyendo que los diseños sin la presencia de dicho smell han sido juzgados y medidos como mejores que los diseños para el mismo sistema pero con el smell [58]. En otras palabras, estos diseños son más completos, correctos y consistentes sin el smell [59]. A su vez, este último junto a *Shotgun Surgery* suelen recibir frecuentemente más modificaciones indicando la necesidad de un mayor esfuerzo de mantenimiento que otras clases [60]. Los smells *God Classes* y *God Methods* solos no tienen ningún efecto, pero comparados con código sin la presencia de ambos estos tienen estadísticamente un incremento significativo en el esfuerzo requerido para llevar adelante modificaciones [61].

Para los smells Feature Envy, Data Class y Refuse Bequest; distintos estudios llevados a cabo demostraron que no existe una correlación significativa entre estos smells y las fallas del sistema [62][63]. Caso contrario es el analizado para Shotgun Surgery que fue asociado positivamente con dichas fallas.

También existe un trabajo que trata de determinar cuál es el entendimiento real de los smells en sí y cual resulta ser el smell más difícil de erradicar y la cantidad de pasos de refactoring necesarios [64]. En pocas palabras, la intención es elaborar un indicador del esfuerzo necesario para eliminar cada uno de los smells.

3.2.2. Lenguajes para la definición y/o detección de smells

En el párrafo anterior se analizaron los smells más importantes, su impacto en la etapa de mantenimiento y su relación con fallas ocurridas en los sistemas bajo estudio. A continuación se describirán algunos de los lenguajes más importantes dentro de la literatura que fueron creados con el objetivo de especificar smells o identificarlos dentro de un modelo.

Un estudio realizado por Moha [65] propone un método denominado DECOR. El corazón de DECOR es un lenguaje específico del dominio que describe smells de código y diseño. A su vez, se presentan algoritmos que permiten realizar un análisis de las normas y generar automáticamente los algoritmos de detección. El trabajo se extiende hasta la detección de antipatrones a partir de la identificación de un conjunto de smells, siendo capaz de detectar todos los antipatrones más conocidos: Blob, Functional Decomposition, Spaghetti Code y Swiss Army Knife.

Otro estudio, liderado por Alikacem [66], propone un lenguaje para detectar violaciones a los principios de calidad y smells en los sistemas. El lenguaje permite la especificación de reglas usando métricas, herencias, o relaciones de asociación entre las clases de acuerdo con las expectativas de los ingenieros. A su vez, permite usar técnicas de lógica fuzzy para expresar los umbrales de las condiciones de las reglas. Las reglas son ejecutadas por un motor de inferencia.

Por otro lado, Endem [52] presenta un estudio empírico para la detección automática de smells basados en la herramienta jCosmos. Los smells, según el autor, pueden ser caracterizados e identificados a través de sus aspectos.

3.2.3. Definición de Smells usando métricas

Dentro del área de la Ingeniería de Software, las métricas están destinadas a conocer o estimar el tamaño y/o las características que presenta un sistema. En este sentido existen dos trabajos muy significativos dentro de la disciplina. Por un lado, un estudio llevado a cabo por Marinescu [67] que presenta un enfoque basado en métricas para revelar la existencia de code smells a partir del uso de *estrategias de detección*. Una estrategia de detección es un algoritmo genérico para computar los valores de las métricas en modelos de código fuente y, mediante heurísticas, capturar las desviaciones de los buenos principios de diseño. El algoritmo de detección de cada code smells se construye automáticamente usando solo un conjunto de operadores que actúan sobre un grupo de clases con algunos valores de métricas manualmente especificados (absoluto o relativo), el cual no puede expresar propiedades estructurales o semánticas.

Por el otro lado, Munro [68] propone heurísticas basadas en métricas para detectar code smells. Las detecciones por medio de heurísticas son derivadas desde plantillas similares a las usadas en los patrones de diseño. También realiza un estudio empírico para justificar la elección de métricas y los umbrales/límites que son usados para la detección.

Las propuestas de Marinescu y Munro, ambas presentan ciertas limitaciones debido a que sólo se enfocan en los code smells y no es posible aplicar sus estrategias de diseño

a defectos de alto nivel, tales como los antipatrones. Asimismo, el uso de métricas es insuficiente para detectar defectos con mayor precisión dado que las métricas no pueden expresar propuestas estructurales y semánticas importantes.

3.2.4. Los smells y sus otros impactos

Lo presentado en las subsecciones anteriores sólo se enfocan en los smells en sí y en el desarrollo de técnicas sobre cómo detectarlos y erradicarlo. A continuación se detallará las implicancias de los smells en diferentes cuestiones tales como la estimación del esfuerzo en la mantenibilidad o las relaciones de los smells con ciertos dominios y/o patrones arquitectónicos de sistemas.

Una aproximación fue realizada por Yamashita [69] asegurando que la identificación de smells es útil para una evaluación de los factores de mantenibilidad como pueden ser el entendimiento que se tiene del sistema -estructura, funcionamiento, etc.- y su modificabilidad. También establece que los smells no son los únicos elementos dado que existen ciertos factores que no son reflejados por los code smells y por lo tanto es necesario contar con otras técnicas para evaluarlos. Finalmente, remarca que los smells son una alternativa a las métricas de software dado que siempre están relacionados con estrategias de refactoreo necesarias para eliminar el smell.

En cuanto al trabajo realizado por Guo [70] toma en cuenta características específicas del dominio y establece que la definición de un smells debería ser más precisa para que se pueda adaptar información de dichos dominios. Un ejemplo puntal: el autor realiza observaciones en sistemas donde se aplicó el patrón de arquitectura MVC (Model-View-Controller) determinando que las clases que componen las interfaces gráficas (GUI) suelen tener demasiados miembros públicos y por lo tanto no deberían ser consideradas *God Class*.

También Bertran [71] realizó su aporte, el mismo es una crítica asegurando que las estrategias convencionales no detectan la mayoría de los code smells relevantes para la arquitectura. Manifiesta la necesidad de determinar qué información de la arquitectura puede ayudarnos a detectar smells que contribuyen en la degradación de los sistemas. Además, establece una correlación entre las estructuras de código potencialmente anómalas y el esfuerzo de mantenimiento.

Khomh [72] después de realizar una revisión de los distintos trabajos en la literatura, llegó a la conclusión de que no existe ningún modelo/enfoque que maneje la incertidumbre. De hecho para este autor todo lo realizado hasta el momento para la detección de smells son del tipo booleano, es decir que los algoritmos determinaban si una situación

pertenecía o no a alguna clase de smells. Las definiciones de los smells son débilmente especificadas debido a que los test de calidad constituyen un proceso centrado en las personas y estas requieren de datos del contexto. El enfoque propuesto es rupturista y está basado en *Bayesian Beliefs Networks* (BBN) para especificar y detectar smells. La salida del BBNs es la probabilidad de que la clase sea un smell, lo que permite rankear clases de acuerdo a su probabilidad. Además cuenta con un mecanismo de calibración que le brinda la posibilidad de ajustar el criterio mejorando la detección de los smells basado en el historial. Esta técnica puede mejorar los resultados pasados aprendiendo de las relaciones entre las diferentes entradas y sus efectos combinadas sobre las salidas.

Hasta el momento existe un gran consenso sobre los efectos que tienen los smells en la arquitectura del sistema y en los esfuerzos de mantenibilidad. De hecho, sobre el último punto citamos diversos estudios sobre la correlación entre los code smells y el esfuerzo dedicado a la mantenibilidad (cantidad de tiempo utilizado para finalizar una tarea). Sin embargo, Sjoberg [73] realiza una medición en dicha relación concluyendo que los 12 smells considerados en el estudio tienen un impacto limitado en los esfuerzos de mantenibilidad. El autor recomienda que para reducir dichos esfuerzos hay que centrarse en reducir el tamaño del código -y las prácticas de trabajo que limitan el número de cambios- puede ser más beneficioso que emprender un proceso de refactoring.

3.2.5. Herramientas y Visualización

Además de las técnicas de detección, múltiples herramientas han sido propuestas para encontrar smells y problemas de implementación o errores sintácticos. En el cuadro 3.2, se describen brevemente tales herramientas. Si bien en la misma predominan aquellas que analizan código Java -dado que es el lenguaje más utilizado-, también existen para otros lenguajes más conocidos e incluso para los lenguajes empleados en los sistemas heredados como es el caso de COBOL.

Una cuestión siempre a tener en cuenta en cualquier organización es el costo de las herramientas informáticas que se usan. En este sentido, la mayoría de las listadas en la tabla son de libre accesos; aunque existen también aquellas que se deben pagar por una licencia de uso. Además, podemos encontrar las que se ejecutan de forma aislada donde solo se le debe proveer como entrada la aplicación bajo estudio. Otras en cambio, dependen de un ambiente de desarrollo como es el caso de FXCOP que requiere de Visual Studio o, por último, los desarrollos bajo el concepto de plugins que deben ser integrados al IDE de Eclipse por citar ejemplos.

En cuanto al funcionamiento, en los inicios eran los usuarios los que debían asistir a estas herramientas a través de anotaciones que luego la aplicación levantaba para el

análisis de los diseños. Sin embargo, esta situación evolucionó a aplicaciones que realizan automáticamente toda la tarea, desplegando los resultados en pantalla así el diseñador defina cuál será el curso acción para emprender las correcciones necesarias. En otras palabras, las aplicaciones modernas toman como entrada el código fuente y las analiza de manera integral en búsqueda de antipatrones, code smells, errores de sintaxis, etc.

No tan solo es necesario una buena definición de la lista de smells y los criterios para detectarlos, sino que además es importante la visualización de los mismos para comodidad de aquellos que toman decisiones. Debe pensarse a la visualización como un medio, no como un fin con el objetivo de generar vistas del sistema para un mejor entendimiento. No existen "balas de plata" a la hora de elegir la mejor visualización, dado que la mayoría de estas técnicas quedarán sujetas al tipo de dominio y la complejidad del análisis.

Existen dos tipos de visualizaciones de acuerdo a las necesidades: (i) La visualización del programa, visualización del código y las estructuras de datos tanto estáticas como dinámicas. (ii) La visualización del algoritmo, es una visualización de las abstracciones de alto nivel las cuales describen el software. Muchas de las formas de visualización que podrían utilizarse son: Treemaps, conos euclidianos, árboles hiperbólicos, mapas de distribución, grafos de Rigi, Radar de evolución, principio de la vista polimétrica y por último, en sistemas con un elevado número de componentes lo recomendado es la creación de lo que se denomina metáfora de la ciudad.

A pesar de las numerosas formas de visualizar los smells identificados, esto se complica aún más cuando el proyecto entra en las etapas de mantenimiento y evolución. El comienzo de estas etapas trae aparejado diversos problemas tales como la escalabilidad, recuperación de la información, la decisión de que visualizar; y si el sistema tiene un alto número de componentes, los problemas que surgen son como visualizarlos y las restricciones de tiempo y recursos asignados a las etapas de erradicación de smells. Además de estas complicaciones, otro problema es la extracción de información debido a que muchas de ellas estarán influenciadas por el lenguaje de programación elegido o de los modelos y sus paradigmas.

Un ejemplo de visualización es el trabajo realizado por Emden [52]. Su aplicación, denominada jCosmos, representa a los smells detectados mediante un grafo estructurado denominado grafo de Rigi que usan los ingenieros para comprender un sistema de software (ver Figura 3.2). Toma el código fuente y a las entidades tales como los paquetes, clases y atributos son representados como nodos mientras que las relaciones entre dichos nodos como arcos, por ejemplo: las clases que pertenecen a un paquete o los atributos de una clase.

CUADRO 3.2: Herramientas relacionadas a los *Code Smells*

Categoría	Descripción
LCLint [74]	*Es una herramienta flexible y eficiente que acepta como entrada programas (escritos en ANSI C) y varios niveles de especificación formal, reportando inconsistencia entre un programa y su especificación.*
ASPECT [75]	*Es una técnica de análisis estático basada en especificaciones formales. Puede manejar la mayoría de las características modernas de los lenguajes de programación imperativos. Fue implementado en CLU.*
Extended Static Checker [76]	*Es un verificador destinado a detectar, de forma estática, errores sencillos en los programas. Actúa como un compilador; tomando un programa con anotaciones como entrada y produce mensajes de error como salida (si los hay).*
SMALLLint [77]	*Es una herramienta que analiza código realizado en SmallTalk para detectar alrededor de 60 tipos comunes de bugs, posibles errores de programación o código sin utilizar. Comenzó como una verificador de estilos pero en sus últimos años se agregaron otros tipos de estilos de reglas conservando el nombre.*
FindBugs [78]	*Es una herramienta de análisis estático que detecta instancias de una variedad de patrones de bug en código JAVA. Para los autores, los patrones de bug son modismos del código que pueden ser errores importante.*
SABER [79]	*Es un enfoque de detección automática de errores de codificación de alto impacto en aplicaciones Java de gran tamaño. Estos errores provocan una serie de degradaciones del rendimiento y cortes en ambientes de producción reales.*
ANALYST4J	*Es una herramienta que según los propios autores ayuda a mantener la entropía de un software bajo control. Desarrollada en Eclipse RCP, permite detectar antipatrones y code smells usando métricas.*
PMD [80]	*Una herramienta con más de diez años en su haber. Es un analizador de código fuente que busca defectos de programación comunes tales como variables sin usar, bloques de excepción vacíos, etc. Provee soporte a diversos lenguajes tales como Java, JavaScript, PLSQL,XML etc.*
CPD [80]	*Es una extensión de PMD que permite detectar código duplicado en Java, C, C#, PHP, Ruby, etc.*
CheckStyle [81]	*Es una aplicación que asiste a los programadores a escribir código Java que se adhiere a un estándar de codificación. Ideal para proyectos que quieren hacer cumplir los estándares adoptados. También puede encontrar problemas en el diseño de clases como en el diseño de métodos.*
FXCOP [82]	*Es un plugin integrado a la suite de Visual Studio 2010 que analiza el código informando sobre violaciones de programación o reglas de diseño definidas en los lineamientos de diseños propuesto por Microsoft. También es considerado como una herramienta con fines educativos.*
HAMMURAPI [83]	*Es una herramienta de inspección holística de código open-source para sistemas de software de gran tamaño que produce reportes consolidados de la aplicación entera. Es una aplicación que puede analizar aplicaciones modernas que constan de artefactos desarrollados en diferentes lenguajes.*
CROCOPAT [84]	*Permite realizar diversos análisis estructurales formalizadas como consultas relacionales. Por ejemplo: la detección de patrones de diseño, patrones de diseño problemáticos, código clonados, código muerto como así también establece la diferencia arquitectónica entre lo programado y lo diseñado.*
BLAST [85]	*Es un software que verifica modelos de programas escritos en C. Dados un programa en C y propiedades de seguridad temporales, BLAST comprueba estáticamente que el programa satisface dichas propiedades.*
MOPS [86]	*Está destinada a la búsqueda de bugs de seguridad en programas escritos en C. y verifica la conformidad de las normas de programación defensiva. Está dirigido a desarrolladores que escriben programas de seguridad crítica y en auditores que verifican la seguridad en código C existente.*
InCode [87]	*Es una nueva generación de herramienta para la evaluación de sistemas hechos en Java, C y C++. Sus características principales son: detección automática de los principales defectos de diseño, exploración visual del diseño, caracterizaciones basado en métricas.*

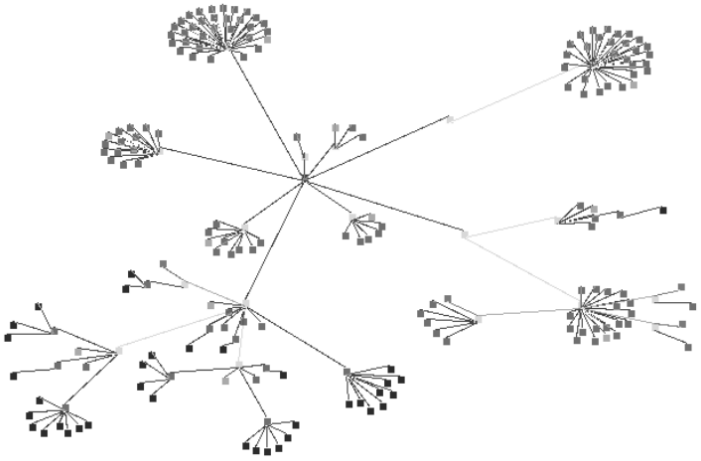

FIGURA 3.2: Grafo de Rigi

Esta representación modificada -el grafo está conformado solo con smells- permite determinar que smells fueron detectados, las partes del sistema afectadas y cuál es la concentración de los mismos. Esta información tiene una importante ventaja estratégica para los directores de proyecto dado que los requerimientos, en su mayoría, afectan a determinados módulos del sistema. A su vez, estos son asignados a una persona o a un pequeño grupo de personas para que sean desarrollados. Contar con esta información nos permitiría determinar cuáles son las violaciones de los estándares de codificación que los desarrolladores poseen y rápidamente reforzar los conocimientos sobre las normas dictaminadas por la empresa.

CUADRO 3.3: Técnicas de Visualización de Smells

Técnica	Ventajas	Desventajas
Treemaps	*100 % uso de pantalla.* *Escalabilidad.*	*Difícil interpretación* *Información solapada.*
Softwarenaut	*Intuitivo.* *Basado en Métricas.* *Visualización Interactiva.*	*Distancia al código fuente.*
Conos Euclideanos	*Mayor info que los gráficos 2D.*	*Falta de profundidad* *Navegación engorrosa.*
Arboles hiperbólicos	*Buen enfoque.* *Dinámico.*	*Propietario.*
Grafos Rigi	*Escabilidad.* *Independencia del Dominio.*	*Falta de semántica del código.*
Metafora de Ciudad	*Posibilidad de representar sistemas complejos.*	

3.3. Code/Design Smells en Sistemas Multiagentes

En cuanto al paradigma orientado a agentes, prácticamente no existen trabajos relacionados a la detección de desviaciones de los estándares en los modelos. Según nuestro conocimiento, el único trabajo fue conducido por Tyriaky [88] y presenta un enfoque para detectar una serie de smells junto a un conjunto de patrones de refactoreo para su erradicación. Su trabajo está basado en una metodología, ágil e incremental junto a un framework de testing denominado AOTDD[1]. Además, los autores proponen un metamodelo de las abstracciones más recurrente en las metodologías SMA, donde una clasificación de tres niveles de smells es realizada (nivel de rol, nivel de tarea y nivel de acción). Así, los smells que fueron identificados para cada uno de estos niveles incluyen a los códigos duplicados, parámetros de acción duplicados (nivel de acción), plan grande, plan incoherente (nivel de plan), rol sobrecargado, y responsabilidad equivocada (nivel de rol), entre otras. Mayor información puede ser encontrada en [88] y [89].

3.4. Conclusiones

La presencia de smells en el código o el diseño pueden tener un severo impacto en la calidad del sistema. Consecuentemente, su detección y corrección/erradicación ha llamado la atención tanto de investigadores como de profesionales quienes han propuesto diversos enfoques para detectar estos defectos en las aplicaciones.

A pesar que la detección de smells alcanzó un buen nivel de madurez, aun así no es posible detectar algunos smells tales como los de alto nivel (mayor abstracción, antipatrones); por ejemplo: Spaguetti Smell. De igual modo, la detección no es 100 % precisa dado que es bastante recurrente caer en falsos positivos como en falsos negativos.

Existen otros inconvenientes que están relacionados a las personas y a la organización. El primero hace referencia al desconocimiento completo de estas técnicas y sus beneficios. Otros conocen la técnica pero no sus ventajas o simplemente no saben cómo introducirlo en la organización. En cambio, los inconvenientes relacionados con la organización muchas veces se deben a presiones presupuestarias y el tiempo asignado al proyecto. Existen situaciones en donde la organización sacrifica márgenes de ganancias en post de posicionarse en un mercado determinado, por lo que la planificación del desarrollo es subestimada. Como consecuencia, la detección de smells y el proceso de refactoreo casi nunca son tenidos en cuenta y por lo que no se les asigna ningún tipo de recursos. Otros inconvenientes están relacionados a los sistemas heredados y a la envergadura que suelen tener, sumado a la alta rotación de personal que existe entre las empresas de tecnologías,

[1] Agent Oriented Test Driven Development

por lo que si una organización intenta capacitar a su personal en estas áreas, muchas veces no puede beneficiarse de ellos.

Capítulo 4

Validación Sintáctica

Uno de los enfoques de desarrollo de sistemas más conocido es el enfoque Desarrollo Dirigidos por Modelos (Model Driven Development, MDD) requiere de la utilización de lenguajes y herramientas para la gestión de modelos con operaciones de soporte tales como la edición, chequeo de consistencia y transformación. El núcleo central y principal características de estas técnicas de gestión es el conjunto de facilidades que brinda para las actividades de navegación y manipulación de instancias de modelos. Por ejemplo, un subconjunto de reglas de OCL[1] puede ser usado para algunas de estas tareas; pero tiene sus limitaciones como lenguaje de propósito general para una variedad de tareas de gestión de modelos.

En MDD dos tecnologías claves son requeridas [90]: (i) Los lenguajes de modelado y metamodelado estándares, los cuales son suficientemente expresivos y ricos para capturar tanto asuntos independientes del dominio como aquellos específicos. (ii) Aspectos de la gestión de modelos: una gestión efectiva requiere de un conjunto de lenguajes y herramientas para la manipulación automatizada de modelos. Un conjunto de herramientas para la gestión de modelos puede incluir los editores de modelos (por ejemplo herramientas para modelar de UML), motores de transformación, motores de validación, entre otros.

Una operación esencial para la gestión de modelos es la *validación de la consistencia*. Esto permite a los diseñadores determinar si la información contenida en uno o más modelos presentan errores y/o contradicciones. La consistencia en los modelos es reconocida como una de las cualidades más importantes solicitadas en gestión de modelos.

Sin embargo, casi toda la literatura sobre calidad de software está enfocada en la calidad del código y los datos, y no en la calidad de los modelos. En este sentido, un número

[1] Object Constraint Language, http://www.omg.org/spec/OCL/

considerable de investigadores adoptan la idea de que el código -producto final- es un reflejo del diseño. La validación de modelos históricamente fue relegada debido a que los modelos muchas veces son usados como un elemento de comunicación. Hablando estrictamente, son utilizados para transmitir una idea o intención del sistema a desarrollarse. Esta situación tiene múltiples orígenes: Por un lado, la envergadura del sistema a desarrollar no es significativo dado que el equipo puede estar utilizando metodologías ágiles y sus principios. Por el otro, existen cuestiones relacionadas a la ejecución del proyecto y sus dificultades, tales como los tiempos de entrega subestimados ya sea por falta de experiencia o asuntos presupuestarios que en muchos casos obliga a los desarrolladores a omitir la generación de ciertos artefactos de diseño. Al fin y al cabo, cualquiera sea la fuente del problema, la realización de los modelos que representa una situación de mundo real siempre es relegada. Sin embargo, para determinadas situaciones en donde el número de componentes del sistema es grande como así también el número de interacciones, es conveniente realizar un modelo que permita entender cabalmente el comportamiento que exhibirá el sistema. Cabe mencionado que en este trabajo cuando se menciona calidad se hace referencia a la calidad de los modelos.

Ahora bien, es posible realizar validaciones de diferentes maneras y abordando diferentes aspectos de un modelo. Consideramos que la validación sintáctica debe de ejecutarse primero antes que las validaciones semánticas (que serán presentadas en el próximo capítulo). El objetivo de la definición de reglas de validación es detectar, de manera temprana, los defectos del modelo para que puedan ser abordados inmediatamente. Realizar esta tarea y que a su vez sea de forma automática trae aparejado un importante número de ventajas. Entre ellas, permitirá reducir drásticamente los costos de desarrollo al mismo tiempo que el producto se acerca a los estándares de calidad deseada por la organización. En consecuencia, garantiza que los desarrolladores siempre cuenten con artefactos del modelo, completos, consistentes, carentes de ambigüedad y correctos. Es por ello que contar con un mecanismo de validación automatizada nos permitirá responder rápidamente a los errores detectados.

La validación sintáctica es crítica para alcanzar las especificaciones del modelo en etapas tempranas en el proceso de desarrollo de software. Este mecanismo habilita a los diseñadores abordar inmediatamente los problemas. A su vez, se realiza este tipo de validación antes de las validaciones semánticas que serán presentadas en el capítulo posterior (5). A continuación se dará introducción a las reglas propuestas formalizadas usando el lenguaje de validación denominado Epsilon Validation Language. Más información sobre EVL remitirse al Apéndice B.

4.1. Presentación de reglas

Tal como se ha mencionado, Janeiro Studio provee un conjunto predefinidas de reglas de validación sintácticas. Por ejemplo, existen reglas que establecen la obligatoriedad de establecer nombres a los conceptos de *Capacity*, *Organization*, *Protocol*, *Role* y *Signal*. Además, la herramienta ofrece la posibilidad de definir nuevas reglas de validación según las necesidades de los diseñadores o del dominio. Para probar la utilidad de las reglas de validación usando EVL aplicados a modelos organizacionales, se presenta a continuación una serie de propuestas:

4.1.1. Verificar existencia de nombre

Como se mencionó anteriormente, el rol representa una parte del comportamiento de la organización. La restricción *HasName* definida en la línea 2 del Listado 4.1 en el contexto del rol, permite validar -a través de su función *isDefined()* (línea 3)- que todos los roles definidos en la organización poseen un nombre.

```
1 context Role {
2   constraint HasName {
3     check : self.name.isDefined()
4     message : 'Role must have a name'
5   }
6 }
```

Listado 4.1: Regla EVL para comprobar que todos los roles poseen nombre

Si bien esta regla es útil para comprobar que se hayan definido nombres para los roles que conforman el modelo, también es posible definir reglas para realizar la misma comprobación en los contextos de Organización, Capacidad, Protocolo, Señales, etc.

4.1.2. Detectar nombres duplicados

Además de la validación presentada en el punto anterior, también resulta importante que en un modelo no existan duplicaciones de nombres dado que puede crear confusión para aquellos que utilizan el modelo. Esta regla es aplicable para los conceptos de capacidad, organización, rol y protocolo. Los dos últimos son componentes de uno de mayor abstracción, la organización; por lo que merecen otro tipo de consideración. Por consiguiente, surgen dos criterios para detectar esta situación: la primera, existen dos conceptos con nombres similares -por ej. roles- dentro de una misma organización lo cual es un error crítico que debe ser resuelto inmediatamente. En cambio, la segunda situación, se refiere a que es posible encontrar nombres iguales en organizaciones distintas lo cual no es en sí un error crítico pero si debe ser advertido para consideración del diseñador.

```
1  context Capacity {
2    constraint DuplicatedName {
3      guard : self.satisfies("HasName")
4      check {
5        var roleList : Set( Capacity );
6        roleList.addAll( self.eContainer().capacities );
7        roleList.remove( self );
8        return not roleList->exists( c | c.name = self.name );
9      }
10     message : 'Duplicated role name: ' + self.name
11   }
12 }
```

LISTADO 4.2: Regla EVL que detecta duplicaciones de nombres en las capacidades

Para ilustrar lo descripto se propone la regla desarrollada en el Listado 4.2 usando el concepto de Capacidad como contexto de la regla. La misma posee una restricción *DuplicatedName* (línea 2) que permite detectar capacidades en el modelo con nombres iguales. Antes de ejecutar *check*, la restricción verifica primero que a las capacidades se les haya asignado un nombre. En el cuerpo de la función se declara una lista donde se recuperan todas las capacidades presentes en el modelo (menos la capacidad sobre el cual está situada la regla) y se verifica con la función *exists()* (línea 8) si hay coincidencia de nombres. Si existe una coincidencia, la función devolverá verdadero por lo que se antepone una negación para que la condición retorne falso y sea violada la restricción.

4.1.3. Verificar la existencia de capacidades aisladas

Una capacidad contribuye en la definición del comportamiento genérico del rol, identificando cuales son las competencias necesarias que debe tener el agente para poder tomar el rol. Un enlace entre el rol y la capacidad indica que la capacidad es requerida por el rol. La regla de validación presentada en el Listado 4.3 permite detectar capacidades que no son enlazadas con ningún rol; en otras palabras, aquellas que no sean requeridas por ningún rol.

```
1  context Capacity {
2    constraint InsolatedCapacity {
3      check {
4        var capacityList : Set( Capacity );
5        for ( r in Role.allInstances ) {
6          capacityList.addAll( r.require );
7        }
8        return  capacityList.includes( self );
9      }
10     message : 'All capacities must be referenced'
11   }
12 }
```

LISTADO 4.3: Regla EVL que valida que todas las capacidades sean referenciadas.

Usando Capacidad como contexto, primero se define un conjunto *capacityList* del tipo *Capacity* (línea 4). A continuación, un lazo *for* recorre todas las instancias que existen

de Rol en el modelo y las agrega a la lista recién definida. Una vez finalizado el lazo, el algoritmo verifica si la capacidad sobre la que estoy situado está presente en la lista *capacityList*. En caso de que la función *includes()* retorne falso, la restricción es violada notificando al usuario que la capacidad no es referenciada o requerida por ningún rol.

4.1.4. Todas las Capacidades deben tener al menos una señal

En el contexto agente, los estímulos generados por un agente son definidos como un conjunto de señales a los cuales el rol reacciona. Los agentes pueden disparar señales que son eventos definidos en la capacidad que permiten comunicar resultados o el estado de la ejecución de las tareas realizadas por la capacidad. Usando *Capacidad* como contexto de la regla, la restricción *HasALeastOneSignal* -definido en la línea 2 del Listado 4.4- recupera a través de la función *size()* el número de señales asociadas a la capacidad. Si el número es igual a cero, la restricción es falsa por lo que el usuario es notificado del error.

```
1 context Capacity {
2   constraint HasALeastOneSignal {
3     check : self.signals.size() > 0
4     message : 'The Capacity ' + self.name + ' must have at least one signal'
5   }
6 }
```

LISTADO 4.4: Regla EVL que verifica que todas las capacidades tengan al menos una señal.

4.1.5. Verificar que existe al menos un rol en una organización

Una vez definido el comportamiento global y representado a través de una organización, el siguiente paso es descomponer ese comportamiento es unidades más pequeñas. Cada uno de estos fragmentos serán los comportamientos exhibidos por los distintos roles. La regla detallada en el Listado 4.5 valida que en las organizaciones definidas en el modelo exista al menos un rol.

```
1 context Organization {
2     constraint HasALeastOneRole {
3     check : self.roles.size() > 0
4     message : 'The Organization ' + self.eClass().name + ' must have at least one role'
5     }
6 }
```

LISTADO 4.5: Regla EVL que comprueba que todas las organizaciones tengan al menos un rol.

Utilizando *Organization* como contexto de la regla, la restricción *HasALeastOneRole* -línea 2- recupera, a través de la función *size()*, el número de roles existentes en la

organización. Si la función es mayor que cero entonces la organización cumple con la restricción; sin embargo, si el número es igual a cero, la restricciones es falsa y por tanto el sistema notifica al usuario del problema.

4.1.6. Comprobar la existencia de al menos un protocolo en una organización

El concepto del protocolo permite al diseñador definir como es la interacción entre los agentes. En otras palabras permitirá detallar la secuencia de paso de información entre los roles participantes.

```
1 context Organization {
2     constraint HasALeastOneProtocol {
3     check : self.protocols.size() > 0
4         message : 'The Organization ' + self.eClass().name + ' must have at least one protocol'
5     }
6 }
```

LISTADO 4.6: Regla EVL que comprueba que todas las organizaciones deben tener al menos un rol

La regla definida en Listado 4.6 permite validar que en las organizaciones se haya definido al menos un protocolo. Usando la organización como contexto, la restricción *HasALeastOneProtocol* (línea 2) permite a la regla navegar a través del atributo *protocols* recuperando el número de instancias presente en dicha organización. Si el número devuelto por la función *size()* es mayor a cero por lo que la restricción evalúa a verdadero; en cambio si es igual a cero, la restricción es falso por lo tanto notifica al usuario sobre la necesidad de corregir este error.

4.1.7. Verificar la existencia de al menos una interacción

Una interacción en un *Diagrama de Interacción* representa un intercambio de información entre dos roles. Esta información puede ser un mensaje o una performativa de FIPA, entre otras posibilidades. La regla presentada en este inciso (Listado 4.7) verifica que exista en los protocolos definidos en la organización al menos una interacción. La restricción *HasALeastOneInteraction* recupera a través de la función *size()* (líneas 2 y 3, respectivamente) el número de instancias de interacciones. Si el número de instancias es igual a cero, el usuario es notificado de que en el protocolo no existen interacciones.

```
1 context Protocol {
2   constraint HasALeastOneInteraction {
3     check : self.interactions.size() > 0
4     message : 'This protocol must have a least one interaction'
5   }
6 }
```

LISTADO 4.7: Reglas EVL que verifica que en los protocolos exista al menos una interacción.

4.1.8. Verificación para auto-mensajes (Señal o llamado a capacidad)

Tanto las señales como los llamados a capacidad son tipos especiales de operaciones que puede realizar un rol. Estos son representados como auto-mensajes que pueden o no llevar parámetros de entrada o salida, sea un llamado a capacidad o una señal, respectivamente. El Listado 4.8 muestra la regla, en la condición *if* -línea 5- la restricción *IsTheSameRole* compara que el origen como el destino sea la misma instancia del rol.

```
1  context OnSignal {
2    constraint IsTheSameRole {
3      check {
4        var band : Boolean = false;
5        if (self.orig = self.dest) {
6            band = true;
7        }
8      return band;
9      }
10       message: 'Origin a destination role must be the same'
11     }
12 }
```

LISTADO 4.8: Regla EVL para verificar auto-mensajes

4.1.9. Detectar parámetros duplicados

Si un agente quiere jugar un rol, el mismo deberá de proveer la implementación de la capacidad que es requerida y así acceder al rol deseado. Dos tipos de llamados a capacidad son posible de realizar: sincrónico, donde la capacidad se ejecuta inmediatamente; y asincrónico, donde la capacidad es puesta en una cola de ejecución de capacidades. De cualquier forma, cuando el agente hace uso de su capacidad realizará un llamado pasándole los parámetros que necesita para poder ejecutarse.

```
1  context Capacity {
2    constraint HasADuplicatedParameter {
3      check {
4        var parameterList : Set(Parameter);
5        parameterList.addAll(self.parameters);
6        var duplicatedParameter : Boolean = false;
7        while ( not parameterList.isEmpty() ) {
8          var parameter : Parameter = parameterList.first();
9          parameterList.remove( parameter );
10         if ( parameterList->exists(n | n.name = parameter.name and
11           n.dataType = parameter.dataType )){
12           duplicatedParameter = true;
13           break;
14         }
15       }
16       return not duplicatedParameter;
17     }
```

```
18      message : 'The Capacity ' + self.name + ' has a duplicated parameter'
19    }
20  }
21
```

LISTADO 4.9: Regla EVL para detectar parámetros duplicados

Estableciendo a Capacidad como contexto, el Listado 4.9 define la restricción *HasADuplicatedParameter*. En la misma se declara una lista -Línea 4- para recuperar y almacenar todos los parámetros definidos en una capacidad e inmediatamente la variable booleana *duplicatedParameter*. Dentro del bucle *while* definido en la línea 7, se toma siempre el primer elemento de la lista a la vez que es eliminado de la misma e iterando hasta que se detecta un coincidencia tanto de nombre del parámetro como su tipo. Si existe una coincidencia, la variable *duplicatedParameter* es seteada en *verdadero* y se sale del bucle para finalmente notificar al usuario que existen parámetros duplicados. La regla notifica al usuario bien encuentre la primera coincidencia, una vez resuelto el problema la regla se ejecutará nuevamente en busca de una nueva coincidencia.

4.2. Conclusiones

La finalidad de este capítulo es presentar un mecanismo para la validación sintáctica de modelos de sistemas multiagentes basados en el enfoque organizacional empleando Janeiro Studio. Detectar un error en etapas tempranas y de forma automática permitirá reducir drásticamente los costos de desarrollo del sistema, al mismo tiempo que se incrementa la calidad del software entregado. Actualmente, este mecanismo está compuesto por un lenguaje de validación denominado EVL integrado a un plugin de Janeiro y por un conjunto de reglas de validación sintáctica predefinidas basadas en la sintaxis del lenguaje elegido. Además, se buscó que estas reglas sean ejecutadas automáticamente por la herramienta de validación sobre los diferentes diagramas que componen el modelo a medida que avanza el modelado.

Se considera que para identificar los defectos de diseño que serán presentadas en el próximo capítulo -identificación de smells organizacionales- es crítico erradicar primero los problemas sintácticos. Además, el plugin propuesto presenta la suficiente flexibilidad como para que los diseñadores o usuarios tengan la posibilidad de definir sus propias reglas con las restricciones que crean conveniente o que surjan del análisis propio del dominio.

Capítulo 5

Organizational Design Smell

5.1. Justificación

En esta tesis se propone un análisis del diseño organizacional de un sistema basado en agentes en pos de encontrar indicadores estructurales y dinámicos de partes mal diseñadas que denominamos "Organization Design Smells" (ODS de ahora en más). Estos indican un pobre o mal diseño que no cumple con los estándares de modelado y/o buenas prácticas. Dicha situación puede tener un impacto negativo en el futuro tanto en la etapa de mantenimiento o en el rendimiento de la ejecución del sistema.

Al igual que los "code smells" de Fowler y Beck, los ODS están destinados a analizar la estructura y relaciones entre los diferentes conceptos de modelado lo más independientemente posible de la meta final (Ejemplo: distribución de tareas, memento, administrador de recursos, etc). Igualmente, las propuestas que serán volcadas en este capítulo han sido definidas lo más abstracta posible dado que la mayoría de los metamodelos organizacionales están compuestos de conceptos comunes o muy similares entre sí. Por el momento se han tenido en cuenta cuatro conceptos básicos:

- Organización/grupo: El concepto de organización o grupo representan cualquier conjunto de agentes que interactúan entre sí en busca de un objetivo. Estos actúan de manera de alcanzar un objetivo específico que puede ser global (común) o local (privada). El concepto de organización adopta diferentes nombre en diferentes metamodelos; por ejemplo: *grupos* en AGR, *organización* en MOCA y CRIO.
- Comportamientos/Roles: Es la representación abstracta de un comportamiento deseado/esperado de un agente en particular dentro de una organización. Recibe el nombre de rol en CRIO como en AGR.

- Interacción/Mensaje: Con el fin de alcanzar sus objetivos, los agentes se comunican intercambiando información específica. Esta información puede ser transmitida a través de diferentes formas: eventos, señales, mensajes estructurados o mensajes ACL como es definido en FIPA.
- Habilidades/Capacidades: El concepto abarca la idea de completar una tarea o alcanzar una meta por un agente. Puede ser encontrado bajo diferentes nombres; tales como know-how, servicios, capacidades, etc.

Estos conceptos expuestos hasta acá, reciben diferentes nombres de acuerdo al metamodelo en el que son usados, pero la idea inicial es la misma. La noción de agente es representada y usada en todos lo metamodelos, pero en este caso el concepto no será sujeto a discusiones dado que nos enfocaremos a la parte organizacional.

5.2. Criterios para la formalización de smells

Proponemos una categorización para las ODS enfocándonos en los problemas relacionados a cada uno de estos bloques de construcción y sus relaciones. Así definimos cuatro criterios a tener en cuenta para la formalización de smells, denominados: *Organization Smells*, *Interaction Smells*, *Skills Behavior Smells* y *Skill Smells*. Cada uno de ellos serán detallados a continuación.

5.2.1. Organization Smells

El enfoque organizacional promueve un nueva forma de descomponer un sistema en grupos en el cual todos los agentes cooperan con el fin de alcanzar un objetivo o tarea. También es posible realizar una descripción de la estructura e interacciones que tienen lugar en un SMA. Cada grupo/organización constituye un contexto común el cual puede consistir de conocimiento compartido, lenguaje común, reglas sociales, etc.

Idealmente, una organización debería tener objetivos bien definidos; si embargo, es posible encontrar un mal diseño en la definición de una organización. Diferentes problemas pueden aparecer asociado a esto:

- Múltiples contexto: Se puede observar que en una misma organización no existe un único contexto. En otras palabras, la organización puede ser representada por diferentes ontologías, mecanismos de comunicación divergentes, etc. Esto significa que, con un detallado análisis, múltiples organizaciones pueden ser identificadas a partir de la observación de una sola.

- Objetivos divergentes/conflictivos: Los agentes o roles que conforman una organización debe enfocarse en objetivos comunes o próximos. Existen casos en el cual objetivos antagónicos pueden ser identificados. Su presencia puede causar problemas, haciendo que la introducción de modificaciones sea una tarea muy complicada.

5.2.2. Interaction Smells

Con el fin de alcanzar una tarea colectiva en una organización, la interacción entre roles es obligatoria. Si bien, todos los miembros son libres de interactuar libremente entre sí, existen patrones de comunicación los cuales representan las interacciones más frecuentes entre ellos. Las interacciones son una secuencia de eventos o acciones cuyo consecuencia afectan los comportamientos de los roles. El contexto de interacción es provista por la organización.

El segundo criterio se denomina *Interaction Smells* y está enfocada en todo lo relacionado con el intercambio de información entre agentes.

- Problemas de rendimiento: Es una organización que tiene un alto intercambio de mensajes entre agentes, llevando a un degradación global del sistema.
- Cuello de botella: Si la capacidad de un agente para procesar los mensajes es menor que el número de mensajes que recibe, esto puede conducir a un problema de cuello de botella (Bottleneck). Tal situación provoca que los demás agentes deban esperar por los resultados del rol.
- Complejidad del mensaje: es un mensaje excesivamente largo que puede afectar la performance de la implementación.

5.2.3. Behavior Skills Smells

Como se ha mencionado anteriormente, el rol es un comportamiento esperado cuyo objetivo es contribuir en alcanzar las metas organizacionales a la cual pertenece. Tienen asociado un conjunto de habilidades/capacidades que definen el comportamiento exhibido. Podemos ver a estas habilidades/capacidades como una noción de las competencias que un agentes debe tener para poder jugar un rol. Esta categoría a la que hemos definido como *Behavior Skill* representa un conjunto de habilidades, capacidades o competencias que define el comportamiento del rol. Un número elevado de estas habilidades puede traer consecuencias negativas en el modelado, ejemplo de ello son los requerimientos excesivos de los agentes. En otras palabras, un agente con muchas capacidades puede

convertirse en un centralizador de servicios, con la consecuencia de que exista un alto número de interacción con otros agentes que conforman la organización.

5.2.4. Skills smells

El último criterio está relacionado con las "habilidades", servicios o competencias de los agentes. En los modelos puede haber agentes que se le han diseñado habilidades que son muy complejas o difíciles de implementar. También se hace mención del mal uso del concepto de *Skill* como abstracción. En este sentido, existe una tendencia en convertir ciertas funciones en skills o el otro extremo es no distinguir los skills asociados a un agente de aquellos relacionadas con el comportamiento abstracto.

5.3. Organizacion Design Smells

En la Figura 5.1 se presentan las diversas propuestas que fueron formuladas a partir del análisis de los distintos proyectos que están desarrollándose dentro del GITIA, como así también del análisis de ejemplos extraídos de la literatura SMA. En lo que resta del capítulo se realizará una descripción más detallada de cada uno de estos smells. La mayoría sigue una determinada estructura: Descripción detallada (propósito del smell), pasos metodológico para su detección (estrategia), diagramas (organizacional e interacción), regla de validación junto a su explicación según la sintaxis y estructura definido para EVL.

5.3.1. Burocracy Role

Este *Design Smells* describe la siguiente situación: Un rol es receptor de un mensaje que no es consumido ni por el propio rol ni por ninguna de las capacidades que tiene asociado. Tal situación puede ocasionar que el verdadero destinatario del mensaje deba esperar por el mismo bloqueándose en algún estado. Esto tiene un efecto negativo en la performance del sistema, no solo porque introduce un latencia innecesaria, en el cual el receptor del mensaje debe esperar, sino que además se le delega al rol la responsabilidad del reenvío del mensaje al destinatario apropiado.

En las Figuras 5.1 y 5.2 se puede observar una organización genérica donde el *RoleB* recibe los mensajes *msg1* y *msg2*. A su vez, este rol llama a su capacidad *Capacity2* que toma como parámetro sólo el mensaje *msg2*. Una vez finalizada la ejecución de la capacidad, la misma envía una señal al rol indicando la finalización de la tarea.

Cuadro 5.1: Lista de los smells de diseño organizacionales.

Nombre del Smell	Descripción
Burocracy Role	Representa un rol que recibe mensajes que no consume. El destinatario de los mensajes debe esperar.
Promiscuos Role	Es un rol que envia el mismo mensaje a diferentes roles. Es un rol que conoce a muchos otros roles. Sobrecarga en el diagrama de comportamiento del rol.
Bottleneck Situation	Identificar un rol con una capacidad que requiere como parámetros de entrada diferentes mensajes de diferentes role. Estos mensajes deben estar presentes al mismo tiempo para evitar un potencial bloqueo del rol.
Selfish Role	Es un rol que requiere un alto número de capacidades resultando en una sobrecarga de la representación su comportamiento (centralizador de servicios).
Different context	Dentro de las organizaciones es posible identificar subgrupos de roles con contextos totalmente diferentes entre sí.
Capacity Abuse	Representa el uso excesivo del concepto de *Capacidad*. La misma puede ser detectada realizando observaciones sobre el número de implementaciones que posee cada capacidad. También es posible detectar este smells observando la complejidad de los servicios que la capacidad ofrece.
Capacity Chain	La misma puede ser detectado cuando los parámetros de salida son los parámetros de entrada de otros. Además, estas capacidades son requeridas nada más que por un rol.
Highly Fragmented Organizations	Consiste en identificar una organización que contiene un número considerable de roles haciendo, en muchos casos, que los comportamientos de cada uno de los roles sean muy sencillos.

Completadas todas las actividades, recién se envía al *RoleC* el mensaje *msg1* necesario para la ejecución de su capacidad: *Capacity1*.

¿Cómo identificarlo?. Pasos Metodológicos.

Paso 1. En el rol receptor, se deben identificar todos los mensajes enviados por los diferentes emisores.

Paso 2. Realizar una comparación de los mensajes recibidos con los parámetros de entrada de las capacidad asociadas al rol.

Paso 3. Identificar los mensajes que no coinciden con ninguno de los parámetros definidos en la capacidad.

```
1 operation ParticipantInstance retrieveInteractionList
2         (participantInstance : ParticipantInstance) : Collection(Interaction) {
3    return Interaction.allInstances->select(i | i.dest = participantInstance);
4 }
5 @cached
```

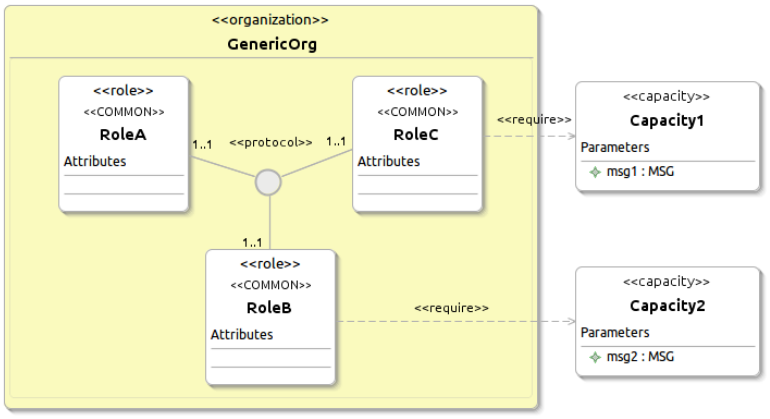

FIGURA 5.1: Burocracy Role - Diagrama Organizacional.

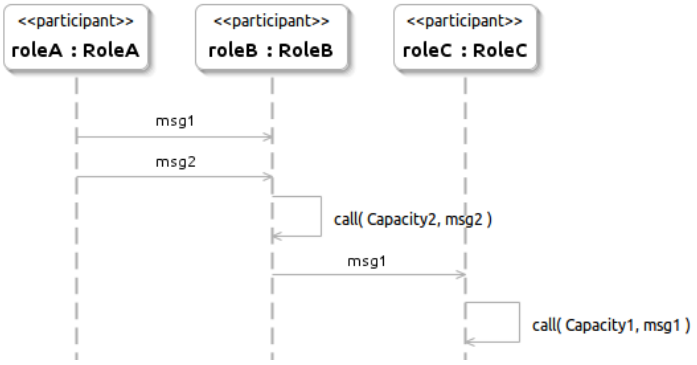

FIGURA 5.2: Burocracy Role - Diagrama de Interacción.

```
6  operation ParticipantInstance retrieveParameterList
7      (participantInstance : ParticipantInstance) : Set(Parameter) {
8     var parameterList : Set(Parameter);
9     for (c in participantInstance.represents.require) {
10       for (p in c.parameters) {
11          parameterList.add(p);
12       }
13    }
14    return parameterList;
15 }
16
17 context ParticipantInstance {
18    critique validation {
19       check {
20          var unconsumedMessageList : Set(Interaction);
21          unconsumedMessageList.addAll(self.retrieveInteractionList(self));
22          for (p in self.retrieveParameterList(self)) {
23             unconsumedMessageList->removeAll(unconsumedMessageList->select(i |i.name = p.name));
24          }
25       return unconsumedMessageList.size() = 0;
26       }
27    message : 'This Role recieve a message that does not consume'
28    }
29 }
```

LISTADO 5.1: Regla EVL para identificar al smell Burocracy Role.

El contexto de la regla (Listado 5.1) es provisto por el concepto de *ParticipantInstance* (Línea 17). Dentro del cuerpo *check* se realiza la comparación entre los mensajes recibidos por el rol y los parámetros de sus capacidades.

La operación *retrieveInteractionList*, definido en la línea 2, tiene por objetivo recuperar todos los mensajes recibidos por el rol sin discriminar remitentes.

Por último, la operación *retrieveParameterList* (línea 7), tiene la misión de recuperar los parámetros definidos en cada una de las capacidades.

5.3.2. Promiscuos Role

Este *design smells* describe un rol que envía el mismo mensaje a diferentes roles que participan en la organización. Esta situación provoca que un rol conozca muchos otros roles, además de la repetitividad de la acción. Llevando a que el diagrama de *Comportamiento* (Statechart), que representa el comportamiento del rol, sea complejo debido a la cantidad de estados y transiciones que posee. Ahora bien, que los roles o capacidades requieran el mismo contenido del mensaje no significa que realicen las mismas acciones y -menos aún- generen los mismo resultados, por lo que no es válido agrupar esas capacidades. El ejemplo ilustrado por las Figuras 5.3 y 5.4 muestran una organización compuesta por un elevado número de roles interactuando entre sí.

¿Cómo identificarlo?. Pasos metodológicos.

Paso 1. Por cada protocolo, recuperar todas las interacciones existentes.

Paso 2. Recorrer las interacciones recuperadas agrupando aquellas que presenten coincidencia (nombre y tipo).

Paso 3. Seleccionar como *Promiscuos Role* al rol que mayor número de un mismo mensaje se corresponda.

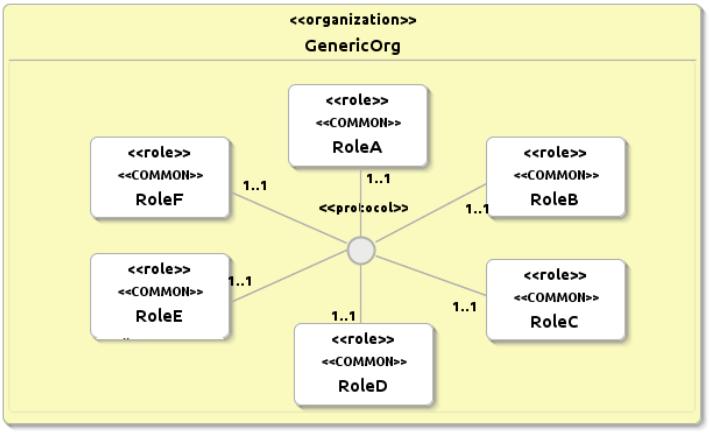

FIGURA 5.3: Promiscous Role - Diagrama Organizacional

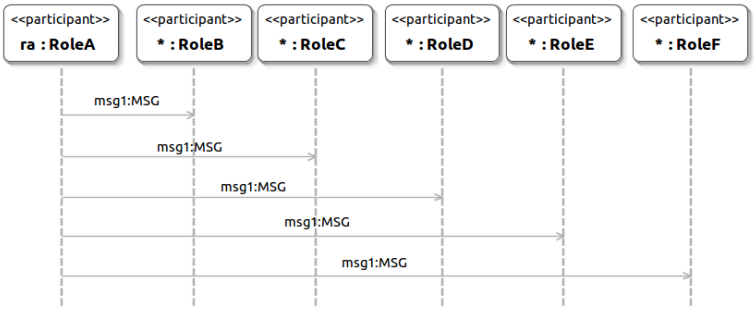

FIGURA 5.4: Promiscous Role - Diagrama de Interacción

```
1  context Protocol {
2    critique forEachProtocol {
3      check {
4        var interactionsList : Set;
5        var promiscuosRole : Role;
6        var number  = 0;
7        interactionsList.addAll(self.interactions);
8        while (not interactionsList.isEmpty()) {
9          var i : Interaction = interactionsList.first();
10         var counter : Integer = interactionsList->
11                         select(n | n.name = i.name and n.orig = i.orig).size();
12         interactionsList.removeAll(interactionsList->
13                         select(n | n.name = i.name and n.orig = i.orig));
14         var pi : ParticipantInstance = i.orig;
15         if ( number < counter) {
16           promiscuosRole = pi.represents;
17           number = counter;
18         }
19       }
20       return (promiscuosRole = null);
21     }
22     message: 'Promiscuos Role: ' + promiscuosRole.name
23   }
24 }
```

LISTADO 5.2: Regla EVL para identificar al smell Promiscous Role.

El contexto de la regla presentada en el Listado 5.2 está determinada por el concepto de Protocolo. Dentro del cuerpo *check* de la restricción *forEachProtocol* (línea 2), se recuperan todas las instancias de *Interaction* que existan en el protocolo y son guardados en la lista *interactionsList*. La secuencia continúa con un bucle *while*, cuya finalidad es contar cuantos mensajes del mismo nombre y tipo existen; y además, si pertenecen a un mismo rol originante. El que mayor número de coincidencias tenga será señalado como *Promiscous Role* (línea 15).

Para la regla presentada se debe hacer una advertencia dado que su detección puede resultar en un falso positivo. El inconveniente tiene su origen en la falta de precisión de la regla dado que no especifica lo que en la teoría de métricas se denominan umbrales y que son los valores a partir de los cuales se puede empezar a considerar la incidencia. Por ejemplo: el número mínimo de interacciones.

5.3.3. Bottleneck Situation

Este *smell*, presentado por las Figuras 5.5 y 5.6, describe un rol que recibe diferentes mensajes de distintos roles emisores. además de que son requeridos por solo una capacidad. Dado la naturaleza asincrónica del intercambio de mensajes entre los agentes puede ocurrir que la capacidad no pueda empezar su ejecución debido a que está bloqueado en algún estado del rol esperando los mensajes restantes. Si bien la teoría no especifica hasta cuantos mensajes pueden encolar los roles, es sabido que estos modelos son implementados en infraestructuras informáticas que si tienen limitaciones en cuanto a hardware, sobre todo si el arribo de mensajes es alto y el bloqueo es de un tiempo considerable.

¿Cómo identificarlo?. Pasos metodológicos.

Paso 1. En el rol receptor, identificar todos los mensajes que provienen de los diferentes roles emisores.

Paso 2. Comparar los mensajes con los parámetros de entrada de las capacidades asociadas a el rol receptor.

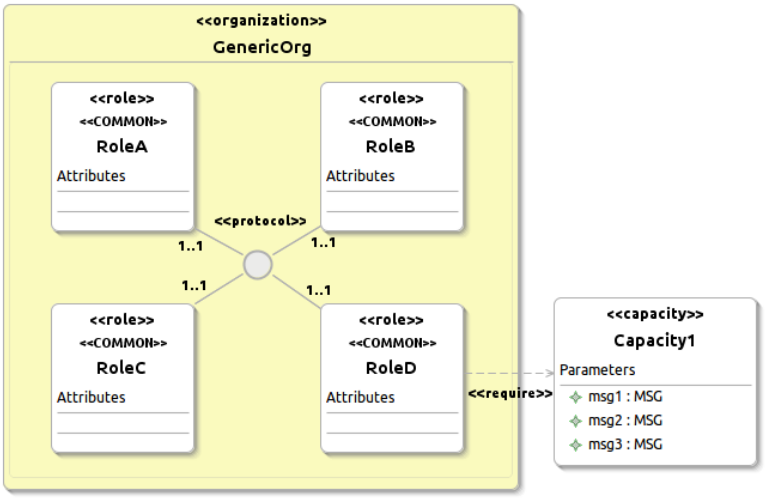

FIGURA 5.5: Bottleneck Situation - Diagrama Organizacional.

```
operation Capacity getParameterNames() : Set(String) {
    var parameterNames : Set(String);
    for ( n in self.parameters.select( m | m.type.asString() = 'input' ) ) {
        parameterNames.add( n.name );
    }
    return parameterNames;
}

operation Protocol getInteractionNames(rs : RoleInstance) : Set(String) {
    var interactionMessages : Set(String);
    for ( n in self.interactions.select( n | n.dest = rs ) ) {
        interactionMessages.add(n.name);
    }
```

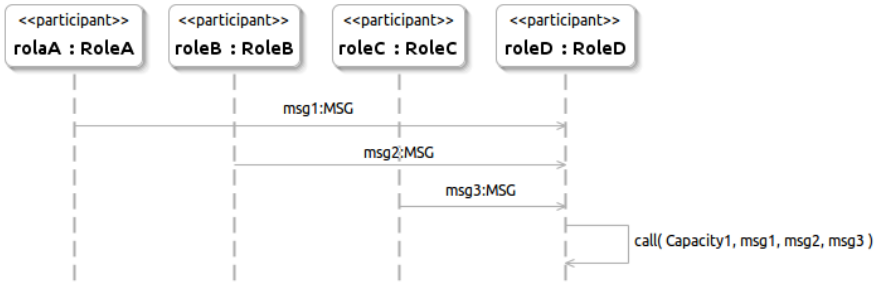

FIGURA 5.6: Bottleneck Situation - Diagrama de Interacción.

```
14      return interactionMessages;
15  }
16
17  operation RoleInstance findMatches() : Boolean {
18      var band : Boolean = false;
19      var interactionNames : Set(String);
20      interactionNames = self.eContainer().getInteractionNames( self );
21      for ( n in self.represents.require) {
22          if ( interactionNames.includesAll( n.getParameterNames() ) ) {
23              band = true;
24              break;
25          }
26      }
27      return band;
28  }
29
30  context Role {
31      critique WaitingForAllMessage {
32          check {
33          var band : Boolean = true;
34          var roleInstance : RoleInstance;
35          for ( p in RoleInstance.allInstances()->select(rs | rs.represents = self ) ) {
36              if ( p.findMatches() ) {
37                  band = false;
38                  roleInstance = p;
39                  break;
40              }
41          }
42          return band;
43      }
44      message : 'Multiple interactions which can be combined in a single one. Protocol: ' +
45                  roleInstance.eContainer().name
46      }
47  }
```

LISTADO 5.3: Regla EVL para identificar al smell Bottleneck Situation

El contexto de esta reglas es provisto por el concepto de rol (línea 30). Dentro del cuerpo *check*, todas las instancias de *roleInstance* correspondiente al rol son ejecutadas por la operación *findmatches*, el cual será explicado más adelante.

La operación *getParameterNames*, definido en la línea 1, tiene el propósito de recuperar una lista con los nombres de los parámetros de la capacidad.

La operación *getInteractionNames* (línea 9) tiene el propósito de retornar una lista con los nombres de las interacciones. Esta lista estará conformada solo por los mensajes que tienen a *roleInstance* como receptor, pasados como parámetros en la operación.

La finalidad de la función definida en la linea 17, *findmatches*, es verificar la existencia de coincidencias entre los nombres de los parámetros de la capacidad y los nombres de los mensajes. Encontrar la primera coincidencia será suficiente para considerar finalizado el lazo *for*. Esto significa que una vez identificada una coincidencia se debe proceder a resolverlo primero y luego a ejecutar la regla de validación nuevamente.

5.3.4. Selfish Role

Este *smell* describe como identificar un rol en el modelo que tiene asociadas demasiadas capacidades -tal como se muestra en las Figuras 5.7 y 5.8- el cual puede representar una anomalía en el modelo. En algunos casos incrementa innecesariamente la complejidad del comportamiento del rol (modelado como un Statechart en CRIO) convirtiéndolo en un centralizador de servicios dentro de la organización. Por las razones mencionadas, la situación ocasiona que muchos de los roles deban comunicarse con él para obtener un servicio. Esto puede incrementar los requerimientos que un agente debe tener para poder jugar el rol (todas las capacidades requeridas por el rol deben estar presentes en el agente). En otras palabras, el rol actúa como un elemento denominado Punto Único de Falla, lo que significa que si esta parte del sistema falla, puede comprometer los objetivos de la organización. Por consiguiente, un punto importante es detectar cuando un rol es un "selfish-role"; basándonos en nuestra experiencia, proponemos un conjunto de criterios subjetivos útiles para detectar un "selfish-role":

- El primer criterio podría ser un cálculo del promedio de las capacidades asociadas a cada rol. Será considerado "selfish-role" aquellas capacidades que superen el promedio calculado.
- Otra propuesta puede ser que el rol posee capacidades que representan conjuntos disjuntos. Puede llegarse a esta conclusión debido que no existe comunicación entre las capacidades disjuntas.
- Capacidades que tienen un alto costo de procesamiento.

¿Cómo identificarlo?. Pasos metodológicos.

Paso 1. Determinar el promedio de capacidades por rol (Otro criterio podría ser usado).

Paso 2. Considerar como *Selfish Role* aquellos roles que superen el promedio.

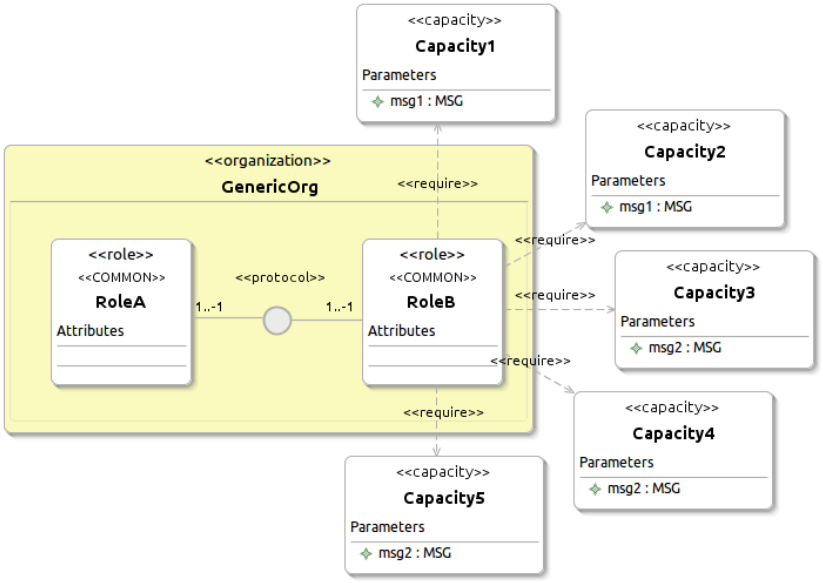

FIGURA 5.7: Selfish Role - Diagrama Organizacional.

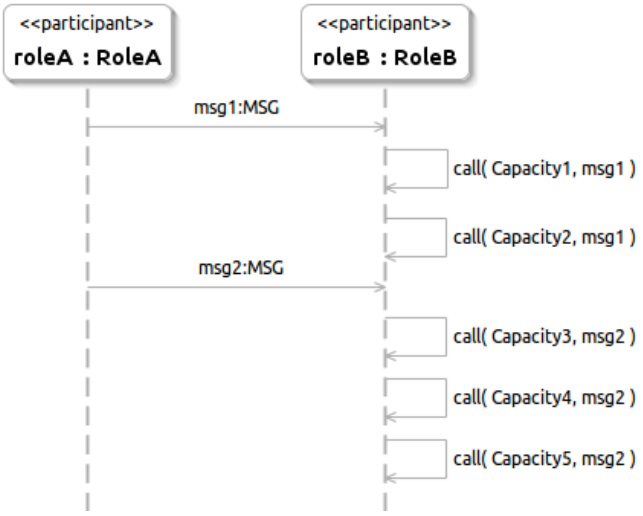

FIGURA 5.8: Selfish Role - Diagrama de Interacción.

```
1  operation Organization numberOfRoles() : Integer {
2      return self.roles.size();
3  }
4
5  operation Role numberOfCapacities() : Integer {
6      return self.require.size();
7  }
8
9  operation Organization averageCapacitiesPerRole() : Real {
10     var nbCapacities : Real = 0.0;
11     for (role in self.roles) {
12         nbCapacities = nbCapacities + role.numberOfCapacities();
13     }
14     return nbCapacities / self.numberOfRoles();
15
16 }
17
18 operation CRIOMetamodel averageCapacitiesPerOrganization() : Real {
19
```

```
20      var average : Real;
21      for (organization in self.organizations){
22          average = average + organization.averageCapacitiesPerRole();
23      }
24      return average / self.organizations.size();
25  }
26
27  context Role {
28      critique selfishRole {
29          check : self.require.size() < self.eContainer().eContainer()
30                  .averageCapacitiesPerOrganization()
31          message : "This role could potentially be a Selfish Role"
32      }
33  }
```

LISTADO 5.4: Regla EVL para identificar al smell Selfish Role

Esta regla de validación tiene definidas cuatro operaciones. Las operaciones "numberOfRole" (línea 1) y "numberOfCapacities" (línea 5) son bastante similares entre sí. La primera, devuelve el número total de roles asociado a una organización; mientras que la segunda, retorna el número total de capacidades de cada rol.

La línea 9 comienza con la definición de la operación *averageCapacitiesPerRole*. El propósito de la misma es calcular, por cada organización, el número de capacidades por rol. El bloque *for* definido entre las líneas 11 y 13 itera a través de todos los roles participante en la organización acumulándolo en la variable *nbCapacities*. Finalmente, la variable *nbCapacities* es promediada con el número de roles calculada para esa organización.

En la línea 18 empieza la definición de la operación *averageCapacitiesPerOrganization*. El principal objetivo es computar el promedio de capacidades presentes en todas las organizaciones involucradas en el modelo.

Finalmente, la línea 27 define el contexto general de la regla el cual a través de la función *check* evalúa si el número de capacidades relacionadas con cada rol es menor que el promedio de capacidades por rol.

5.3.5. Different Context

En ciertas situaciones es posible observar que en una misma organización existe más de un contexto. Se puede llegar a esta conclusión a partir del análisis de los diferentes artefactos detectando -potencialmente -la existencia de diferentes ontologías, mecanismos de comunicación, etc. En otras palabras, es posible detectar subgrupos dentro de una organización con fuertes interacciones entre los roles de un mismo subgrupo. A su vez, existe escasa interacción entre los subgrupos. Lo ideal en una modelización es contar con organizaciones de único contexto dado que provee una clara separación de intereses (separation of concerns).

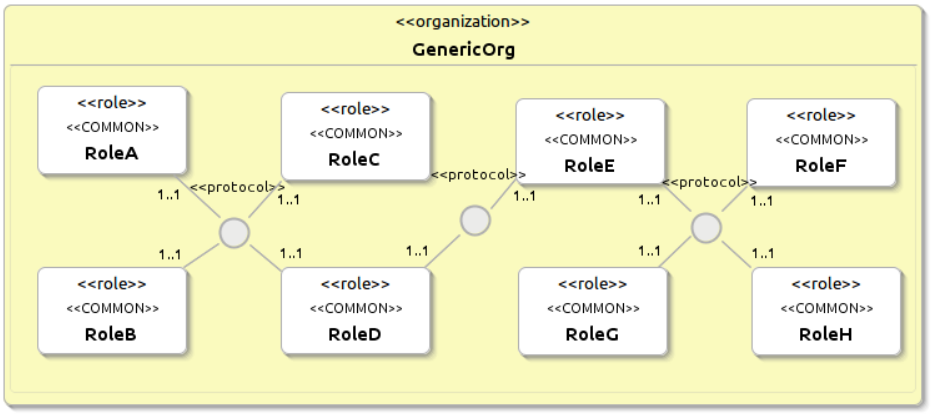

FIGURA 5.9: Different Context - Diagrama Organizacional.

En la Figura 5.9 se puede observar una organización compuesta por múltiples roles y un conjunto de protocolos que relacionan estos roles. Con un análisis más detallado es posible detectar que existen subconjuntos de roles dentro de una misma organización. Tal dedución surge a partir de la observación de los protocolos 1 y 3 -contando de izquierda a derecha- que son ilustrados en las Figuras 5.10 y 5.11 respectivamente, en donde existe una alta interacción entre los roles que participan en cada protocolo. En cambio, en el protocolo dos el número de agentes participantes como así también las interacciones entre ellos es mucho menor que en los casos anteriores (Figura 5.12).

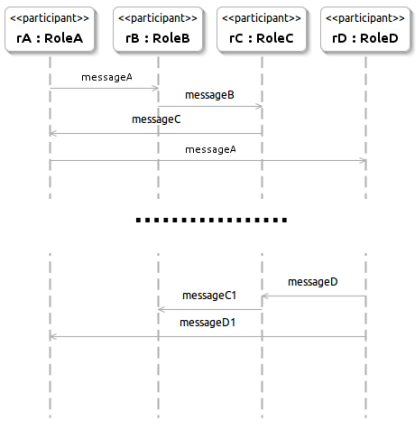

FIGURA 5.10: Different Context - Diagrama de interacción correspondiente al primer protocolo (a la izquierda de la Figura 5.9)

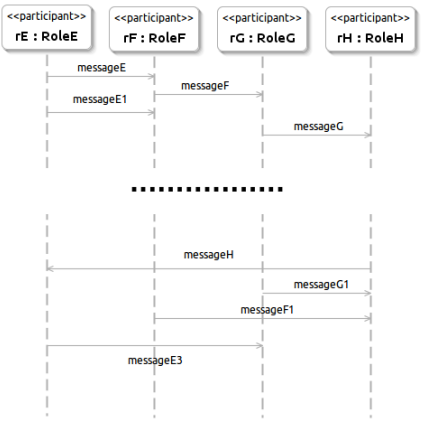

FIGURA 5.11: Different Context Smells - Diagrama de interacción correspondiente al tercer protocolo (a la derecha de la Figura 5.9)

La propuesta realizada no posee actualmente una regla para su detección debido a que para llegar a la conclusión de que estamos en presencia del smells es necesario un análisis mucho más detallado de la semántica asociada. Tal análisis puede involucrar, por ejemplo al *Diagrama de Ontologías del Problema*, que es uno de los posibles artefactos que ayudarían a detectar si en la organización bajo estudio existen contextos diferentes.

Sin embargo, con la definición del smell provista es posible detectarla "manualmente" y tener en consideración para una posible división en dos o más -según lo analizado- de la organización.

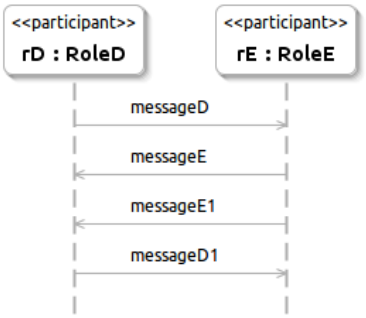

FIGURA 5.12: Different Context Smells: Diagrama de interacción correspondiente al segundo protocolo (en el centro de la Figura 5.9)

5.3.6. Capacity Abuse

Este *Design Smells* describe una situación que denominamos "Call Capacity Abuse", el cual resulta de la observación del concepto de capacidad. Esta abstracción fue creado para promover su reusabilidad y la modularidad. El concepto en sí puede ser considerado como una interface (de competencias requeridas por el agente para poder jugar el rol), permitiendo que una capacidad puede tener diferentes implementaciones asociadas a él. El ejemplo más común para esta características de la capacidad es el algoritmo *Encontrar el camino más corto*. Es conocido que existen muchas implementaciones posibles de este algoritmo que busca el camino mas corto entre dos vértices de un grafo. Por ejemplo: Bellman-Ford, Dijkstra, Ant Colony, etc. En nuestro caso, la capacidad tienen un sola implementación posible. Así, consideramos que no todas las *Skills* pueden ser abstraídos al concepto de capacidad. La situación surge dado que existen ciertos problemas que son resueltos de una única forma y no existen otra solución posible. Esto puede darse por cuestiones de legislación, debido a que es una proceso bien definido o porque es una tarea muy simple o sencilla.

5.3.7. Capacity Chain

Este *design smells* describe como identificar un conjunto de capacidades de un mismo rol que potencialmente pueden ser agrupadas en una sola. La situación queda en evidencia cuando una capacidad genera ciertas salidas que son utilizadas como entradas por otra. Una condición adicional es que estas capacidades no deben ser requeridas por ningún otro rol del modelo.

¿Cómo identificarlo?. Pasos metodológicos.

Paso 1. Identificar una secuencia de capacidades. Esto significa que las salidas de una de las capacidades es el parámetro de entrada de otra.

Paso 2. Verificar que ninguna de las capacidades que pertenece al encadenamiento sea requerida por otros roles del modelo.

Paso 3. Verificar si cada capacidad tiene solo una implementación.

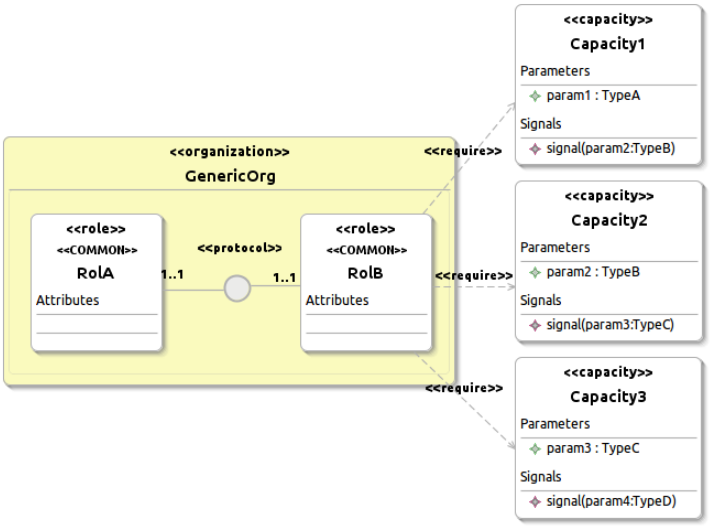

FIGURA 5.13: Capacity Chain - Diagrama Organizacional.

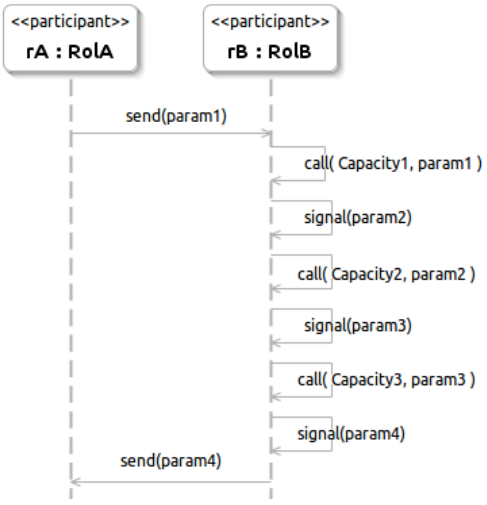

FIGURA 5.14: Capacity Chain - Diagrama de Interacción

5.3.8. Highly Fragmented Organizations

Como se ha mencionado con anterioridad, es posible relacionar cada comportamiento global a una única organización. A su vez, dicho comportamiento es fragmentado en

partes de menor tamaño que represetan a los roles (comportamiento esperado). En otras palabras, es una división en unidades de responsabilidades que los agentes que jueguen dichos roles deberán cumplir con las obligaciones y ejercer sus derechos acorde a las normas definidas en la organización.

Existe en la metodología ASPECS una actividad especialmente dedicada a lo descripto, denominada Identificación de Roles e Interacciones. En la misma se provee los lineamientos sobre como descomponer el comportamiento global en comportamientos de interacción más pequeños. Sin embargo, se ha podido observar que existen modelos con organizaciones compuestas por un alto número de roles dentro del mismo. Esta situación puede generar una excesiva e inecesaria burocracia dentro de la organización con roles de comportamientos muy simples.

¿Cómo identificarlo?. Pasos metodológicos.

Paso 1. Determinar el número promedio de roles en todas organizaciones.

Paso 2. Comparar el número de roles por cada organización con el promedio. Si el primero es mayor al segundo estamos en presencia de un HFO.

Paso 3. Otra cuestión sería definir umbrales que sirvan de referencia para la identificación del smell.

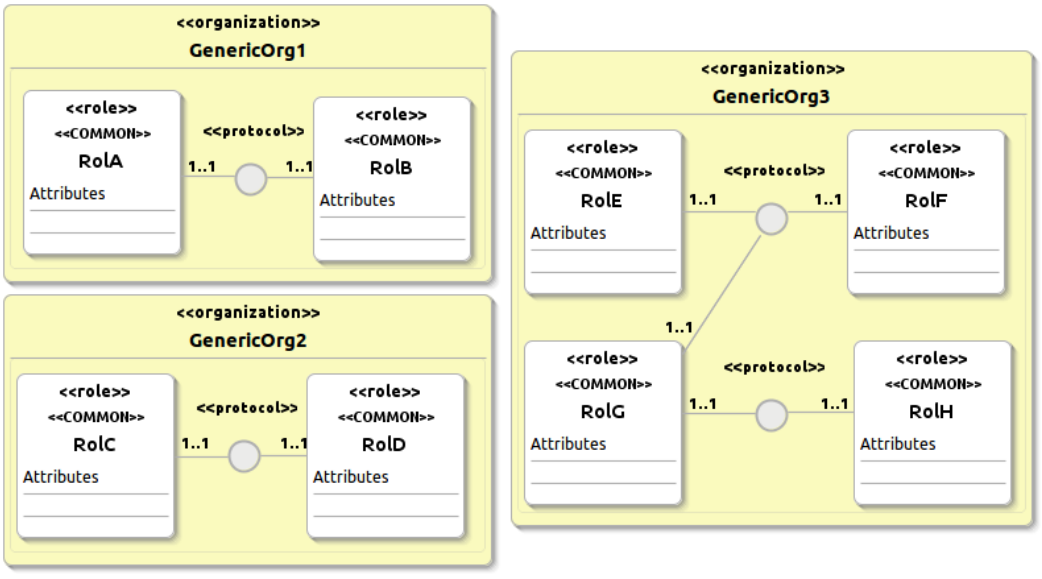

FIGURA 5.15: Highly Fragmented Organizations - Diagrama Organizacional

```
1 operation Organization numberOfRolesPerOrganization() : Integer {
2   return self.roles.size();
3 }
4
5 operation CRIOMetamodel averageRolesPerOrganization() : Real {
6   var average : Real = 0.0;
7   for (organization in self.organizations) {
```

```
8       average = average + organization.numberOfRolesPerOrganization();
9     }
10    return average / self.organizations.size();
11  }
12
13  context Organization {
14    constraint largeNumberofRole {
15      check : self.numberOfRolesPerOrganization()
16                    < self.eContainer().averageRolesPerOrganization()
17      message: "This organization could potentially be a HFO"
18    }
19  }
```

LISTADO 5.5: Regla EVL para identificar al smells Highly Fragmented Organizations

La implementación de la regla detallada en el Listado 5.5 cuenta con dos operaciones. La operación *numberOfRolesPerOrganization* (línea 1) retorna, a través de la función *size()*, el número total de roles existentes por cada organización definida.

En la línea 5 es definida la segunda operación *averageRolesPerOrganization* cuyo objetivo es calcular el promedio de roles por organizaciones presentes en el modelo. El cuerpo de la operación es definida una variable del tipo real para el promedio y un lazo *for* que recorre todas las organizaciones recuperando el número de roles. Una vez finalizado el lazo se calcula el promedio dividiendo la variable con el número de organizaciones.

El contexto de la regla está situado en el concepto de Organización (línea 13). En el cuerpo *check* se ejecuta una condición que permite determinar si el número de roles en una organización es menor que el promedio de roles por organización. Si bien en la regla presentada se utiliza un criterio demasiado simple dado que con el solo hecho de haber superado el promedio de roles por organización ya se lo puede considerar Highly Fragmented Organizations. Sin embargo, es posible establecer criterios basado en la experiencia que tienen los desarrolladores, como por ejemplo establecer umbrales para considerar que tal o cual regla se dispare. La definición de estos límites contribuye en reducir los falsos positivos que puedan presentarse.

5.4. Impacto de los smells en la diferentes etapas

La presencia de los smells descriptos puede traer complicaciones en diferentes etapas del ciclo de vida de desarrollo del sistema. Las tres etapas donde más impactan están detalladas en la Tabla 5.2. La primera columna, *Implementación*, trata sobre los smells que tienden a incrementar la complejidad de la implementación final del sistema haciendo al proyecto más difícil de entender y extender. La segunda columna es *Rendimiento* del sistema. El origen del inconveniente puede estar relacionado con el mal uso de los recursos computacionales o por una degradación de los tiempos de respuesta del sistema, entre otros. Y por último, y no por ello el menos importante, la columna correspondiente al

Metodologías	Etapas: Implementación	Rendimiento	Mantenimiento
Burocracy Role		✓	
Promiscuos Role	✓		✓
Bottleneck Situation	✓	✓	
Selfish Role	✓		✓
Different Context	✓		✓
Capacity Abuse			✓
Capacity Chain			✓
HFO	✓		✓

CUADRO 5.2: El impacto de los ODS en las diferentes etapas.

Mantenimiento dado que no erradicar estos smells puede conducirnos a un incremento de los costos de introducir modificaciones en el sistema ya sea para agregar nuevas funcionalidades o en la realización de mantenimiento correctivo.

Es importante destacar que los smells propuestos tienen un impacto en cualquiera de las tres etapas enunciadas; sin embargo, en la tabla se tilda aquellas donde la incidencia es más notable.

5.5. Hacia la definición de un proceso para la erradicación de Smells

En el capítulo tres de la tesis se mostró todos los trabajos que se han realizado sobre smells. Entre ellos se destaca un método, denominado DECOR, que sirve para describir smells y sus características principales. Sin embargo, en la literatura no existe un trabajo que especifique un método dedicado a la erradicación de smells considerando todas las etapas que podrían involucrar dicho proceso.

En la figura 5.16 se detalla una primera aproximación de lo que sería un proceso para la erradicación de smells. Dicha propuesta surge como una necesidad que atienda a esta actividad tan importante en el desarrollo del software. La misma está comprendida desde las cuestiones referidas a los recursos humanos hasta el procesos de refactoring, sin dejar atrás a la planificación que se debe realizar y los recursos que se deben asignar para el éxito del proceso.

A su vez, consideramos que un futuro y una vez que se haya refinado e incorporado mejoras a la propuesta, la misma debe ser incorporada a la planificación del proceso de desarrollo. Esto traerá múltiples beneficios a la organización, tales como un incremento de la calidad del producto final como así también el aprendizaje por parte del personal de las buenas prácticas y los estándares impuestos.

A continuación se describe el proceso para la erradicación de smells:

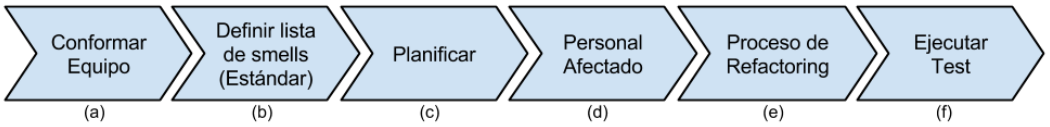

FIGURA 5.16: Procesos para la erradicación de smells.

(a) **Conformar equipo**: Por cada proyecto nuevo se debe definir cuales son las personas que definirán los estándares a ser utilizados; además, servirán como referencia dentro del grupo de desarrollo. Generalmente el criterio de selección está basado en la experiencia que hayan obtenido en diversos proyectos, arquitecturas, tecnologías como así también familiaridad con el dominio del nuevo sistema.

(b) **Definir lista de Smell**: Una vez finalizada la conformación de los grupos de profesionales, se debe proceder a la definición de cuales serán los criterios de calidad -entre ellos la lista de smells-. La composición de la lista está restringida a cuestiones relativas a la características del proyecto, dominio, personal asignado (habilidades y formación), presupuesto y fechas de entregas límites.

(c) **Planificar**: Alcanzado un número crítico de smells se puede considerar iniciar el proceso de refactoring. Hay que tener en cuenta que dicho proceso deberá ser considerado al inicio de la planificación general del sistema dado que es del tipo *time consuming*. El criterio para establecer cuando se realizará el refactoring sobre un sistema es considerada una tarea subjetiva y que es decisión exclusiva del ingeniero de software a cargo del proyecto.

(d) **Personal afectado**: se trata de las personas que serán las encargadas de ejecutar el refactoring. Pueden ser seleccionadas de acuerdo a su conocimiento del estándar, el menor número de errores cometidos o incluso pueden ser las mismas personas que introdujeron los errores como una forma de reforzar el conocimiento que tienen del estándar.

(e) **Proceso de refactoring**: este proceso permite eliminar los smells que fueron detectados. Cabe destacar que los módulo, y de acuerdo a la concentración de smells que posea, estará bloqueado para que no se le agreguen nuevas funcionalidades.

(f) **Ejecutar Test**: dado que el procesos de refactoring es modificar la estructura del sistema sin alterar el comportamiento del mismo, una vez finalizado es de vital importancia ejecutar los test diseñados para verificar que los modificaciones hechas no impactaron sobre el comportamiento anterior del sistema. Puede suceder que algunos cuestiones no fueron contempladas en las pruebas y por lo tanto no sean detectadas.

5.6. Conclusiones

Consideramos que la necesidad de identificar y formalizar estructuras que pueden potencialmente perjudicar la arquitectura y/o la evolución del sistema representa un aporte significativo al campo del aseguramiento de calidad de los SMA basados en el enfoque organizacional. En este capítulo se dio introducción a los Organizacional Design Smells y el papel que juega en los modelos organizacionales de los SMA. Además, una primera aproximación de los criterios para la identificación de los smells organizacionales es presentada. En la misma cuatro categorías fueron introducidas: *Organization Smells*, *Interaction Smells*, *Behavior Skill Smell* y por último, *Skill Smells*.

A continuación de los criterios, se describe una lista de 8 smells junto a un breve detalle del propósito de cada una. Inmediatamente se realiza una descripción más detallada de cada uno compuesto por el propósito, los pasos metodológicos necesarios para poder detectarlo en el modelo, los diagramas y la reglas EVL con su respectiva explicación. Solo dos smells no poseen reglas por los motivos expuestos en la subsección 5.3.5, indicando que es necesario realizar un análisis más complejo de los artefactos que componen el modelo.

En el futuro nos concentraremos en generalizar la propuesta a otros enfoques organizacionales y proponer actividades específicas para la identificación y erradicación de estos smells. También se evaluará si las propuestas para erradicar estos smells pueden convertirse en patrones de diseño.

Capítulo 6

Casos de Estudio

En la presente sección se describirán dos casos que fueron objetos de estudio: el primero pertenece al conjunto de proyectos homologados que se desarrollan dentro del Grupo de Investigación en Tecnologías Informáticas Avanzadas (GITIA), mientras que el segundo fue extraído de la literatura SMA. Para los mismos, se realizará una introducción a la problemática describiendo el contexto y las restricciones que tienen asociados.

6.1. Zafra

6.1.1. Introducción al problema

El transporte de la caña de azúcar desde los productores en los campos hasta los ingenios para su procesamiento es un problema complejo. Una solución realista para el problema requiere tener en cuenta una gran cantidad de aspectos diferentes y restricciones a respetar. A pesar de la importancia económica de esta actividad para la provincia de Tucumán (la cual se estima en un 40 % del PBI de la provincia) en la actualidad la plantación, cosecha y transporte no presentan una planificación optimizada. Esta situación se agrava por otros elementos socio-económicos (mal mantenimiento de los vehículos, etc) que reducen la rentabilidad de la actividad tanto para productores como para los equipos.

Tucumán, una provincia de noroeste de la República Argentina, es la principal productora de caña de azúcar del país. Esta actividad es además una de las industrias más grandes de la provincia, responsable del 10,5 % del Producto Bruto Interno (PBI) y del 42 % de la manufactura industrial.

A pesar de la importancia económica de la actividad, la misma no está optimizada. Situación que no sólo reduce las ganancias tanto de los productores de la caña de azúcar y los dueños de los ingenios, sino que además tienen un impacto importante en el tráfico de las rutas, la seguridad de los caminos y el medio ambiente en general. Una cadena de suministro planificada y organizada podría brindar, por ejemplo, fechas precisas para el plantado y la cosecha basada en la capacidad de los ingenios o proveer planificaciones para minimizar los costes de transporte.

Existen diversos esfuerzos para la optimización de la cadena de suministro en la literatura pero con éxitos relativo. Sin embargo, y debido a las diferencias sustanciales entre los procesos de las cadenas de suministros entre los países, las soluciones propuestas en los trabajos no son directamente aplicables a nuestro escenario.

En el área de Tucumán existen 15 ingenios que pueden procesar la mayor parte de la producción (lo restante va a otras partes del país). Los ingenios tienen asociado capacidades con cargas máximas que pueden procesar. Aunque pueden operar con estas cantidades, la eficiencia picos ocurre cuando la cantidad de caña ronda el 85 % de su capacidad máxima.

La producción anual de caña de azúcar en Tucumán es de aproximadamente 40 millones de toneladas distribuidos durante la temporada de cosecha (de Marzo a Octubre) con un pico en los meses de Julio-Agosto. Los campos de caña no son todos cosechados de la misma manera: cerca del 66 % es realizado automáticamente usando máquinas cosechadoras, el 24 % de forma semi-automatica; y finalmente, el 10 % de los campos son cosechados manualmente.

Durante el periodo de cosecha (180 días aproximadamente), todos los días la caña es cosechada y tomada de los campos hacia algunos de los ingenios. La cosecha general para un determinado día no es constante, esta es afectada por diversos factores ambientales; el primero y más importante es el cambio de estación el cual afecta la producción de forma global pero predecible. Es menester advertir que debido a esta situación, mientras en el comienzo y hacia el final de la zafra los ingenios tal vez no procesen la suficiente caña para cubrir sus costos operacionales. En cambio, en la mitad de la zafra puede haber demasiada caña de azúcar para ser procesados por los ingenios.

En segundo lugar, los otros factores son estocásticos y afectan a la producción de una manera más local; tales como escarchas del piso, huelga de trabajadores. Estos hacen de la producción de la caña de azúcar versus el patrón de tiempo parecerse a una distribución Gaussiana con un ruido de frecuencia alto.

Los ingenios realizan negociaciones con cada uno de los productores para asegurarse que suficiente materia prima le llegue todos los días. Típicamente, el ingenio de mayor

tamaño es capaz de comprar más caña de la que necesita, contrario es el caso de los ingenios de menor tamaño que no pueden comprar suficiente y a menudo tiene que operar por debajo de su capacidad. Idealmente, todos los ingenios deberían siempre tratar de operar al 85 % de su capacidad el cual es donde su productividad alcanza picos. Si esto no es posible (debido a que la producción de la caña de azúcar es alta o baja más que la alcanzada por esta meta) el objetivo debería ser maximizar la eficiencia general del sistema.

Sin embargo, una solución integral a los problemas enunciados debería incluir una completa planificación y un esquema de candelarización del plantado, cosechado y transporte a lo largo de toda la temporada. El alcance del problema se reduce a la situación de considerar solo el transporte de la producción de la caña desde los campos a los ingenios en un solo día con un costo mínimo, tomando en cuenta un conjunto de restricciones.

Se han presentado diferentes contribuciones por parte de integrantes del GITIA que representan versiones preliminares de este problema desarrollándose un modelo matemático y diferentes algoritmos heurísticos para resolverlos [91][92]. En todos los casos, los mejores resultados se obtuvieron con heuríticas basadas en SMA. A continuación se presentará el modelo propuesto para el problema presentado y los smells detectados en el mismo.

6.1.2. Modelo organizacional

En la Figura 6.1 se ilustra el diagrama organizacional para el problema, modelado mediante Janeiro Studio. En la misma puede observarse a la organización Zafra compuesta por dos roles: *Consumidor* (Ingenio) y *Productor* (Parcela), ambos relacionados a través de un protocolo (círculo gris). Además, los roles tienen asociadas capacidades: *Consumidor* requiere de las capacidades *CalcularRendNorm* (abreviación de Calcular Rendimiento Normalizado), *CalcularFuerza*, *CalcularEnergía* y *CalcularAcumulador*; mientras que el rol *Productor* sólo tiene asociada la capacidad *ElegirConsumidor*. La finalidad de cada una de estas capacidades serán descriptas más adelante.

6.1.3. Smells detectados

Una vez ejecutadas las reglas sobre el modelo propuesto para Zafra, el sistema el sistema de validación arrojó que fueron detectados dos potenciales smells: el primero de ellos es Selfish Role y el segundo es Capacity Chain.

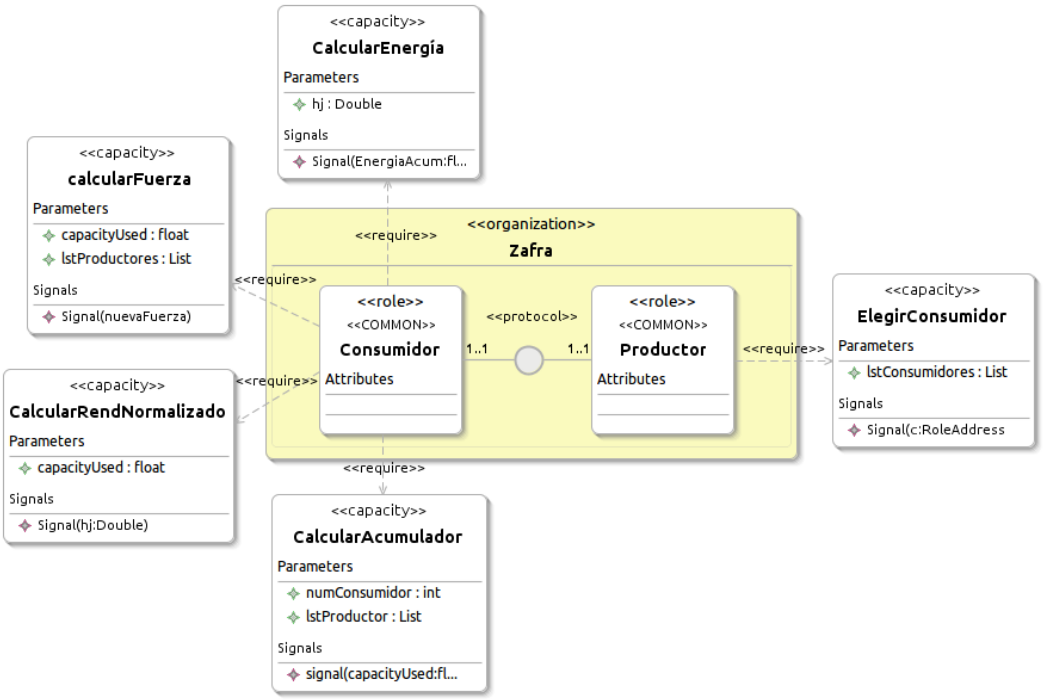

FIGURA 6.1: Zafra: Diagrama Organizacional.

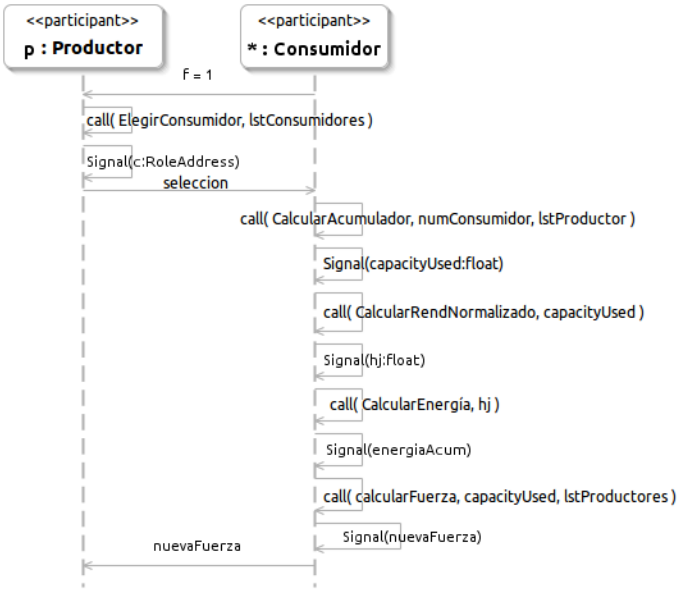

FIGURA 6.2: Zafra: Diagrama de Interacción.

Selfish Role

El primer smell detectado es el Selfish Role -descripto 5.3.4- que resulta en un rol que concentra un alto número de responsabilidades dentro de la organización. En el caso de la organización Zafra ilustrado en la Figura 6.1, el rol *Consumidor* tiene asociado cuatro capacidades en contraste a la única capacidad que está relacionada con el rol *Productor*. Tal como se había indicado en la descripción del smells, en el diagrama es posible observar un desbalance de las responsabilidades atribuidas a la organización. Este modelo desembocó en una compleja representación e inclusive implementación del comportamiento del rol. Parte de esta complejidad se ilustra en la Figura 6.2, donde el rol *Consumidor* tiene por cada capacidad una llamada a la misma y una señal. Tal como oportunamente se ha indicado, implementar este sistema resultó en una estructura de código complicada de leer y seguir. De hecho introducir modificaciones en el sistema, si bien no es imposible, si resulta bastante engorrosa de realizar.

Capacity Chain

El segundo smells detectado es *CapacityChain*, descripto en la Subsección 5.3.7. Esta regla identifica una secuencia de capacidades en donde la salida que genera una capacidad es utilizada como parámetro de entrada por otra capacidad asociada al mismo rol. En la Figura 6.2 se puede visualizar mejor al rol *Consumidor* donde la secuencia de capacidades encadenadas comienza con la capacidad *CalcularAcumulador* que retorna la variable *capacityUsed*. Dicho valor de retorno es utilizado por la capacidad *CalcularRenNormalizado*. Finalizadas las tareas definidas para dicha capacidad, la misma devuelve el valor *hj* y que es tomado como parámetro de entrada por *CalcularEnergia*. Además, se cumple con la restricción de que las capacidades no son requeridas por ningún otro rol del modelo.

La presencia de este smell puede permitirnos repensar el modelo, dado que si se cumple el mencionado encadenamiento junto a la restricción de que las capacidades no son requeridas por ningún otro rol, se podría evitar que las líneas de códigos dedicadas a las llamadas de capacidad, como de las señales, sean significativas en la implementación. Se ha considerado que hubiera sido mejor modelar todas estas capacidades encadenadas como una sola.

En este ejemplo se da una situación que merece una explicación: En casi todas las capacidades tienen definida como parámetro de entrada una lista denominada lstProductor que advertimos que no debe ser considerada en la definición y ejecución de la regla de validación, dado que obedece a una cuestión meramente implementatoria.

6.2. MicroGrid

6.2.1. Introducción al problema

En el contexto actual de la demanda eléctrica, las características de las redes eléctricas tradicionales se vieron superadas debido al incremento del consumo eléctrico. Parte de este suceso devino gracias al surgimiento de conceptos y tendencias tales como la generación distribuida, aplicación y uso de energías renovables, almacenamiento de energía y gestión eficiente del consumo. Todo lo mencionado trajo aparejado la necesidad de administrar de manera más eficiente la red, dando introducción al concepto de *Smart-Grid*. Las *Smart-Grid* son, en pocas palabras, un sistema eléctrico funcionando en paralelo con un sistema informático. Estos sistemas deben ser capaces de monitorear y controlar el estado de la red a través de diferentes dispositivos, monitores y actuadores, con el objetivo de lograr una administración más eficiente del uso de la energía.

Otro concepto asociado a los Smart-Grid son las denominadas *MicroGrid*, los cuales hacen referencia a una porción o parte de una red que puede separarse de la red de abastecimiento principal de energía. Una *MicroGrid* está conformada por un sistema informático que debe ser capaz de gestionarla, un sistema de generación de energía distribuida, un sistema de almacenamiento y por supuesto el sistema eléctrico que interconecte los consumidores (viviendas, negocios, edificios públicos, etc.) que la conforman. Las Microgrid tienen la capacidad de autoabastecerse y gestionar el consumo, de tal manera que puedan funcionar independientemente de la red principal, aunque de ser necesario, pueden llegar a conectarse a ésta última para satisfacer su demanda o proveer de energía.

6.2.2. Modelo Organizacional

Teniendo en cuenta que la tecnología de sistemas multiagentes es utilizada mayormente para el modelado de sistemas complejos, es habitual que los diagramas cuenten con un alto número de conceptos. Por cuestiones de espacio, en la Figura 6.3 se presenta la organización *MicroGrid* que está constituida por cuatro roles, tres de los cuales se encuentran asociados a la simulación del funcionamiento propio de la *MicroGrid* (*EnergyDevice*, *Regulator*, *Transmiter*), mientras que el restante (*TimeManager*) se utiliza para gestionar el tiempo de la simulación. En más detalle, el rol *EnergyDevice* representa un dispositivo que consume energía, la genera o ambas en un determinado momento, tal como sucede con los sistemas de almacenamiento, dado que estos consumen al momento de la carga y proveen energía en la descarga según la necesidad de la red. Por otro lado, el *Transmiter* cumple la función de simular el conductor o red a la cual se encuentran conectados

los diferentes dispositivos o *EnergyDevices*. El *Regulator* tiene la función de regular y estabilizar la energía que se encuentra "fluyendo" por el *Transmiter*. Es un dispositivo del tipo monitor y regulador, siendo el mismo el encargado de determinar las conexiones y desconexiones de los dispositivos. Por último, el *TimeManager* cumple con el rol de ser el responsable de manejar el tiempo o velocidad actual de la simulación que se desee ejecutar.

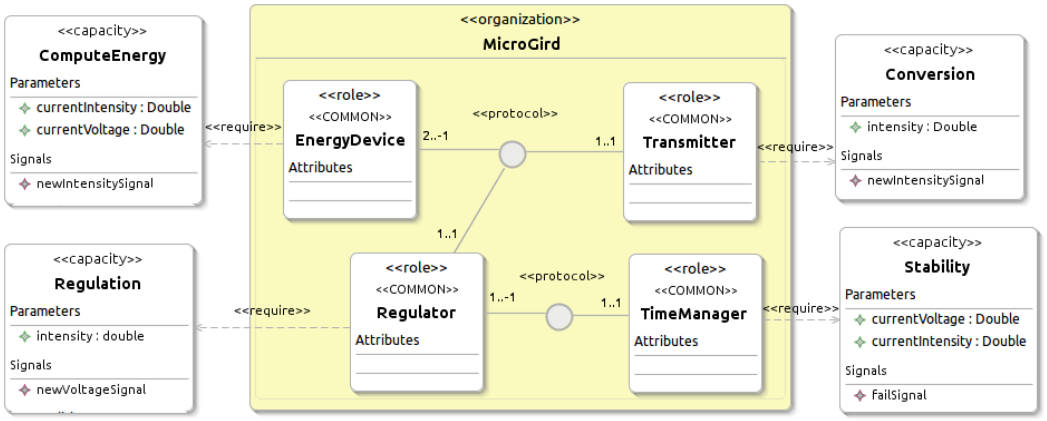

FIGURA 6.3: MicroGrid: Diagrama Organizacional.

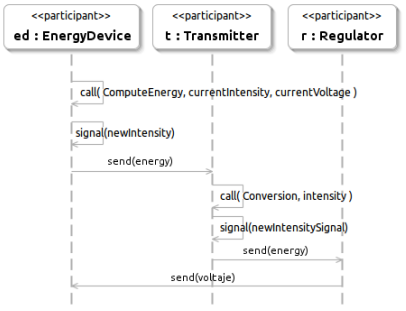

FIGURA 6.4: Microgrid: Diagrama de Interacción para el protocolo 1.

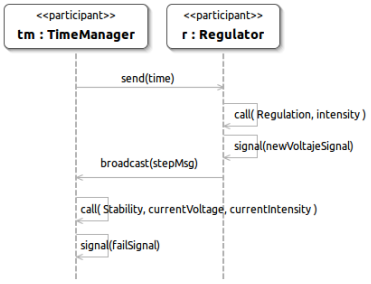

FIGURA 6.5: Microgrid: Diagrama de Interacción para el protocolo 2.

6.2.3. Smells detectados

Para el caso del MicroGrid, se han detectado potencialmente dos smells. Una aclaración importante es que en el modelo original, realizado y refinado en el laboratorio del IRTES-SeT, consta de 2 organizaciones adicionales. Una que modela la zona de simulación (dos roles) y el otro que está relacionado con la interfaz con el usuario (también compuesto de dos roles).

Highly Fragmented Organization

Así, el sistema de validación arroja que uno de los smells detectado es el HFO, dado que la organización principal está conformado por cuatro roles lo cual -en un primer término- supera el promedio de roles por organizaciones que existen en todo el modelo. Sin embargo, y después de un análisis más detallado de la organización, concluimos que el smell resultó en un falso positivo dado que los comportamientos de los roles -si bien son triviales- son significativos para ser representados y tratados de forma individual.

De todas forma, para considerar que este smell es positivo en ciertas ocasiones es recomendable recurrir a los otros diagramas para un análisis más detallado y así llegar a un conclusión muchos más certera de este smell. Tal análisis ayudará en un futuro en la descomposición de la organización tanto en dos o más organización; o en el otro extremo, permitirá determinar cuáles son los roles que pueden ser combinados.

Capacity Abuse

El segundo smells detectado es producto de un análisis manual: el mismo es *Capacity Abuse*. Consideramos que el smell está presente después de analizar el código generado para el simulador del MicroGrid. En dicho código, la mayoría de las implementaciones de las capacidades son tan solo operaciones de suma. De hecho, se introducen más códigos en las declaraciones de las capacidades, implementación en los agentes y en los llamados a las capacidades dentro de los roles que en la implementación en sí de las operaciones. Un segundo criterio, además de la simplicidad de las capacidades, es que no existen diferentes implementaciones para realizar dichas operaciones bajo estudio. Por lo tanto, podemos indicar que en el modelo en observación es necesario eliminar el smell.

6.3. Conclusiones

Para validar parte de las propuestas realizadas en el Capítulo 5, se utilizaron dos modelos. A partir de los análisis realizados en dichos modelos, se extrajeron que tres restricciones fueron violadas, siendo uno de ellas un falso positivo. Además, se llevó a cabo un análisis manual de los artefactos de ambos modelos y se pudo identificar en el segundo proyecto la presencia de un smell que representa el mal uso que se hace de uno de los conceptos fundamentales de CRIO.

El primero es de un proyecto propio del grupo de investigación en donde fue desarrollada esta tesis. En la misma se detectaron dos smells: En primer término se detectó un importante número de capacidades asociadas a un rol convirtiéndolo en un centralizador

de servicios o Selfish-Role. Además, en la misma organización se identificó también un encadenamiento de tres capacidades, las cuales ninguna es requerida por otro rol; por los que concluimos que estamos en presencia del smell *Capacity Chain*.

Una continuación a este trabajo sería determinar qué estrategia se debería seguir para la erradicación de los smells y su prioridad. Si la estrategia para la eliminación del Selfish-Role o para el Capacity Chain.

El segundo proyecto bajo estudio es un desarrollo realizado en el laboratorio IRTES-SeT y está relacionado con un simulador para los Smart-Grid. En la misma se pudo identificar también dos smells. El primer smell, denominado HFO, resultó en un falso positivo dado que un análisis posterior se concluyó que los comportamientos que representan los roles identificados se corresponden a un correcta partición de la organización.

El segundo smell identificado surgió no tan solo del análisis del modelo, sino que se tomó en consideración el código del sistema implementado en Janus. En el mismo se observó que las líneas de código dedicadas a la declaración de la capacidad, implementación de las misma y los llamados a las capacidades que tienen lugar en los roles, son muchos más que la implementación de la lógica de las operaciones. Así, se concluyó que se hizo de un abuso en el uso del concepto de capacidad en el modelo. Uno de los motivos de porque se declararon tales capacidades puede deberse a que los diseñadores quisieron remarcar cuales son las operaciones más importantes de cada rol tal vez sin dimensionar el costo de implementar dicha solución.

Capítulo 7

Conclusiones y Trabajos Futuros

7.1. Conclusiones Generales

En el presente trabajo de Tesis se ha propuesto un enfoque para realizar validaciones sintácticas y semánticas en el área del Desarrollo Dirigido por Modelos, una disciplina de la Ingeniería de Software aplicados a Sistemas Multiagentes (SMA). Existe en la literatura un importante número de contribuciones acerca de los code/design smells aplicados al paradigma orientado a objetos pero muy pocos trabajos para el dominio de los SMA. Por este motivo consideramos crítico adaptar o redefinir esta técnica dado que los SMA están construidos sobre conceptos y/o tecnologías diferentes.

Dos grandes objetivos fueron definidos: El primero de ellos fue la definición de un conjunto de reglas de validación sintáctica que permiten llevar a que un modelo sea conceptualmente válido respetando las restricciones impuestas por el metamodelo y aquellas definidas por el usuario (Capítulo 4). El segundo objetivo fue la identificación y formalización -también a través de reglas- de situaciones en el modelado que pueden conducirnos a una degradación de la calidad del modelo (validación semántica) -y por lo consiguiente su implementación- impactando en diferentes etapas ya sea de mantenimiento y/o rendimiento del sistema, entre otras (Capítulo 5). Para la realización de las validaciones semántica se introdujo a los *Organizational Design Smells* (ODS) y sus principales características. Una primera aproximación de los ODS es que los mismos representan una desviación a las buenas prácticas de modelado. En otras palabras, es un indicador de un diseño pobre o defectuoso que no cumple con los estándares de diseño organizacional y que puede impactar negativamente en etapas posteriores.

Tanto la validación sintáctica como la detección de smells, se apoyan en el análisis de las instancias del metamodelo organizacional CRIO. El metamodelo debe su nombre a los

cuatro conceptos principales: Capacidad, es una descripción de un know-how/servicio y una abstracción de alto nivel que promueve su reusabilidad y modularidad; Rol, es el comportamiento esperado y un conjunto de derechos y obligaciones en el contexto de la organización; Interacción, es una secuencia sistematizada de intercambio de información entre roles; y finalmente, Organización, que es definida por una colección de roles que interactúan usando patrones institucionalizados de interacciones con otros roles en un contexto común. Es importante mencionar que dicho metamodelo es la piedra angular de la metodología ASPECS [1], reconocida actualmente como una de las más completas para el modelado de SMA [21].

Para estas definiciones se requirió del desarrollo de una extensión a la herramienta Janeiro Studio [3]. Dicha herramienta es un prototipo cuyo objetivo es asistir a los analistas/diseñadores en el uso de ASPECS. Además, busca facilitar la representación gráfica de los diagramas definidos por la notación de la metodología y proveer los lineamientos para un correcto armado de los modelos. En la actualidad, la herramienta cuenta con una adaptación informática del metamodelo CRIO empleando Eclipse Modeling Framework (EMF) permitiendo cubrir la primera fase de ASPECS: *Dominio del Problema*.

Para analizar las instancias y detectar la presencia de ODS en el metamodelo fue necesaria la adopción de un lenguaje que cuente con las características que permita una implementación parcial o total de los smells. Para tal labor se seleccionó Epsilon Validation Language (EVL), que es parte importante de una gran familia de lenguajes que conforman Epsilon desarrollado por la Eclipse Foundation. La combinación de EVL y la extensión de Janeiro permiten la creación, modificación, eliminación e importación de nuevas reglas que vayan surgiendo de acuerdo a las necesidades de los usuarios como así también la ejecución automática sobre las instancias del metamodelo de las reglas definidas.

Para la tesis se han presentado ocho ODS de las cuales seis tienen definidas reglas de validación. Consideramos además que dichas reglas deben ser -en lo posible- ejecutadas automáticamente sobre los modelos. La detección de estos "problemas" permite a los diseñadores abordarlos en etapas tempranas del ciclo de vida de desarrollo. En cambio, una detección tardía -en la etapa de codificación o peor aun cuando el sistema esta siendo utilizado por los usuarios finales- impactará negativamente en los costos del proyecto y en la calidad del software en general. Las dos reglas que quedaron sin una formalización se debe a que estas dependen en gran medida del dominio de aplicación y los diferentes contextos que pueden presentarse en una organización. Esta situación requiere de un análisis más profundo de la semántica asociada del que se realiza en esta tesis, involucrando artefactos que no se tuvieron en cuenta tales como las ontologías, etc. Sin

embargo, la detección de estos smells es posible realizarlas con las tradicionales técnicas de búsquedas manuales de smells.

Para probar la validez de algunas de las propuestas se utilizaron dos modelos: el primero es un modelo que surge de un proyecto homologado por la UTN y que es desarrollado en el Grupo de Investigación en Tecnologías Informáticas Avanzadas (GITIA) -lugar donde el tesista desempeña sus actividades-. El mismo está relacionado con la optimización del transporte de la caña de azúcar en la provincia de Tucumán. El segundo modelo proviene de la literatura de los SMA y es el diseño de un simulador Smart-Grid y los distintos componentes que lo conforman.

7.2. Perspectiva y Trabajo Futuros

En este trabajo se introdujo el concepto de ODS en cual permite la detección temprana de problemas de diseño en un SMA. De esto se desprende un conjunto de líneas de trabajo para mejorar la precisión y detalle de la detección y proponer un conjunto de soluciones adaptadas a las problemáticas de los SMA presentadas.

A continuación detallamos las principales líneas de trabaja futuros.

7.2.1. Propuestas de Refactoring

La tesis está fuertemente enfocada en la detección y formalización de los problemas de diseño que fueron detectado a partir del estudio de los distintos proyectos de investigación que se encuentran en desarrollo en el GITIA y de la experiencias de algunos de sus miembros. Consideramos que es importante proveer por cada smells, estrategias de refactoring bien definidas para evitar que la introducción de las modificaciones necesarias para la eliminación de los smells tenga impacto en el comportamiento que exhibe el sistema. A su vez, estimamos que es posible proponer más de una estrategia por cada ODS.

7.2.2. Formalizar los ODS usando Métricas

Las métricas de software están destinadas a conocer o estimar el tamaño y/o las características que presentan un sistema de software. Existen en el campo de los SMA trabajos que destacan las ventajas en la adopción de este tipo de técnicas, entre ellos podemos citar a Cossentino [93], Cernuzzi [94], García-Magariño [95], etc.

En esta línea de investigación ya estamos trabajando en la caracterización de la mayoría de los ODS usando métricas, en nuestro caso, aplicados a los modelos organizacionales. El objetivo principal de realizar mediciones es determinar el grado de modularidad, extensibilidad y mantenibilidad, entre otros atributos no funcionales de los diseños.

7.2.3. Patrones de Diseño Organizacionales

Un patrón de diseño es una plantilla de una propuesta de solución a un problema recurrente en un modelo con indicaciones sobre cómo adaptar el patrón a determinados dominios. El concepto de patrones nació en el campo de la arquitectura tradicional y debido a las ventajas que ofrece rápidamente fue adoptado por la industria del software. Dicha técnica permite capitalizar experiencias pasadas de un conjunto de personas para que puedan ser aplicadas en el futuro.

Así como los smells son defectos en el diseño y la especificación de los mismos está sujeta muchas veces al contexto y a la cantidad de ocurrencia dentro del modelo. En este sentido, está en consideración explorar si dichas soluciones pueden ser consideradas Patrones de Diseño Organizacionales (PDO). Además, se evaluará la posibilidad de proponer técnicas para capitalizar las ventajas de la combinación de patrones.

7.2.4. Proponer un proceso de validaciones

Consideramos que las actividades de validación sintáctica y erradicación de smells deben ser realizadas antes de aplicar los potenciales patrones que se desarrollen en un futuro. Por este motivo -y paralelamente a la formalización de dichos patrones- se está trabajando en la definición de un proceso para la aplicación de patrones de diseño organizacionales. Dicho proceso permitirá identificar oportunidades para la aplicación de patrones detectando problemas recurrentes, fuerzas del dominio, a su vez que debe proponer el patrón que mejor se adapte y asistir al diseñador en la aplicación del mismo. A su vez, deberá generar toda la documentación acerca de la aplicación del patrón como así también los elementos que permitirán realizar una trazabilidad del mismo. Nuestra intención es que dicho proceso sea especificado siguiendo las normas Software & Systems Process Engineering Metamodel (SPEM) de la Object Management Group (OMG) para luego integrarlas dentro de la metodología ASPECS.

Apéndice A

Janeiro Studio CASE Tool

La Ingeniería de Software Orientada a Agentes (ISOA o AOSE por sus siglas en inglés) requiere para el análisis, diseño e implementación, de cuatro elementos fundamentales: el metamodelo y los lenguajes que se utilizarán para describir los modelos; la metodología que define la secuencia de pasos a seguir y los actores involucrados para la obtención de un diseño del producto; la plataforma de implementación sobre la cual se ejecutarán estos modelos; y por último, la herramienta CASE (Computer Aided Software Engineering) utilizada para asistir al diseñador en el proceso de desarrollo. Consideramos que estos elementos representan lo que denominados *Ecosistema* y que son necesarios para abordar un proyecto de desarrollo.

En la literatura orientada a agentes existe un importante número de herramientas de desarrollo. La lista está comprendida por aproximadamente 24 aplicaciones comerciales y 40 proyectos académicos. Esta cantidad, y a diferencia del paradigma orientado a objetos que alcanzó un nivel de madurez considerable, refleja la constante evolución de la teoría agentes. El propósito de estos desarrollos es, en la mayoría de las veces, reforzar las propuestas de algún grupo de investigación ya sea una metodología en particular, teoría de agencia, arquitectura o algún lenguaje agentes.

Dada la vasta diversidad de metáforas utilizadas para el modelado de sistemas multi-agentes, realizar una comparativa de todas ellas está fuera del alcance de este documento. Es por ello que la tabla A.1 sólo se centra en las herramientas que están basadas en el enfoque organizacional. Muchas de estas aplicaciones están desarrolladas para metamodelos diferentes, proveyendo soporte visual para algunos o la mayoría de sus diagramas más importantes.

La tabla A.1 consta de una serie de criterios que describiremos brevemente a continuación.

CUADRO A.1: Metodologías y sus herramientas

	Metodología	Diagramas Cubiertos	Verificación de Modelos	Verificación Cruzada	Generación de Código	Soporte
agentTool III	O-MaSE	Todos	Si	si	Si	Si
GAIA4E	GAIA	Todos	N/E	N/E	No	N/S
IDK	Ingenias	Principales	No	No	Si	Si
PDT	Prometheus	N/E	Si	Si	Si	?
Metameth / PTK	PASSI	Todos	Si	Si	Si	No/Si
OpenTool	Adelfe	?	Si[1]	No	?	No
Rebel	ROADMAP	Principales	No	No	No	No
T-Tool	Tropos	Principales	Si[2]	No	No	No

Metodología. Indica cual es el proceso de desarrollo de software a la que brinda soporte.

Diagramas Cubiertos. Cada metodología especifica una serie de diagrama. Este ítem indica en qué medida estos diagramas cubren las distintas fases que componen la metodología.

Verificación de Modelos. Permite encontrar inconsistencias en un mismo diagrama.

Verificación Cruzada. Permite validar la consistencia del significado de un mismo concepto en diferentes diagramas que conforman el modelo.

Generación de Código. Contemplamos este criterio dado que deja entrever si la herramienta acompaña el proceso de desarrollo. Es importante remarcar que ninguna de ellas genera un código que directamente es compilable sino más bien nos proporcionan un esqueleto del código.

Soporte. Indica si se continúa con el desarrollo de la herramienta. Muchas de ellas fueron pioneras en área de las tecnologías multiagentes. Otras en cambio fueron creadas para demostrar la efectividad de algún concepto y nunca salieron del ámbito académico. Distinta suerte tuvieron otras como O-MaSE que cuentan en su haber aplicaciones académicas e industriales.

En el presente apéndice se presenta los avances realizados en la herramienta CASE Janeiro Studio [3]. Si bien el objetivo final de la mencionada herramienta será proveer de un soporte adecuado tanto para el metamodelo organizacional CRIO como la metodología ASPECS [1], actualmente Janeiro se encuentra en sus primeras etapas de desarrollo abarcando sólo la primera fase de la metodología. Desde su inicios fue concebido como una herramienta CASE multiplataforma de libre distribución, open-source, completamente desarrollada en JAVA que facilita el modelado de sistemas multiagentes basado en el enfoque organizacional. En la Figura A.1 se ilustra la interfaz creada para Janeiro. La misma se divide en cinco secciones que serán detalladas a continuación:

(A) **Explorador de Proyectos**. Contiene a todos los proyectos que fueron creados en el espacio de trabajo. Además, cada proyecto es posible visualizarlo estructurado

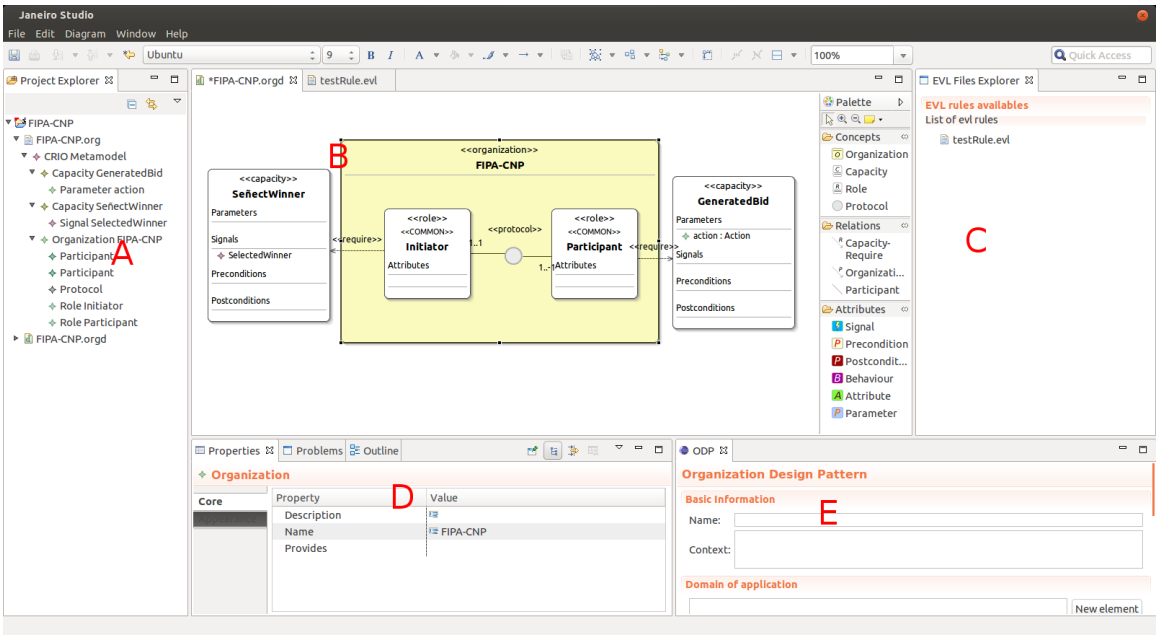

FIGURA A.1: IDE del prototipo Janeiro Studio.

en forma de árbol conformado por los elementos definidos por el usuario que están presentes en los diagramas. Esta estructura facilita la navegación y búsqueda de los conceptos plasmados en los distintos diagramas.

(B) **Editores**. Actualmente fueron definidos cinco editores para la manipulación de las instancias de los diagramas. Los mismos serán explicados más adelante en esta sección. Para cada tipo de diagrama se despliega una paleta con accesos directos de los conceptos gráficamente representados que pueden utilizarse para el armado del modelo (Palette).

(C) **Explorador de reglas de validación**. Janeiro cuenta con un validador y un conjunto de reglas de validación integradas, las mismas son para la validación sintácticas además de ser genéricas para todos los proyectos en los cuales se esté trabajando. Con esta vista también es posible definir reglas que atañen a situaciones particulares de cada proyecto.

(D) **Conjuntos de *View-Part* (*Properties*, *Problems*, *Outline*)**. Consiste en tres "tabs" de soporte que proveen información sobre los diagramas que se están manipulando en el editor que tiene el foco de la aplicación. El primero de ellos, *Properties*, posibilita la especificación de propiedades de en conceptos que pueden figuran o no en la notación gráfica. El segundo, *Problems*, muestra los errores arrojados por las validaciones sintácticas y semánticas realizadas. Por último, *Outline* permite tener una vista global del diagrama sobre el cual se está trabajando.

(E) **ODP** (*Organization Design Pattens*, actualmente en desarrollo). La idea detrás de la creación de esta vista es brindar la posibilidad de registrar patrones de diseños que puedan presentarse en el modelo como así también la posibilidad de utilizar dichos patrones en nuevos diseños.

A.1. Plataformas

Para el desarrollo de Janeiro se requirió de un conjunto de frameworks que permiten el desarrollo de la aplicación. Entre ellas están: Eclipse RCP, EMF y GMF. Las mismas serán explicadas brevemente a continuación.

La base de Janeiro está construida usando RCP dada las ventajas que ofrece dicha plataforma para el desarrollo de herramientas integradas. Provee un entorno rico en opciones y debido a su popularidad posee una importante comunidad que le brinda soporte. También destacamos su desarrollo basado en el concepto de *plugins* que permite una alta modularización de la aplicación. Un plugin es definido como una unidad de modularidad, de hecho todo en Eclipse es desarrollado bajo este concepto. Básicamente, un plugin es autodescriptivo y explicita una lista de los otros plugins a los cuales depende para un funcionamiento adecuado. El framework a su vez, nos ofrece una manera clara y sencilla de gestionarlos para construir aplicaciones complejas y funcionales.

El segundo framework en importancia es EMF, parte del proyecto denominado Eclipse Modeling Project[3] (EMP). Es una librería que permite a los expertos construir y personalizar sus propias herramientas basadas en un metamodelo de datos estructurados denominado *Ecore*. Un *Ecore* simplifica la definición de un metamodelo facilitando la programación requerida para la implementación de un Lenguaje Específico del Dominio (o Domain Specific Language). Adicionalmente, el uso de EMF permite el modelado de jerarquías donde un modelo es metamodelo de otro. Los conceptos usados en este framework son mucho más simples (abstractos) que aquellos definidos por el Object Management Group[4] (OMG). Dicha característica tiene como finalidad abarcar una vasta variedad de situaciones, además de obtener rápidamente un código ejecutable. Algunos de los elementos más importantes del *Ecore* son: *EPackage* (paquete), *EClass* (clase), *EAttribute* (atributo), *EDataType* (tipo de dato), como así también la posibilidad de representar a través del concepto de *EReference* (referencia) las relaciones de agregación y asociación.

En el párrafo anterior se introdujo la tecnología utilizada para representar los conceptos teóricos de CRIO en un metamodelo implementado. Sin embargo, la manipulación de

[3]Eclipse Modeling Project, www.eclipse.org/modeling/
[4]Object Management Group, www.omg.org

estos *Ecore* no es para nada sencilla por el formato de árbol que utiliza, situación que motivó la creación de un framework específico. Eclipse proporciona desde sus inicios un framework gráfico denominado Graphical Editor Framework (GEF de ahora en más)[5]. No obstante su uso para la construcción de editores gráficos es bastante complejo, puesto que es un entorno genérico e independiente del modelo subyacente; además de poseer una extensa librería para personalizar editores. El problema de la personalización de modelos radica en que gran parte del esfuerzo de implementación se invierte en mecanismos para tratar con ellos, así como en tareas de carga/serialización. El siguiente paso lógico en el desarrollo de los editores fue adoptar el uso de EMF como framework de modelado subyacente. Sin embargo, la integración de EMF y GEF no es trivial y presenta diversas dificultades técnicas. Son por estos motivos que Graphical Modeling Framework (GMF) surge como un framework capaz de unificar ambos entornos, siguiendo una filosofía dirigida por modelos que simplifica significativamente los esfuerzos necesarios para el desarrollo de aplicaciones basadas en metamodelos. El objetivo de GMF es permitir la definición de la metáfora gráfica de los conceptos siguiendo una serie de pasos que incluyen la definición de las herramientas y de las formas que adopten los conceptos. El último paso es realizar un mapeo o establecer una correspondencia entre los conceptos volcados en el *Ecore* y la representación gráfica que desplegarán los distintos editores de la herramienta. También, con GMF es posible definir junto con cada editor una serie de menúes, toolbar, views o cualquier otro elemento que sea necesario para una correcta manipulación de las instancias de los diagramas.

A.2. Infraestructura de desarrollo

Para que un proyecto sea ejecutado exitosamente en una organización no tan solo se debe contar con personal capacitado sino que además se debe disponer de una infraestructura que permita gestionar adecuadamente el proyecto. Dicha infraestructura de desarrollo tiene una marcada importancia estratégica dentro cualquier organización dado que puede limitar o potenciar el crecimiento de los proyectos. Es por esto se debe formar recursos humanos que tengan un conocimiento del potencial y el impacto de la utilización de las nuevas tecnologías en los proyectos.

Los ámbitos de desarrollo de software agrupan y organizan un conjunto de tecnologías que sustentan los proyectos de diversas formas. Un diseño pobre o poco robusto de dicho ambiente no tan solo no aprovecha al máximo los recursos disponibles, sino que podría llevar a la deriva un proyecto de desarrollo al punto de producir un sistema de difícil mantenimiento, fuera del presupuesto, fuera de plazos de entrega, que no cumple

[5] https://www.eclipse.org/gef/

con los objetivos de calidad acordados o incluso el fracaso total. La elección de dichos componentes no es una tarea trivial sino que debe ser una actividad meticulosa y detallada dado que debe existir entre estos, además de las prestaciones individuales, una integración armoniosa y fluida para evitar cualquier tipo de inconvenientes.

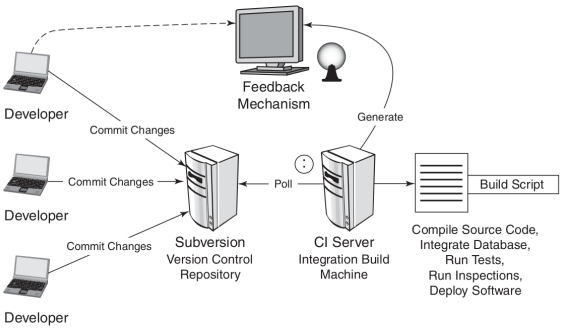

FIGURA A.2: Infraestructura de desarrollo.

En ella mencionamos: (i) el sistema de control de versiones, que permite llevar un histórico de las modificaciones que se están realizando en la aplicación. Para nuestro desarrollo hemos elegido al sistema denominado GIT, dado que una de sus características más importantes es el versionado distribuido. (ii) El sistema de compilación y gestión de dependencias Maven el cual asegura una unificación de los criterios de compilación y empaquetado de software como así también garantiza que todos los colaboradores trabajen sobre las mismas versiones de las dependencias. (iii) las técnicas de integración continua establecen un conjunto de buenas prácticas a seguir para incrementar las posibilidades de éxito en el desarrollo de un sistema. Usamos Jenkins para correr los test bien se detectan modificaciones en el código principal y por el sistema de devolución de feedbacks acerca del estado del sistema. Estas y otras características hacen de la integración continua una herramienta imprescindible en la actualidad.

Consideramos que en la actualidad es imposible no contar con algún tipo de infraestructura de desarrollo más allá del tamaño del proyecto y del número de integrantes. Estas infraestructuras permiten a los arquitectos de software gestionar ordenadamente tanto los recursos humanos como la marcha del proyecto llevando a las aplicaciones desarrolladas a un estándar alto de calidad. La necesidad de tal infraestructura está motivada por el objetivo de fomentar y facilitar la participación de desarrolladores externos al centro de investigación en donde fue desarrollada esta tesis.

La necesidad de tal infraestructura está motivada por el objetivo de fomentar y facilitar la participación de desarrolladores externos al centro de investigación en donde fue desarrollada esta tesis.

A.3. Conclusiones

En el presente apéndice se describió lo que es el primer prototipo de la herramienta de desarrollo *Janeiro Studio*. Para alcanzar este objetivo fueron desarrollados tres Ecores para proveer soporte a los cinco diagramas principales de CRIO. El *Diagrama de Descripción del Dominio* y el *Diagrama de Ontología* son soportados cada uno por un Ecore, mientras que el *Diagrama Organizacional*, de *Interacción* y *Comportamiento* son "vistas" que surgen del Ecore restante. Esto significa que fue necesaria una adaptación del metamodelo organizacional CRIO a la tecnología EMF de Eclipse que concluyeron en la definición de los tres Ecores que actualmente posee la herramienta. Es importante destacar que dicha adaptación no es una relación uno a uno sino más bien fue necesario la incorporación de conceptos que habilitan contemplar determinadas situaciones y que surgen como limitaciones propias de la tecnología adoptada (Por ejemplo: no es posible representar un mismo concepto de EMF en dos vistas o diagramas diferentes). Estos cinco diagramas en su conjunto, permiten abordar la primera de las tres fases del proceso de desarrollo de ASPECS; denominada *Dominio del Problema*.

Para el desarrollo de Janeiro fue necesario una serie de framework que permiten construir una aplicación de calidad como así también ayudan a reducir los tiempos de desarrollo. Como base del proyecto, Eclipse RCP que simplifica el desarrollo de aplicaciones ricas y robustas haciendo uso intensivo de los conceptos de reutilización. Elementos como los menues (principal y contextuales), barra de botones, herramientas de ayuda, vistas, paneles y demás conceptos, son posibles de utilizar sin tener que definirlos desde cero. El segundo framework fundamental es EMF. El mismo permite la definición de un metamodelo, simplificando la programación subyacente del mismo como así también su rápida implementación. Por último, GMF es un framework que permite la definición de la metáfora gráfica que tendrán cada uno de los elementos declarados en el metamodelo EMF.

Finalmente, se ha montado una infraestructura de desarrollo dedicada para Janeiro Studio. La misma consta de un "build tool" denominada Maven, el sistema de control de versiones GIT y el mecanismos de integración contínua Jankins. Todas estas herramientas/técnicas nos ayuda a una correcta gestión del proyecto a su vez que permite que personas ajenas al grupo de desarrollo, o porque no al grupo de investigación, puedan participar en el desarrollo de Janeiro.

Apéndice B

Epsilon Validation Language

Dentro del campo de los lenguajes tanto de validación como de expresividad de modelos, el seleccionado para cumplir con el objetivo propuesto es *Epsilon Validation Language* (EVL de ahora en más). El mencionado lenguaje es parte importante de una gran familia de lenguajes y herramientas que conforman el framework Epsilon[1]. Estos lenguajes permiten la validación, transformación, migración, merging y refactoreo de modelos como así también la generación de código. A continuación se describirán brevemente los lenguajes que conforman el universo Epsilon:

- Epsilon Object Languaje (EOL): es un lenguaje de programación imperativo que provee un conjunto reusable de elementos genéricos para la gestión de modelos y arriba de él se pueden implementar lenguajes para tareas especificas. Además, es posible utilizar el lenguaje de forma *standalone* para la creación, consulta y modificación de modelos EMF.
- Epsilon Transformation Language (ETL): es un lenguaje de transformación de modelo a modelo basado en reglas. ETL fue diseñado como un lenguaje híbrido (mezcla imperativo y declarativo). El mismo permite el manejo de un número arbitrario de modelos de entrada transformándolos en un número, también arbitrario, de modelos de salida.
- Epsilon Comparison Language (ECL): es un lenguaje híbrido que permite definir algoritmos de comparación para determinar correspondencias entre dos modelos, tanto homogéneos como heterogéneos. El objetivo de analizar estas correspondencias es para encontrar elementos que estén involucrados en una relación de interés.
- Epsilon Merging Language (EML): es un lenguaje híbrido basado en reglas para la fusión de modelos heterogéneos u homogéneos. En realidad, este lenguaje reusa

[1]Extensible Platform of Integrated Languaje

la sintaxis y semántica de ETL, extendiendo conceptos específicos para la fusión de modelos.

- Epsilon Wizard Language (EWL): es un lenguaje adaptado para la transformación interactiva de los elementos del modelo seleccionados directamente por el usuario. Además, tiene una alta integración con los editores de EMF y GMF. Este lenguaje es particularmente útil para la automatización de tareas recurrente en la edición de modelos (Ej: refactoreo, aplicación de patrones o la construcción de sub-árboles consistentes de elementos similares).
- Epsilon Generation Language (EGL): es un lenguaje utilizado para la transformación de modelos a varios tipos de artefactos de texto capaz de generar código ejecutable, documentos, imágenes y otros artefactos textuales tomando como entrada los modelos creados.

Este conjunto de lenguajes representa una evolución, que tiene su inicio a partir de un cuidadoso análisis de los lenguajes y frameworks de gestión de modelos existentes en la literatura, en especial de OCL. La novedad es que los lenguajes son independientes de una tecnología en particular y pueden ser usados para administrar modelos de diversas tecnologías tales como MOF, EMF y XML. Además, también son independientes del metamodelo dado que no están ligados a uno específico. De hecho, pueden ser usados tanto como un lenguaje para la gestión de modelos genéricos, como así también usarlo de base para la definición de lenguajes de gestión de modelos específicos de las tareas.

A pesar de que OCL es el lenguaje más utilizado para la validación de modelos, el mismo es inherentemente limitado a las restricciones que se especifican en el contexto de un único modelo. En otras palabras, no puede ser utilizado tal cual para expresar reglas de consistencia a través de diferentes modelos. Por esta y otra razones elegimos EVL en vez de OCL dado que posee diversas ventajas prácticas. Algunas de ellas son:

- Permite al usuario definir restricciones que pueden depender entre sí.
- Es posible definir una secuencia de sentencias permitiendo a los ingenieros descomponer consultas complejas en otras mucho más simples. La descomposición de consultas promueve la modularización facilitando la lectura y el mantenimiento de las reglas.
- Permite especificar mensajes mucho más personalizados cuando un invariante no se satisfizo. Esta es la característica más significativa dado que el mecanismo de *feedback* provisto por OCL es limitado ya que solo muestra el nombre del invariante.

- Además de personalizar los mensajes descripto en el ítem anterior, en EVL es posible la diferenciación por tipos de *feedback*. Dos tipos son posibles: *error* y *warning*. Por un lado, error, indica la violación de un invariante deteniendo la ejecución de la validación e indicando que el problema es crítico y debe ser enmendado inmediatamente. Por el otro, *warning*, indica que una restricción no se cumplió pero no detiene la ejecución del sistema.
- Puede ser usado para expresar restricciones inter-modelos. En otras palabras, es posible definir reglas que cubren múltiples modelos.
- Permite que los usuarios puedan definir entradas y salidas en las operaciones.

La combinación de EVL, junto a las instancias de los *Ecores* de EMF y el framework de Eclipse RCP, permite construir modelos altamente expresivos en los cuales los diseñadores pueden formular propiedades adicionales (validaciones) que no podrían ser posibles usando solo la notación gráfica.

Para ilustrar como es el funcionamiento de EVL, vamos a considerar el ejemplo de un grafo dirigido. Un metamodelo sencillo para el grafo dirigido es presentado en la Figura B.1. En el ejemplo, el grafo está compuesto de *GraphElements* pudiendo ser nodo o arista. Un nodo es identificado con una etiqueta junto a un conjunto de aristas de entradas y salidas. Una arista tiene un nodo que representa el origen y otro el destino junto con el peso de la arista.

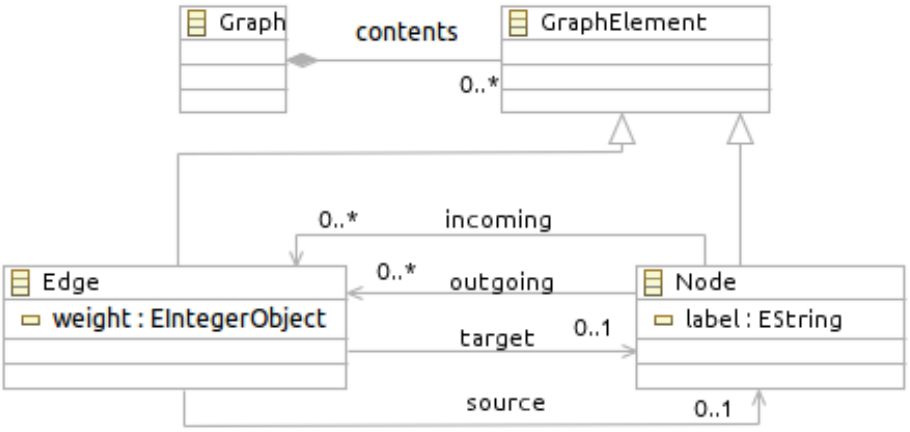

FIGURA B.1: Metamodelo para modelar un grafo dirigido

Si se desea definir una regla para asegurar que el grafo no contenga ningún ciclo podemos hacerlo tal como se muestra en el Listado B.1. En la línea 1 se define el contexto de la regla (ej. Node). Dentro, un conjunto de restricciones para el modelo puede ser definido, tal como se hace en la linea 2. La palabra reservada *constraint* en EVL representa un

error crítico que invalida el modelo. En cambio, la palabra *critique* puede ser usado para expresar errores no críticos que no invalidan el modelo, pero que deberían se abordados por el diseñador en algún punto. Tanto *constraint* como *critique* representan invariantes; el cuerpo del invariante es definido dentro de *check* (línea 3). Si el invariante no se cumple, el sistema muestra un mensaje de error que fue definido por el usuario (línea 4). Incluso, podemos usar la sección *fix* para definir un conjunto de acciones en Epsilon Object Language de forma de resolver este problema de los ciclos. Mayor información de EVL y el resto del ecosistema Epsilon se puede encontrar en su página oficial[2].

```
1  context Node {
2    constraint NotInCycle {
3      check : not self.allIncoming().includes(self)
4      message : "Node " + self.label + " is part of a directed cycle"
5    }
6  }
7
8  operation Node allIncoming(visited : Set) : Set {
9    for (n in self.incoming.collect(e|e.source)) {
10     if (not visited.includes(n)) {
11       visited.add(n);
12       visited.addAll(n.allIncoming(visited));
13     }
14   }
15   return visited;
16 }
```

LISTADO B.1: Regla EVL para un Grafo Dirigido

[2] http://www.eclipse.org/epsilon

Bibliografía

[1] Massimo Cossentino, Nicolas Gaud, Vincent Hilaire, Stéphane Galland, and Abder Koukam. Aspecs: an agent-oriented software process for engineering complex systems. *Autonomous Agents and Multi-Agent Systems*, 20(2):260 – 304, 2010. ISSN 1387-2532. doi: 10.1007/s10458-009-9099-4.

[2] Massimo Cossentino, Vincent Hilaire, Ambra Molesini, and Valeria Seidita. *Handbook on Agent-Oriented Design Processes*. Springer, 2014 edition edition, 2014. ISBN 3642399746.

[3] Pedro Araujo and Sebastián Rodríguez. Janeiro studio. In *1er Congreso Nacional de Ingeniería Informática/Sistemas de Información*, Córdoba, Argentina, 2013.

[4] Pedro Araujo and Sebastian Rodriguez. Enfoque para la validación sintáctica de modelos organizacionales de sistemas multiagentes. *Revista de Ciencia y Tecnología - Universidad de Palermo*, 14, 2014. ISSN 1850-0870.

[5] Pedro Araujo, Diego Lizondo, Sebastian Rodriguez, and Vincent Hilaire. An approach for organizational design smells identification within multi-agent systems. page 10, Buenos Aires, 2015. Springer.

[6] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley Press, 2009.

[7] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2), 1995.

[8] Gerhard Weiss. *Multiagent Systems*. Intelligent Robotics and Autonomous Agents. MIT Press, second edition, 2013. ISBN 0262018896.

[9] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000. ISSN 1387-2532, 1573-7454. doi: 10.1023/A:1010071910869. URL http://link.springer.com/article/10.1023/A%3A1010071910869.

[10] Richard Evans, Paul Kearney, Giovanni Caire, F Garijo, J Gomez Sanz, J Pavon, F Leal, P Chainho, and P Massonet. Message: Methodology for engineering systems of software agents. *EURESCOM, EDIN*, pages 0223–0907, 2001.

[11] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From agents to organizations: An organizational view of multi-agent systems. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *Agent-Oriented Software Engineering IV*, number 2935 in Lecture Notes in Computer Science, pages 214–230. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-20826-6, 978-3-540-24620-6. URL http://link.springer.com/chapter/10.1007/978-3-540-24620-6_15.

[12] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of the 3rd International Conference on Multi Agent Systems*, ICMAS '98, pages 128–, Washington, DC, USA, 1998. IEEE Computer Society.

[13] Sebastian Rodriguez, Nicolas Gaud, Vincent Hilaire, Stéphane Galland, and Abderrafiâa Koukam. An analysis and design concept for self-organization in holonic multi-agent systems. In Sven A. Brueckner, Salima Hassas, Márk Jelasity, and Daniel Yamins, editors, *Engineering Self-Organising Systems*, number 4335 in Lecture Notes in Computer Science, pages 15–27. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-69867-8, 978-3-540-69868-5. URL http://link.springer.com/chapter/10.1007/978-3-540-69868-5_2.

[14] Bryan Horling and Victor Lesser. A survey of multi-agent organizational paradigms. *Knowl. Eng. Rev.*, 19(4):281–316, 2004. ISSN 0269-8889. doi: 10.1017/S0269888905000317. URL http://dx.doi.org/10.1017/S0269888905000317.

[15] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. *Developing Multiagent Systems: The Gaia Methodology*. 2003.

[16] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos, and Anna Perini. Tropos: An agent-oriented software development methodology. 2003.

[17] JuanC. Garcia-Ojeda, ScottA. DeLoach, Robby, WalamitienH. Oyenan, and Jorge Valenzuela. O-mase: A customizable approach to developing multiagent development processes. In Michael Luck and Lin Padgham, editors, *Agent-Oriented Software Engineering VIII*, volume 4951 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-79487-5.

[18] Juan Pavón and Jorge Gómez-Sanz. Agent oriented software engineering with ingenias. In Vladimír Mařík, Michal Pěchouček, and Jörg Müller, editors, *Multi-Agent Systems and Applications III*, number 2691 in Lecture Notes in Computer Science, pages 394–403. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40450-7, 978-3-540-45023-8. URL http://link.springer.com/chapter/10.1007/3-540-45023-8_38.

[19] M. Cossentino and C. Potts. *PASSI: a Process for Specifying and Implementing Multi-Agent Systems Using UML*. 2002.

[20] Lin Padgham and Michael Winikoff. Prometheus: A methodology for developing intelligent agents. In Fausto Giunchiglia, James Odell, and Gerhard Weiß, editors, *Agent-Oriented Software Engineering III*, number 2585 in Lecture Notes in Computer Science, pages 174–185. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-00713-5, 978-3-540-36540-2. URL http://link.springer.com/chapter/10.1007/3-540-36540-0_14.

[21] David Isern, David Sánchez, and Antonio Moreno. Organizational structures supported by agent-oriented methodologies. *J. Syst. Softw.*, 84(2):169–184, 2011. ISSN 0164-1212. doi: 10.1016/j.jss.2010.09.005. URL http://dx.doi.org/10.1016/j.jss.2010.09.005.

[22] UML. Unified Modeling Language (UML) Superstructure V 2.4.1, au 2011. Version 2.4.1.

[23] Bernhard Bauer, Jörg P Müller, and James Odell. Agent uml: A formalism for specifying multiagent software systems. *International journal of software engineering and knowledge engineering*, 11(03):207–230, 2001.

[24] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. Moise+: Towards a structural, functional, and deontic model for mas organization. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1*, AAMAS '02, pages 501–502, New York, NY, USA, 2002. ACM. ISBN 1-58113-480-0. doi: 10.1145/544741.544858. URL http://doi.acm.org/10.1145/544741.544858.

[25] Marc Esteva, David de la Cruz, and Carles Sierra. Islander: An electronic institutions editor. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 3*, AAMAS '02, pages 1045–1052, New York, NY, USA, 2002. ACM. ISBN 1-58113-480-0. doi: 10.1145/545056.545069. URL http://doi.acm.org/10.1145/545056.545069.

[26] Virginia Dignum, Frank Dignum, and John-Jules Meyer. An agent-mediated approach to the support of knowledge sharing in organizations. *Knowl. Eng. Rev.*, 19(2):147–174, 2004. ISSN 0269-8889. doi: 10.1017/S0269888904000244. URL http://dx.doi.org/10.1017/S0269888904000244.

[27] B. Horling and V. Lesser. Using odml to model multi-agent organizations. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 72–80, 2005. doi: 10.1109/IAT.2005.139.

[28] Virginia Dignum, Javier Vázquez-Salceda, and Frank Dignum. Omni: Introducing social structure, norms and ontologies into agent organizations. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, number 3346 in Lecture Notes in Computer Science, pages 181–198. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-24559-9 978-3-540-32260-3. URL http://link.springer.com/chapter/10.1007/978-3-540-32260-3_10.

[29] Danny Weyns, Robrecht Haesevoets, and Alexander Helleboogh. The macodo organization model for context-driven dynamic agent organizations. *ACM Trans. Auton. Adapt. Syst.*, 5(4):16:1–16:29, 2010. ISSN 1556-4665. doi: 10.1145/1867713.1867717. URL http://doi.acm.org/10.1145/1867713.1867717.

[30] Vincent Hilaire. *Vers une approche de spécification, de prototypage et de vérification de Systèmes Multi-Agents*. Phd thesis, Université de Technologie de Belfort-Montbéliard, 2010.

[31] Sebastian Rodriguez. *From analysis to design of holonic multi-agent systems: A framework, methodological guidelines and applications*. PhD thesis, Universitè de Technologie de Belfort-Montbèliard and Universitè de Franche-Comptè, 2005.

[32] Nicolas Gaud. *Holonic Multi-Agent Systems: From the analysis to the implementation. Metamodel, Methodology and Multilevel simulation*. PhD thesis, Université de Technologie de Belfort-Montbéliard, Belfort. France, 2007.

[33] Juan Pablo Gruer, V. Hilaire, A. Koukam, and P. Rovarini. Heterogeneous formal specification based on object-z and statecharts: semantics and verification. *Journal of Systems and Software*, 70(1-2):95–105, 2004. ISSN 0164-1212. doi: 10.1016/S0164-1212(02)00161-9. URL http://www.sciencedirect.com/science/article/pii/S0164121202001619.

[34] Roger Duke, Gordon Rose, and Graeme Smith. Object-z: A specification language advocated for the description of standards. *Computer Standards & Interfaces*, 17

(5-6):511–533, 1995. ISSN 0920-5489. doi: 10.1016/0920-5489(95)00024-O. URL http://www.sciencedirect.com/science/article/pii/092054899500024O.

[35] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. ISSN 0167-6423. doi: 10.1016/0167-6423(87)90035-9. URL http://www.sciencedirect.com/science/article/pii/0167642387900359.

[36] Ian Sommerville. *Software Engineering*. Addison-Wesley, Boston, 9 edition edition, 2010. ISBN 9780137035151.

[37] Alexander Pokahr and Lars Braubach. A survey of agent-oriented development tools. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini, editors, *Multi-Agent Programming:*, pages 289–329. Springer US, 2009. ISBN 978-0-387-89298-6, 978-0-387-89299-3. URL http://link.springer.com/chapter/10.1007/978-0-387-89299-3_9.

[38] Juan C. Garcia-Ojeda, Scott A. DeLoach, and Robby. agenttool process editor: Supporting the design of tailored agent-based processes. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 707–714, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8. doi: 10.1145/1529282.1529430. URL http://doi.acm.org/10.1145/1529282.1529430.

[39] Luca Cernuzzi and Franco Zambonelli. Gaia4e: A tool supporting the design of mas using gaia. In *ICEIS (4)*, pages 82–88. Citeseer, 2009.

[40] John Thangarajah, Lin Padgham, and Michael Winikoff. Prometheus design tool. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 127–128, New York, NY, USA, 2005. ACM. ISBN 1-59593-093-0.

[41] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform*. The Eclipse Series. Pearson Education, 2 edition, 2010. ISBN 0321603788.

[42] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003. ISBN 0131425420.

[43] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. ISBN 0-07-249668-1.

[44] Mario Barbacci, Mark H Klein, Thomas A Longstaff, and Charles B Weinstock. Quality attributes. Technical report, Carnegie Mellon University, 1995.

[45] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. ISBN 0-201-19930-0.

[46] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1 edition, 1999. ISBN 0201485672. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201485672.

[47] William C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1 edition, 2003. ISBN 0321109295.

[48] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings*, pages 381–384, 2003. doi: 10.1109/ICSM.2003.1235447.

[49] F.A. Fontana and M. Zanoni. On investigating code smells correlations. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 474–475, 2011. doi: 10.1109/ICSTW.2011.14.

[50] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976. ISSN 0018-8670. doi: 10.1147/sj.153.0182.

[51] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 47–56, New York, NY, USA, 1999. ACM. ISBN 1-58113-238-7. doi: 10.1145/320384.320389. URL http://doi.acm.org/10.1145/320384.320389.

[52] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings*, pages 97–106, 2002. doi: 10.1109/WCRE.2002.1173068.

[53] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 87–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1339-5. URL http://dl.acm.org/citation.cfm?id=823457.824038.

[54] CJ Kasper and MW Godfrey. "cloning considered harmful" considered harmful: Patterns of cloning in software". *Empirical Software Engineering*, 13(6):645–692, 2008.

[55] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: 10.1145/1081706.1081737. URL http://doi.acm.org/10.1145/1081706.1081737.

[56] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, pages 227–236, 2008. doi: 10.1109/ICSM.2008.4658071.

[57] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070547. URL http://dx.doi.org/10.1109/ICSE.2009.5070547.

[58] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2):127–139, 2003. ISSN 0164-1212. doi: 10.1016/S0164-1212(02)00054-7. URL http://www.sciencedirect.com/science/article/pii/S0164121202000547.

[59] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2): 129–143, 2004. ISSN 0164-1212. doi: 10.1016/S0164-1212(03)00240-1. URL http://www.sciencedirect.com/science/article/pii/S0164121203002401.

[60] S. Olbrich, D.S. Cruzes, V. Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *3rd International Symposium on Empirical Software Engineering and Measurement, 2009. ESEM 2009*, pages 390–400, 2009. doi: 10.1109/ESEM.2009.5314231.

[61] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 181–190, 2011. doi: 10.1109/CSMR.2011.24.

[62] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120–1128, 2007. ISSN 0164-1212. doi: 10.1016/j.jss.2006.10.018. URL http://www.sciencedirect.com/science/article/pii/S0164121206002780.

[63] M. D'Ambros, Alberto Bacchelli, and M. Lanza. On the impact of design flaws on software defects. In *2010 10th International Conference on Quality Software (QSIC)*, pages 23–31, 2010. doi: 10.1109/QSIC.2010.58.

[64] S. Counsell, H. Hamza, and R.M. Hierons. The #x2018;deception #x2019; of code smells: An empirical investigation. In *2010 32nd International Conference on Information Technology Interfaces (ITI)*, pages 683–688, 2010.

[65] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, Laurence Duchien, and Alban Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 22(3-4):345–361, 2009. ISSN 0934-5043, 1433-299X. doi: 10.1007/s00165-009-0115-x. URL http://link.springer.com/article/10.1007/s00165-009-0115-x.

[66] E.-H. Alikacem and H.A. Sahraoui. A metric extraction framework based on a high-level description language. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, 2009. SCAM '09*, pages 159–167, 2009. doi: 10.1109/SCAM.2009.27.

[67] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*, pages 350–359, 2004. doi: 10.1109/ICSM.2004.1357820.

[68] M.J. Munro. Product metrics for automatic identification of "bad smell"design problems in java source-code. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 15–15, 2005. doi: 10.1109/METRICS.2005.38.

[69] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, 2012. doi: 10.1109/ICSM.2012.6405287.

[70] Yuepu Guo, C. Seaman, Nico Zazworka, and F. Shull. Domain-specific tailoring of code smells: an empirical study. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 167–170, 2010. doi: 10.1145/1810295.1810321.

[71] Isela Macia Bertran. Detecting architecturally-relevant code smells in evolving software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1090–1093, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1986003. URL http://doi.acm.org/10.1145/1985793.1986003.

[72] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *9th International Conference on Quality Software, 2009. QSIC '09*, pages 305–314, 2009. doi: 10.1109/QSIC.2009.47.

[73] D.I.K. Sjoberg, A. Yamashita, B.C.D. Anda, A. Mockus, and T. Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.89.

[74] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 44–53, New York, NY, USA, 1996. ACM. ISBN 0-89791-795-2. doi: 10.1145/231379.231389. URL http://doi.acm.org/10.1145/231379.231389.

[75] Daniel Jackson. *Aspect: A Formal Specification Language for Detecting Bugs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1992.

[76] David L. Detlefs. An overview of the extended static checking system. In *In Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, 1995.

[77] SMALLLINT. `http://c2.com/cgi/wiki?SmallLint`.

[78] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39 (12):92–106, 2004. ISSN 0362-1340. doi: 10.1145/1052883.1052895. URL http://doi.acm.org/10.1145/1052883.1052895.

[79] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. Saber: Smart analysis based error reduction. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 243–251, New York, NY, USA, 2004. ACM. ISBN 1-58113-820-2. doi: 10.1145/1007512.1007545. URL http://doi.acm.org/10.1145/1007512.1007545.

[80] PMD. `https://pmd.github.io/`.

[81] CHECKSTYLE. `http://checkstyle.sourceforge.net/`.

[82] FXCOP. `https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx`.

[83] HAMMURAPI. `http://sourceforge.net/projects/hammurapi/`.

[84] CROCOPAT. `http://www.sosy-lab.org/ dbeyer/CrocoPat/`.

[85] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007. ISSN 1433-2779, 1433-2787. doi: 10.1007/s10009-007-0044-z. URL http://link.springer.com/article/10.1007/s10009-007-0044-z.

[86] Hao Chen and David Wagner. Mops: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 235–244, New York, NY, USA, 2002. ACM. ISBN 1-58113-612-9. doi: 10.1145/586110.586142. URL http://doi.acm.org/10.1145/586110.586142.

[87] InCode. `www.intooitus.com/products/incode`.

[88] Ali Murat Tiryaki, Erdem Eser Ekinci, and Oguz Dikenelli. Refactoring in multi agent system development. In Ralph Bergmann, Gabriela Lindemann, Stefan Kirn, and Michal Pěchouček, editors, *Multiagent System Technologies*, number 5244 in Lecture Notes in Computer Science, pages 183–194. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-87804-9, 978-3-540-87805-6. URL http://link.springer.com/chapter/10.1007/978-3-540-87805-6_17.

[89] Ali Murat Tiryaki, Sibel Öztuna, Oguz Dikenelli, and Riza Cenk Erdur. Sunit: A unit testing framework for test driven development of multi-agent systems. In Lin Padgham and Franco Zambonelli, editors, *Agent-Oriented Software Engineering VII*, number 4405 in Lecture Notes in Computer Science, pages 156–173. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-70944-2, 978-3-540-70945-9. URL http://link.springer.com/chapter/10.1007/978-3-540-70945-9_10.

[90] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon object language (eol). In *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA'06, pages 128–142, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35909-5 978-3-540-35909-8. doi: 10.1007/11787044_11. URL http://dx.doi.org/10.1007/11787044_11.

[91] N. Majorel Padilla, A. Décima, A. Will, S. Rodriguez, and O. Diez. Optimization of sugar cane transportation in tucuman using multiagent systems. *Iberoamerican Journal of Industrial Engineering*, 3:103–119, 2011.

[92] Agustín Décima, Nicolás Majorel Padilla, Adrián Will, Sebastián Rodríguez, and Oscar Diez. Optimizacion del transporte de caña de azúcar utilizando sistemas

multiagentes y algortimos genéticos grouping. In *Mecánica Computacional, Volume XXX. Number 32. Industrial Applications (A)*, 2011.

[93] Massimo Cossentino, Carmelo Lodato, Salvatore Lopes, Patrizia Ribino, and Valeria Palermo. Metrics for evaluating modularity and extensibility in hmas systems. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '15, pages 1061–1069, Richland, SC, 2015. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-1-4503-3413-6. URL http://dl.acm.org/citation.cfm?id=2772879.2773286.

[94] Luca Cernuzzi and Gustavo Rossi. On the evaluation of agent oriented modeling methods. In *IN PROCEEDINGS OF AGENT ORIENTED METHODOLOGY WORKSHOP*, pages 21–30, 2002.

[95] Iván García-Magariño, Massimo Cossentino, and Valeria Seidita. A metrics suite for evaluating agent-oriented architectures. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 912–919, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7. doi: 10.1145/1774088.1774278. URL http://doi.acm.org/10.1145/1774088.1774278.