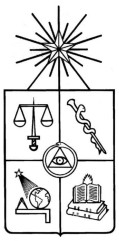


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

TEXTRAM, UN LENGUAJE DE ESPECIFICACIÓN DE DOMINIO TEXTUAL, PARA EL DESARROLLO Y MANEJO DE MODELOS RAM

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN TECNOLOGÍA DE LA INFORMACIÓN

MAREL JOSUÉ OLIVA RODRÍGUEZ

PROFESOR GUÍA:

JOHAN FABRY

MIEMBROS DE LA COMISIÓN:

SERGIO OCHOA DELORENZI
ÉRIC TANTER
PEDRO ROSSEL CID

SANTIAGO DE CHILE

2015

Resumen

La programación orientada a aspectos (AOP) sirve para separar las preocupaciones transversales desde la perspectiva de implementación de un sistema de software. El modelamiento orientado a aspectos (AOM) ofrece una alternativa para representar las preocupaciones transversales en niveles más altos de abstracción, en etapas anteriores a la implementación del software.

"Reusable Aspect Models" (RAM) es un enfoque de AOM, que permite el diseño detallado de un sistema de software a través de módulos llamados "aspect models". La representación gráfica de RAM está inspirada por UML; los aspectos son definidos estáticamente a través de diagramas de clases y dinámicamente por medio de diagramas de estado y de secuencia. La consistencia entre modelos está asegurada por RAM, gracias a la jerarquía de modelos y estructuras de dependencias. TouchRAM es una herramienta gráfica para el modelado de aspectos RAM. Actualmente TouchRAM continúa en desarrollo y es la única herramienta disponible para la creación de RAMs. TouchRAM no está disponible para el público general, dificultando su adopción en otros proyectos de software.

El objetivo de la presente tesis es presentar TextRAM, un modelador textual para la definición de modelos RAM. TextRAM se presenta como una alternativa a TouchRAM. El código fuente de TextRAM está disponible para su descarga y su implementación pretende estimular el desarrollo e investigación de RAM. Los modelos resultantes de TextRAM, podrán ser utilizados por TouchRAM y viceversa. Los usuarios de RAM, podrán elegir entre un modelamiento gráfico con TouhRAM o de un modelamiento textual con TextRAM. Los objetivos de TextRAM son: (1) definición de una abstracción adecuada para la sintaxis concreta textual, (2) implementar una extensión de Eclipse que permita la edición de modelos con la sintaxis concreta de TextRAM, (3) tranformación de modelos desde TextRAM a TouchRAM y viceversa, (4) Aplicar las validaciones semánticas alineadas a las reglas de TouchRAM, (5) demostrar la validez de TextRAM, por medio de un caso de estudio llamado "Slot Machines".

Dedicatoria

A Bueso y a Guzmán.

Agradecimientos

Para iniciar, doy gracias a mi profesor guía Johan Fabry, por darme la oportunidad de desarrollar la tesis en temas que son de mi interés; además de su paciencia y dedicación durante el desarrollo del presente trabajo. También, quiero agradecer a todos los profesores del DCC de la Universidad de Chile, por cumplir satisfactoriamente con las expectativas que tenía del programa; quiero dar gracias, especialmente a los profesores que impartieron las materias que más me cautivaron: Éric Tanter, Verónica Virgilio, Sergio Ochoa, Alexandre Bergel, Didier de Saint Pierre, Marcelo Monsalves, Johan Fabry, Daniel Perovich y Andrés Vignaga. Junto a los profesores, aprovecho la oportunidad de agradecer a Christian Bridevaux, Teresa Huenunguir y Yordy Árevalo, por su refrescante eficacia, orientación y buena disponibilidad.

Muchas gracias a Luis Fernando Zelaya y a Ludwing Nelson Espinal, porque con su apoyo y confianza impulsaron mis estudios de postgrado. Finalmente, el mayor agradecimiento a mis padres por su ejemplo y templanza; los quiero mucho.

Tabla de Contenido

Capítulo 1

Introducción

1.1. Programación orientada a aspectos

La modularidad es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y con bajo acoplamiento. La conexión entre dichos módulos, se realiza en base a los supuestos que los módulos hacen entre sí [6]. Parnas ofrece un criterio para la descomposición de sistemas en módulos: "... inicia con una lista de decisiones difíciles de diseño que son susceptibles al cambio. Entonces, cada módulo es diseñado para ocultar dicha decisión a los demás" [20]. En consecuencia, los sistemas modulares son más fáciles de razonar. Un sistema de software complejo, por lo general, está compuesto por diversas preocupaciones que interactúan entre sí para alcanzar un objetivo. Una preocupación, es un conjunto de información cohesiva que el desarrollador debe representar dentro de un sistema. Lamentablemente, los mecanismos de descomposición jerárquica de los lenguajes de programación orientados a objetos no son suficientes para modularizar todas las preocupaciones interesantes de un sistema complejo [15]. Las preocupaciones transversales atraviesan la modularidad natural de un sistema, causando problemas de dispersión y entrelazamiento de código (por ejemplo el monitoreo y la traza de eventos, manejo de excepciones, gestión de transacciones, etc). El código disperso y entrelazado no se puede reutilizar, además es díficil de mantener y las preocupaciones transversales no se pueden razonar en forma local; esto tiene como consecuencia una complejidad en la evolución del software [15].

En 1997, Kiczales et. al. [15] realizan una propuesta llamada programación orientada a aspectos (AOP) como alternativa a la modularización de las preocupaciones transversales. AOP resuelve los problemas inherentes a la dispersión y entrelazamiento de código, ofreciendo al programador mecanismos para separar preocupaciones transversales en forma limpia y concisa. La AOP define los siguientes conceptos básicos [18]: (1) "*Aspect*": es la definición modular de una preocupación transversal. (2) *Código base*, es toda preocupación que no es transversal. (3) *"Weaving"* es el mecanismo que combina el código base con las preocupaciones transversales para formar el sistema final. (3) *"Join Points"* son los puntos de ejecución identificables dentro del sistema. (4) "*Pointcuts*" es un mecanismo para la selección de un conjunto de "*Join Point*s".

1.2. Modelamiento orientado a aspectos y "Reusable Aspect Models"

1.2.1. Modelamiento orientado a aspectos

Desde la perspectiva de la implementación de un sistema, AspectJ [18] ha surgido como uno de los protagonistas principales en la implementación de AOP. La aplicación de AOP no se limita al nivel de programación, también está presente en las demás etapas del ciclo de vida del desarrollo de software (ingeniería de requisitos, análisis y diseño, etc.). El desarrollo de AOP en ámbitos distintos a la programación, se ha visto estimulado por el surgimiento de paradigmas como la ingeniería de software orientada a modelos (MDSE), que ve al software como un producto de transformaciones y modelos [22]. El enfoque de modelamiento orientado a aspectos (AOM), los aspectos identifican preocupaciones que atraviesan las diferentes vistas de un modelo[12]. Los enfoques existentes de AOM, se han convertido en una estrategia exitosa para la separación y composición de modelos.

1.2.2. "Reusable Aspect Models"

"Reusable Aspect Models" (RAM) [16] es un enfoque simétrico de AOM que permite crear modelos en forma escalable. Los aspectos modelados se llaman "*aspect models*". Los RAM, permiten descomponer una funcionalidad compleja en funcionalidades más simples, por medio de la reutilización de los "aspect models" y la aplicación de una cadena de dependencias que generan una jerarquía de aspectos. La correcta composición de "aspect model*s*" es verificada por mecanismos de consistencia. El proceso "weaving" es el encargado de resolver la dependencia de aspectos y de la generación del modelo final.

Existen dos tipos de dependencias: (1) extensión de modelos y (2) personalización de modelos [4]. Cuando *A extiende* a *B*, la intención del modelador es adicionar nuevas estructuras o comportamientos al modelo *B*. Cuando *A* depende de *B* (personalización de modelos), la intención del modelador es adaptar la estructura y comportamiento de *A* con las características provenientes de *B*.

Gráficamente los "aspect models" son representados por diagramas UML adaptados con la semántica de RAM. Así, la vista estructural se representa por medio de diagramas de clase; la vista dinámica por medio de diagramas de estado y secuencia. Actualmente la única herramienta disponible para modelar aspectos en RAM, es TouchRAM una herramienta gráfica táctil que permite modelar la vista estructural y la vista de dinámica de los modelos de RAM. También permite la definición de la jerarquía de aspectos y soporta el proceso de "weaving" de "aspect models".

1.3. TextRAM

El objetivo de la presente tesis es presentar a TextRAM como una herramienta para el modelado de aspectos en RAM. TextRAM se presenta, no sólo como una alternativa a TouchRAM, su motivación también se encuentra en los siguientes puntos:

- Es una oportunidad para estimular el desarrollo e investigación de RAM. Los usuarios de RAM, disfrutarán de dos herramientas para la definición de "aspect models". Ellos podrán elegir la alternativa más adecuada a sus preferencias: textual o gráfico.
- Comprobar empíricamente lo que Voelter afirma en [32]: "el formato textual es por lo general más usado, escala mejor y construirlo requiere menos esfuerzo", y así ofrecer una herramienta potencialmente más amigable que TouchRAM.

- Para gozar de todos los beneficios de TouchRAM, es necesario poseer una pantalla táctil de grandes dimensiones. Con TextRAM, no se requiere, ya que los modelos se definirán en archivos de texto plano, validados por la herramienta.

- A través de una representación textual, se pretende ofrecer una alternativa de comunicación de los modelos RAM. Actualmente para razonar un modelo RAM, se utilizan diagramas basados en UML; ahora, también se podrá utilizar la sintaxis concreta textual de TextRAM para analizar un modelo RAM.

- TouchRAM no está disponible para el público general, dificultando su adopción en otros proyectos de software [24]. El código fuente de TextRAM está disponible para su descarga en [28].

La tesis de un Magíster en Tecnología de la Información está orientada a la aplicación de los conceptos aprendidos durante el transcurso del programa; dicha tesis, no está orientada a la comprobación de una hipótesis, tal y como lo hacen el Magíster en Ciencias con mención Computación o el Doctorado en Ciencias con mención Computación. La intención del presente documento es describir el proceso de creación de TextRAM y presentar dicha herramienta como una alternativa a TouchRAM.

Actualmente los modelos RAM, solo se pueden definir por medio de TouchRAM. TouchRAM es un DSL gráfico para la definición de modelos RAM. TextRAM se presenta como una alternativa textual para definirs RAMs. Las características que debe soportar TextRAM, son las siguientes:

- Jerarquía de aspectos.

- Definición de la vista estructural.

- Definición de la vista de mensajes.

- Interoperabilidad con TouchRAM (por medio de los modelos RAM).

- Validación semántica de los modelos.

- Generación automática de modelos RAM.

- Generación automática de la representación textual en base a un modelo existente de RAM (ingeniería inversa).

- Exportación de la representación textual a una representación gráfica, por medio del uso de Graphviz [14].

1.3.1. Motivación

La naturaleza textual de TextRAM, ofrece ventajas sobre el DSL gráfico TouchRAM. En [32], Voelter hace una comparación entre las notaciones gráficas y textuales: “el formato textual es por lo general más usado, escala mejor y construirlo requiere menos esfuerzo”. También, los modelos de TextRAM se pueden utilizar como una herramienta efectiva para el razonamiento y comunicación de los modelos RAM; para expresar un modelo de RAM, se puede utilizar su notación textual y para compartirlo solo se necesita de un simple archivo de texto plano. Otra de las motivaciones de TextRAM, es estimular el desarrollo e investigación de RAM.

1.3.2. Implementación

TextRAM es una aplicación generadora de modelos, por tanto, se implementó con los lineamientos dictados por la ingeniería de softwares orientada a modelos (MDSE): dicho paradigma, indica que el software es el resultado de la interacción entre modelos y transformaciones. Los modelos son abstracciones de la realidad, los metamodelos son modelos que describen modelos. El metamodelo de TextRAM está restringido al metamodelo de RAM.

Un modelador textual está compuesto por 3 ingredientes:

1. La sintaxis concreta textual (TCS): notación para expresar la representación de los modelos.
2. La sintaxis abstracta: estructura de datos que persiste la semántica reflejada por instancias de la sintaxis concreta textual.
3. La semántica: describe el significado de los elementos definidos en el lenguaje y el resultado de combinar dichos elementos. La semántica de TextRAM son los conceptos definidos por RAM.

La definición de la TCS, constituye el primer paso para la construcción de TextRAM, con dicha definición se deriva la sintaxis abstracta y la gramática (reglas formales de transformación de instancias de la TCS a instancias de la sintaxis abstracta).

La generación de instancias de TouchRAM, por medio de TextRAM se realiza por una transformación M2M (modelo a modelo). El M2M se realizó a través de la definición de equivalencias entre modelos de TextRAM y TouchRAM. A este proceso se le llamó exportación de modelos desde TextRAM a TouchRAM; la transformación M2M no se limitó a la exportación de modelos, también existe un proceso de importación de modelos TouchRAM a TextRAM, esto permite la reutilización de modelos existentes de TouchRAM y su representación en forma automática en TextRAM.

La integridad semántica de los modelos de TextRAM se garantiza a través de una validación de modelos realizados por TextRAM. Para soportar la jerarquía y reutilización de aspectos, fue necesario crear reglas de importación de modelos y restriccciones de visibilidad entre elementos ("scoping"). El resultado de la importación de modelos TouchRAM a TextRAM es un archivo de texto plano con la representación textual de RAM; los formatadores son programas que se construyerón con el objetivo de dar legibilidad humana a la TCS de TextRAM.

1.3.3. Validación

"Slot Machines" (SM) es un caso de estudio no trivial, con requisitos dispersos en muchas fuentes y con restricciones impuestas por distintos campos de estudio [10]; la relevancia de dicho trabajo es el análisis del comportamiento de interacciones complejas dentro del contexto de AOP, en un proyecto complejo obtenido de la industria. Para validar el funcionamiento de TextRAM, se ha elegido el caso de estudio SM. Una SM es una máquina tragamonedas que funciona al introducirle créditos; el jugador apuesta dichos créditos y si gana, la SM entrega un premio en forma de créditos canjeables en monedas, "tickets" o transferencias electrónicas. Previamente, el equipo de trabajo que construyó RAM identificó y desarrolló los siguientes "aspect models": *Betting, Blockable, CriticalError, Demo, DemoNetworkResolution, Game, InterceptReturnMetering, Map, Metering, Named, NetErrorsArbiter, NetworkedError, NormalError, Observer, Random, ReturningCommand, SMNetwork, SMNetworkArbiter, Singleton, SlotMachineMetering, WeakError, ZeroToManyOrdered*. Para validar el funcionamiento de TextRAM, se desarrollaron todos los "aspect models" mencionados anteriormente.

1.4. Objetivos

Para poder construir TextRAM y cumplir con la generación de modelos conformes a RAM, se han definido los siguientes objetivos:

- Definir una sintaxis concreta textual, que permita la definición de RAMs.
- Implementar una extensión de Eclipse que permita la edición de modelos con la sintaxis concreta textual de TextRAM.
- Implementar la transformación de modelos TextRAM a modelos equivalentes de TouchRAM.
- Implementar la transformación de modelos existentes de TouchRAM a representaciones equivalentes de TextRAM.
- Aplicar validaciones semánticas alineadas a las reglas de TouchRAM.
- Mostrar la validez de TextRAM, a través del caso de estudio "Slot Machines".

1.5. La herramienta TextRAM

TextRAM, es una herramienta textual cuyo objetivo es la creación de modelos conformes a RAM. Por lo tanto, TextRAM soporta las siguientes características: (1) jerarquía de aspectos, (2) vista estructural, (3) vista de mensajes (diagramas de secuencia), (4) generación de instancias válidas de modelos RAM.

1.6. Organización del texto

El presente trabajo, se organiza de la siguiente forma:

- Capítulo 2, describe los fundamentos teóricos necesarios para la construcción de TextRAM. Inicia con los conceptos básicos de los lenguajes de dominio específico (TextRAM es un lenguaje de dominio específico a RAM). Después, se presenta la evaluación de las distintas herramientas para el desarrollo de TextRAM; luego se presenta una intrucción al MDSE y a la AOP, para finalizar con la teoría de RAM.
- Capítulo 3, presenta el metamodelo de TextRAM. Inicia con la definición de las sintaxis concreta textual, luego se presenta la definición de la grámatica y finalmente, se detallan todos los elementos del metamodelo de TextRAM.
- Capítulo 4, explica los detalles de implementación de TextRAM: configuración del lenguaje, conversor de valores, mecanismo de "scoping", validaciones semánticas, formateradores y termina con la descripción de la exportación e importación de modelos RAM en TextRAM por medio de mecanismos de transformación de modelos.
- Capítulo 5, su objetivo es validar el funcionamiento de TextRAM, por medio del caso de estudio "Slot Machines". Inicia con la descripción del caso de estudio, continúa con la identificación de las principales preocupaciones e interacciones, y termina con la implementación de todas los aspectos identificados en la herramienta TextRAM.
- Finalmente, el capítulo 6 presenta las conclusiones del presente trabajo a través de la experiencia obtenida en la construcción de la herramienta; además describe los elementos que hay que tomar en cuenta en futuras evoluciones de TextRAM y dedica la última sección al trabajo futuro.

Capítulo 2

Marco Teórico

La sección 2.1; introduce el concepto de los DSL. Un "language workbench" es una herramienta especializada para la definición de DSLs, la sección 2.2 presenta una lista relevante de "language workbenches" investigados para la implementación de TextRAM. Los DSLs están contenidos dentro un campo de estudio más amplio llamado Ingeniería de Software Orientado a Modelos (MDSE), dicho paradigma se estudia en la sección 2.3. Para la comprensión de RAM es necesario el estudio de la programación orientada a aspectos (ver sección 2.4 en la página 23). La sección2.5, explica los conceptos básicos de RAM y en la sección 2.6, se presenta TouchRAM la única herramienta que existe actualmente para modelar aspectos en RAM.

2.1. Introducción a los Lenguajes de Dominio Específico.

En todas las ramas de la ciencia y la ingeniería, se pueden distinguir dos enfoques: genéricos y específicos. Los enfoques genéricos proporcionan una solución general para los problemas de un área en particular, pero su solución puede no ser óptima. Un enfoque específico, puede proporcionar una mejor solución para un conjunto pequeño de problemas [31]. En las ciencias de la computación, esta dicotomía también existe, con los lenguajes de programación general (GPLs) y los lenguajes de especificación de dominio (DSLs). Un lenguaje de dominio específico (DSL) es un lenguaje de programación o un lenguaje de especificación ejecutable que ofrece, a partir de notaciones y abstracciones adecuadas, un poder de expresividad enfocado y por lo general restringido en un problema de dominio particular [31]. De la definición anterior, se destacan 4 elementos:

1. Lenguaje de programación de computadoras: Un DSL es usado por humanos para instruir a la computadora que realice cierta acción.

2. Naturaleza del lenguaje: Un DSL es un lenguaje de programación, y como tal, debe tener una sensación de fluidez, en donde la expresividad, debe originarse no solo de expresiones individuales, si no que también de la forma en que la mismas pueden ser compuestas como un conjunto.

3. Expresividad limitada: Un lenguaje de programación general proporciona muchas capacidades: soporte de variables, control y abstracción de estructuras. Todo esto es útil, pero es más difícil de aprender y usar. Un DSL soporta, un conjunto mínimo de funcionalidades para soportar un dominio. Por medio de un DSL no se puede construir software que no este enmarcado dentro del dominio del problema.

4. Enfoque en el dominio: Un lenguaje limitado, es solamente útil si tiene un enfoque claro en un dominio pequeño. El enfoque al dominio, es lo que convierte a un lenguaje limitado en algo de real valor.

2.1.1. Lenguajes de programación general vs lenguajes de especificación de dominio

Los GPLs pueden ser utilizados para implementar cualquier artefacto compatible con la máquina de Turing. Entonces, ¿Por qué existen diversos GPLs? mucho depende de la estrategia de ejecución. Por ejemplo, en el lenguaje de programación C es compilado a un código nativo eficiente, mientras que Ruby, en general, es ejecutado por una máquina virtual (una combinación entre compiladores e intérpretes). Las características que ofrecen cada uno de los lenguajes, están optimizadas para las tareas que son relevantes en sus respectivos dominios. Por ejemplo, en C se puede manipular la memoria (aspecto importante para comunicarse con dispositivos de bajo nivel). Por otro lado, en el lenguaje de programación Ruby no se puede manipular memoria, pero gracias al recolector de basura el programador no tiene que preocuparse de ubicar y liberar memoria manualmente.

Incluso en el campo de los GPLs, existen diferentes lenguajes, cada uno exponiendo diferentes funcionalidades, ajustadas a una tarea específica. Entre más específica se vuelve una tarea, es más adecuado el uso de lenguajes especializados. Un DSL, simplemente es un lenguaje que está optimizado para una clase de problemas o un dominio en particular. Los lenguajes de dominio específico, sacrifican cierta flexibilidad para expresar cualquier tipo de programa, en favor de la productividad y precisión de programas relevantes en un dominio en particular.

2.1.2. Ingredientes de un lenguaje de programación

Un DSL y un GPL deben tener los siguientes ingredientes principales [7]:

1. Sintaxis concreta: define la notación que usarán los usuarios para definir los programas. Es la representación específica del lenguaje a modelar.

2. Sintaxis abstracta (AST): es la estructura de datos que persiste la semántica de la información relevante expresada por un programa. Por lo general es un árbol o un grafo. No debe contener detalles de notación.

3. Semántica: describe el significado de los elementos definidos en el lenguaje y el significado de combinar dichos elementos.

La Figura 2.1.1, muestra la relación entre los tres ingredientes: la semántica define el significado de la sintaxis abstracta e indirectamente el de la sintaxis concreta; la sintaxis concreta representa la sintaxis abstracta.

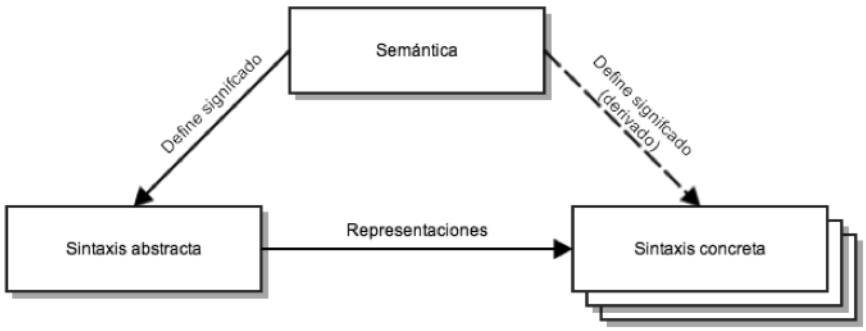


Figura 2.1.1: Ingredientes principales de un lenguaje

2.1.3. Clasificación de los DSLs

Los DSLs se pueden clasificar en dos categorías [13]:

1. DSL externo, es un lenguaje de programación que se construye, desde cero y tiene una infraestructura independiente para el análisis léxico, técnicas de parseo, interpretación y generación de código.

2. DSL interno, es un lenguaje que utiliza la infraestructura de un lenguaje de programación existente, para construir la semántica de especificación de dominio encima de él. (ejemplo: "Rails en Ruby"). Por ende, los DSL internos están embebidos dentro de un lenguaje de programación general. Usualmente, el lenguaje huésped es de tipos dinámicos y la implementación del DSL es basada en meta-programación.

2.1.4. Sintaxis concreta y abstracta

La sintaxis concreta de un lenguaje es la interfaz que utilizan los usuarios para crear programas; la sintaxis abstracta es la representación semántica de dicho lenguaje [32]. La sintaxis abstracta, es una estructura de datos o modelo, que actúa como una API para aplicar herramientas de validación, transformación y generación de código. Existen dos estrategias para el desarrollo de la sintaxis concreta y abstracta[32]:

- Iniciar por la definición de la sintaxis concreta: la sintaxis abstracta es derivada a partir de la sintaxis concreta, ya sea de forma automática o con la ayuda de especificaciones durante la definición de la sintaxis concreta.

- Iniciar por la definición de la sintaxis abstracta: la sintaxis concreta se define a partir de las especificaciones de la sintaxis abstracta.

Para la creación de la sintaxis abstracta, existen dos técnicas [32]:

1. "Parsers": utilizan una definición formal llamada gramática, para poder derivar la sintaxis abstracta a partir de la sintaxis concreta.

2. Proyección: la sintaxis abstracta es generada a partir de acciones realizadas por el usuario en un editor. La sintaxis concreta es dinámica y es generada a partir de los cambios ocurridos en la sintaxis abstracta. La proyección no utiliza una gramática.

Los compiladores tradicionales utilizan "parsers" creados en forma manual; la consecuencia de esto son programas grandes y monolíticos. "Parser Generator" es una técnica mencionada por Fowler [11] que indica cómo el "parser es generado en forma automática en base a la especificación de una gramática. Este es el enfoque que utilizan la mayoría de los compiladores actuales. La ventaja de los generadores de "parsers" es que el modelador no debe ser un experto en la definición del lenguaje (a diferencia de un "parser" construido en forma manual); sin embargo los "parser" hechos a la medida ofrecen un mejor rendimiento.

2.2. Herramientas de implementación

Los "Language Workbenches" son herramientas que soportan la reutilización, composición y definición eficiente de lenguajes, a través de un entorno integrado de desarrollo (IDE) [19]. Existen diversas alternativas:

1. *Enso* (http://www.enso-lang.org): permite el desarrollo de software basado en la interpretación e integración de lenguajes de especificación ejecutables. El proyecto está basado en Ruby. Su objetivo es la exploración de nuevos enfoques para el paradigma MDSE.

2. *Más* (http://www.mas-wb.com): es una herramienta web para la creación de modelos o DSLs. Más, utiliza edición proyeccional para brindar un ambiente amigable a los usuarios no técnicos. La semántica del lenguaje es por medio de *activaciones*, que consistenten en plantillas declarativas para la generación de código. El objetivo de Más, es reducir la barrera de entrada para la creación de lenguajes; Más, busca la adopción rápida del MDSE por parte de otras disciplinas e industrias.

3. *MetaEdit+* (http://www.metacase.com): es un "language workbench" gráfico, maduro para el modelamiento de DSLs. MetaEdit+ reduce la complejidad accidental ya que guía al usuario en la construcción de lenguajes productivos. Estudios empíricos han demostrado la mejora de productividad por parte de los desarrolladores en un factor 5-10, en comparación a la programación convencional. Su licencia es comercial.

4. *MPS* (http://www.jetbrains.com/mps): es un "language workbench" de código libre desarrollado por JetBrains. Su característica más distintiva es el uso de un editor proyeccional que soporta notación textual, simbólica y tabular.

5. *Onion*: es un "language workbench" cuya infraestructura está construida en la plataforma .NET; los objetivos principales de Onion son proveer herramientas para agilizar la creación de DSL en diferentes notaciones (gráficas, textuales proyeccionales) y proveer escabilidad a grandes modelos, a través del particionamiento y sincronización de modelos.

6. *Spoofax* (http://www.spoofax.org): es un "language workbench" basado en el IDE de Eclipse. Está orientado a la construcción de DSL textuales. En Spoofax, los lenguajes son específicados declarativamente por medio de una meta-DSL.

7. *SugarJ* (http://www.sugarj.org): es un lenguaje de programación extensible basado en Java. SugarJ utiliza una extensión del IDE de Spoofax. SugarJ está recomendado para implementaciones de DSL, que disfrutan de los beneficios que puede brindar el uso simultáneo de DSLs internos y externos.

8. *Whole Platform* (http://www.whole.sourceforge.net): es un proyecto maduro que soporta la programación orientada a lenguajes. Se utiliza principalmente en el dominio de las finanzas, debido a su habilidad para definir formatos de datos y "pipelines" para la transformación de los modelos sobre grandes cantidades de datos. "Whole Platform" también es utilizado para interactuar con sistemas legados por medio del uso de formatos de datos basados en la gramáticas definidas por el modelador.

9. *Rascal* (http://www.rascal-mpl.org): es un lenguaje de metaprogramación extensible, con un IDE para el análisis de código fuente y transformaciones. Rascal combina y unifica características encontradas en otras herramientas para la manipulación de código fuente y "language workbenches". También, proporciona una interfaz simple de programación para extender el IDE de Eclipse.

10. *Xtext* (http://www.eclipse.org/xtext): es una marco de trabajo de código libre, para el desarrollo de lenguajes de programación y DSLs. Su diseño está basado en construcciones de compilación probados y patrones que se expresan en muchos de los lenguajes de programación más utilizados.

Para la construcción de TextRAM, se evaluaron dos "Language Workbenches": Xtext y Rascal. A continuación se presenta una descripción de ambas tecnologías, con el objetivo de compararlas y justificar la elección de la herramienta más adecuada para TextRAM.

2.2.1. Xtext

Xtext es una herramienta que sirve para implementar lenguajes de programación (DSLs o GPLs), dentro del marco de trabajo de Eclipse [5]. Xtext es un generador de "parsers" y ofrece toda la infraestructura para definir restricciones, manejo de tipos, "scoping", generación de código, interpretes, "quickfixes" y todas las características de un lenguaje de programación moderno [34]. Para la generación del "parser", Xtext se apoya en ANTLR [2] (pronunciado: "Antler, ANother Tool for Language Recognition"). La definición de la gramática en ANTLR se realiza en un solo archivo. Un modelo semántico [11] es un modelo de objetos en memoria, que un DSL debe completar. El modelo semántico y la sintaxis abstracta son términos equivalentes. Xtext se apoya en EMF Ecore (ver sección 2.3.5) para persistir el modelo semántico en memoria.

La gramática (piedra angular de Xtext) es la definición formal de la sintaxis concreta. El objetivo de la gramática es indicar cómo se mapea la sintaxis concreta con la sintaxis abstracta representada en memoria. El modelo es construido dinámicamente por el "parser" cuando se consume una entrada válida de texto. Xtext permite la reutilización de otras gramáticas.

"Parser"

Las reglas del "parser" son vistas como un plan para la creación de *EObjects* que forman el modelo semántico. Los elementos que están diponibles en las reglas del "parser" son [5]:

1. Grupos
2. Alternativas
3. Palabras claves.
4. Reglas de llamada.

A continuación, se describe cómo algunas expresiones proporcionan "hints" para la construcción directa del AST:

Asignaciones

La creación de instancias se realiza en las asignaciones. El tipo de la entidad asignada es un *EClass* inferido desde el lado derecho de la asignación. El nombre del *EClass* puede ser definido explícitamente o implícitamente utilizando el nombre de la regla. A continuación se presenta una regla que retorna un *EClass* llamado Aspect:

```
Aspect: 'aspect' name=ID '{' structure=Structure '}';
```

La declaración sintáctica de Aspect inicia con la palabra clave "aspect" seguido de la asignación name=ID; el lado derecho de dicha asignación puede ser una llamada a otra regla, una palabra clave, una referencia cruzada o una alternativa. Los tipos involucrados en la asignación deben ser compatibles.

Operadores de asignación

1. El carácter = es usado cuando se espera un solo elemento.
2. El carácter += espera una lista con múltiples valores y adiciona el valor a dicha lista.
3. El carácter ?= espera un elemento de tipo EBoolean y asigna "true" si el elemento del lado derecho fue referenciado.

Referencias cruzadas

Xtext permite la definición de referencias cruzadas dentro de la gramática. Ejemplo:

```
Transition : event=[Event] '=>' state=[State];
```

La regla "Transition" está formada por dos referencias cruzadas, una apuntando a "event" y otra a "state". El texto entre corchetes no se refiere a otra regla, se refiere a un *EClass*. En Ecore, la clase *EReference* tiene una propiedad que indica si la referencia es contenedora o no. Una referencia contenedora es un apuntador a un objeto. Un objeto puede tener un solo contenedor. Las referencias cruzadas son referencias no contenedoras. La determinación de alcance ("scoping") se encarga de resolver este tipo de referencias a través de un identificador único ("index") que almacena toda la información del objeto. La resolución de las referencias cruzadas se hace en una etapa llamada enlazamiento "linking".

Determinación de alcance y enlazamiento

El enlazamiento resuelve las referencias de los símbolos de un lenguaje basado en "parser". La resolución de referencias es por medio de la convención de nombres. La determinación de alcance, es el mecanismo principal detrás de la visibilidad y la resolución de referencias cruzadas. Desde el momento en que el DSL necesita estructurar el código, se necesita una implementación de la definición de alcance.

Por lo general, le determinación de alcance de una referencia cruzada, depende de:

1. El espacio de nombres en donde viven los elementos.
2. La ubicación dentro de la estructura del sitio que contiene la referencia cruzada.
3. Algún aspecto, que no necesariamente es estructural por naturaleza.

Xtext proporciona una API de Java para implementar la determinación de alcance. Con dicha API un desarrollador puede definir varios niveles de alcance [5]:

1. Alcance simple y local.
2. Alcance anidado.
3. Alcance global.

Restricciones

No todos los programas que cumplen con la especificación del metamodelo son válidos. La definición de un lenguaje incluye restricciones que no pueden ser expresadas solamente por el metamodelo. Las restricciones son condiciones booleanas que deben ser evaluadas como verdaderas para poder indicar la validez de un modelo. Un mensaje de error debe ser reportado si la expresión a evaluar es falsa. Se pueden distinguir dos tipos de restricciones [5]:

1. Restricciones que exigen que los elementos estén bien formados. Ejemplo: unicidad de los nombres en una lista de elementos.
2. Sistema de tipos: las reglas de los sistemas de tipos, son diferentes porque verifican la correcta definición de los tipos dentro de un programa. Por ejemplo, el sistema de tipos se asegura que no se pueda asignar un *float* a un *int*.

Las restricciones pueden ser implementadas por cualquier lenguaje o "framework" que pueda consultar un modelo y reportar errores al usuario. La definición eficiente de restricciones, debe contemplar [5]:

1. Instrucciones para navegar y filtrar el modelo.
2. Soporte a funciones de orden superior, para la escritura de algoritmos genéricos y estrategias transversales.
3. Definición declarativa de las restricciones, con asociación a los conceptos del lenguaje (o patrones estructurales).

Inferencia del modelo de Ecore

Los elementos de Ecore son inferidos a partir de las reglas definidas en la gramática de Xtext.

- *EPackage*: es creado después de la directiva "generate". El nombre del paquete se forma a partir de sus parámetros y de su *nsUri*. Se acepta el uso de alias.
- *Enum*: se crea por reglas que utilizan enumeración.
- *EDataType*: por defecto es EString, se crea a partir del tipo de dato de cada regla terminal o de una regla de tipo de dato.
- *EAttribute*: es EBoolean si se utiliza el operador ?= si se utilizan los operadores = o += en las reglas terminales, se crea un atributo con un tipo igual al tipo de retorno de la clase llamada.
- *EReference*: se crea si hay una regla que llama a otra regla, también se crea una EReference por cada asignación de una acción; en ambos casos el tipo será igual al tipo de retorno de la regla llamada.

Transformaciones

La transformación se refiere a la creación de un artefacto a partir de un modelo semántico. Puede ser de dos tipos:

1. Modelo a texto (M2T): los modelos son convertidos a texto (usualmente código fuente, XML, archivos de configuración, etc).
2. Modelo a modelo (M2M): los modelos son transformados en otros modelos (conversión de un modelo semántico a otro equivalente).

Xtend [33] puede ser usado como una alternativa de la transformación M2T.

2.2.2. Rascal

Rascal es un lenguaje de meta-programación extensible que proporciona mecanismos de análisis de código fuente y transformación [30]. El código fuente, es la materia prima de Rascal.

Rascal proporciona una interfaz simple de programación para extender el IDE de Eclipse. Actualmente es utilizado como vehículo de investigación para analizar código existente e implementar DSLs. EASY (Extracción, análisis y síntesis) es el paradigma que propone Rascal, (Figura2.2.1). EASY se puede resumir en los siguientes pasos:

- Extracción: los meta-programas extraen información ("facts") desde un programa de entrada.
- Análisis: los "facts" derivados son computados y la información es enriquecida.

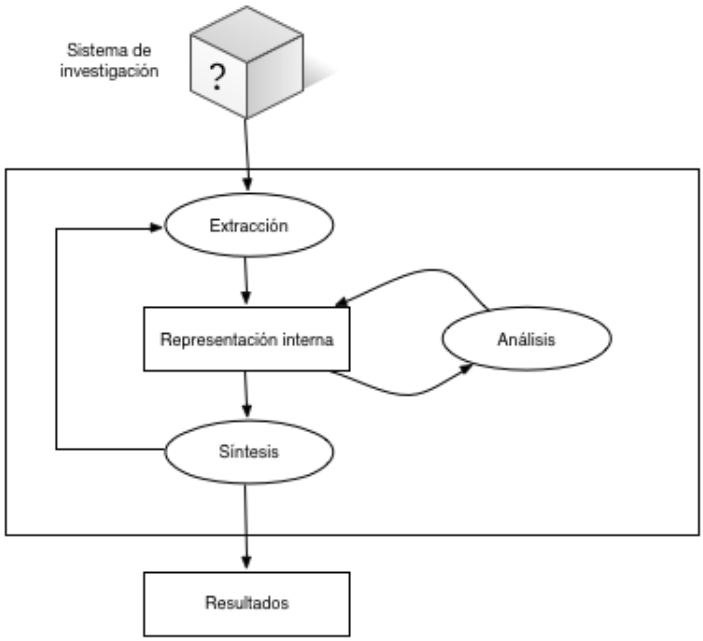


Figura 2.2.1: Paradigma EASY

- Síntesis: finalmente, el meta-programa producirá un tipo de resultado. Ejemplos: transformación de un código fuente (eliminación de código muerto), reportes (estadística del número públicas sin utilizar), visualización.

Un DSL puede ser implementado con RASCAL aplicando el paradigma EASY:

- Extracción: La información de entrada es el código fuente del DSL. El AST puede ser derivado a partir la sintaxis concreta.
- Análisis: el producto de la extracción es el AST. En la etapa de análisis se pueden realizar las validaciones, restricciones, verificaciones de tipo y restricciones de alcance ("scoping").
- Síntesis: cubre las tareas de visualización, generación de código y optimización.

Las principales cualidades de Rascal son:

- Sintaxis familiar y flujo de control: la sintaxis de Rascal es parecida a C, Java, Javascript o C#. Las estructuras de control obedecen las mismas reglas sintácticas de un GPL basado en llaves.
- Datos inmutables: El cambio de un valor, siempre tendrá como resultado otro valor.
- Tipos de datos integrados y "pattern matching": Rascal integra una colección de tipos integrados (integer, boolean, string, real, tuple, list, set, relation,map, parse trees, source locations, date-time). Adicionalmente soporta tipos de datos abstractos (TAD).
- Construcciones específicas de dominio: un ejemplo es la sentencia `visit` que emula el funcionamiento del patrón de diseño Visitor. `visit` puede ser utilizado en el análisis y transformación de código fuente.
- Gramática arbitraria libre de contexto: gracias a esto, Rascal genera en forma automática "parsers" a partir de la gramática.
- Plantillas de cadenas: utilizadas para la generación de código.

- Integración con Java: algunas tareas requieren de cualidades que Rascal no proporciona. Para resolver este problema, es posible utilizar código Java por medio de anotaciones en los encabezados de las funciones de Rascal.
- Integración del IDE con Eclipse: Rascal puede ser instalado en un IDE basado en Eclipse, dando como resultado el soporte a resaltado de sintaxis, visualización del esquema del código, visualización interactiva y también tiene un REPL ("Read-Eval-Print-Loop").

2.2.3. Similitudes entre Xtext y Rascal

Xtext y Rascal tienen la denominación de un "Language workbench", esto es una herramienta que proporciona mecanismos de alto nivel para la implementación de lenguajes específicos a dominio [9]. Diversos "languages workbenches" son estudiados y comparados en una competición anual llamada "Language Workbench Challenge" [19]. La Figura 2.2.2 muestra la base de competición de un "language workbench" y las características mínimas que deben soportar dichas herramientas.

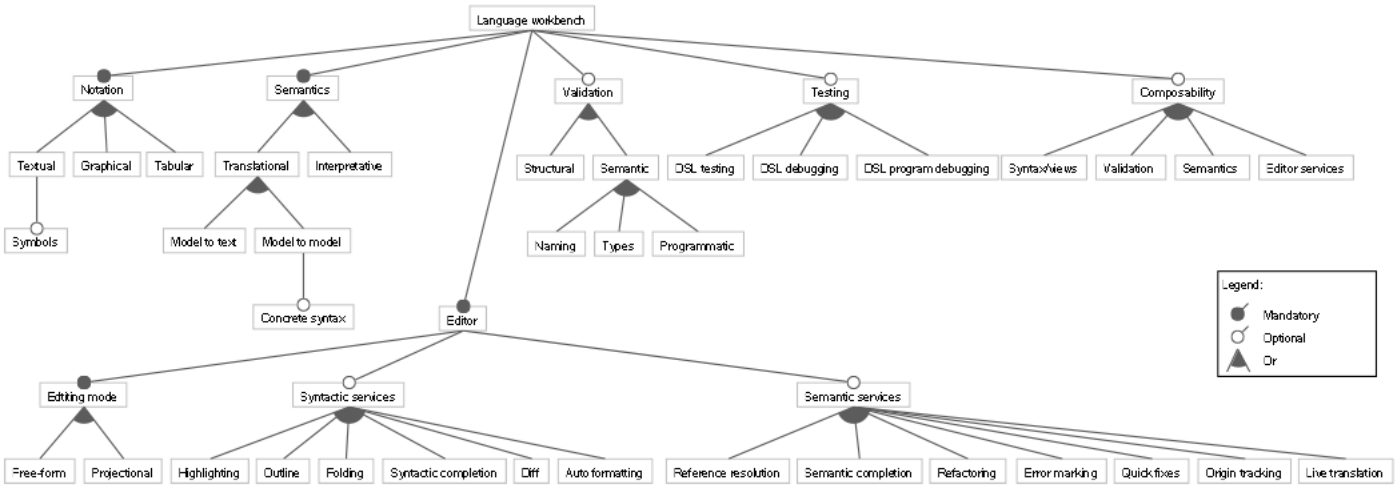


Figura 2.2.2: Modelo de un "Language Workbench". tomado de [19]

Rascal y XText comparten las mismas cualidades descritas en la Figura 2.2.2. TextRAM fue desarrollado en XText por los siguientes motivos:

1. XText proporciona una curva rápida de aprendizaje.
2. En comparación con Rascal, XText es un producto con más años en el mercado.
3. Tiene muy buena aceptación por parte de la industria.
4. Su documentación es completa y actualizada.
5. Tiene una comunidad de usuarios amplia y dispuesta a resolver dudas; además tiene una gran cantidad de recursos para resolver problemas: publicaciones, blogs, stackoverflow, twitter, etc.
6. Xtext es una herramienta ligada intimamente a Ecore (ver 2.3.5), que es el estándar de metamodelado utilizado por RAM.
7. Rascal es mas flexible y ofrece más funcionalidades que Xtext. Pero el fuerte de Rascal es la manipulación del código fuente; en cambio Xtext es una herramienta especializada para la creación de DSLs, por tanto es más fácil construir un DSL en Xtext que en Rascal.

2.2.4. Xtend

Xtend es un lenguaje de programación estático, fuertemente tipificado que se traduce a un código fuente comprensible de Java [33]. Xtend es un lenguaje de programación inspirado en Java, pero que remueve su ruido sintáctico. Actualmente Xtend soporta la generación de código para Java 8. Xtend promete no tener problemas de interoperabilidad con Java, realiza inferencia de tipos y su sistema de tipos está conforme con el lenguaje de especificación de Java. La motivación del uso de Xtext, se justifica con sus cualidades básicas [5]:

1. Parte del "core" de Xtext, fue construido en Xtend.
2. Métodos de extensión: permite adicionar nuevos métodos a tipos existentes sin modificarlos. El nombre Xtend, proviene de está característica. El mecanismo sintáctico de esta opción es utilizar el primer argumento de una llamada de una función, como la entidad que recibe la llamada. Ejemplo:

```
"hello".toFirstUpper()
//en lugar de: StringExtensions.toFirstUpper("hello")
```

3. Expresiones lambda: brindan una forma clara y concisa para representar la interfaz de un método por medio de una expresión. La funcionalidad es tratada como argumento de un método y el código es visto como datos. Una expresión lambda está rodeada de corchetes. El tipo del parámetro puede ser inferido por el contexto. Ejemplo:

```
textField.addActionListener([ e |
  textField.text = "The command was": + e.actionCommand
]);
```

4. Anotaciones activas: permiten la participación del programador dentro del proceso de traducción Xtend a código Java. Esto permite omitir el uso del patrones de diseño Visitor u Observer para afectar el proceso de traducción.

5. Expresiones "Switch" poderosas: estás expresiones son distintas a sus equivalentes en Java. Un ejemplo ilustra su diferencia:

```
switch myString {
  case myString.length > 5: "a long string."
  case 'some': "It's some string"
  default: "It's another short string."
}
```

6. "Dispatch Methods": Generalmente, la resolucion y enlazamiento de métodos es efectuado estáticamente en tiempo de compilación de Java. Las llamadas a los métodos son enlazadas en base a los tipos sintácticos de los argumentos. Algunas veces éste no es el comportamiento deseado, especialmente en el contexto de los métodos de extensión en donde se desea obtener un comportamiento polimórfico. Los métodos "dispatch" son particularmente útiles para la conversión de modelos M2T y M2M ya que eliminan la necesidad de inspección del tipo de dato por medio del operador `typeof` de Java; además, no es necesario el uso de la conversión explícita de tipos de datos ("casting"). Un método "dispatch" es declarado por medio de la palabra clave `dipatch:`

```
def dispatch printType(Number x) {
  "it's a number"
}

def dispatch printType(Integer x) {
  "it's an int"
}
```

7. Expresiones de plantilla: estas expresiones son fundamentales para la conversión de modelos M2T y M2M; las plantillas brindan legibilidad a la concatenación de cadenas. Las plantillas están rodeadas por comillas triples (`'''`). Las cadenas se pueden representar con simple o doble comilla. "Guillemets `«»`" pueden ser usados para insertar expresiones. Un ejemplo típico del uso de las expresiones de plantilla es el siguiente:

```
def someHTML(String content) '''
   <html>
      <body>
                « content »
          </body>
   </html>
'''
```

8. No existen las sentencias, todo es una expresión: en Xtend todo es una expresión que tiene un tipo de retorno. Las sentencias no existen. Un ejemplo de esto se puede visualizar con la siguiente expresión `try-catch`:

```
val data = try {
    fileContentsToString('data.txt')
  } catch (IOException e) {
    'dummy data'
}
```

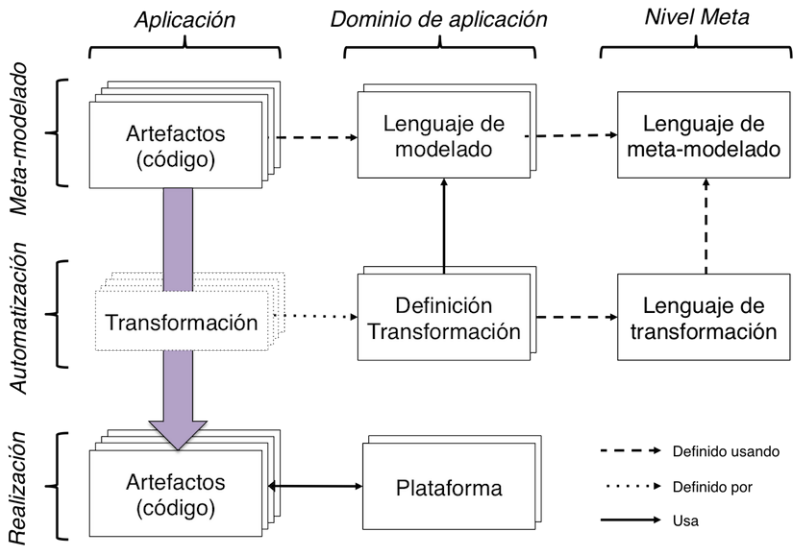


Figura 2.3.1: Visión general de la metodología MDSE. tomado de [7].

9. Acceso a propiedades: Si no existe un campo con el nombre proporcionado y además no existe un método sin parámetros con dicho nombre, Xtend enlazará el método getter (Java-Bean) correspondiente:

```
// generará myObj.getMyProperty()
myObj.myProperty
```

2.3. Ingeniería de Software Orientada a Modelos (MDSE)

MDSE es un marco de trabajo conceptual unificado en donde todo el ciclo de vida del software es visto como un proceso de producción, refinamiento e integración de modelos [23]. MDSE es una metodología para aplicar las ventajas del modelado a las actividades de ingeniería de software. En el contexto del MDSE, el software se obtiene como resultado de la siguiente ecuación:
`Modelos + Transformaciones = Software` [7].

Un lenguaje de modelado es la notación por la cual se expresan los modelos y las transformaciones. El proceso dirigido por modelos define qué tipos de modelos (orden de los mismos y su nivel de abstracción) son necesarios de acuerdo a un determinado tipo de software. La definición y el uso de DSLs es un sabor del MDSE, que se aplica por medio de la creación de representaciones formales, que son específicas a un aspecto particular de un sistema de software y son procesables con la ayuda de herramientas. La Figura 2.3.1, da un vistazo general de los principales aspectos considerados en MDSE, y resume cómo los diferentes problemas son resueltos de acuerdo a las siguientes dimensiones ortogonales: conceptualización (columnas) e implementación (filas).

El problema de implementación consiste en la definición de tres conceptos:

1. El nivel de modelamiento: define los modelos.

2. El nivel de realización: implementa soluciones, a través de artefactos que son usados dentro de sistemas en ejecución (código en caso de software)

3. Nivel de automatización: resultado de la correspondencia entre los niveles de modelamiento y realización.

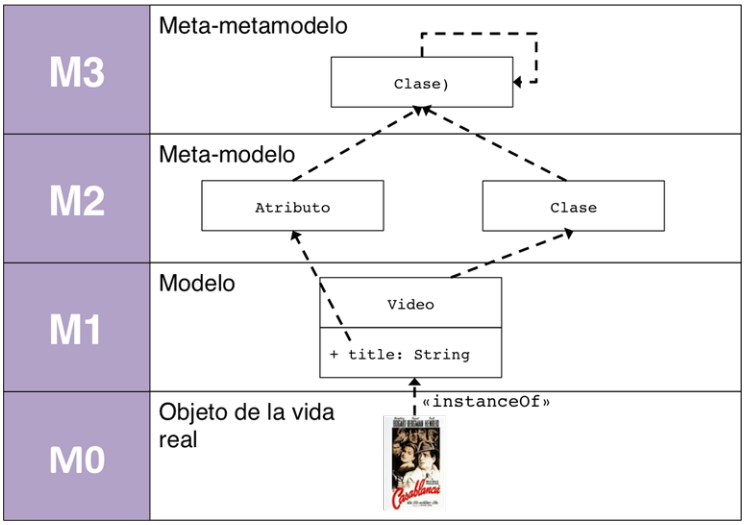


Figura 2.3.2: Modelado, metamodelos y meta-metamodelos. tomado de [7].

Los problemas de conceptualización, están orientados a definir modelos conceptuales para describir la realidad. Esto puede ser aplicado a varios niveles:

1. Nivel de aplicación: lugar en donde se definen los modelos de las aplicaciones, las reglas de transformación son aplicadas, y los componentes del sistema en ejecución son generados.

2. Nivel del dominio de aplicación: define el lenguaje de modelamiento, transformaciones, y plataformas de implementación para un dominio específico.

3. Nivel meta: la conceptualización de los modelos y transformaciones son definidos.

El flujo principal del MDSE parte con los modelos de aplicación hasta la realización, a través de transformaciones subsecuentes del modelo. Esto permite la reutilización de modelos y ejecución de sistemas en diferentes plataformas. Estos transformaciones son posibles gracias a la especificación de modelos, de acuerdo al un lenguaje de modelado que define la semántica de los elementos de los modelos. Es posible definir modelos de la realidad y luego modelos que describen modelos (metamodelos), después modelos recursivos que describen metamodelos (llamados meta-metamodelos). En teoría, se pueden definir instancias infinitas para los niveles de los metamodelos, pero en la práctica, se ha demostrado que los meta-metamodelos se pueden definir así mismos. La figura 2.3.2, muestra un ejemplo completo de metamodelado.

Los metamodelos pueden ser útiles para:

1. Definir nuevos lenguajes para el modelado o programación.

2. Definir nuevos lenguajes de modelado para intercambiar y almacenar información.

3. Definir nuevas propiedades y funcionalidades que pueden ser asociadas a información existente (meta data)

2.3.1. Transformaciones

Las tranformaciones son un ingrediente crucial para el MDSE, ya que permiten la definición de asignaciones (mapeo) entre diferentes modelos. Las transformaciones son ejecutadas entre el código fuente y el modelo destino, pero realmente son definidas por sus respectivos metamodelos. MDSE provee lenguajes apropiados

para la definición de transformaciones, por tanto brinda a los diseñadores soluciones óptimas para especificar reglas de transformación. Las reglas de transformación, pueden ser escritas manualmente por el programador, o pueden ser definidas como especificaciones refinadas de un modelo existente. Como alternativa, las transformaciones se pueden producir a sí mismas en forma automática, a partir de un mapeo de alto nivel entre modelos. Está técnica se realiza en dos pasos [7]:

1. Definición de mapeo entre elementos del modelo a elementos de otro modelo ("model mapping" o "model weaving").

2. Generación automática de las reglas de transformación a través de un sistema que recibe como entrada el mapeo y la definición de ambos modelos (origen y destino).La principal ventaja de este enfoque, es permitir al desarrollador concentrarse en los aspectos conceptuales de las relaciones entre modelos y delegar la producción de las reglas de transformación a un marco de trabajo.

2.3.2. Clasificación de los modelos

Los modelos se clasifican según su nivel de abstracción, en el dominio del problema en donde son aplicados. Por ejemplo en la etapa de diseño de un sistema, se pueden aplicar las siguientes abstracciones:

- Describa requerimientos y necesidades a un nivel abstracto, sin hacer referencia a aspectos de implementación.

- Defina el el comportamiento de los sistemas en términos de almacenamiento de datos y ejecución de algoritmos, sin mencionar detalles técnicos o tecnológicos.

- Defina todas los aspetos de tecnología en detalle.

Los modelos describen dos dimensiones principales: la parte estática (o estructural) y la parte dinámica (o de comportamiento). La definición de ambas dimensiones es la siguiente [7]:

- *Modelos estáticos:* se concentran en los aspectos estáticos de un sistema en términos de gestión de datos, figuras estructurales y arquitectura de un sistema.

- *Modelos dinámicos:* Se preocupa del comportamiento dinámico de un sistema, visualizando la secuencia de ejecución de acciones y algoritmos, las colaboraciones entre los componentes de sistema y los cambios de estado interno de los componentes y aplicaciones.

Está separación es importante para tener diversas vistas de un mismo sistema. Una representación completa de un modelo debe considerar las dos vistas por separado, pero con sus apropiadas interconexiones; las notaciones juegan un rol importante en este aspecto.

2.3.3. Observaciones al MDSE

Friedrich Steimann realiza la siguiente crítica con respecto al MDSE [25]: "Los modelos ofrecen un gran valor para las personas que no programan. Desafortunadamente, esto no se debe a que los modelos son más expresivos que los programas (en el sentido que son capaces de expresar aspectos complejos de forma simple), si no a que brindan una versión muy simplificada del problema. La construcción de un programa útil, solo es posible creando un modelo complejo; causando que la gente que no sabe programar no lo entienda, y personas que si pueden programar prefieren escribir un programa a dibujar el correspondiente modelo. Sin embargo, la extra-simplificación puede ser útil a veces, pero ciertamente no es suficiente para la creación satisfactoria de un producto".

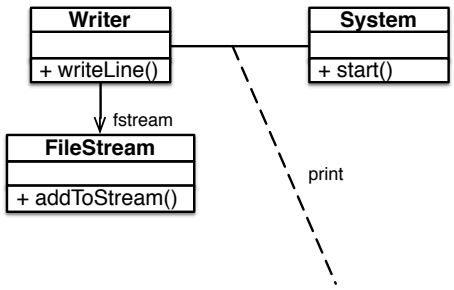


Figura 2.3.3: Dibujo con artefactos UML sin valor semántico

Es cierto que los modelos son atractivos porque proporcionan una representación simplificada de la realidad. Esta es su naturaleza inherente. Pero en lugar de ser una debilidad, esto representa una fortaleza. Lo anterior se demuestra porque varias soluciones reales se han ideado a partir de los estudios de modelos simplificados. Los modelos son cruciales para describir sistemas complejos y representan una poderosa herramienta para la consecución del producto final. No hay que olvidar que el MDSE puede aplicar jerarquía de modelos, transformación de modelos y agregación de modelos complementarios para describir un sistema. Por lo tanto, aunque los modelos sean una versión simplificada de la realidad, un conjunto coherente de modelos puede brindar una fácil compresión de todo el sistema.

2.3.4. Herramientas de dibujo versus herramientas de modelado

Las herramientas de modelado y las herramientas de dibujo no son conceptos intercambiables; solo algunas herramientas de dibujo son al mismo tiempo herramientas de modelado. Existen herramientas de modelado que utilizan una *sintaxis textual concreta* para la especificación de modelos, por tanto no tienen soporte para notaciones gráficas. Por otro lado, muchas herramientas gráficas no son herramientas de modelado. Una herramienta gráfica puede ser considerada una herramienta de modelado si la herramienta "entiende" el dibujo; ejemplo: la herramienta no se relaciona solo con figuras, líneas y flechas, si no que entiende lo que representa una clase, una asociación y otros conceptos de modelado. Debe existir al menos una validación con la especificación del modelo. Por ejemplo, una herramienta de modelado, puede permitir al diseñador crear una figura rara dento del modelo. Los íconos y las figuras pueden ser las utilizadas para acercarse a la representación de los conceptos de modelamiento (notaciones UML), pero el modelo como tal no tiene sentido, porque sus elementos rompen las reglas del lenguaje. En la Figura 2.3.3 se aprecia que la línea *print*, no tiene significado semántico para UML, esto es un ejemplo del dibujo de artefactos y no del diseño de modelos. Si no hay semántica, no se puede crear un metamodelo del problema, por tanto no hay forma de derivar varias representaciones de la misma abstracción.

2.3.5. Eclipse Modeling Framework

MDSE necesita herramientas que den soporte al desarrollo de modelos y permitan su integración dentro del proceso de desarrollo de software. Dichas herramientas deben [26]:

- Permitir la manipulación de modelos (por medio de la exposición de una API).
- Garantizar un cierto grado de alineamiento semántico y calidad en el modelo, por medio de la conformancia con el metamodelo.
- Transformar modelos.

"Eclipse Modeling Framework" (EMF) [26], es una de las herramientas de modelado de Eclipse para el MDSE. EMF tiene una amplia variedad de servicios y herramientas para persistir, editar y transformar modelos.

Para la definición de metamodelos, el EMF utiliza una versión simple y pequeña de UML llamada Ecore. Las principales características de Ecore son:

- Diversas representaciones (código Java, XML, UML) del metamodelo.
- Mecanismos automáticos de transformación entre las diferentes representaciones.
- Clases generadas por EMF.Edit que sirven como punto de partida para la implementación de código y el desarrollo de una aplicación.
- Un editor genérico para la creación, visualización y edición de modelos.
- Una API de consultas para la obtención de la estructura del metamodelo.
- Mecanismos de reflexión para la manipulación de instancias del metamodelo.
- Ecore y la serialización XMI (ver sección: 2.3.5) son el centro del universo EMF.

Las principales representaciones de EMF son XML, código Java y UML, las transformaciones de dichas representaciones son intercambiables; EMF también permite la integración y definición de otras representaciones (por ejemplo: diagramas entidad entidad relación).

Los modelos básicos de Ecore son [26]:

- *EClass*: representa los modelos del lenguaje (elementos de la sintaxis abstracta).
- *EAtrribute*: describe el estado de un EClass.
- *EDataType*: indica el tipo de un atributo. Un tipo de dato puede ser primitivo o un tipo de objetos como ser `java.util.Date`
- *EReference*: representa asociaciones entre EClases. En forma opcional los EReferences pueden tener semántica de contenedores.
- *EObject*: representa instancias de EClasses (por ejemplo nodos AST). Cada EObject puede ser contenido de al menos una instancia de EReference.
- *EPackage*: agrupa clases y tipos de datos relacionados.

El diagrama UML de dichos modelos se puede ver en la figura 2.3.4.

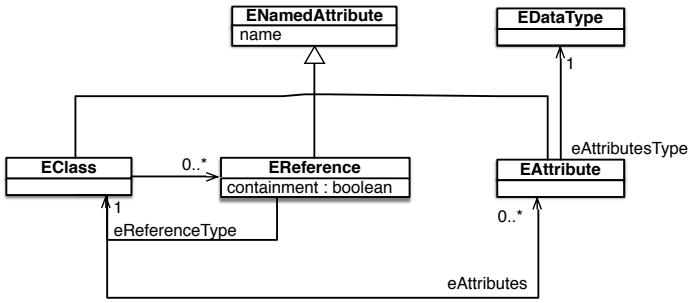


Figura 2.3.4: metamodelo básico de Ecore

Existen tres alternativas para crear y editar modelos:

1. Edición directa de Ecore: EMF incluye un árbol para la edición de modelos de Ecore. También se pueden utilizar herramientas gráficas que soportan la notación de Ecore:

   a) "Topcased's Ecore Editor" (http://topcased.org).

   b) "Omondo's Eclipse UML" (http://www.omondo.com/)

   c) "Soyatec's eUML" (http://www.soyatec.com/).

2. Importación desde UML: EMF sólo soporta archivos de tipo "Rational Rose"; está exclusividad se debe a que el núcleo de arranque del EMF fue construido en está herramienta.

3. Exportación desde UML: similiar a la segunda opción, pero la conversión es invocada desde una herramienta UML externa y no desde los asistentes de EMF.

Serialización XMI

XMI es la representación estándar para guardar modelos de Ecore. XMI quiere decir: intercambio de metadata por medio de XML. XMI no almacena información extra del modelo (al contrario de los esquemas XML); por está razón Ecore utiliza este formato para serializar sus modelos.

EMF incluye un serializador por defecto de XMI que puede ser usador genéricamente para guardar en disco objetos de cualquier modelo, no solo de Ecore. Para guardar en disco un modelo es necesario el uso de un "Resource". La interface "Resource" es utilizada para representar una ubicación física de almacenamiento (ejemplo: un archivo). Un modelo puede contener referencia a otros modelos, esto es manejado por una clase llamada "ResourceSet".

El "Runtime Framework" de EMF

Además de la transformación de modelos y generación de código, EMF ofrece las siguientes características [26]:

- *Notificación del cambio de un modelo*: cuando un modelo cambia de estado, el EMF notifica el evento por medio de "listeners" llamados "adapters". Los "adapters" utilizan el patrón de diseño "observer", pero además permiten la extensión del comportamiento de un modelo (ejemplo: soporte a interfaces adicionales sin la necesidad de realizar "subclassing"). La extensión de comportamiento se realiza por medio de un "adapter factory".

- *API reflectiva eficiente para la manipulación de objetos EMF en forma genérica*: en lugar de escribir métodos concretos para leer y escribir el modelo, se puede utilizar la API reflectiva. El serializador XMI genérico y el generador de código por defecto se sirven de está API.

- *Bases para la integración de datos*: existen utilidades que ayudan a la gestión de objetos compartidos. Por ejemplo: `EcoreUtil.CrossReferencer` que puede encontrar, limpiar, eliminar referencias cruzadas a otros objetos, también puede encontrar los "proxies" no resueltos de un recurso.

2.4. Introducción a la programación orientada a aspectos (AOP)

La modularización es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y con baja acoplación. La conexión entre dichos módulos, se realiza en base a los supuestos que los módulos hacen entre sí.[6].

Los mecanismos de descomposición jerárquica de los lenguajes de programación orientados a objetos son bastante útiles, pero no suficientes para modularizar todas la preocupaciones interesantes de un sistema complejo. Existen preocupaciones que atraviesan la modularidad natural de un sistema, causando problemas de dispersión y entrelazamiento de código. Las consecuencias de la dispersión y entrelazamiento de código son:

- Dificultad para la evolución y mantenimiento del código
- La detección y resolución de errores es compleja.
- Las preocupaciones transversales no se pueden reutilizar.
- El razonamiento local de una preocupación transversal es difícil.

AOP es una propuesta que modulariza las preocupaciones transversales y mejora la separación de preocupaciones en el software [15]. AOP resuelve los problemas inherentes a la dispersión y entrelazamiento de código, ofreciendo al programador mecanismos para separar preocupaciones transversales en forma limpia y concisa. Un *aspecto* es la definición modular de dicha preocupación transversal. *Código base*, es toda preocupación que no es transversal. *"Weaving"* es el mecanismo que combina el código base con las preocupaciones transversales para formar el sistema final.

Para implementar un sistema de AOP, es necesario incluir muchos de los siguientes conceptos [18]:

- "*Join points*": son puntos de ejecución identificables dentro del sistema (por ejemplo: ejecución de métodos, creación de objetos, ejecución de excepciones, etc). Los "join points" están presentes en todos los sistemas, incluso en aquellos que no utilizan técnicas de AOP.
- "*Pointcuts*": son mecanismos para la selección de "join points". La definición de los "pointcuts" incluye criterios que deben ser satisfechos para la selección de un conjunto determinado de "join points". Un "pointcut" puede utilizar otro "pointcut" para formar una selección compuesta. Los "pointcuts" también recolectan contexto en base a los "join points" seleccionados (argumentos de un método, objeto llamador, etc).
- "*Join point model*": es la definición de los tipos de "join points" disponibles y la descripción de cómo son utilizados.
- "*Advice*": es el mecanismo para alterar el comportamiento del programa. Un "advice" puede modificar el comportamiento antes, después o alrededor de los "join points" seleccionados. El "advice" es una forma de una transversalidad dinámica porque afecta la ejecución del sistema.
- "*Inter-type declarations*": modifican la estructura estática de un sistema (modificación de la estructura de una clase, cambios en la jerarquía de clases, etc).
- *Aspectos*: son módulos que contienen los "*pointcuts*", "*advice*" e "*inter-type declarations*".

2.5. Modelamiento de aspectos reutilizables

El modelado multi-vista permite describir el software desde distintas perspectivas y diferentes notaciones. Dicho modelado enfrenta dos retos: escalabilidad y consistencia [17]. En aplicaciones complejas, los modelos tienden a crecer en tamaño, dificultando la comprensión de las vistas.

Las técnicas de orientación a aspectos, han resuelto el problema de identificación y de modularización de preocupaciones transversales, lo que permite al desarrollador razonar sobre una preocupación individual. Por

tanto, estas técnicas son una buena alternativa para resolver los problemas de escalabilidad y consistencia inherentes al modelado multi-vista.

Los enfoques existentes del modelado orientado a aspectos (AOM), se han convertido en una estrategia exitosa para separar y componer modelos. En el contexto del modelado multi-vista, AOM puede ser aplicado sobre vistas individuales para resolver el problema de escalabilidad; sin embargo, esto dificulta la consistencia entre modelos.

"Reusable Aspect Models" (RAM) [16] es un enfoque de modelamiento orientado a aspectos que permite crear modelos en forma escalable y consistente entre múltiples vistas. RAM posibilita expresar la estructura y el comportamiento de sistemas complejos, por medio de diagramas de clase, estado y secuencia en un paquete UML especial llamado "aspect model". Las características de RAM son [16] :

1. Integración de diagramas de clase, estado y de secuencia por medio de las técnicas de modelado orientado a aspectos.

2. Reutilización de “aspect models” en forma segura y flexible.

3. Soporta la creación de cadenas complejas de dependencia. Esto permite modelar aspectos de funcionalidad compleja, descomponiéndolos en aspectos que proveen una funcionalidad simple.

4. Ejecuta revisiones de consistencia para verificar la correcta composición de aspectos y su reutilización.

5. Define un detallado algoritmo de “weaving” que resuelve la dependencia de aspectos para generar “aspect models” independientes, que son aplicados en el modelo final.

2.5.1. Antecedentes del modelamiento orientado a aspectos

RAM compone modelos que representan diferentes vistas del mismo concepto. El resultado esperado es un modelo compuesto a partir de un modelo origen y un modelo destino. Para que esto sea posible, se deben cumplir dos requisitos:

1. Los elementos del modelo a componer deben ser del mismo tipo sintáctico.

2. Los elementos del modelo a componer deben ser instancias de la misma clase del metamodelo.

Si existe un elemento que no está presente en el modelo destino (y viceversa), dicho elemento es incluido en el modelo final compuesto. Pareo de elementos ("element matching"), es el proceso de identificar elementos del modelo a componer. En el pareo automático de elementos, cada tipo de elemento es asociado con una firma que determina su unicidad dentro del espacio de tipos: dos elementos con firmas equivalentes representan el mismo concepto, por tanto dichos elementos son incluidos en la composición.

En la Figura 2.5.1, se visualiza un diagrama de clases base, en donde hay una entidad que produce salidas (clase “Writer”) a un dispositivo de salida (clase “FileStream”). El objetivo es desacoplar la producción de salida de sus dispositivos, por medio de un “Buffer”; esto es posible, gracias al modelamiento de un aspecto que incluye la clase “Buffer” y a la unión de dicho aspecto con el diagrama de clases base. La composición final es visualizada en el modelo “Result”.

2.5.2. “Weaving” de los diagramas de estado y secuencia

Para tejer los diagramas de estados de aspectos, se deben especificar dos tipos de diagramas:

1. Un diagrama de estados para el “pointcut” (especificación del comportamiento a detectar)

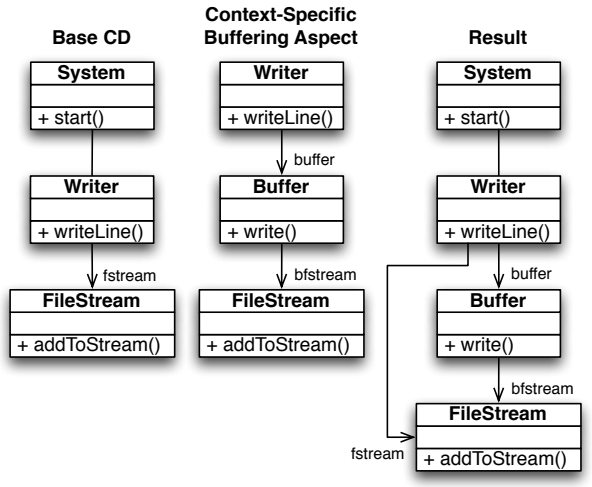


Figura 2.5.1: Ejemplo de composición de diagramas de clase; tomado de [17]

2. Un diagrama de estados para el "advice" (comportamiento esperado para cada "join point").

El resultado de este enfoque es la extensión del comportamiento con uno nuevo o la eliminación de este. Un comportamiento puede ser insertado alrededor , antes o después de un "join point", por lo tanto el comportamiento original se puede sustituir, extender o remover. Un ejemplo de esto se visualiza en la Figura 2.5.2, por medio de un diagrama de secuencia.

En el ejemplo de la Figura 2.5.2, los diagrama de secuencia base y destino muestran una interacción entre el usuario y el servidor:

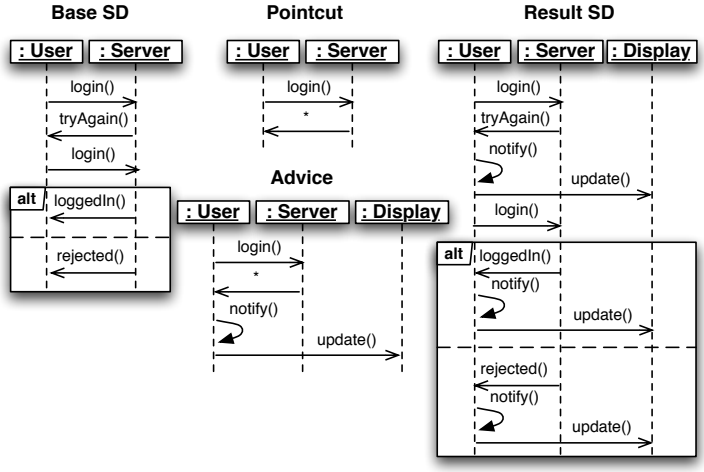


Figura 2.5.2: "Weaving" de un diagrama de secuencia; tomado de [17]

1. El usuario envía un mensaje de `login` al servidor.

2. El servidor responde con `tryAgain`

3. El usuario realiza un nuevo intento.

4. El diagrama de secuencia muestra un escenario alternativo (`alt`) que describe que mensajes son enviados después, dependiendo si el `login` es aceptado o rechazado.

El aspecto especificado en la Figura 2.5.2, consiste en un "pointcut" y un "advice". El "pointcut" intercepta cualquier interacción entre el usuario y el servidor, empezando por método `login`. En la especificación del "pointcut", es posible utilizar expresiones regulares en el nombre de los mensajes. El asterisco `(*)` captura cualquier mensaje desde el servidor al usuario. El "advice" indica que el mensaje `notify` y el mensaje `update` de un objeto de tipo `Display` son adicionados después del retorno del mensaje desde el servidor.

2.5.3. Conceptos básicos de RAM

Una preocupación en RAM, se modela por medio de 3 tipos de vistas: vista estructural, vista de estados y vista de mensajes; dichas vistas son agrupadas en un paquete UML especial llamado "aspect model".

RAM no diferencia entre código base y aspectos, por tanto se clasifica como un modelador simétrico. En RAM, cualquier preocupación es modelada como aspecto; no importa si el aspecto se utiliza solo una vez dentro de una misma aplicación; además, un aspecto puede ser reutilizado por otros aspectos. Para lograr mayor reusabilidad de los modelos, RAM recomienda modelar aspectos simples y pequeños.

Vista estructural

Es la primera sección de un "aspect model". La vista estructural utiliza diagramas de clase con atributos, métodos y asociaciones. Los miembros de clase tienen tres tipos de visibilidad:

1. Miembros privados: los métodos solo son visibles dentro de la clases en donde fueron definidos. Estos métodos se anotan con el carácter guión `(-)`

2. Miembros públicos: representan la interfaz pública de los aspectos de RAM y son visibles al exterior del paquete de aspectos. Estos métodos se anotan con el carácter más `(+)`

3. Miembros intra-aspectos: únicamente pueden ser llamados desde otros objetos que son parte del aspecto. Su anotación es el carácter virgulilla `(~)`.

Completitud de la clases

Las clases dentro de la vista estructural no necesitan estar completas. Dichas clases solo necesitan especificar los miembros que son relevantes dentro de la preocupación modelada. La clases incompletas reciben el nombre de clases parciales y se anota mediante el carácter de barra vertical |.

Para ser usadas en una aplicación, es requerido completar las clases parciales; no es posible crear una instancia de una clase parcial, ya que las mismas no definen constructores. Todas las clases parciales de un aspecto son exportadas como **parámetros de instanciación obligatoria**; gráficamente, dichos parámetros son representados en la esquina superior derecha del "aspect model". Para poder usar el aspecto y tejerlo en el modelo destino, los parámetros de instanciación obligatoria deben ser mapeados a los elementos del modelo del diagrama de clases destino.

Después, la composición de clases se realiza por medio del "weaver" enlazando o instanciando el modelo de aspectos con el modelo de clases base.

Vista de mensajes

Para proveer la funcionalidad relacionada a una preocupación, los elementos del modelo dentro del aspecto deben colaborar en "run-time". En RAM, la colaboración entre objetos es representada por medio de la vista de mensajes. Los criterios para definir las reglas de mensaje, son las siguientes:

1. Los diagramas de secuencia UML son utilizados para representar la vista de mensajes.

2. Se debe definir una vista de mensajes para cada operación pública que involucra intercambio de mensajes entre objetos de la vista estructural.

3. Deben incluirse los mensajes que muestran un intercambio de información entre entidades.

La vista de mensajes se divide en dos: en un "pointcut" y en un "advice". El "pointcut" especifica el método que deben existir en el digrama de secuencia destino; el "advice" muestra los detalles de ejecución que serán adicionados al método especificado en el "pointcut". En ocasiones, el "pointcut" puede representar comportamientos más complejos: mensajes de secuencia entre distintos objetos. En estos casos, el "advice" muestra cómo los mensajes nuevos son adicionados dentro del comportamiento especificado en el "pointcut" o inclusive, cómo los mensajes interceptados son reemplazados.

La Figura 2.5.4, ilustra la clase parcial |`Checkpointable`, que es instanciada para su futura reutilización por parte del "aspect model" `Checkpointable`.

La Figura 2.5.3 muestra el "aspect model Game" extraído del caso de estudio Slot Machines [35]; dicho "aspect model" contiene 4 vistas de mensaje (`initialize, play, getCurrentSymbol, getCurrentPosition, setCurrentPosition`). `Initialize` Uno de los mensajes del modelo describe: la inicialización de la máquina, el carrete de azar y la configuración del carrete. El comportamiento de la vista de mensajes "play" indica el resultado del juego, después del giro de los carretes.

RAM, clasifica la vista de mensajes en 3 tipos principales:

- "*Message Views*": vista de mensajes o vista normal de mensajes; definen el comportamiento de un método. Su objetivo es representar el intercambio de mensajes involucrados cuando el método en cuestión es llamado. Adicionalmente, por medio de la palabra clave "*affected by*" una vista de mensajes conoce que aspectos extienen su funcionalidad.

- "*Aspect Message Views*": vista de mensajes de aspectos; definen el comportamiento que debe ser extendido por otras vistas de mensajes. Contiene un *pointcut* que describe la llamada de un método y un *advice* que define el comportamiento a ser insertado en un punto determinado ("*join point*").

- "*Message View Reference*": vista de mensajes de referencia; permite la referencia de una vista de mensajes proveniente de un aspecto derivado. Un aspecto que instancia otro aspecto puede extender el comportamiento de aspectos derivados.

Vista de estados

La vista de estados es opcional dentro de un "aspect model". Los principales usos de la vista de estados son:

1. Documentación.

2. Verifican la consistencia entre la invocación de operaciones.

3. Establecen el orden de las llamadas entre operaciones (cualidad que no puede ser definida en la vista de mensajes).

La definición de la vista de estados, está basada en el modelado de protocolos (PM) [4] que permite reutilizar comportamientos y que además puede ignorar, aceptar o reutilizar eventos. Las máquinas de estado, son utilizadas para representar los protocolos reutilizables que componen el modelo. Los estados representan el estado lógico del objeto. Las transiciones conectan dos estados (estado inicial y final); una transición corresponde a una operación de una clase en la vista estructural. Las transiciones pueden tener un "guard

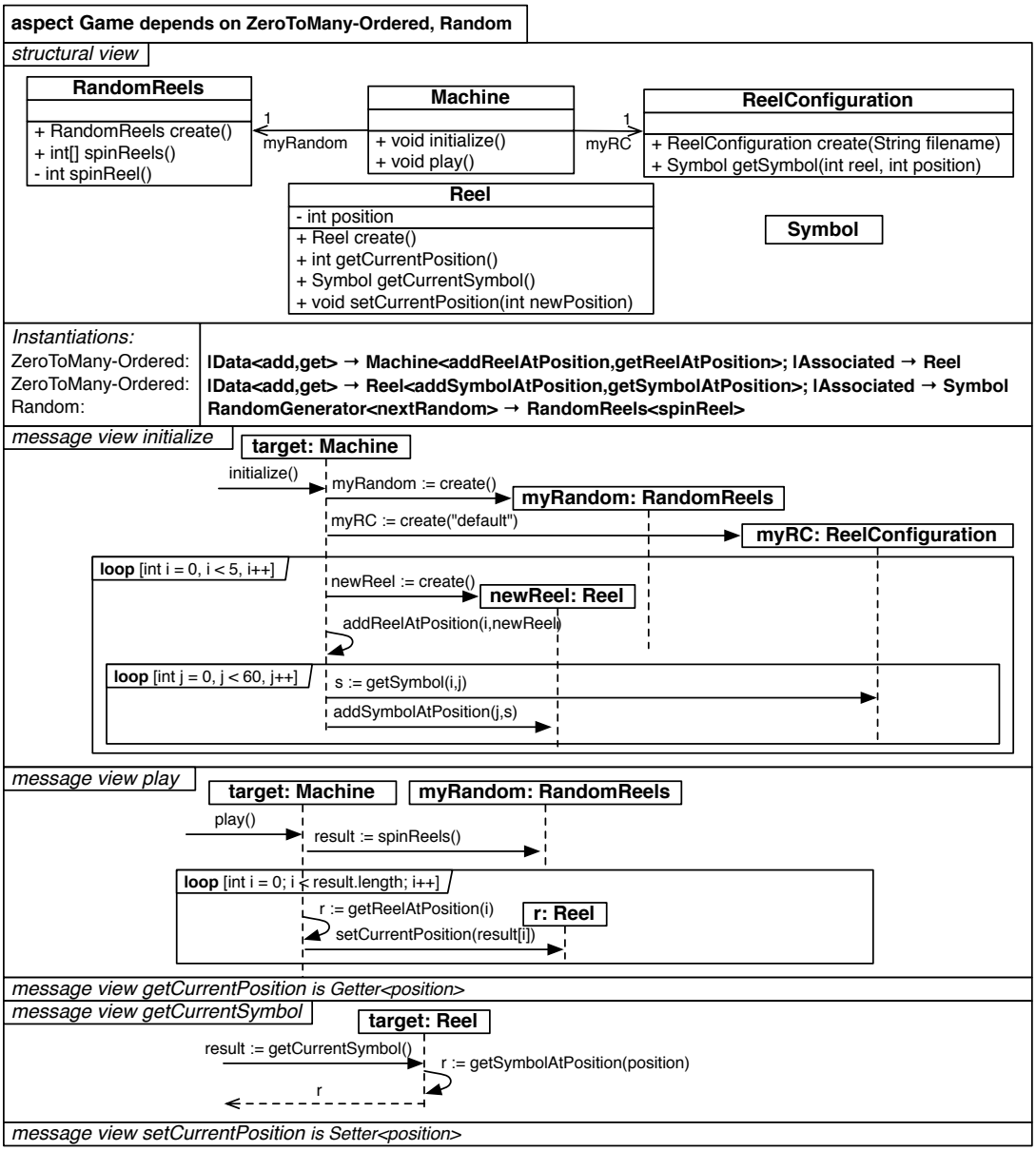


Figura 2.5.3: Aspect Model Game

condition" que tiene que evaluarse como verdadero para que el protocolo acepte la llamada a la operación de la transición. La figura 2.5.4, muestra la vista de estado llamada "Checkpoint" con dos estados: `Empty` y `CheckpointExist`; dicha vista de estados define una verificación que no puede ser expresada en la vista de mensajes: no se puede restaurar un Checkpoint (`restoreCheckpoint`), si el mismo está en estado `Empty`. TextRAM omite la vista de estados, ya que los mismo estaban en proceso de definición cuando se inició la presente tesis; el resto del presente documento se concentra en la vista estructural y la vista de mensajes.

2.5.4. Dependencia de aspectos, reutilización, enlaces e instanciación

Uno de los objetivos de RAM, es proveer escalabilidad por medio del modelamiento multi-vista. Para mantener los "aspect models" relativamente pequeños, los aspectos que necesitan representar una funcionalidad compleja, deben tener la capacidad de reutilizar la funcionalidad de otros aspectos. Si un aspecto `A` reutiliza modelos proporcionados por un aspecto `B`, entonces `A` depende de `B`. Las dependencias deben ser mostradas en el encabezado del paquete de aspectos. Ejemplo: `aspect A depends on B`.

Instanciación

En RAM, si `A` depende de `B`, `A` debe indicar explícitamente que reutiliza la funcionalidad de `B` instanciando `B`. Cada una de las vistas (estructural, estados y mensajes) pueden indicar parámetros de instanciación obligatoria, por medio de clases parciales definidas en la esquina superior derecha de cada vistas. Si `A` desea reutilizar B, `A` debe proporcionar al menos una directiva de instanciación que corresponda a los parámetros de instanciación obligatoria de la vista de `B` (estructural, estado o mensajes). Clases en `B` que no son parámetros de instanciación obligatoria pueden ser instanciadas en forma opcional.

El formato de instanciación es el siguiente:

`AspectoOriginal.ClaseParcial -> NombreEntidad`

En la Figura 2.5.4, se ejemplifica como el aspecto `Checkpointable` reutiliza el aspecto `Copyable`:

1. El encabezado del aspecto, indica que `Checkpointable` reutiliza `Copyable`: `aspect Checkpointable depends on Copyable`.

2. La vista estructural de `Copyable` se instancia en la vista estructural de `Checkpointable`, pareando la clase incompleta `|Copyable` con la clase incompleta `|Checkpointable`. El significado de la instanciación es el siguiente: todas las instancias de `|Checkpointable` además de exponer los métodos `restoreCheckpoint, discardCheckpoint`, también define los métodos, atributos y asociaciones definidas en `|Copyable`.

3. La vista de estados de `|Checkpointable` especifica que una instancia de la clase incompleta `|Checkpointable` acepta cualquier número de llamadas de `establishing`, seguido de por lo menos, el mismo número de llamadas de `restoreCheckpoint` y `discardCheckpoint`. De igual forma, `replaceWith` puede ser llamado solamente después de una llamada de `restoreCheckpoint` o `discardCheckpoint`.

Enlazamiento ("Binding")

En el escenario de que un aspecto `A` depende de un aspecto `B`, puede pasar que una clase parcial `|X` en la vista estructural de `A`, necesite ser compuesta para completar la clase `Y` definida en `B` (o en uno de los aspectos de los cuales depende `B`). En este caso la vista de estados `X` en `A` también necesite refinar la vista de estados `Y` para tomar en cuenta la funcionalidad de `A`. De igual forma, `A` puede necesitar, refinar o sobre-escribir los mensajes de secuencia especificados en la vista de mensajes definidos en `Y` para tomar en

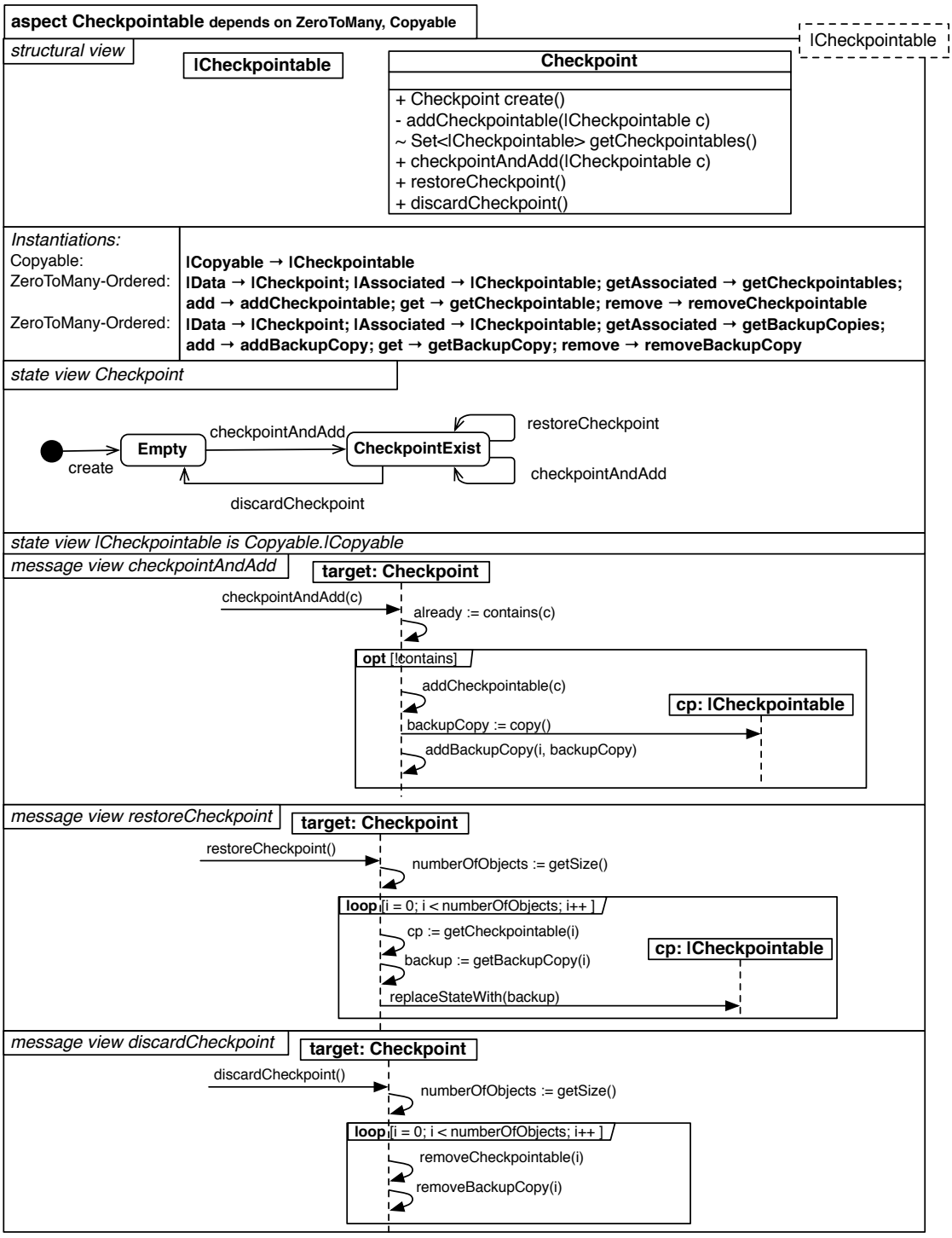


Figura 2.5.4: |Checkpointable: ejemplo de una clase parcial

cuenta la funcionalidad proporcionada por `A`. La sintaxis para la directiva de enlazamiento es el siguiente: `NombreEntidadIncompleta -> Destino.NombreEntidad`.

Si una vista contiene una directiva de enlazamiento, los elementos enlazados no pueden aparecer al mismo tiempo como parámetros de instanciación obligatorios en la directiva de instanciación de dicha vista. Las directivas de instanciación y enlazamiento pueden ser de uno-a-muchos o de muchos-a-uno si es necesario. En este caso, se pueden utilizar "wildcards" para instruir al "weaver" que realice "pattern matching" en el modelo, para determinar el conjunto de elementos que van a ser usados en la directiva.

Reutilización

Uno de los principales objetivos de RAM, es permitir el diseño de modelos de aspectos altamente reutilizables. La idea es evitar la dispersión de modelos por medio de la definición de funcionalidades relacionadas y evitar el entrelazamiento de elementos, a través de diferentes funcionalidades. Para que la reutilización sea posible, se deben seguir las siguientes reglas:

1. Si un aspecto `A` expone una funcionalidad, cuyo diseño necesita la funcionalidad del aspecto `B`, entonces `A` depende de `B`; solo en este caso `A` puede instanciar vistas de `B`, o enlazar elementos de `A` con elementos de `B`.

2. Las dependencias circulares están prohibidas.

2.5.5. Metamodelo de RAM

Comprender el metamodelo de RAM es importante para la construcción de TextRAM, debido a que la TCS de TextRAM es un representación textual que debe instanciar modelos conformes al metamodelo de RAM. Se puede decir que RAM es un sabor de UML, ya que su metamodelo está basado en él. Las ventajas de usar UML como base del metamodelo de RAM, son las siguientes:

1. Adopción rápida de RAM por parte de los usuarios que utilizan UML.

2. Importación de modelos existentes de UML, para después completarlos con la información específica de RAM.

2.5.5.1. Metamodelo de la vista estructural

El elemento raíz de un modelo en RAM es el *aspecto*. El *aspecto* tiene un nombre que es obligatorio; todos los elementos con nombre, tiene como clase padre *NamedElement*. El contenido del aspecto está conformado por una *vista estructural*, un *"layout"*, *instanciaciones, vistas de mensajes* y *estados*. La *vista estructural* está basada en los diagramas de clase UML. "*Classifier*" es una abstracción que permite modelar clases o implementaciones de clase (ejemplo: `java.util.Set` de java). La vista estructural, expone una lista de tipos (cuya clase base es "*Type*"), una lista de "*Classifiers*" y una lista de *asociaciones*. Los "*Classifiers*" contienen *operaciones* que tienen un nombre, una visibilidad y propiedades que indican si la operación es *"abstract", "partial"* o *"static"*. Una *operación* también tiene un *tipo de dato* de retorno y una lista ordenada de *parámetros*. Cada *parámetro* define un nombre y un *tipo de dato*. Un *parámetro,* a su vez es un "*TypedElement*". A parte de los miembros especificados por "*Classifier*", una *clase* define una lista de *super-tipos* (de tipo *"Classifier"*) y una lista de *atributos*. Los *atributos* son propiedades de clase que pueden ser estáticos y exigen un tipo de dato:*"PrimitiveType"*. Los *"AssociationEnd"* especifican si la clase pertenece a un *"Association"*, además tienen un nombre e indican la mínima y máxima ocurrencia de la asociación. Las asociaciones contienen dos extremos de tipo *"AssociationEnd"*; el tipo de la *asociación* se deriva de la *clase* especificada en el otro

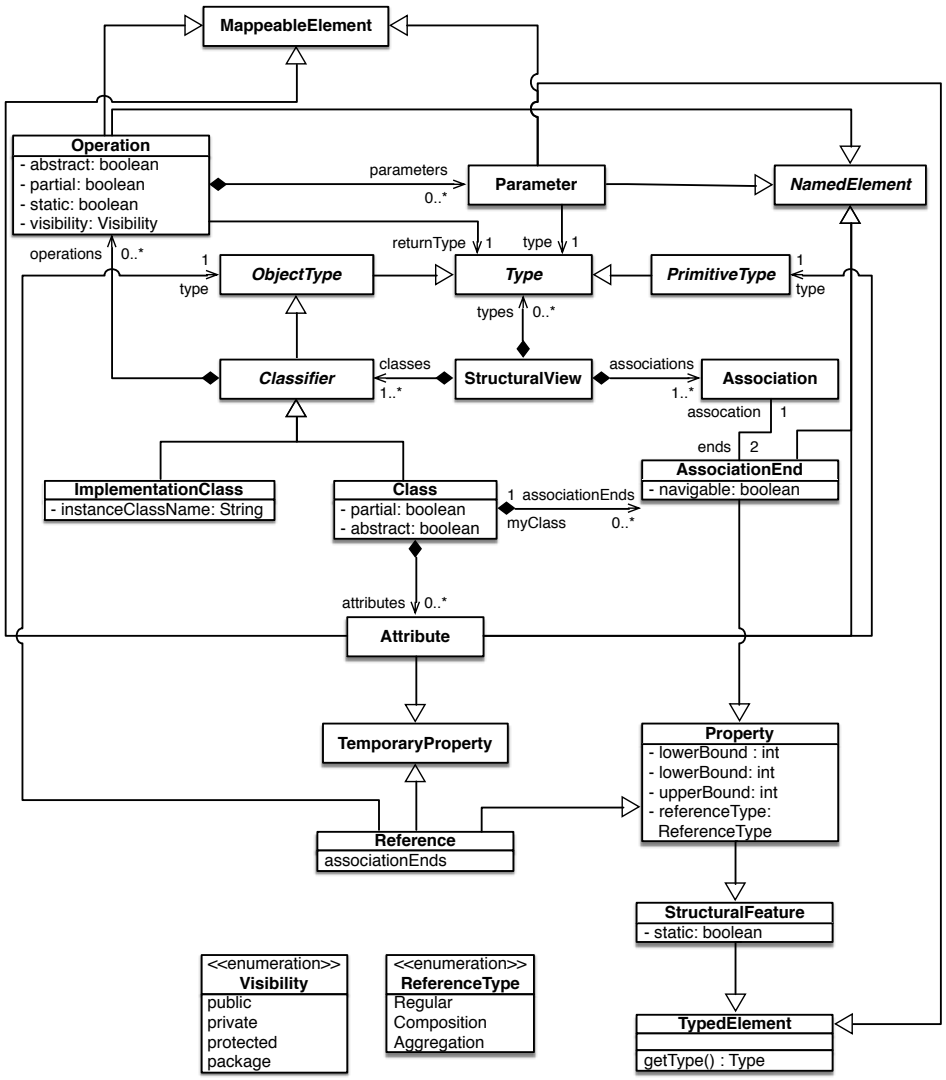


Figura 2.5.5: Metamodelo de la vista estructural de RAM.

extremo de la asociación. "*Property*" es una clase abstracta que indica el tipo de referencia y los límites de una asociación. Los tipos de referencia se representan por la enumeración "*ReferenceType*". Los valores de "*ReferenceType*" son: "*Composition*", "*Aggregation*", "*Regular*"; las especializaciones de "*Property*" son: "*Reference*" y "*AssociationEnd*".

Además, cada clase es también un "*ObjectType*". Las clases y las operaciones son mapeables gracias a que son especializaciones de "*MappableElement*". La vista estructural permite la definición de "*PrimitiveTypes*" y tipos especiales como *void* y *any*; dichos tipos pueden ser utilizados como valor de retorno de una operación. *Any* indica que cualquier cosa puede ser retornado por una operación parcial. En la Figura2.5.5 se visualiza el metamodelo de la vista estructural de RAM.

Un diseñador de RAM, debe preocuparse de modelar únicamente los puntos relevantes de la vista estructural (propiedades de clase, cantidad de clases involucradas, asociaciones entre clases).

2.5.5.2. Metamodelo de la vista de mensajes

La vista de mensajes describe el comportamiento del *aspecto*. El metamodelo de vista de mensaje se puede visualizar en la Figura 2.6.1. La base para la definición de la vista de mensaje son los diagramas de secuencia UML. Se debe crear una vista de mensajes para cada operación pública de la vista estructural. Dentro de la vista de mensajes, se aceptan solo llamadas de la clase representada o de sus clases asociadas. Esto incluye operaciones y clases disponibles a través del mecanismo de mapeo efectuado en las *instanciaciones*.

Además de los 3 tipos mencionados en 2.5.3, una vista de mensajes define:

- "Returns": se utiliza si una operación define una respuesta. Las "*Temporary Properties*" son utilizadas para persistir el valor de retorno de un método y también para ser enviadas como argumentos de una operación. Las propiedades temporales existen solo dentro del ciclo de vida del "*Lifeline*" involucrado.
- El "pointcut" y el "advice" pueden contener un elemento que representa el *comportamiento original* del mensaje involucrado. La notación para este elemento es una caja cuyo contenido es el carácter "`*`". El orden de ejecución del "advice" ("after", "before", "around"), depende de la posición del "`*`" dentro de la caja del modelo.
- Captura de excepciones: UML no tiene un operador para representar un bloque `try-catch`; el operador utilizado para representar excepciones es el fragmento combinado "*disruptable*".
- Llamadas a un "*Lifeline*": todas las vistas de mensajes, contienen un "*Lifeline*" que representa al llamador del mensaje. El primer mensaje entre el llamador y el destino sirve para marcar el punto de inicio del mensaje. Los "*Lifelines*" representan a los participantes individuales de una *interacción*.
- Instanciaciones por defecto: definen restricciones de los "*lifelines*" y clases. (ejemplo: no pueden existir instancias de una determinada clase).

Un *aspecto* contiene varios tipos de vistas de mensajes, esto es posible, gracias a la abstracción "*AbstractMessageView*". Para los "*message view references*" es obligatorio especificar el "*aspect message view*" afectado. En la vista de mensajes se especifica la operación representada; sin embargo, esto no es obligatorio para las clases parciales, ya que las mismas no poseen una especificación. El "*advice*" de un "*aspect message view*" es la especificación de la vista de mensajes, y es además una *interacción*. Hasta el momento, todos los modelos de RAM utilizan una especificación simple de "*pointcut*", por tanto el "*pointcut*" siempre hace referencia a una *operación*.

La *interacción* es la unidad básica de encapsulamiento del comportamiento en la vista de mensajes. El objetivo de las *interacciones* es describir el intercambio de mensajes entre las instancias de los "*Classifiers*". Una *interacción* contiene "*lifelines*", "*messages*", "*fragments*" (que son un conjunto ordenado) y "*formal gates*". A parte de las *interacciones*, los otros elementos que pueden contener fragmentos son: "*InteractionOperand*" y "*CombinedFragment*". El elemento "*FragmentContainer*" es utilizado para representar un conjunto ordenado de fragmentos. "*FragmentContainer*" es la clase padre de "*Interaction*" e *"InteractionOperand"*. Las interacciones pueden contener propiedades locales "*Temporary Properties*", usadas para definir instancias de un "*Lifeline*". Las *referencias* y los *atributos* son del tipo "*Temporary Property*". Una *variable temporal,* está viva solo durante la ejecución de un método de la vista de mensajes.

Un "*Lifeline*" conoce a que "*InteractionFragment*" pertenece y conoce las instancias que representa por medio de su propiedad "*Represents*". En RAM, las especializaciones de "*Represents*" son los *"attributes", "properties"* y los "*parameters*"; adicionalmente el "*Represents*" de un "*Lifeline*" puede ser un *"AssociationEnd",* ya que el mismo es un subtipo de "*TypedElement*", clase base de "*Parameter*" y *"StructuralFeature".* Un "*Lifeline*" puede representar una instancia de un "*Classifier*" o por medio de la propiedad "*static*", puede

representar al "*Classifier*" como tal. La visualización gráfica de este tipo de "*Lifelines*" es por medio del estereotipo `<<metaclass>>`.

Un *mensaje*, describe la comunicación entre dos "*Lifelines*". El "*messageSort*" define el tipo de mensaje: "*synchronous*", "*call*", "*reply*", "*create*" o "*delete*". La firma de los mensajes, se deriva de la operación referenciada. Cada mensaje, posee eventos de envio y recepción que definen los "*Lifelines*" que están conectados entre sí. Los eventos pueden ser un "*Gate*" o un "*MessageOcurrenceSpecification*". El valor de retorno de una llamada de operación puede ser asignado a un "*StructuralFeature*" que permite la referencia a una *clase* existente o una *propiedad temporal.*

Los *argumentos* definen si la llamada a una operación contiene parámetros. El metamodelo tiene un conjunto ordenado de argumentos, en donde el argumento es definido por "*ValueSpecification*". Un "*ValueSpecification*" especifica el valor concreto del parámetro o literales como ser *String* e *Integer.* El orden de los argumentos debe coincidir con el orden de los parámetros de la operación. Por medio de "*ParameterValueMapping*" se pueden mapear los parámetros formales de una operación con sus valores actuales.

"*InteractionFragment*" es la clase base de "*OcurrenceSpecification*", "*OriginalBehaviorExecution*" y "*CombinedFragment*". "*OcurrenceSpecification*" se utiliza para describir una ocurrencia convencional de un mensaje o para indicar la destrucción de un mensaje por medio de "*DestructionOcurrenceSpecification*". "*OriginalBehaviorExecution*" es utilizado por "*AspectMessageView*" para indicar la ejecución original del método seleccionado por el "*pointcut*". "*CombinedFragment*" tiene un conjunto ordenado de "*InteractionOperands*" para soportar la ejecución de otras interacciones (esto es posible porque "*InteractionOperand*" es una especialización de "*FragmentContainer*"). Los operadores soportados por "*CombinedFragment*" son: "*alt*", "*opt*", "*loop*", "*critical*" y "*disruptable*". Se debe definir un "*constraint*" para cada uno de los operadores, a excepción de "*critical*". El "*constraint*" debe ser del tipo "*ValueSpecification*"; también se puede definir una "*OpaqueExpression*"; ejemplo, `int index = 0; index < size; index++`.

2.5.6. Conclusiones de RAM

Las principales contribuciones de RAM son:

1. RAM es el primer enfoque AOM, que integra diagramas de clase, estado y de secuencia en técnicas de orientación a aspectos. Como resultado los "aspect models" pueden describir la estructura y el comportamiento de una preocupación en particular.

2. La reutilización de "aspect models" en RAM es flexible. La flexibilidad se alcanza por medio de la composición o extensión de elementos ("bindings").

3. El "weaver" obliga a que todos los parámetros de instanciación obligatoria sean definidos cuando un aspecto es instanciado.

4. Un enlazamiento definido en un aspecto superior, debe ser compatible con los enlazamientos del aspecto a reutilizar.

5. La vista estructural describe las clases, propiedades, asociaciones e instancias del aspecto.

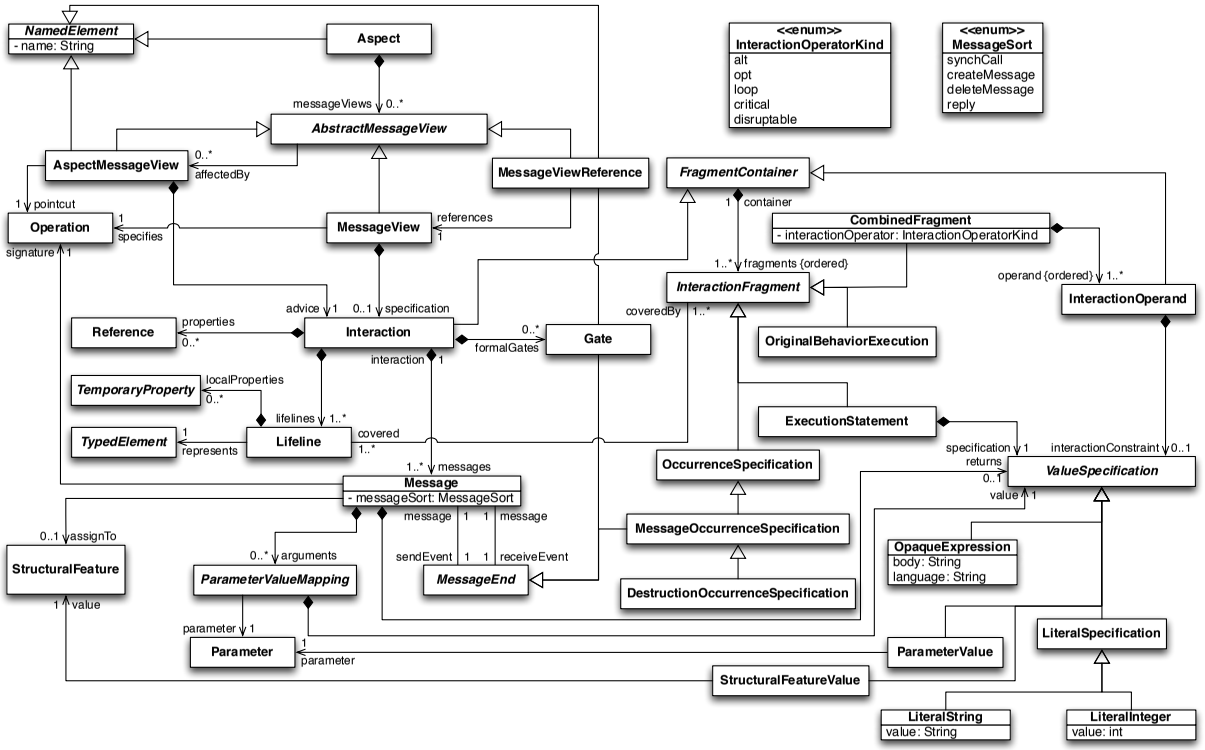
6. La vista de mensajes define el comportamiento del aspecto.

2.6. TouchRAM

TouchRAM [1] es una herramienta con una notación gráfica táctil que sirve para diseñar modelos RAM en forma ágil. Esto es posible, debido a que TouchRAM se apoya en el metamodelo y las reglas de AOM

definidas en RAM. Las funcionalidades soportadas por TouchRAM son:

1. Jerarquía de aspectos.
2. Vista estructural: edición, visualización y "weaving" de diagramas de clases.
3. Vista de mensajes: visualización y "weaving" de diagramas de secuencia.
4. Vista de estados: solo visualización de diagramas de estado.
5. Verificaciones simples de consistencia.



Voelter

Figura 2.6.1: Metamodelo de la vista de mensaje de RAM. tomado de [24]

2.7. Resumen

El presente capítulo introdujo los conceptos necesarios para el desarrollo de TextRAM. Los DSL fueron presentados como un lenguaje de especificación ejecutable que ofrece, a partir de notaciones y abstracciones adecuadas, un poder de expresividad enfocado y restringido a un problema de dominio en particular. Los ingredientes de un DSL son: (1) sintaxis concreta: define la notación del programa a utilizar. (2) sintaxis abstracta: estructura de datos que persiste el resultado de la sintaxis concreta. (3) semántica: describe el significado de los elementos representados en la sintaxis concreta y persistidos en la sintaxis abstracta. Las dos alternativas evaluadas para la implementación del DSL fueron XText y Rascal; XText es un generador de "parsers" que deriva la sintaxis abstracta a partir de la gramática; Rascal es un lenguaje de meta-programación extensible que ofrece mecanismos de análisis de código fuente y abstracción de código. Para la construcción de TextRAM se eligió XText, sobre Rascal debido a las siguientes razones: (1) corta curva de aprendizaje de XText, (2) El foco único de XText es la definición de DSLs, (3) Completitud de la documentación XText y su íntima relación al EMF (marco de trabajo de RAM). AOP es una propuesta que modulariza las preocupaciones transversales y mejora la separación de módulos en el software, resolviendo los problemas inherentes de modularización de los lenguajes de programación a objetos. RAM es un enfoque de modelamiento orientado a aspectos que resuelve los problemas de escabilidad y consistencia, presente en otros enfoques de modelamiento AOP. El resultado de RAM, son "aspect models" que se modelan por medio de diagramas de clase, estado y de secuencia. TextRAM es una representación textual de RAM y es una alternativa a TouchRAM, la única herramienta que existe para la definición de "aspect models", además el desarrollo de TextRAM estimulará el desarrollo e investigación de RAM. El siguiente capítulo explica el metamodelo de TextRAM y su relación con el metamodelo de RAM.

Capítulo 3

Metamodelo de TextRAM

El primer paso para construir un DSL en XText, es la especificación de la sintaxis concreta textual (TCS); a partir de la TCS se deriva la gramática y el modelo semántico del lenguaje (metamodelo). El objetivo del presente capítulo, es describir la TCS de TextRAM (ver sección 3.1), describiendo los detalles de la vista estructural y la vista de mensajes. La sección 3.2, expone los elementos que componen la gramática del lenguaje; la sección 3.3, describe el metamodelo de TextRAM y su equivalencia al metamodelo de RAM.

3.1. Sintaxis concreta textual

El metamodelo de RAM describe tres vistas (estructural, mensajes y estados). TextRAM contempla la vista estructural y la vista de mensajes (se omite la vista de estados ya que la misma estaba en proceso de definición). El metamodelo de RAM es la base del metamodelo de TextRAM. Para trabajar con metamodelos existentes, XText ofrece una metodología de trabajo que permite derivar la TCS a partir de un metamodelo existente; lo cual ayuda a refinar dicho metamodelo hasta alcanzar el resultado deseado. El modelo semántico final reutiliza partes del metamodelo de RAM y define nuevas entidades necesarias para la consecución de lo objetivos antes mencionados.

Según lo descrito en la sección 2.2.1; XText es un generador de parser que deriva la TCS y el AST a partir de la gramática. La gramática es la piedra angular del marco de trabajo de XText. Por tanto, para describir el metamodelo de TextRAM fue necesario iniciar por la conceptualización de la TCS, ya que la misma es la inspiración de la gramática. Una hoja de ayuda de la TCS se incluye en el apendíce 6.

3.1.1. Vista estructural

Un aspecto es un bloque de código que contiene la vista estructural, las instanciaciones y la vista de mensajes. El inicio de un bloque se representa por medio de la apertura de llave ({) y finaliza con el cierre de llave (}). La palabra clave para definir un aspecto es *aspect*; la vista estructural se representa por medio de la palabra *structure*; la palabra reservada *class* se utiliza para definir una clase dentro de la vista estructural. La vista estructural puede contener una lista de clases (sintácticamente, es correcto definir una vista estructural sin clases). Una clase posee operaciones y atributos. La definición de una operación consta de:

1. Un modificador de acceso, su sintaxis es:

   a) + para miembro públicos.

   b) - para miembros privados.

Tipo de Dato	Palabra clave
Verdadero/Falso	boolean
Entero	int
Carácter	char
Cadena variable de texto	String
Número flotante	float
Número flotante de doble precición	double

Cuadro 3.1: Tipos de datos primitivos

```
1  aspect Observer {
2    structure {
3      class |Subject {
4        + |Subject |create()
5        ~ Set<|Observer> getObservers()
6        + void |modify()
7        + void addObserver(|Observer a)
8        + void removeObserver(|Observer a)
9      }
10     class |Observer {
11       + void startObserving(|Subject s)
12       + void stopObserving()
13       ~ void |update(|Subject s)
14     }
15   }
16 }
```

Cuadro 3.2: Ejemplo de elementos básicos de una vista estructural

c) ~ para miembro intra-aspectos.

2. Un tipo de dato de retorno. El tipo de dato puede ser otra clase, un tipo de dato primitivo (ver tabla 3.1). También se puede utilizar dos tipos de datos especiales: `Set<TipoDato>` para la representación de un conjunto de elementos que representan el mismo tipo y `Any` para indicar que se puede retornar cualquier cosa. Si la operación no retorna nada se utiliza la palabra clave `void`.

3. Un nombre válido, debe respetar las siguientes reglas: cadena de carácteres de longitud variable de la letra 'a' a la 'z', el carácter guión bajo '_', números del '0' al '9'; el nombre permite la combinación de mayúsculas y minúsculas. Un nombre válido no debe contener espacios. En el caso de operaciones parciales se acepta el carácter de barra vertical |.

4. Una lista de parámetros. Los parámetros se definen dentro de un bloque de paréntesis, deben estar separados por coma y deben indicar como primera instancia un tipo de datos, seguidos de un nombre que los identifique.

Un atributo tiene un tipo de dato y un nombre; los mismos no tienen modificador de acceso y los únicos tipos de datos permitidos son los primitivos (ver Cuadro: 3.1). Las operaciones y atributos no necesitan ser definidas en un orden específico. El punto y coma se utiliza como delimitador de sentencias opcional; si se omite el punto y coma, el salto de línea es utilizado como delimitador de sentencias. Las clases y las operaciones pueden ser parciales; se utiliza una barra vertical `(|)` para definir la parcialidad de una clase u operación, dicho carácter debe ser el primer carácter del nombre. El cuadro 3.2 presenta un ejemplo con los elementos básicos de la vista estructural.

Las asociaciones son utilizadas para definir la relación entre dos clases y las mismas se declaran dentro de bloque *associations*. El nombre se define en la última sección del bloque *associations*. Los extremos de una

asociación se definen de izquierda a derecha. La primera sección de un extremo es opcional y representa su cardinalidad, para representar una multiplicidad de muchos elementos se utiliza el carácter asterisco (*). La segunda sección es el nombre de la clase involucrada. En una asociación unidireccional solo una clase está al tanto de la relación (->), en una asociación birideccional, ambas clases se conocen entre sí (&). En una relación de *agregación*, el hijo puede existir sin el padre; en una relación de *composición* un hijo no puede estar huérfano. La *agregación* se representa por medio del carácter (o) y la composición se representa por el carácter asterisco (*). La relación de composición o agregación debe ir antes de la dirección unidireccional o bidireccional. Un ejemplo de *agregación*, es la relación entre una sección y una empresa; una sección dentro de una empresa no puede existir por sí sola (ver Cuadro: 3.3).

Las directivas de instanciación son utilizadas para describir una jerarquía entre modelos; cuando se utiliza una instanciación se está describiendo una dependencia sobre otro aspecto. Los tipos de dependencia pueden ser:

1. Extensión: cuando *A extiende B*, se entiende como la adición de elementos estructurales o de comportamiento al aspecto *B*; en este caso *A* brinda propiedades alternativas, complementarias o adicionales al aspecto *B*. La parabra clave de una extensión es *extends*. En este caso *A* tiene permitido ver todos los miembros de *B* (incluso los privados y los intra-aspectos).

2. Personalización: cuando *A depende de B*, se entiende como la adaptación de la estructura y comportamiento de un modelo base *B*. En este caso *A* debe alterar o aumentar las propiedades del modelo base *B*, el objetivo es transformar dichas propiedades en una nueva funcionalidad. En la personalización, *A* solo tiene acceso a los miembros públicos de *B*.

La declaración del tipo de instanciación se realiza en la parte superior del aspecto, inmediatamente después del nombre del mismo. El mapeo de elementos entre el aspecto actual y sus aspectos dependientes, se realiza en el bloque *instantiations* que debe ir después del bloque de asociaciones de la vista estructural.

En el ejemplo del Cuadro 3.2, *Observer* ofrece una nueva semántica a los comportamientos que expone *ZeroToMany*; gracias a la reutilización de *ZeroToMany*, *Observer* es capaz de crear, obtener, eliminar y adicionar nuevos *Observers*. *Observer* puede ser reutilizado por todos los aspectos que necesiten implementar dicho patrón de diseño.

Para indicar que una clase hereda de otra se utiliza el carácter dos puntos (:); en RAM es posible la herencia múltiple, se debe utilizar coma como separador de supertipos. Las clases y operaciones abstractas se señalan con la palabra clave *abstract*. Los miembos estáticos son representados por la palabra clave *static*. Actualmente un modelo en RAM, se puede representar y manipular gráficamente por medio de TouchRAM (ver sección 2.6); para indicar la posición de las clases dentro de TouchRAM se utilizan coordenadas (*x*,*y*);

```
1  aspect Empresa {
2    structure {
3      class Empresa {
4      }
5
6      class Seccion {
7      }
8      associations {
9        1 Empresa o-> * Seccion { miSeccion }
10     }
11   }
12 }
```

Cuadro 3.3: Ejemplo de agregación

```
1  messages {
2    lifelines {
3      ref target:|Subject {
4        observers:|Observer
5        o:|Observer
6      }
7      ref o:|Observer
8      assoc mySubject:mySubject
9    } ...
```

Cuadro 3.4: Declaración de *Lifelines*

en TextRAM dichas coordenas se ubican opcionalmente arriba de la declaración de una clase; un ejemplo de coordenadas de posición es: (`@@x=297.0003 @@y=119.009`).

3.1.2. Vista de mensajes

La vista de mensajes inicia con el bloque *messages,* y se divide en dos: el encabezado y el cuerpo de los mensajes. El encabezado del mensaje contiene la declaración de todos los *lifelines* utilizados por la vista de mensajes. Un *lifeline* se define con su nombre y la entidad que representa (separados por dos puntos). Los *lifelines* pueden representar cuatro tipos de artefactos de la vista estructural: *referencias a clases, la instancia misma de una clase, asociaciones y atributos.* Opcionalmente un *lifeline* puede declarar *propiedades locales* separadas por punto y coma; una *propiedad local* se utiliza para guardar el valor de retorno de un método o para mantener el valor de un parámetro.

Existen dos tipos de *propiedades locales*: *referencias* y *atributos*; las *referencias* son apuntadores a clases dentro de la vista estructural y los atributos son nuevas declaraciones dentro del *lifeline,* incluso se puede declarar valores por defecto para los *atributos* declarados dentro del *lifeline.* El ejemplo del Cuadro, muestra tres ejemplos de *lifelines*: (1) *target* en la línea 3, que hace referencia a la clase parcial *Subject* y tiene dos *propiedades locales*: *observers y o*; ambas propiedades representan a la clase *Observer;* (2) *lifeline* cuyo nombre es representado por la letra *o* en la línea 5 y hace referencia a *Observer*; (3) *mySubject* en la línea 8, que representa a una asociación.

Los mensajes pueden ser de tres tipos: vista de mensajes de aspectos, vista de mensajes de referencia y la vista de mensajes convencional. La vista de mensajes de aspectos se declara por medio del bloque *aspectMessageView* y consta de dos secciones: *pointcut* y *advice.* En RAM los *pointcut* especifican operaciones; la especificación de la operación no puede ser ambigua (se debe especificar la firma concreta de la operación y la clase a la que pertenece). El *advice* puede extender, remover o modificar el comportamiento de la operación; el cuerpo del *advice* está compuesto por un conjunto de *interacciones.* La vista de mensajes de referencia utilizan la palabra clave *affectedby* y también especifican una operación. La vista de mensajes convencional hace referencia a una operación dentro de la vista estructural y contiene una lista de *interacciones.* Las *interacciones* pueden ser de tres tipos:

1. Interacciones convencionales: representan la interacción entre dos *lifelines*; para indicar la dirección de la interacción se utiliza la siguiente flecha: (=>). Se utiliza la parabra clave *new* cuando se hace la llamada a un constructor. En la vista de mensajes de aspectos se puede hacer referencia al comportamiento original, esto se representa por el carácter (*). El mensaje de la interacción se encuentra encerrado entre llaves. Para indicar la destrucción de un objeto, se utiliza la letra 'X' después de la flecha de dirección de la interacción.

2. Elementos combinados: son interacciones que representan bloques de tipo: *"loop", "alt", "opt".* Los ele-

mentos combinados son bloques que pueden contener otras interacciones. Un elemento combinado tiene una restricción. Un ejemplo de restricción dentro de un *loop,* puede ser: "`for Observer o : observers`".

3. Interacción que representan el retorno de la secuencia: se representan por la palabra clave *return* y pueden devolver una *propiedad temporal, un parámetro o un lifeline*; si la operación involucrada tiene un tipo de dato *void,* la palabra *return* se especifica sin un valor de retorno. La vista de mensajes del aspecto *Observer* se presenta en el cuadro 3.5.

```
1  aspect Observer dependsOn ZeroToMany {
2    structure {
3      ...
4    }
5  }
6  messages {
7    lifelines {
8      ref target:|Subject {
9        observers:|Observer
10       o:|Observer
11     }
12     ref o:|Observer
13     assoc mySubject:mySubject
14   }
15   aspectMessageView notification {
16     pointcut |Subject.|modify()
17     advice {
18       target =>*
19       target => target { observers := getObservers() }
20       loop [ |Observer o : observers ] {
21         target => o { |update() }
22       }
23     }
24   }
25   messageView |Observer.startObserving(s) {
26     o => mySubject { addObserver() }
27   }
28   messageView |Observer.stopObserving() {
29     o => mySubject { removeObserver() }
30   }
31   messageView |Subject.|modify() affectedBy notification
32   }
33 }
```

Cuadro 3.5: Vista de mensajes del aspecto *Observer*

3.2. Gramática

En XText, el metamodelo de un DSL se deriva a partir de la gramática. La sección 2.2.1 describe las reglas para la definición de una gramática en XText. El nombre de la gramática de TextRAM es: *"http://cl.pleiad.textram/1.0"* e importa dos metamodelos: (1) el metamodelo de Ecore: *"http://www.eclipse.org/emf/2002/* (2) el metamodelo de RAM: *"http://cs.mcgill.ca/sel/ram/2.1"*. Las principales reglas de la gramática de TextRAM son:

- *TAspect*: representa un aspecto, declara una lista de dependencias y contiene la vista de clases y secuencias de RAM.

- *TStructuralView*: reutiliza los conceptos de la vista estructural de RAM y contiene la definición de las clases y asociaciones.
- *TClass*: representa una clase dentro de TextRAM. Permite la definición no ordenada de operaciones y atributos.
- *TClassMember*: es una regla abstracta que representa los miembros de una clase: *TOperation* y *TAttribute*. La regla *TOperation* puede contener una lista de parámetros *TParameter*.
- *TAssociation*: representa una asociación con dos extremos *TAssociationEnd*. Un *TAssociationEnd* indica la referencia de clase y los límites inferiores y superarioes de la asociación.
- *TInstantiationHeader*: definen una lista de los aspectos externos que se están extendiendo o personalizando.
- *TClassifierMapping*: son declarados dentro de la regla *Instantiation* y permiten el mapeo entre los miembros (*TClassMember*) de clase del aspecto actual y los miembros de clase de los aspectos externos.
- *TAbstractMessages*: abstracción que permite la declaración de dos tipos de vista de mensajes: *TAspectMessageView* y *TMessageView*.
- *TLifeline*: representa un *lifeline* y define el tipo de referencia del mismo (*TLifelineReferenceType*). Un *TLifeline* tiene un tipo de referencia *TLifelineReferenceType* y puede definir una lista de propiedades locales (*TTemporaryProperty)*.
- *TLifelineReferenceType*: es una enumeración que representa el tipo de referencia de un *TLifeline* (*reference, association, attribute)*.
- *TTemporaryProperty*: es una abstracción que permite la representación de referencias (*TReference*) y atributos locales (*TLocalAttribute*).
- *TInteraction*: es una abstracción que permite a la vista de mensajes contener distintos tipos de interacción: *TInteractionMessage* para interacciones convencionales, *TCombinedFragment* para interacciones que representan un bloque de decisión o un ciclo, *TOcurrence* para represntar el comportamiento original o la destrucción de una interacción y *TReturnInteraction* para representar el retorno de un mensaje.
- *TMessage*: representa un mensaje dentro de una interacción. Puede contener una firma que apunta a una operación (*TOperation*), una lista de argumentos (*TValueSpecification*) y un *TMessageAssignableFeature* para indicar que el mensaje será asignado a una *TAssociation, TTemporaryProperty* o un *TLifeline*. Un *TValueSpecification* es una abstracción de un *TTemporaryProperty, TParameter, TLifeline y* una regla especial llamada *TDummyValueSpecification* utilizada para no entrar en conflicto con las otras reglas.

La gramática completa se puede apreciar en el apendíce 6.

3.3. Metamodelo de TextRAM

El metamodelo de TextRAM define nuevos elementos y reutiliza entidades del metamodelo de RAM (descritas en 2.5.5). La transformación de modelos (RAM a TextRAM y viceversa) exige el conocimiento de las similitudes y diferencias de dichas herramientas. La afinidad entre ambos metamodelos es la siguiente:

- *Aspect* vs *TAspect*: un aspect en RAM contiene la vista estructural, la vista de mensajes, la vista de estados, las instanciaciones y las coordenadas del "layout" gráfico. Un aspecto en TextRAM solo posee la vista estructural, la vista de mensajes y las instanciaciones; en TextRAM el "*layout*" se define a nivel de cada clase y la vista de estados aún no está soportada. Además, *TAspect* permite la definición de múltiples aspectos externos por medio de *TInstantiationHeader*.

- *StructuralView* vs *TStructuralView*: *TStructuralView* hereda todas los elementos de *StructuralView* (una lista de "*Classifiers*", asociaciones y todos los tipos de datos permitidos en RAM). La diferencia radica en que *TStructuralView* soporta las asociaciones específicas de TextRAM (*TAssociations*)

- *Class* vs *TClass*: ambos son un "*Classifier*", pero *Class* exige una TCS en donde el bloque de las operaciones se define antes de la declaración de los atributos, *TClass* permite definir operaciones y atributos en forma intercalada por medio de *TClassMember*; *TClass* también brinda la flexibilidad de que el carácter de barra vertical sea parte del nombre de una clase o una superclase (el carácter "|" es utilizado para identificar una clase parcial). Finalmente el "*layout*" es definido a nivel de *TClass,* no a nivel de *StructuralView*.

- *Operation* vs *TOperation*: estos elementos no comparten una abstracción común, pero si comparten una visibilidad, un valor de retorno, un nombre y una lista de parámetros. La abstracción base de *TOperation* es *TClassMember*.

- *Attribute* vs *TAttribute*: estos elementos no comparten una abstracción común, pero si comparte la definición de un tipo de dato y un nombre. El tipo de dato de ambos elementos debe ser un *PrimitiveType*.

- *Parameter* vs *TParameter*: ambos son utilizados por las operaciones, pero no comparten un ancestro común; dichos elementos permiten la definición de un tipo de dato y de un nombre.

- *Association* vs *TAssociation*: *Association* tiene un nombre y una lista de extremos de tipo *AssociationEnd*; en cambio *TAssociation* soporta solo dos extremos de tipo *TAssociationEnd* y también soporta la definición del tipo de multiplicidad entre asociaciones (unidireccionales o bidireccionales).

- *AssociationEnd* vs *TAssociationEnd*: los elementos de tipo *Class* tienen asociado un lista de *AssociationEnd*; está relación en *TextRAM* se realiza desde *TAssociationEnd* a *TClass*; por tanto un *AssociationEnd* puede estar asociado a una sola clase, brindadno mayor simplicidad al metamodelo. Ambos elementos soportan la definición de límites inferiores y superiores de la asociación; dichos límites son del tipo *ElementBound* (permite la definición de un número entero y se utiliza un "*" en caso de existir múltiples ocurrencias de una clase).

- *ClassifierMapping* vs *TClassifierMapping*: el objetivo de ambos es mapear los elementos del aspecto actual con los elementos de un aspecto externo. Su diferencia radica en que *ClassifierMapping* utiliza diferentes listas para mapear atributos y operaciones; en cambio *TClassifierMapping* utiliza una sola lista para representar el mapeo de atributos y operaciones por medio de la abstracción *TClassMember*.

- *AbstractMessageView* vs *TAbstractMessageView*: la vista estructural de RAM soporta múltiples vistas de mensajes de diferentes tipos (*AspectMessageView*, *MessageViewReference* y *MessageView*); para soportar la flexibilidad deseada en la TCS, TextRAM tuvo que redefinir todos los elementos de la vista de mensajes, lo que significa que los elementos de la vista de mensajes de TextRAM no heredan las características de RAM, simplemente se construyeron elementos equivalentes que permiten describir todas la variaciones de la vista de mensajes de RAM. *TAspect* solo tiene un *TAbstractMessageView,*

dicho elemento sirve como encabezado de la vista de mensajes, contiene la declaración de los *TLifelines* y una lista de *TAbstractMessages* que es la abstracción para la definición de *TMessageView* y *TAspectMessageView*.

- *Lifeline* vs *TLifeline*: Un *Interaction* contiene muchos *Lifelines*, a su vez un *Lifeline* conoce todos los *InteractionFragment* que forman parte de su ciclo de vida; en cambio *TLifeline* es un bloque cuyo propósito es declarar todos los *lifelines* a utilizar en las distintas vistas de mensajes de los aspectos. Un *TLifeline* también posee una lista de propiedades locales (*TTemporaryProperty*) que serán usados como valor de retorno de una operación o como parámetro de operación.

- *MessageViewReference, MessageView* vs *TMessageView*: el modelo de TextRAM contempla dos conceptos de RAM: *MessageViewReference* y *MessageView*. *MessageView* es una vista de mensajes convencional que soporta la declaración de distintas interacciones, *MessageViewReference* permite hacer referencia a una vista de mensajes de un aspecto derivado, en este tipo de vista de mensajes, es obligatorio indicar el aspecto derivado por medio de la palabra clave *affectedBy*. Si la palabra clave *affectedBy* aparece dentro de un *TMessageView*, el mismo es considerado como un equivalente de *TMessageViewReference,* en caso contrario es considerado como un *MessageView*. Semánticamente no es correcto tener interacciones dentro de un *MessageViewReference*, por tanto se valida que no existan interacciones dentro de un *TMessageViewReference*.

- *AspectMessageView* vs *TAspectMessageView*: ambos comparten la misma semántica y soportan la definición de *pointcut* y *advice*.

- *Interaction* vs *TInteraction:* el primero es una especialización de *FragmentContainer* y un *FragmentContainer* posee una lista de *InteractionFragment*; *InteractionFragment* es un elemento clave dentro del contexto de *Interaction*, ya que este permite soportar varios tipos de interacciones por la referencia polimórfica que exite entre *InteractionOperand* e *Interaction*. *TInteraction* es una abstracción *TInteractionMessage, TCombinedFragment, TOcurrence* y *TReturnInteraction*. *TCombinedFragment* tiene dos propiedades: *containers* y *elseContainers*; debido a que ambas propiedades son de tipo *TInteraction* es posible expresar cualquier tipo de interacción dentro de un *TCombinedFragment*.

- *Message* vs *TMessage*: ambos están gobernados dentro de una interacción. *Message* contiene dos eventos (envío y recepción), dichos eventos son de tipo *InteractionFragment;* esto permite a un mensaje soportar *MessageOcurrenceSpecification, Gates* o *CombinedFragments* ya que los mismos son especializaciones de *InteractionFragment*. *TMessage* representa la llamada a una operación por medio de su propiedad *signature* y el resultado de dicha operación se puede asignar opcionalmente a un *TMessageAssignableFeature* (*TAssociation*, *TTemporaryProperty*, *TLifeline*).

- *Gates:* estos son utilizados para representar la entrada y salida de una interacción; TextRAM omite este concepto debido a que es redundante escribir la entrada y salida de todas las ocurrencias de una interacción. En TextRAM, el valor de retorno de una interacción se define explícitamente por medio de *TReturnInteraction*.

- *ValueSpecification* vs *TValueSpecification*: ambas permiten describir el valor de retorno de un mensaje y expresar restricciones dentro de un *CombinedFragment*.

3.4. Resumen

XText es un generador de "parsers" que deriva la TCS y el AST a partir de la gramática. La conceptualización de la TCS fue el punto de partida para la definición de la gramática. Los elementos de la gramática derivan el metamodelo de TextRAM. Cada elemento de la gramática tiene una equivalencia con el metamodelo de RAM. A partir del metamodelo de TextRAM se pueden construir herramientas complementarias al lenguaje: definición de alcance, formateadores de la TCS, generadores de código y validaciones necesarias para cumplir con las restricciones semánticas del lenguaje; el siguiente capítulo, explica la implementación del lenguaje de TextRAM y sus herramientas complementarias.

Capítulo 4

Implementación

XText ofrece un marco de trabajo bien definido, que tiene como primer requisito la configuración del lenguaje por medio del motor del flujo de trabajo para el modelado (MWE2); la configuración del lenguaje se explica en la sección 4.1. La gramática de TextRAM necesita la interpretación de ciertos símbolos especiales, dicha interpretación se realiza a través de un mecanismo llamada conversor de valores (explicado en la sección 4.2). La definición de alcance ("scoping") es necesaria para la definición de la visibilidad de los elementos, y su completitud es necesaria para la transformación de modelos, el "scoping" específico a TextRAM es estudiado en la sección 4.3. Un metamodelo necesita de restricciones conformes a la semántica del lenguaje, las validaciones semánticas de TextRAM están descritas en la sección 4.4. El formateador es un mecanismo que ayuda a la comprensión de la TCS, las reglas de formato son vistas en la sección 4.5. La transformación de un modelo de TextRAM a un modelo de RAM (y viceversa) se realiza gracias a los generadores de código descritos en la sección 4.6.

4.1. Configuración del lenguaje

El desarrollo en XText inicia por la configuración del lenguaje, a través de fragmentos definidos en el motor de flujo de trabajo para el modelado (MWE2). El MWE2, permite componer grafos de objetos en forma declarativa y compacta. Su eje central es la transformación de elementos EMF a una API de Java destinada a gestionar el modelo Ecore resultante. Las reglas de transformación se definen declarativamente en forma de fragmentos, por defecto XText define un conjunto estándar de fragmentos que acompañan al proceso de construcción del DSL (ejemplo: validadores, serializadores, generadores de código, etc.). La configuración de los fragmentos de TextRAM se puede visualizar en la Figura 4.1.1.

El metamodelo de RAM fue reutilizado en la gramática de TextRAM, esto fue posible gracias a la importación del modelo Ecore de RAM. Los requisitos básicos para importar un modelo Ecore existente son:

1. Registro del proyecto o ejecutable a importar en el *classpath* del proyecto actual.
2. Registro del paquete a importar en el MWE2 (fragmento *Standalone*, propiedad *registerGeneratedEPackage*).
3. Registro del archivo físico a importar en el MWE2 (fragmento *Standalone,* propiedad *registerGenModelFile*).
4. Definición de puntos de extensión dentro del archivo plugin.xml del proyecto DSL y el proyecto GUI de TextRAM.

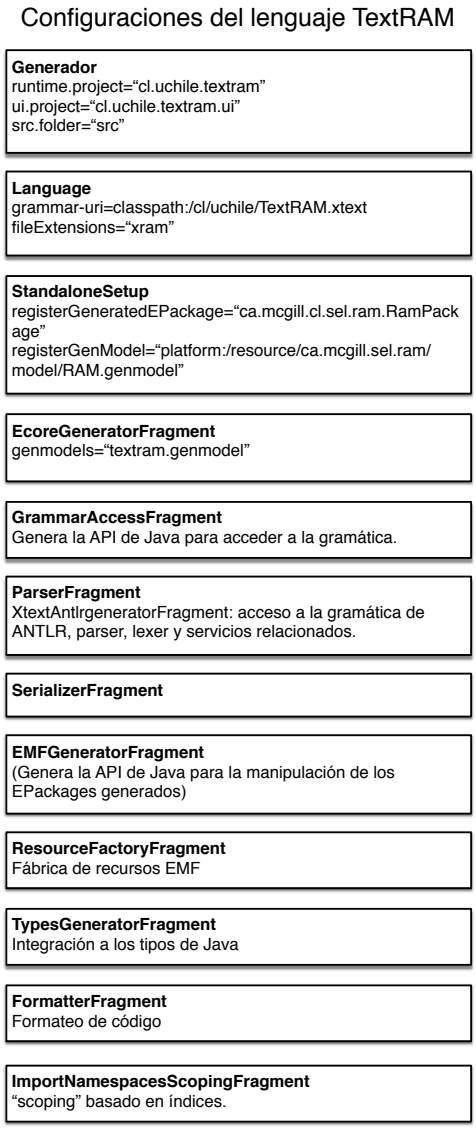


Figura 4.1.1: Configuración del Lenguaje TextRAM

La clase *TextRAMStandaloneGenerated* es una clase que implementa la interfaz *ISetup* y fue generada automáticamente por el marco de trabajo de XText; el objetivo de dicha clase es registrar todos los recursos EMF del proyecto e inyectar la dependencia de los objetos (el "framework" de inyección de depencia por defecto es Google Guice). Para poder ejecutarse como una aplicación *Standalone,* el paquete EMF de TextRAM fue registrado en *TextRAMStandaloneGenerated.*

Los conceptos específicos de la gramática de TextRAM fueron discutidos en la sección 3.2 como parte de la descripción del metamodelo. Los ingredientes que complementan la gramática son los diversos fragmentos de configuración del lenguaje discutidos en las secciones que siguen a continuación.

4.2. Conversor de valores

En ocasiones la representación textual de una regla de la gramática no coincide con la definición de su tipo de dato; éste es el caso de la representación textual de las ocurrencias de una asociación. Su regla indica

```
1  package cl.uchile.pleiad.converter;
2
3  import org.eclipse.xtext.common.services.DefaultTerminalConverters;
4  import org.eclipse.xtext.conversion.IValueConverter;
5  import org.eclipse.xtext.conversion.ValueConverter;
6  import org.eclipse.xtext.conversion.ValueConverterException;
7  import org.eclipse.xtext.nodemodel.INode;
8  import org.eclipse.xtext.util.Strings;
9
10 public class TextRAMConverter extends DefaultTerminalConverters {
11
12   @ValueConverter(rule = "ElementBound")
13     public IValueConverter<Integer> ElementBound() {
14         return new IValueConverter<Integer>() {
15
16       @Override
17       public Integer toValue(String string, INode node)
18           throws ValueConverterException {
19
20         if (Strings.isEmpty(string))
21                   throw new ValueConverterException("Couldn't convert empty string to
                         int", node, null);
22                 else if ("*".equals(string.trim()))
23                   return -1;
24                 try {
25                   return Integer.parseInt(string);
26                 } catch (NumberFormatException e) {
27                   throw new ValueConverterException("Couldn't convert '"+string+"' to
                         int", node, e);
28                 }
29       }
30
31       @Override
32       public String toString(Integer value)
33           throws ValueConverterException {
34                 return ((value == -1) ? "*" : Integer.toString(value));
35       }
36         };
37   }
38 }
```

Cuadro 4.1: Ejemplo de implementación de un "Value Converter"

que su tipo de dato es un número entero que sirve para indicar el número de instancias de una clase dentro de una asociación, pero la representación de un número infinito de ocurrencias se realiza por medio de un "*" un carácter que obviamente no es un número entero.

Los "Value Converters" se utilizan para convertir el texto "parseado" a su tipo de dato correspondiente y viceversa. El registro de los conversores de valores se puede realizar en forma declarativa por medio de la implementación del servicio *AbstractDeclarativeValueConverterService* y el registro de métodos anotados con *IValueConverter*. El conversor de valores que resuelve la representación textual de las ocurrencias de una asociación se puede ver en la clase del Cuadro 4.1.

4.3. Alcance ("Scoping")

El "scoping" se preocupa de la visibilidad de los objetos y de la resolución de las referencias cruzadas. La API de "scoping" se apoya en los conceptos de EMF para definir que elementos son referenciables por una determinada referencia. Los principales conceptos del "scoping" son:

- *Contenedores de referencia:* la referencia de un objeto está contenida dentro del objeto referenciado

```
TOperation returns textRam::TOperation:
  name=EString '('
  (parameters+=TParameter(","parameters+=TParameter)*)?')';
```

Cuadro 4.2: Ejemplo de un contenedor de referencia

```
Instantiation returns Instantiation:
        externalAspect=[Aspect|EString];
```

Cuadro 4.3: Ejemplo de referencia cruzada

(también llamado contenedor). Un elemento es una referencia (tipo *EReference*) cuando el mismo no es un tipo básico (string, integer, etc). La clase *EReference* de *Ecore* tiene una propiedad llamada *containment* que define si una referencia es de tipo contenedora o no. En el ejemplo del Cuadro 4.2, *TParameter* es un miembro contenido dentro de *TOperation*; por tanto, *parameters* es una contenedora de referencias multi-valor de tipo *TParameters.*

- *Referencias cruzadas:* El objeto referenciado se guarda en un lugar distinto al objeto que hace la referencia; el objeto referenciador guarda un apuntador al elemento referenciado. La gestión de las referencias cruzadas se realiza sobre los elementos cuya propiedad *containment* es falsa. El fragmento de código del Cuadro 4.3, ilustra una referencia cruzada al objeto *Aspect*; *Instantiation* tiene un apuntador a una instancia de *Aspect* que está almacenada en otro lugar.

- *El índice*: en un principio, todos los objetos que tienen un nombre son visibles dentro de un DSL modelado en XText; el componente encargado de gestionar la visibilidad es el índice; dicho mecanismo almacena la información de cada objeto por medio de la abstracción *IEObjectDescription.* Por cuestiones de rendimiento y eficiencia, el índice solo almacena el nombre del objeto y su ubicación (URI).

Las etapas involucradas en el "scoping" son las siguientes:

- *"Parsing"*: XText "parsea" el programa y crea el correspondiente modelo EMF (modelo semántico); durante esta etapa las referencias cruzadas aún no han sido resueltas.
- *"Indexing"*: Todos los componentes de los modelos EMF son procesados, la descripción del objeto es incluida dentro del índice.
- "*Linking*": Este servicio resuelve las referencias cruzadas por medio de proveedor de alcance ("scope provider").

La declaración del "scoping" se puede hacer declarativamente; XText genera una clase XTend que hereda de *AbstractDeclarativeScopeProvider* y su nombre por defecto es [Nombre DSL]ScopeProvider.xtend (en este caso la clase se llama TextRAMScopeProvider.xtend). Cada método dentro de TextRAMScopeProvider resuelve la visibilidad de un "feature" contenido en las reglas de la gramática. Cada uno de los métodos debe respetar la nomenclatura del Cuadro 4.4. Con dicho patrón se indica el deseo de implementar el alcance para el <nombre de feature> de la clase <nombre de EClass> cuando el contexto sea <tipo de contexto>. El ejemplo del Cuadro 4.5 muestra la definición del "scoping" para el "feature" *returnType* de la regla *TOperation.*

```
def IScope scope_<nombre de EClass>_<nombre de feature> (<tipo de contexto> context,
    EReference reference)
```

Cuadro 4.4: Nomenclatura para la declaración de un "scoping"

```
def IScope scope_TOperation_returnType(TStructuralView structuralView, EReference
    reference)
```

Cuadro 4.5: Ejemplo de "scoping"

La serialización es el proceso que convierte la instancia de un modelo EMF a su representación textual. El "scoping" juega un papel importante dentro del proceso de serialización, ya que dicho mecanismo depende del "scoping" para la resolución de las referencias cruzadas. La serialización de los modelos de TextRAM requirió la definición del alcance para todas las reglas que tienen referencias cruzadas. La descripción del "scoping" de TextRAM se puede observar en la siguiente tabla:

Regla	"Feature"	Descripción
Instantiation	externalAspect	Permite hacer referencia a los aspectos externos definidos en el encabezado del aspecto.
TOperation	returnType	Los tipos de datos de retorno permitidos son: los tipos primitivos, clases y todos los tipos Set.
TAttribute	type	La definición de un atributo, permite solo tipos de datos primitivos.
TParameter	type	Los tipos de datos permitidos son los tipos primitivos, clases y todos los tipos Set.
ClassifierMapping	fromElement	Son visibles todas las clases provenientes de los aspectos externos definidos en la instanciación actual.
ClassifierMapping	toElement	Permite solo las clases definidas dentro del aspecto actual.
ClassifierMapping	fromMembers	Son visibles los miembros pertenecientes a la clase especificada en el fromElement del ClassifierMapping actual.
ClassifierMapping	toMembers	Son visibles los miembros pertenecientes a la clase especificada en el toElement del ClassifierMapping actual.
TAbstractMessages	class	Son visibles todas las clases de los aspectos externos definidos en el encabezado del aspecto actual.
TAbstractMessages	specifies	Son visibles todas las operaciones que pertenecen a la clase especificada en la vista de mensajes actual (es permitido definir clases de aspectos externos).

Regla	"Feature"	Descripción
TLifeline	represents	Son visibles todas las asociaciones, atributos, clases del aspecto actual o de los aspectos externos.
TMessage	assignTo	Son visibles las propiedades locales del *TLifeline* de la izquierda, todas las asociaciones y el *TLifeline* de la derecha. El contexto de visibilidad es el *TInteractionMessage* que se está editando.
TMessage	signature	Filtra todas las operaciones definidas en la clase involucrada en la interacción del mensaje. "Signature" debe cumplir con la firma de la operación seleccionada.
TReturnInteraction	return	Permite retornar las propiedades temporales, parámetros, *TLifelines* definidos dentro del mensaje actual.
TMessage	arguments	Son visibles los parámetros que pertenecen a la operación especificada por el mensaje, todos los *TLifelines* involucrados en el mensaje, y las propiedades locales de los *TLifelines* especificados en la interacción actual.
TMessageView	affectedBy	Obtiene todos los "aspect messages views" del aspecto actual.
TAbstractMessages	arguments	Son visibles todos los parámetros de la operación especificada en la vista de mensajes actual.
TAssociationEnd	classReference	Obtiene todas las clases pertenecientes a las clases externas definidas en el encabezado del aspecto.

Cuadro 4.6: Definición de "scoping"

4.4. Validaciones

La validación es el mecanismo por el cual se aplican validaciones semánticas sobre el modelo. La generación de código no se efectúa si el modelo no es válido. Las validaciones se dividen en dos grupos:

1. *Validaciones por defecto*: son validaciones genéricas incorporadas dentro del marco de trabajo de XText; un ejemplo clásico es la resolución de las referencias cruzadas, si la referencia no se encuentra, el editor muestra el siguiente error: "Couldn't reference to...". Los validadores por defecto se configuran en el archivo MWE2 de cada proyecto.

2. *Validaciones personalizados:* son las validaciones semánticas del modelo. Las validaciones personalizadas se definen en una clase XTend generada automáticamente dentro del paquete "validation"; la clase que realiza las validaciones de TextRAM se llama *TextRAMValidator*; la herramienta interpreta los métodos

```
1  @Check
2  def checkNoDuplicatesAttributes(TAttribute attr) {
3    val owner = TextRamEcoreUtil.getRootContainerOfType(attr, TextRamPackage.Literals.
       TCLASS) as TClass;
4
5    val count = owner.members.filter( m | m.name == attr.name ).size
6
7    if ( count > 1 ) {
8      error('''Duplicate member 'Â« attr.name Â»' ''',TextRamPackage.eINSTANCE.
         TClassMember_Name, DUPLICATE_ELEMENT)
9    }
10 }
```

Cuadro 4.7: Ejemplo de un validador

anotados con *@Check* como métodos validadores del modelo. El parámetro de dichos métodos debe ser la instancia del modelo a validar; el nombre del método no es considerado como parte del proceso de configuración de la validación. El fragmento de código del Cuadro 4.7, valida que no existan atributos duplicados dentro de una clase.

TextRAM aplica las siguientes validaciones semánticas:

- En el contexto de una clase, no es permitido definir atributos con el mismo nombre.
- En el contexto de una clase, no es permitido definir operaciones con el mismo nombre.
- En el contexto de un aspecto, no es permitido definir clases con el mismo nombre.
- En el contexto de una interacción en la vista de mensajes, se valida que la firma de la operación especificada exista dentro de la clase definida en la interación.
- En el contexto de una vista de mensajes, se valida que la firma de la operación especificada exista dentro de la clase definida en la interación.
- Se valida que el nombre de una clase parcial inice con una barra vertical '|' y no contenga espacios a la derecha.
- Se valida que el nombre de una operación parcial inice con una barra vertical '|' y no contenga espacios a la derecha.
- Si el modelador especifica la sección "messages" se valida que exista al menos una vista de mensajes.

4.5. Formateadores

Un modelo semántico de EMF no contiene información acerca de su representación textual; por tanto los espacios en blanco no son considerados como parte del modelo EMF. XText mantiene la información sintáctica (longitud y el "offset") en un modelo en memoria llamada *node model*. XText utiliza un formateador por defecto para obtener una representación textual de las partes del modelo. El modelador puede definir un formateador personalizado por medio de una clase XTend generada automáticamente dentro del paquete "formatting"; en TextRAM dicha clase se llama *TextRAMFormatter*. Los fragmentos del modelo a formatear, se enuncian a continuación:

- Se realiza un salto de línea después del inicio de un bloque.

```
1  class TextRAMFormatter extends AbstractDeclarativeFormatter {
2
3    @Inject TextRAMGrammarAccess g
4
5    override protected void configureFormatting(FormattingConfig c) {
6      val f = grammarAccess
7
8      // structural view's indent
9      c.setIndentation( g.TStructuralViewAccess.leftCurlyBracketKeyword_2 , g.
           TStructuralViewAccess.rightCurlyBracketKeyword_4_4 )
10   }
11 }
```

Cuadro 4.8: Especificación por nombre

- Se realiza un salto de línea antes y después del cierre de un bloque.
- Elimina los espacios antes y después de las ocurrencias de la apertura y cierre de los parentésis.
- Elimina los espacios antes y después de las ocurrencias del signo mayor y menor.
- Elimina los espacios en blanco después del carácter de barra vertical "|"
- Elimina los espacios en blanco antes y después de un punto.
- Elimina los espacios en blanco después de la lista de comas definidas en el encabezado de las instanciaciones.
- Elimina los espacios en blanco antes y después del carácter ':' contenido en la definición de una propiedad local.
- Indenta las secciones anidadas dentro del aspecto a modelar.

El formateador también es utilizado dentro del proceso de serialización. El tema de serialización se verá en la sección 4.6. La desventaja del formateador es que ciertas definiciones son explícitas y se apoyan de la estructura del metamodelo; si el metamodelo cambia, el formateador corre el riesgo de quedar obsoleto. El ejemplo del Cuadro 4.8 muestra la especificación por nombre de los símbolos de apertura y cierra del bloque de la vista estructural (ver línea 9). En cambio, el ejemplo del Cuadro 4.9 (ver líneas de la 11 a la 9), es más genérico debido a que hace referencia a los símbolos '{' y '}' y no tiene dependencia del metamodelo. Dicho ejemplo indica como incluir un salto de línea después de la ocurrencia de un inicio de bloque y un salto de línea antes y después del cierre de un bloque.

4.6. Generadores

La generación de código es la parte más importante del EMF. Dicha generación permite la transformación de modelos a texto (M2T) y la transformación de modelos a otros modelos (M2M). La generación de código en XText ocurre después de que el modelo ha sido "parseado" y validado. La resolución de dependencias a otros modelos es resuelta por la infraestructura de XText. Los artefactos generados se almacenan relativamente dentro del proyecto actual; la abstracción *IFileSystemAccess* se preocupa de almacenar el artefacto generado en la ubicación relativa definida por el modelador. La transformación de los modelos TextRAM a los modelos RAM es gestionada por la clase *TextRAMGenerator*. En XText la generación de código se realiza por medio de las plantillas XTend descritas en el punto 7.

```
1  class TextRAMFormatter extends AbstractDeclarativeFormatter {
2
3    @Inject TextRAMGrammarAccess g
4
5    override protected void configureFormatting(FormattingConfig c) {
6
7      val f = grammarAccess
8
9          f.findKeywords("{").forEach[ k | c.setLinewrap.after(k) ]
10
11     f.findKeywords("}").forEach[ k |
12       c.setLinewrap.before(k)
13       c.setLinewrap.after(k)
14     ]
15   }
16 }
```

Cuadro 4.9: Especificación por símbolo

Como se menciona en la sección 2.5.4, un aspecto en RAM puede reutilizar otros aspectos, lo que genera un grafo de dependencias que se debe resolver antes de la transformación de modelos. Las dependencias son definidas en la sección de instanciación de un aspecto. La resolución de dependencias se hizo gracias a la aplicación del algoritmo "Topological Sorting"; la implementación Java de dicho algoritmo se obtuvo de [29].

4.6.1. Transformación de modelos.

El Cuadro **??** en la página ??, muestra la correspondencia entre los modelos de TextRAM y TouchRAM.

Elemento de TextRAM	**Elemento de TouchRAM**	**Observaciones**
TAspect	Aspect	TAspect hereda de Aspect, pero además define una lista de TInstantiationHeader.
TStructuralView	StructuralView	TStructuralView hereda de StructuralView, pero además define una lista de TAssociations.
TClass	Class	TClass es un Classifier que define una lista de TClassMembers y los parámetros del "layout" de una clase.
TOperation	Operation	TOperation es un TClassMember que define "features" equivalentes a Operation.
TAttribute	Attribute	TAttribute es un TClassMember que define "features" equivalentes a Attribute.

Elemento de TextRAM	**Elemento de TouchRAM**	**Observaciones**
TParameter	Parameter	TParameter no tiene relación de herencia con Parameter, pero define "features" equivalentes a él.
TAssociation	Association	TAssociation define dos extremos de asociación (desde y hasta), además indica la multiplicidad y el tipo de asociación definida. TAssociation no tiene relación de herencia con Association.
TAssociationEnd	AssociationEnd	TAssociationEnd representa el extremo de una asociación e indica la cardinalidad del extremo. TAsscociationEnd no tiene relación de herencia con Association.
TInstantiationHeader, Instantiation	Instantiation	TInstantiationHeader es el encabezado de definición de las clases externas de una instanciación.
TClassifierMapping	ClassifierMapping	TClassifierMapping representa las clases y miembros a mapear entre el aspecto actual y un aspecto externo. TClassifierMapping es una especialización de ClassifierMapping.
TAttribute	AttributeMapping	Utilizado para el mapeo de atributos dentro de un TClassifierMapping.
TOperation	OperationMapping	Utilizado para el mapeo de operaciones dentro de un TClassifierMapping.
TParameter	ParameterMapping	Utilizado para el mapeo de parámetros dentro de un TClassifierMapping.
TClass.layoutX, TClass.layoutY	Layouts	layoutX y layoutY son propiedades de TClass.

Elemento de TextRAM	**Elemento de TouchRAM**	**Observaciones**
TAbstract-MessageView	AbstractMessage-View	RAM utiliza una lista de *AbstractMessageView,* en cambio TextRAM utiliza el *TAbstractMessageView* como root de todas las vistas de mensajes representadas por *TAbstractMessageView*
TMessageView	MessageView-Reference	Si *TMessageView* contiene la propiedad *affectedBy* es transformado a *MessageViewRefence,* en caso contrario es transformado a *MessageView*
TMessageView	Interaction	Para TextRAM la unidad mínima de interacción es un mensaje (TMessageView).
TAspectMessage-View	AspectMessage-View	TAspectMessageView no tiene relación de herencia con AspectMessageView. TAspectMessageView especifica un TOperation para la definición del "pointcut" y una lista de TInteraction para la definición del "advice".
TLifeline	Lifeline	TextRAM tiene un encabezado que define los *TLifelines* globales a la vista de mensajes. RAM utiliza distintas instancias de *Lifelines* definidas en forma local en cada una de las vistas de mensajes.
TReference, TLocalAttribute	Temporary-Property	Utilizados como variables locales de un TLifeline.
TTemporary-Property, TParameter, TLifeline	ValueSpecification	*ValueSpecification* es utilizado por *InteractionOperand* (expresión del *CombinedFragment*) y por *ParameterValueMapping.*
TInteraction	Interaction-Fragment	Un TInteraction es transformado a un InteractionFragment.
TInteraction	Message, Gate	Por defecto se crea un *Gate* de entrada por cada interacción.

Elemento de TextRAM	**Elemento de TouchRAM**	**Observaciones**
TInteraction-Message	Message, Message-OcurrenceSpecification	A partir de un *TInteractionMessage* se crean dos instancias de *MessageOcurrenceSpecification*, una para el mensaje emisor y otro para el mensaje receptor.
TCombined-Fragment	Combined-Fragment	TCombinedFragment es definido para soportar bloques de tipo: alt, opt y loop. TCombinedFragment es transformado a CombinedFragment.
TOcurrence	OriginalBehavior-Execution.	La instancia de TOcurrence es transformada a OriginalBehaviorExecution.
TReturnInteraction	Message, Message-OcurrenceSpecification	A partir de un *TReturnInteraction* se crea un *Message* con dos *MessageOcurrenceSpecification*
TLocalAttribute	Attribute, Temporary-Property	TLocalAttribute es una referencia a un *Attribute* dentro de la vista de mensajes.

Cuadro 4.10: M2M entre TextRAM y RAM

La transformación del modelo TextRAM a RAM necesito de las siguientes consideraciones:

1. Los tipos de datos de RAM deben estar instanciados antes de la definición de cualquier elemento. El cambio de un elemento del modelo de TextRAM gatilla la generación de código y es en este evento en donde se registran todos los tipos de datos (incluyendo los tipos de datos simples y las referencias).

2. Fue necesario el uso de la clase padre TClassMember para brindar flexibilidad en el orden de la definición de atributos y operaciones dentro de una clase.

3. La vista estructural de TextRAM, posee una cierta equivalencia con la vista estructural de RAM, pero para la vista de mensajes se tuvo que realizar una definición sin relación de jerarquía entre los elementos de TextRAM y RAM."

La transformación de un modelo de RAM a una representación textual de TextRAM fue gracias al proceso de serialización de modelos. La serialización de modelos es un proceso *Standalon*e que necesita la creación de un *ResourceSet*. El proyecto *commons* de RAM contiene una utilidad llamada *ResourceManager* que permite obtener un *ResourceSet* con todos lo necesario para el funcionamiento de RAM. El *ResourceSet* de RAM permite cargar modelos existentes, guardar nuevos modelos y también inicializa el lenguaje de restricciones de objectos (OCL) que validan la integridad semántica de los modelos de RAM. El proceso de serialización, también requiere la inclusión del framework de inyección de dependencias y una referencia al fragmento

de serialización utilizado por el proyecto y definida en el MWE2. En TextRAM, la clase que gestiona la serialización se llama *RamToTextRAM*. Para hacer efectiva la serialización fue necesaria la configuración de:

1. "Scoping" para la resolución de todas las referencias cruzadas.
2. "Formatting" para la correcta visualización de la representación textual del modelo.
3. Servicios que indican que ciertos elementos del modelo no son serializables. La clase que indica los elementos transitivos se llama *TextRAMTransientValueService* y es una especialización de *DefaultTransientValueService* e implementa la interfaz *IDefaultTransientValueService*.

La ejecución del proceso de serialización se realiza en el IDE de Eclipse por medio de la opción de menú TextRAM -> Load RAM model...; la habilitación de una opción de TextRAM dentro del IDE fue posible gracias a la configuración de los siguientes componentes dentro del proyecto GUI:

1. Adición de un comando de menú dentro del archivo plugin.xml del proyecto GUI.
2. Creación de un "handler" que captura el evento de lectura del modelo RAM:
   - Creación de la clase *GenerationHandler* e implementación de *IHandler*.
   - Obtención de instancias para de *ResourceSet* y *IResourceDescription* de TextRAM gracias al framework de inyección de dependencias.
   - El método *execute* instancia un cuadro de dialogo para la lectura de archivos, lee el modelo RAM y lo serializa con el apoyo de la API construída en el proyecto DSL de TextRAM.
   - Se obtiene una instancia del *Workbench* actual y se visualiza la representación textual del modelo TextRAM.

4.7. Resumen

El presente capítulo describió la implementación de los diversos mecanismos para la implementación del DSL de TextRAM. La configuración del lenguaje requiere el nombre de la gramática y la definición declarativa de los diversas herramientas de apoyo del DSL. El metamodelo de RAM es referenciado dentro de la configuración del lenguaje. El "scoping" se preocupa de la visibilidad de los objetos; el "scoping" también es importante en los procesos de transformación de modelos, debido a la resolución de las referencias cruzadas; por tanto fue necesario detallar el "scoping" de cada uno de los elementos de TextRAM. Las validaciones son el mecanismo, por el cual se aplican las validaciones semánticas del modelo. La transformación de modelos no se efectúa si el modelo no es válido. XText ofrece generadores que exponen el metamodelo del lenguaje para realizar transformaciones a otras representaciones. La transformación de un modelo de TextRAM a un modelo de RAM (y viceversa) requirió de un mapeo entre elementos de ambos metamodelos. Hemos finalizado la descripción de TextRAM, y en el capítulo siguiente, se valida la implementación con un caso de estudio llamado "Slot Machines."

Capítulo 5

Validación con SlotMachines

El objetivo principal del presente capítulo, es demostrar el funcionamiento de TextRAM por medio de una caso de estudio llamado "Slot Machines". La sección 5.2 resume la funcionalidad general de un "Slot Machine". La sección 5.3 identifica las principales preocupaciones del presente caso de estudio. Las interacciones entre las distintas preocupaciones son explicadas en la sección 5.4. Finalmente, se hace una demostración de TextRAM por medio del aspecto *Game* (una de las preocupaciones principales del presente caso de estudio); la implementación de *Game* se puede ver en la sección 5.6.

5.1. Descripción del caso de estudio "Slot Machines"

"Slot Machines" (SM) es un caso de estudio no trivial, extraído de la industria, con requisitos dispersos en muchas fuentes y con restricciones impuestas por distintos campos de estudio (legales, técnicos, administración y negocios). Dicho trabajo fue realizado por Fabry, Zambrano y Gordillo en [10, 35, 36]. Se ha seleccionado el presente caso de estudio para demostrar el modelamiento de TextRAM. Una SM es una máquina tragamonedas que funciona al introducirle créditos (monedas, billetes o "tickets"); el jugador apuesta dichos créditos y si gana, la SM entrega un premio en forma de créditos canjeables en monedas, "tickets" o transferencias electrónicas; la SM por lo general, tiene cinco carretes que giran cuando el botón "play" es presionado. La máquina tiene premios preconfigurados que son pagados de acuerdo a los símbolos que se muestran en pantalla después de la última revolución de los carretes. Los símbolos son visualizados al azar. Una vez que la sesión ha finalizado los créditos remanentes pueden ser reclamados.

El "core" del SM es un diseño rudimentario basado en "loop", un patrón de juegos estándar (ver figura 5.1.1). El primer paso de dicho patrón, es la lectura de datos que son originados por el usuario, "drivers" de hardware o eventos programados. La información de entrada es procesada cambiando el estado interno del juego. Finalmente, la pantalla es actualizada con la información más reciente del juego. Las acciones originadas por el jugador son: jugar, seleccionar una línea de apuestas, insertar créditos, entre otras cosas. Un ejemplo de "loop" es el siguente: cuando el jugador presiona el botón "play", el resultado final es presentado por el SM en pantalla.

5.2. Funcionalidades principales del SM

Toda actividad del SM es registrada por un conjunto de contadores llamados "meters". Para acceder a dichas métricas, se utiliza una interfaz de gestión que permite leer y verificar el valor de los contadores; dicho panel es utilizado por los fiscalizadores de las entidades reguladoras para auditar el correcto funcionamiento

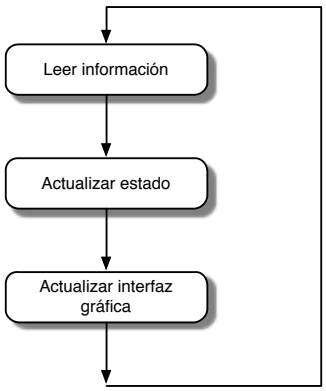


Figura 5.1.1: Patrón de juego "loop"

de la máquina. Existen muchos tipos de métricas; siendo las más relevantes:

- *Cantidad de créditos*: créditos disponibles para el jugador.
- *Créditos insertados*: cantidad total de créditos insertados y almacenados dentro de la SM.
- *Créditos entregados:* cantidad total de créditos entregados, por juego ganado.
- *Juegos ejecutados*: cantidad de juegos ejecutados desde la última vez que las métricas fueron reiniciadas.
- *Juegos ganados*: número total de juegos ganados.
- *Juegos pérdidos*: número total de juegos pérdidos.
- *Cantidad de monedas:* cantidad de monedas ingresadas al sistema.
- *Monedas inválidas:* monedas inválidas registradas por el sistema.
- "*Total Jackpot*": total de créditos pagados por la operadora.
- "*Jackpot*": último conjunto de créditos pagados por la operadora.
- "*Créditos cancelados*": monto total de créditos pagados por el SM.
- "*Cumulative progressive*": representa la suma de todos los premios pagados en forma progresiva.
- "*Billetes ingresados*", cantidad de billetes ingresados y aceptados por el validador.
- "*Billetes por denominación*": contadores individuales para cada tipo de denominación de la moneda.

La información "core" de la SM son las métricas y la configuración. En caso de una falla eléctrica, las métricas deben recuperarse a un estado válido y consistente. En caso de falla, también es importante recuperar el estado de un juego. Este comportamiento de recuperación es llamado "*reaunudación de programa*" y es requerido por las entidades legales que regulan el juego. Las regulaciones también obligan el uso de un "*Game Recall*", que es una especie de recuperación del estado de los últimos N juegos. Este comportamiento es requerido para resolver disputas entre los jugadores.

Los sistemas de monitoreo se aseguran de la correcta contabilización de las ganancias de la SM (necesarias para la distribución de ganancias a los dueños de la máquina, dueños del casino e inversionistas).

Protocolos de comunicación son establecidos para entregar la información de monitoreo a los involucrados. Los protocolos de comunicación pueden proveer muchos tipos de mensajes. Su agrupación es la siguiente:

- *Consulta de los mensajes de métricas*: consultan el estado de las métricas almacenadas en la SM. Estás métricas son utilizadas en los procesos de contabilidad.
- *Configuración general del SM:* el SM puede ser configurado remotamente, por medio de los protocolos de comunicación. Por ejemplo, en máquinas multijuego es posible seleccionar el juego que será desplegado por el SM.
- *Mensajes de boletas recibidas y entregadas:* En algunos casinos en lugar de dinero se utilizan "tickets" como créditos para la máquina. Físicamente, el "ticket" solo contiene un identificador y no la información del valor monetario. Cuando el mismo es insertado al SM, los créditos correspondientes son transferidos a la máquina. Toda la información de los "tickets" válidos es entregada a los servidores centralizados.
- *Reportes en tiempo real*: durante el juego, se ejecutan diversos eventos (ejemplo: inicio y fin de juego); dichos eventos son reportados al sistema de monitoreo.

Los mecanismos de regulación exigen la existencia de distintos modos: *modo certificación* y *modo demo.* Estos modos deben estar presentes en cada juego antes de ser ingresado al mercado. Los laboratorios de certificación son instituciones que aseguran el cumplimiento de las regulaciones. La certificación es un proceso meticuloso en donde muchos de los aspectos del juego son comparados con el comportamiento esperado. Desde el punto de vista del software, los elementos analizados durante el proceso de certificación son:

- Generación aleatoria de números: aseguran ciertas propiedades en la distribución probabilística de resultados.
- Pago: el juego debe comportarse consitentemente en la relación al pago reportado por cada configuración.
- Tabla de pagos: define el premio a partir de la combinación de símbolos generados por la máquina.

5.3. Identificación de preocupaciones

Con el objetivo de entregar un modelamiento orientado a aspectos a los programadores del software de la SM, se han identificado las siguientes preocupaciones:

- *"Game"* (juego): es la lógica base del juego de azar. Los usuarios pueden ingresar créditos dentro de la máquina y después jugar. La salida es determinada en forma aleatoria y cuando el jugador gana, es recompensado con un conjunto de créditos.
- *"Meters"* (métricas): son un conjunto de contadores que se utilizan para auditar la actividad del juego. Existen métricas que cuentan el número de jugadas, el monto total de la apuesta, lo total ganado, la ocurrencias de los errores, entre otros.
- *"Program interrumption and resumption"* (interrupción y reanudación del programa): son mecanismos de persistencia y recuperación de requerimientos. Estos requerimientos indican la forma que la máquina se debe comportar en caso de una falla de energía. El sistema debe recuperar su último estado funcional después de una falla eléctrica.
- *"Game Recall"* (vuelta atrás de los juegos): permite al SM recordar y reproducir las últimas jugadas, con el objetivo de resolver cualquier disputa de un jugador.
- *"Error Conditions"* (condiciones de error): bajo ciertas circunstancias, el juego debe detectar condiciones de error y comportarse en forma correcta.

- *"Communications"* (comunicaciones): el SM es conectado a un sistema de monitoreo dentro del casino. Está precupación define los tipos de datos y el formato de la información a intercambiar entre el SM y el sistema de monitoreo.

- *"Demo"* (demostración): permite al SM cumplir con las regulaciones exigidas por las entidades reguladoras. Recrea todos los escenarios posibles de la máquina.

5.4. Interacciones

Las interacciones entre preocupaciones son complejas. La comprensión de las interacciones es un proceso vital para los diseñadores y programadores. Para clasificar las interacciones se ha hecho uso de la siguiente taxonomía: conflictos, dependencia, reforzamiento y "mutex". dentro del dominio del problema, se pueden identificar las siguientes taxonomías::

- *Conflictos.* Entre la preocupación *Demo* y *Meters* existe un conflicto, ya que *Meters* solo trabaja en forma correcta sin la presencia de *Demo,* pero si *Demo* está activo, su actividad debe ser registrada por *Meters.*

- *"Mutex"*: prohibido tener dos protocolos proporcionando la misma funcionalidad al mismo tiempo.

- "*Dependencia*": existe dependencia en la relación entre *Communications* y *Meters*; el protocolo necesita comunicar el estado del SM, que a su vez es representada por *Meters.*

- *"Reforzamiento"*: la detección de una *condición de error,* permite a los *protocolos de comunicación* brindar funcionalidad "extra" en este caso el error es comunicado en tiempo real al sistema de reportes del SM.

Las interacciones, que poseen relaciones transversales son:

- *Demo* a *Game:* para cumplir con las regulaciones legales los requisitos de *Demo* afectan muchas de las definiciones de *Game.*

- *Game Recall* a *Game*: los requisitos de *Game Recall* afectan el comportamiento de *Game*, ya que se utiliza un registro de actividades para cada juego y otros eventos relacionados a resolver las disputas entre jugadores.

- *Meters* a *Game:* las métricas cuentan las actividades y muchas funciones definidas en *Game*, por ejemplo: jugar, pagar, insertar monedas, etc

- *Program resumption* a *Game, Game Recall, G2S* y *Meters*: *Program resumption* es persistencia, atraviesa todos los lugares en donde existen datos importantes, que necesitan ser restaurados y cambiados.

- *G2S* a *Game:* está preocupación atraviesa muchos de los requisitos de *Game*, ya que ciertos eventos de *Game* deben ser reportados, monitoreados y comunicados al sistema de reportería.

- *SCP a Game*, se refiere al otro protocolo utilizado para monitorear el comportamiento de *Game.*

- *Error conditions* a *Game:* el comportamiento asociado con las condiciones de error necesitan ser incluidas al comportamiento de *Game.* Son varias las condiciones que puede generar un error: desde un atasco de papel hasta una puerta abierta.

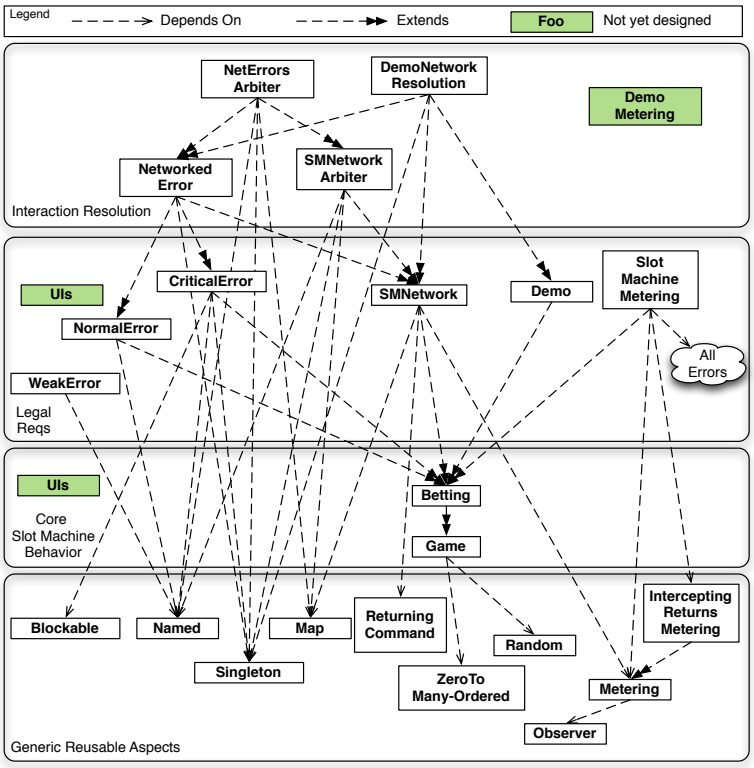


Figura 5.5.1: Jerarquía de aspecto "Slot Machines"

5.5. Aspectos modelados en TouchRAM

El caso de estudio "Slot Machines" ya fue implementado en TouchRAM; el resultado de dicha implementación, fue un subconjunto de las preocupaciones identificadas en la sección 5.3. El subconjunto de aspectos es el siguiente: *Betting, Blockable, CriticalError, Demo, DemoNetworkResolution, Game, InterceptReturnMetering, Map, Metering, Named, NetErrorsArbiter, NetworkedError, NormalError, Observer, Random, ReturningCommand, SMNetwork, SMNetworkArbiter, Singleton, SlotMachineMetering, WeakError, ZeroToManyOrdered.* La jerarquía del subconjunto de aspectos de "Slot Machines" se puede ver en la Figura 5.5.1.

Dichos aspectos fueron modelados en TextRAM, validando así el funcionamiento de la herramienta. En [27], hay un hipervínculo en donde se pueden descargar los aspectos modelados en TextRAM junto con los modelos RAM generados automáticamente por el software.

5.5.1. Problemas conocidos

A continuación, se presenta una lista de los problemas conocidos en las herramienta TextRAM. Dichos detalles no son inhabilitantes para la definición de un aspecto, pero sí, deben de tomarse en cuenta a la hora de modelar. Los problemas conocidos son:

1. La definición de una instanciación incluye el mapeo de miembros de una clase actual con miembros de una clase externa. La sintaxis para dicho mapeo se puede observar en el anexo 6. La sintaxis del lenguaje exige un espacio entre el nombre de la clase y la apertura de llaves que indica el inicio de los miembros de la clase a mapear. Un ejemplo inválido sería `|Data<add, get>` (se observa, que no hay espacio entre el nombre de la clase y la sección que mapea los miembros).

2. En ocasiones, cuando se está editando la vista de mensajes, aparece un error en la vista estructural (ver

```
1 aspect Observer {
2   structure {
3     class |Subject {
4       + * |modify()
5     }
6   }
7 }
```

Cuadro 5.1: Patrón para especificar la aceptación de cuaquier tipo de dato como valor de retorno en un método parcial

figura 5.7.8). Para solucionarlo, primero hay que asegurarse de tener una sintaxis válida en la vista de mensajes; luego, se debe modificar cualquier aspecto de la vista estructural (por ejemplo la inclusión de un espacio en blanco después del cierre de llaves es suficiente); finalmente, se debe guardar el modelo para borrar el error.

3. La vista de clases permite el concepto de herencia. Si el aspecto A extiende el aspecto B, A puede visualizar todas las clases de B; por tanto las clases de A pueden heredar las clases de B. Actualmente TextRAM no soporta la visibilidad de clases que están dentro de los aspectos extendidos.

4. La sección de instanciación permite mapear elementos del aspecto actual con elementos de aspectos externos. Cuando se realiza un mapeo entre métodos, los parámetros del método del aspecto actual deben coincidir en número y tipo con los parámetros del método del aspecto externo. Dicha validación está pendiente.

5. TextRAM carece del concepto de paquetes, por tanto existe colisión de nombres entre los elementos del aspecto actual y los elementos de los aspectos externos; en caso de colisión de nombres, TextRAM no genera los modelos RAM. Esto se podría resolver implementando el concepto de paquetes, mediante el soporte para paquetes proveído por XText.

6. Para que la transformación de los modelos RAM a TextRAM funcione, es necesario desactivar el "scope" de los aspectos externos y las instanciaciones de los aspectos externos. La figura 5.7.7 indica el fragmento de código a descomentar del archivo TextRAMScopeProvider.xtend. La forma de resolver este problema es investigando el funcionamiento del índice de XText durante su utilización como una aplicación "Standalone".

7. RAM permite la definición de métodos parciales con la especificación de un patrón que indica la aceptación de cualquier tipo de dato como valor de retorno. Dicho patrón se representa por el carácter (*). Actualmente TextRAM no soporta esta característica. En el cuadro 5.1, se observa que el método parcial *modify* acepta cualquier tipo de dato de retorno.

5.6. Definición del aspecto "Game" en TextRAM

A continuación se presenta un ejemplo del proceso de validación. Se modelará el aspecto *Game* haciendo uso de pequeños pasos que ilustrarán las distintas funcionalidades de TextRAM. El objetivo es ejemplificar las siguientes funcionalidades:

1. *Transformación automática de un modelo de RAM a un modelo de TextRAM*: el aspecto *ZeroToMany-Ordered* será modelado en TouchRAM (ver sección 2.6) y luego será exportado a su TCS en TextRAM.

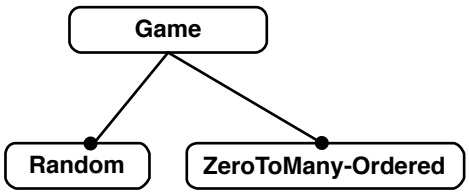


Figura 5.6.1: Jerarquía del aspecto Game

2. *Transformación automática de un modelo TextRAM a un modelo de RAM:* para demostrar este punto, se hará uso del aspecto *Game*; se definirá la vista estructural, las instanciaciones y la vista de mensajes. Para cada sección del modelo, se hará una conversión a RAM y se ilustrará el resultado en TouchRAM.

3. *Problemas conocidos*: se enunciarán los problemas conocidos y se darán las indicaciones para poder actuar sobre ellos. Se recomienda la lectura de la sección 5.5.1, antes de proceder a crear un modelo de TextRAM por primera vez.

El aspecto *Game* fue definido en la sección5.3, y su modelo fue presentado en la figura 2.5.3. *Game* depende de los aspectos *ZeroToMany-Ordered* y *Random* la jerarquía de dependencias se puede ver en la figura 5.6.1. Por tanto, para definir *Game*, es necesario iniciar por el modelamiento de *ZeroToMany-Ordered* y *Random*.

5.6.1. Transformación automática de un modelo de RAM a un modelo de TextRAM

La mejor forma de comprender la TCS de TextRAM es transformando modelos existentes de RAM a TextRAM. Para ejemplificar dicha transformación, se utilizará el aspecto *ZeroToMany-Ordered;* el objetivo de dicho aspecto es reproducir la interfaz de la clase Java, *java.util.Set* por medio de tres clases `|Data`, `Sequence` y `|Associated`. La clase `Sequence` enlaza de forma ordenada una instancia de la clase `|Data` con muchas instancias de la clase `|Associated`. El aspecto *ZeroToMany-Ordered* se puede contemplar en la Figura 5.6.2 a través de la herramienta TouchRAM.

El anexo 6, muestra la instancia serializada de *ZeroToMany-Ordered*; el resultado de la serialización es un archivo *ecore* conforme al metamodelo de RAM. Los pasos para transformar un modelo existente de RAM a uno de TextRAM, se enuncian a continuación:

1. Seguir los pasos enunciados en el anexo 6, para crear la creación de un nuevo proyecto TextRAM.

2. Ir a TextRAM > Load RAM model...

3. El IDE de Eclipse visualizará la representación textual del modelo en pantalla (lo muestra como un archivo de texto plano).

4. Ir a File > Save As... y guardar el aspecto en el proyecto correspondiente.

5. Cerrar y abrir de nuevo el archivo (esto es necesario para la detección de la opción "XText nature").

El modelo TextRAM generado se puede ver en la figura 5.6.3. En dicha figura se observa como trabaja el "highlighting" de sintaxis del lenguaje y el funcionamiento del formateador de texto definido en la sección 4.5.

A continuación se resume el proceso de implementación para la transformación de un modelo TouchRAM a un modelo TextRAM:

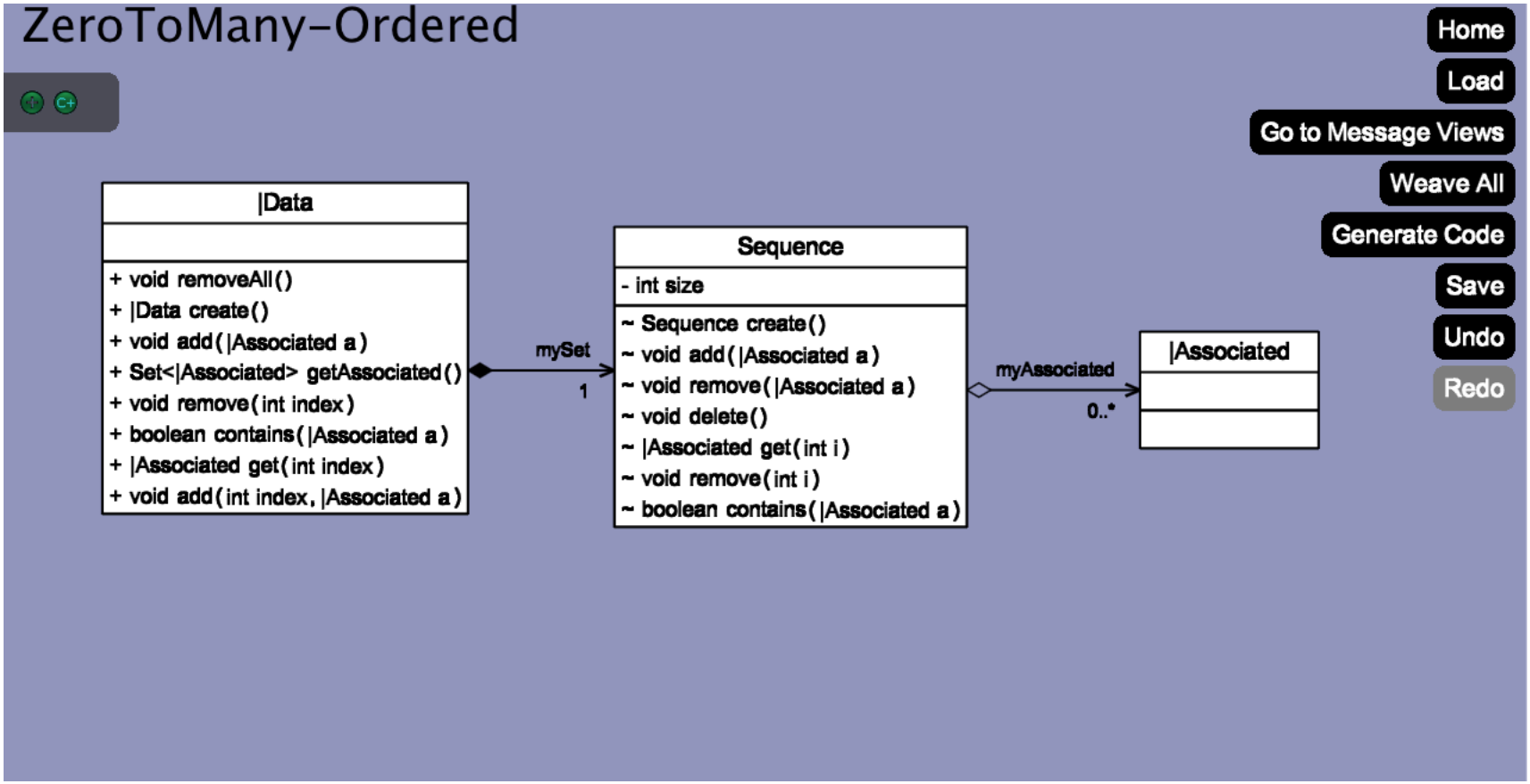


Figura 5.6.2: Aspecto *ZeroToMany-Ordered* representado en TouchRAM

1. El *workbench* crea una instancia de la clase *GenerationHandler* que captura el evento clic de la opción de menú "Load RAM model..."
2. Se presenta un cuadro de diálogo que sirve para seleccionar el archivo.
3. A través del archivo seleccionado se crea una instancia RAM. La instancia de RAM es cargada gracias a la clase utilitaria *ResourceManager* definida dentro de uno de los paquetes de TouchRAM llamado *ca.mcgill.sel.commons*. Dicha clase inicia el *ResourceSet* asociado y prepara el contexto del OCL.
4. El modelo RAM obtenido en el paso 3, es transformado a un modelo de TextRAM.
5. El modelo TextRAM obtenido en el paso 4, es serializado y su contenido es visualizado en una página del Eclipse "workbench".
6. La herramienta aplica las reglas de validación definidas en la sección 4.4.
7. El modelo RAM correspondiente a la TCS de TextRAM es generado automáticamente en la carpeta *src-gen/aspects*.

5.6.2. Transformación automática de un modelo TextRAM a un modelo de RAM

Como se menciona en la sección 5.3, el aspecto *Game* encapsula la lógica base del juego de azar; su responsabilidad es hacer girar los carretes en forma aleatoria. Por medio de *Game* se demostrará el modelamiento en TextRAM. Para el presente ejercicio se necesitan las siguientes dependencias: *ZeroToMany-Ordered* (explicado en la sección 5.6.1) y *Random* (ver figura 5.7.1). Los pasos para modelar *Game* son:

1. Reutilizar el proyecto definido en la sección 5.6.1 (dicho proyecto ya contiene el aspecto *ZeroToMany-Ordered).*
2. Definir el aspecto *Random* (ver figura 5.7.1). Las instrucciones para crear un nuevo aspecto, son:
   a) Ir a File > New > File
   b) Se presentará una ventana de diálogo con una lista de proyectos. Seleccione el proyecto que contendrá el nuevo aspecto.
   c) En la caja de texto *File name,* indique el nombre del archivo (se sugiere que sea igual al nombre del aspecto) e indique la extensión .xram (*importante para el reconocmiento de "XText nature".*
   d) Acepte la opción "XText nature".
   e) Se presentará el área de trabajo correspondiente para la edición del nuevo aspecto.
3. Defina la vista estructural del aspecto *Game*, ver figura 5.7.2.
4. Modele la sección de instanciaciones del aspecto *Game*, ver figura 5.7.3.
5. Defina la vista de mensajes, ver figura 5.7.4.
6. Los modelos generados serán almacenados en la carpeta *src-gen/aspects*
7. Cree una nueva instancia de TouchRAM.
8. Seleccione Load... y seleccione el modelo *Game* generado en la carpeta *src-gen/aspects* del paso 6.
9. El aspecto *Game* modelado previamente en TextRAM, estará disponible en TouchRAM. La vista de mensajes generada se puede observar en la figura 5.7.6.

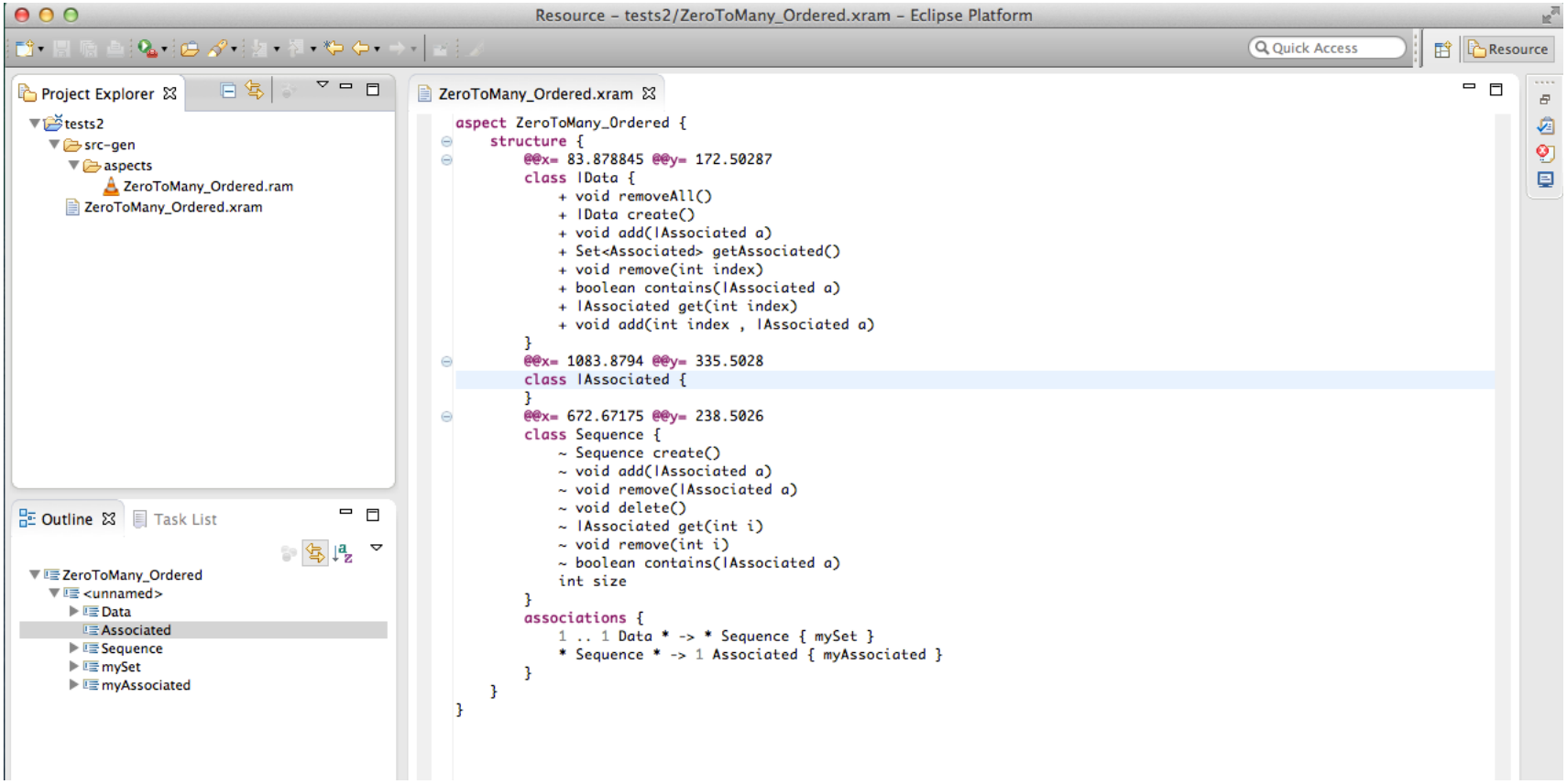


Figura 5.6.3: Aspecto *ZeroToMany-Ordered* representado en TextRAM

Figura 5.7.1: Aspecto *Random* representado en TextRAM

5.7. Resumen

"Slot Machines" es un caso de estudio no trivial, que describe el funcionamiento de las máquinas tragamonedas utilizadas en los casinos. Dicho caso de estudio fue elegido para demostrar la funcionalidad de TextRAM. Todos los modelos de Slot Machines modelados en TouchRAM fueron modelados en TextRAM. El aspecto *Game* fue elegido para demostrar el funcionamiento de TextRAM, por medio de una serie de iteraciones que involucraron la definición de modelos en TextRAM y en TouchRAM. El capítulo siguiente detalla el trabajo futuro, en relación a TextRAM.

Game.xram

```
aspect Game dependsOn ZeroToMany_Ordered, Random {

    structure {

        class ArrayOfInt {}
        class Symbol {}

        class Reel {
            int position
            + int getCurrentPosition()
            + Reel create()
            + Symbol getCurrentSymbol()
            + void setCurrentPosition(int newPosition)
            + void addSymbol(Symbol s)
            + void addSymbolAtPosition(int index, int newPosition)
            - Symbol getSymbol(int position)
        }

        class ReelConfiguration {
            - Symbol getSymbol(int reel, int position)
            - ReelConfiguration create(String filename)
        }

        class Machine {
            + void initialize()
            + void play()
            + void addReelAtPosition(int index, Reel a)
            + Reel getReelAtPosition(int index)
        }

        class RandomReels  {
            + void create()
            + ArrayOfInt spinReels()
            - int spinReel()
        }

        associations {
            Machine -> 1 ReelConfiguration { myReelConfiguration }
            Machine -> 1 RandomReels       { myRandomReels }
        }
    }
```

Figura 5.7.2: TCS de la vista estructural del aspecto *Game* (aspecto definido en TextRAM)

Game.xram

```
aspect Game dependsOn ZeroToMany_Ordered, Random {

    structure {...}

    instantiations {
        ZeroToMany_Ordered {
            !Data <add,get> -> Machine <addReelAtPosition,getReelAtPosition>;
            !Associated -> Reel
            !Data <add,get> -> Reel <addSymbol,getSymbol>;
            !Associated -> Symbol
        }

        Random {
            RandomGenerator <nextRandom> -> RandomReels <spinReel>
        }
    }

    messages {
        lifelines {...}
    }

}
```

Figura 5.7.3: TCS de las instanciaciones del aspecto *Game* (aspecto definido en TextRAM)

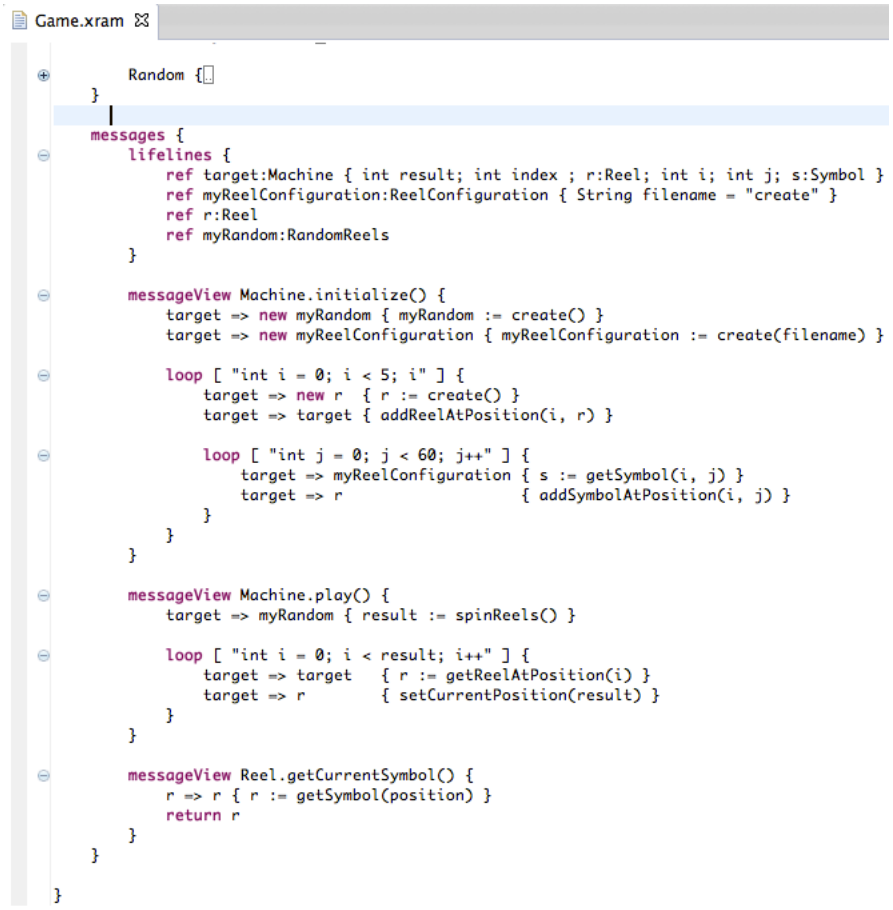


Figura 5.7.4: TCS de la vista de mensajes del aspecto *Game* (aspecto definido en TextRAM)

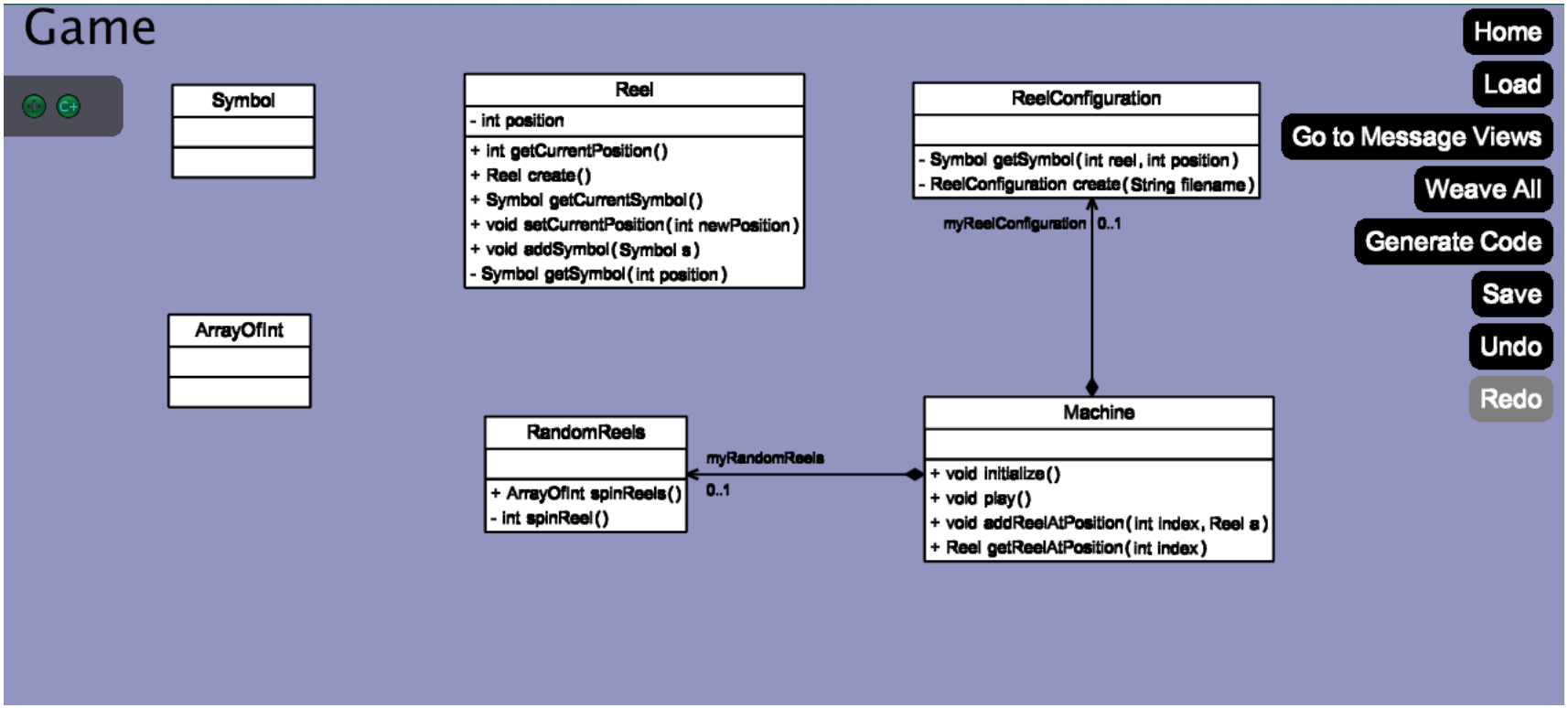


Figura 5.7.5: Vista estructural del aspecto *Game* (aspecto definido en TextRAM)

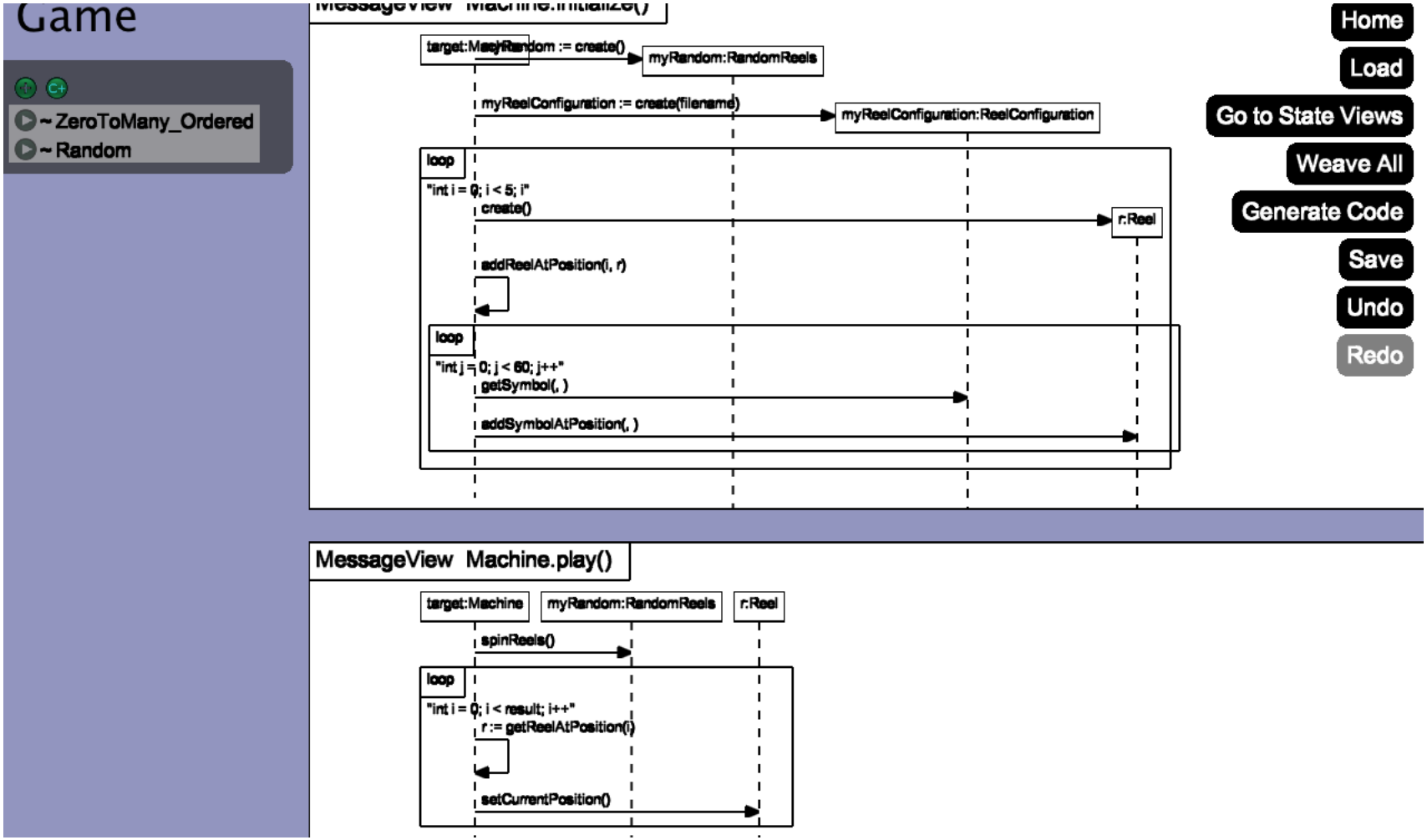


Figura 5.7.6: Vista de mensajes del aspecto *Game* (aspecto definido en TextRAM)

```
class TextRAMScopeProvider extends AbstractDeclarativeScopeProvider {

    extension ModelScopeProvider scopeProvider = new ModelScopeProvider

//  def IScope scope_Instantiation_externalAspect(TAspect aspect, EReference reference) {
//      Scopes::scopeFor( aspect.getExternalAspectsFromHeader )
//  }

//  def IScope scope_TInstantiationHeader_externalAspects( TAspect aspect, EReference reference ) {
//      val externalAspects = TextRamModelUtil::collectExtendedAspects( aspect )
//
//      Scopes::scopeFor( externalAspects )
//  }

    def IScope scope_TOperation_returnType(TStructuralView structuralView, EReference reference) {
        Scopes::scopeFor( structuralView.getTypesFor )
    }

    def IScope scope_TAttribute_type(TStructuralView structuralView, EReference reference) {
        Scopes::scopeFor( structuralView.getPrimitiveTypes )
    }
```

Figura 5.7.7: Generación de modelos RAM desde TextRAM

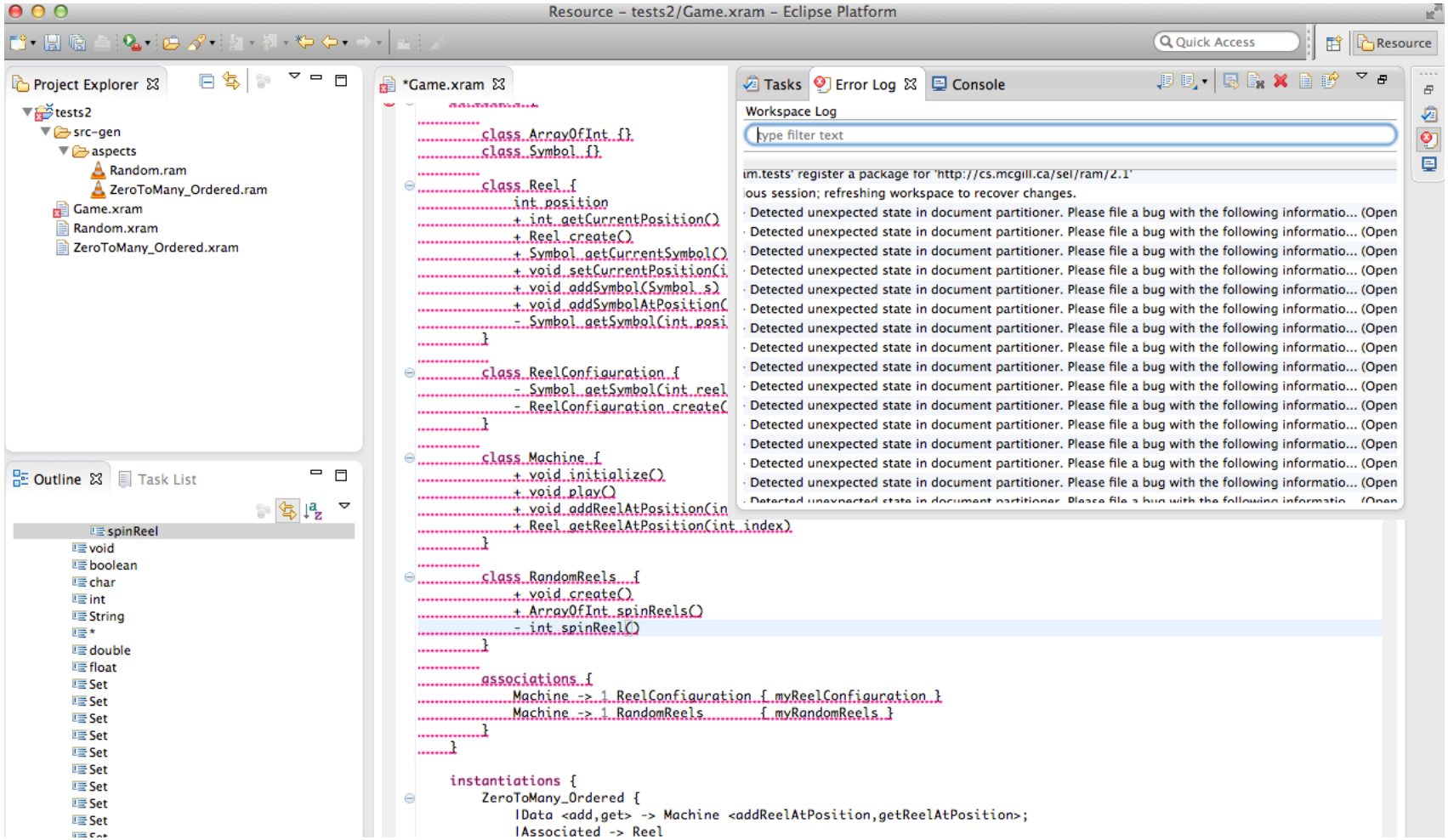


Figura 5.7.8: Error conocido en la vista de estructural

Capítulo 6

Conclusiones y Trabajo Futuro

El objetivo de la presente tesis fue la construcción de TextRAM, un DSL textual para la construcción de modelos conformes a RAM. La motivación principal para la implementación de TextRAM fue presentar una alternativa a la única herramienta que permite diseñar modelos en RAM, el DSL gráfico TouchRAM. El desarrollo de TextRAM, exigió el conocimiento de los siguientes conceptos:

1. Lenguajes de especificación de dominio (DSL): a partir de notaciones y abstracciones, ofrecen un poder de expresividad enfocado y restringido a un problema de dominio en particular.

2. Herramientas para implementación de DSLs: los "language workbenches" son aplicaciones orientadas al desarrollo de DSLs; se hizo una investigación de los diversos "language workbenches" disponibles. El resultado de la investigación dejó dos "language workbenches" adecuados para la construcción de TextRAM: Rascal y XText; este último fue elegido debido a su corta curva de aprendizaje y su íntima relación con EMF (marco de trabajo de RAM).

3. La ingeniería de software orientado a modelos (MDSE): es un marco de trabajo conceptual unificado en donde todo el ciclo del software es visto como un proceso de producción, refinamiento e integración de modelos. Los DSL son un ingrediente de la MDSE.

4. Herramientas de soporte al MDSE: el "Eclipse Modeling Framework" (EMF) es la principal alternativa de Eclipse para la implementación del MDSE. El metamodelo de RAM fue definido en EMF.

5. Programación orientada a aspectos (AOP): es una técnica para modularizar preocupaciones transversales. El estudio de AOP fue necesario debido a que RAM es una herramienta, cuyo objetivo es modelar por medio de preocupaciones transversales.

6. "Reusable Aspect Models" (RAM): enfoque de modelamiento orientado a aspectos que permite crear modelos en forma escalable y consistente entre múltiples vistas.

7. XText y XTend: XText es un generador de "parsers" que deriva la TCS y el AST a partir de la gramática. Xtend es un lenguaje de programación estático, tipificado que se traduce a un código fuente comprensible de Java; Xtend fue utilizado para la manipulación del AST, la validación semántica y la transformación de modelos.

Después del estudio de los conceptos anteriores, se procedió a la implementación de TextRAM. El primer paso de la implementación, fue la definición del metamodelo; para la definición del metamodelo fue necesaria la conceptualización de la TCS. El TCS se construyó en base a dos objetivos: (1) conformidad al metamodelo

de RAM (excluyendo la vista de estados), (2) la TCS debe ser flexible, económico y expresivo (dichas características delinearon la definición de la gramática). Los elementos de la gramática derivan el metamodelo de TextRAM. Para implementar el lenguaje, fue necesario el desarrollo de los siguientes elementos:

1. Configuración del lenguaje: describe las propiedades de TextRAM (indica la reutilización del metamodelo de RAM).

2. El "scoping": se preocupa de la visibilidad de los objetos; el "scoping" también es importante en los procesos de transformación de modelos, debido a la resolución de las referencias cruzadas.

3. Validaciones: mecanismo por el cual se aplican las validaciones semánticas del modelo. La transformación de modelos no se efectúa si el modelo no es válido.

4. Generadores: necesarios para la transformación de modelos.

Para validar la implementación de RAM se hizó uso del caso de estudio "Slot Machines" (describe el funcionamiento de las máquinas tragamonedas utilizadas en los casinos). Los 19 aspectos definidos en este caso fueran codificados en TextRAM y exportados al formato de TouchRAM. El aspecto *Game* fue elegido para demostrar el funcionamiento de TextRAM, por medio de una serie de iteraciones que involucraron la definición de modelos en TextRAM y en TouchRAM.

Los lineamientos para la construcción de TextRAM se presentarón en la sección 1.3; de dichos requerimientos, no se pudó desarrollar la exportación gráfica de TextRAM por medio de Graphviz. Actualmente TouchRAM es la única alternativa para visualizar en forma gráfica, los modelos desarrollados en TextRAM. Además, hay una lista de problemas conocidos enunciados en la sección 5.5.1, que deben ser resueltos para que TextRAM sea considerada como una aplicación robusta. Sin embargo, dichos problemas no son inhabilitantes para el modelamiento efectivo de aspectos. Ahora mismo, en comparación con TextRAM, TouchRAM es una aplicación más madura porque tiene más tiempo de vida y su desarrollo estuvo a cargo de un equipo de trabajo que sigue realizando mejoras e implementando nuevas funcionalidades.

En mi experiencia, TouchRAM tiene una curva de aprendizaje más corta que TextRAM debido a que su sintaxis gráfica esta basada en UML, que es un lenguaje de modelado familiar a todo ingeniero de software. Sin embargo, una vez aprendida la sintaxis de TextRAM, la creación de modelos es más rápida que en TouchRAM, eso gracias a su naturaleza textual. Considero que la complejidad en el uso de las dos herramientas mencionadas anteriormente, radica en la compresión de la teoría de RAM y no tanto en la conveniencia de un DSL gráfico o textual. Debido a que TextRAM permite modelar aspectos completos en RAM y compartirlos en TouchRAM para su visualización gráfica, se concluye que se ha cumplido con el objetivo de la tesis.

La visualización gráfica de los modelos de RAM en TouchRAM no es estándar; por tanto, no es posible compartir dichos gráficos con otras tecnologías. Una alternativa para poder dar portabilidad a la representación gráfica de RAM es utilizar el estándar del lenguaje DOT [8]. El lenguaje DOT es un fichero de texto plano que describe gráficos; Graphviz [14] es una herramienta de visualización de código abierto que permite la definición de ficheros DOT. Para futuras versiones de TextRAM se puede adicionar un módulo de transformación de modelos RAM a DOT y así TextRAM soportará en forma nativa la visualización gráfica de modelos RAM, con la ventaja adicional de portar los gráficos a distintas tecnologías.

El proceso más costoso de la implementación de TextRAM fue la transformación de modelos. Para la transformación de modelos, se hizó uso de el lenguaje de programación XTend, debido a su versatibilidad para manipular el AST de un DSL. Pero XTend es un lenguaje de programación, no es un componente especializado para realizar tareas de transformación de modelos. En lugar de utilizar XTend, se puede estudiar la factibilidad de utilizar otras herramientas enfocadas en la transformación de modelos, como ser: (1) Atlas

Transformation Languge de Eclipse [3], (2) QVT "Operational Mappings Language" [21], especificado por la "Object Management Group" (OMG).

RAM modela aspectos por medio de 3 vistas (clases, estados y secuencia). TextRAM solo soporta la definición de la vista de clases y secuencias. La vista de estados no se implementó debido a que la misma estaba en proceso de definición. Futuras versiones de TextRAM deben soportar la vista de estados.

TextRAM permite la transformación de modelos de RAM a TextRAM y viceversa, pero en el proceso se pierden los comentarios de los modelos, debido a que el metamodelo de RAM no soporta comentarios. Una vez que dicho metamodelo incluya los comentarios, TextRAM debe soportar la transformación de comentarios.

TextRAM puede incluir el concepto de librerías reutilizables por medio de la implementación de modelos que son usados frecuentemente por otros modelos; la librería de aspectos reutilizables, puede incluir los siguientes modelos: (1) patrones de diseño: *Observer, Command, Singleton*, (2) utilitarios*: Copyable, KeyCounter, Map, Named, ZeroToMany, ZeroToMany-Ordered.*

Finalmente, la sección 5.5 se presentó una lista de los problemas conocidos, que deben de tomarse en cuenta en las futuras evoluciones de TextRAM.

Bibliografía

[1] Wisam Al Abed, Valentin Bonnet, Matthias Schöttle, Engin Yildirim, Omar Alam, and Jörg Kienzle. Touchram: A multitouch-enabled tool for aspect-oriented software design. In *Software Language Engineering*, pages 275–285. Springer, 2013.

[2] ANTLR. Antlr (another tool for language recognition) www.antlr.org.

[3] ATL. Atl - a model transformation technology https://eclipse.org/atl/l.

[4] Abir Ayed and Jörg Kienzle. Integrating protocol modelling into reusable aspect models. In *Proceedings of the 5th ACM SIGCHI Annual International Workshop on Behaviour Modelling - Foundations and Applications*, BMFA '13, pages 2:1–2:12, New York, NY, USA, 2013. ACM.

[5] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend.* Packt Publishing Ltd, 2013.

[6] Grady Booch, Robert A Maksimchuk, Michael W Engel, Bobbi J Young, Jim Conallen, and Kelli A Houston. *Object oriented analysis and design with applications*, volume 3. Addison-Wesley, 2008.

[7] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice.* Synthesis Lectures on Software Engineering. Morgan And Claypool Publishers, 2012.

[8] DOT. The dot language http://www.graphviz.org/doc/info/lang.html.

[9] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *Software Language Engineering*, pages 197–217. Springer, 2013.

[10] Johan Fabry, Arturo Zambrano, and Silvia Gordillo. Expressing aspectual interactions in design: Experiences in the slot machine domain. In *Model Driven Engineering Languages and Systems*, pages 93–107. Springer, 2011.

[11] Martin Fowler. *Domain Specific Languages.* Addison-Wesley Professional, 1st edition, 2010.

[12] R. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented approach to early design modelling. In *IEE Proceedings - Software*, pages 173–185, 2004.

[13] Debasish Ghosh. *DSLs in Action.* Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[14] Graphiz. Graphviz - graph visualization software.

[15] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. In *ECOOP97 Object-Oriented Programming.* Springer Berlin Heidelberg, 1997.

[16] Jorg Kienzle, Wisam Al Abed, Franck Fleurey, Jean marc Jezequel, and Jacques Klein. Aspect-oriented design with reusable aspect models, 2010.

[17] Jorg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 87–98, New York, NY, USA, 2009. ACM.

[18] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.

[19] LanguageWorkBench2013. Language workbench challenge - comparing tools of the trade www.languageworkbenches.net.

[20] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[21] QVT. Qvt operational component is a partial implementation of the operational mappings language defined by the omg standard specification (mof) 2.0 query/view/transformation. https://projects.eclipse.org/projects/modeling.mmt.qvt-omll.

[22] Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer, and Gerti Kappel. A survey on aspect-oriented modeling approaches. *Reporte TÃ©cnico, Vienna University of Technology*, 2007.

[23] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006.

[24] Matthias Schöttle. Aspect-oriented behavior modeling in practice. Master's thesis, M. Sc. Thesis, Department of Computer Science, Karlsruhe University of Applied Sciences (September 2012), http://mattsch.com/papers/masterthesis.pdf, 2012.

[25] Friedrich Steimann and Thomas Kühne. Coding for the code. *Queue*, 3(10):44–51, 2005.

[26] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[27] TextRAM. Modelos ram generados en textram https://github.com/mjorod/textram/tree/master/docs/sm.

[28] TextRAM. Textram's source code https://github.com/mjorod/textram.git.

[29] TopSortJava. Implementacion en java del algoritmo topological sorting http://www.keithschwarz.com/interesting/code/topological-sort/topologicalsort.java.html.

[30] Tijs van der Storm. The rascal language workbench. Technical report, CWI Technical Report SEN-1111, CWI, 2011.

[31] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[32] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[33] Xtend. Xtend is a statically-typed programming language which translates to comprehensible java source code http://www.eclipse.org/xtend/documentation.html.

[34] Xtext. Xtext is a framework for development of programming languages and domain specific languages. http://www.eclipse.org/xtext.

[35] Arturo Zambrano, Johan Fabry, Guillermo Jacobson, and Silvia Gordillo. Expressing aspectual interactions in requirements engineering: Experiences in the slot machine domain. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2161–2168, New York, NY, USA, 2010. ACM.

[36] Arturo Federico Zambrano Polo y La Borda. *Addressing aspect interactions in an industrial setting: experiences, problems and solutions.* PhD thesis, Facultad de Informática Universidad de la Plata, 2014.

Anexo A
Gramática de TextRAM

```
1  grammar cl.uchile.pleiad.TextRAM with org.eclipse.xtext.common.Terminals
2
3  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
4  import "http://cs.mcgill.ca/sel/ram/2.1"
5
6  generate textRam "http://cl.pleiad.textram/1.0" as textRam
7
8  Aspect :
9    TAspect
10 ;
11
12 TAspect returns textRam::TAspect:
13 'aspect' name=EString (headerInstantiations+=TInstantiationHeader (","?
       headerInstantiations+=TInstantiationHeader)*)?
14 '{'
15   structuralView=TStructuralView
16   ('instantiations' '{' instantiations+=Instantiation ( ";"? instantiations+=
         Instantiation)* '}' )?
17   ('messages' '{' messageViews+=TAbstractMessageView '}' )?
18 '}';
19
20 terminal TEMPLATE: (ID'<' ID '>')*;
21 terminal FLOAT returns ecore::EFloat: ('+'|'-')? (INT '.' INT? | '.' INT);
22
23 EString returns ecore::EString:
24     STRING | ID | TEMPLATE;
25
26 EInt returns ecore::EInt:
27     '-'? INT;
28
29 EFloat returns ecore::EFloat:
30   FLOAT
31 ;
32
33
34 //AbstractStructuralView returns StructuralView:
35 //     TStructuralView
36 //;
37
38 TAbstractMessageView returns AbstractMessageView:
39   {textRam::TAbstractMessageView}
```

```
40    'lifelines' '{'
41      lifelines+=TLifeline (';'? lifelines+=TLifeline)*
42    '}'
43
44    messages+=TAbstractMessages (messages+=TAbstractMessages)*
45  ;
46
47  TStructuralView returns StructuralView:
48      {textRam::TStructuralView}
49      'structure'
50      '{'
51          ( classes+=Classifier (classes+=Classifier)* )?
52          ('associations' '{' tAssociations+=TAssociation ( ";"? tAssociations+=
                TAssociation)* '}' )?
53      '}';
54
55  Classifier returns Classifier:
56      TClass;
57
58  //Type returns Type:
59  //     TClass | RVoid | RBoolean | RInt | RChar | RString | RAny |  RDouble | RFloat |
        RSet;
60
61  ObjectType returns ObjectType:
62      TClass | RBoolean | RInt | RChar | RString  | RDouble | RFloat | RSet;
63
64  PrimitiveType returns PrimitiveType:
65      RBoolean | RInt | RChar | RString | RDouble | RFloat;// | REnum;
66
67  RString: {RString}; RFloat: {RFloat}; RDouble: {RDouble}; RAny: {RAny}; RChar: {RChar};
        RInt: {RInt}; RBoolean: {RBoolean}; RVoid: {RVoid}; RSet: {RSet};
68
69  AbstractClass returns Class:
70    TClass
71  ;
72
73  TClassMember returns textRam::TClassMember:
74    TAttribute | TOperation
75  ;
76
77  TClass returns textRam::TClass:
78      {textRam::TClass}
79      ('@@x=' layoutX=EFloat)?
80      ('@@y=' layoutY=EFloat)?
81      (abstract?='abstract')?
82      'class'
83      (partial?='|')?
84      name=EString (':' (partialSuperType?='|')? superTypes+=[Classifier|EString])?
85      '{'
86        ( members+=TClassMember  ( ";"? members+= TClassMember)*  )?
87      '}';
88
89  TAttribute returns textRam::TAttribute:
90      (static?='static')?
91      type=[PrimitiveType|EString]
92      (partial?='|')? name=EString;
93
```

```
94  TOperation returns textRam::TOperation:
95      {textRam::TOperation}
96      (abstract?='abstract')?
97      (static?='static')?
98      visibility=Visibility
99      (partialReturnType?='|')?
100     returnType=[Type|EString]
101     (partial?='|')?
102     name=EString
103     '('
104       ( parameters+=TParameter ( "," parameters+=TParameter)* )?
105     ')';
106
107 TParameter returns textRam::TParameter:
108   {textRam::TParameter}
109     (partialType?='|')?
110     type=[Type|EString]
111     name=EString;
112
113 TAssociation returns textRam::TAssociation:
114     {textRam::TAssociation}
115     fromEnd=TAssociationEnd (referenceType=ReferenceType)? directionMultplicity=
            AssociationDirectionMultplicity toEnd=TAssociationEnd '{' name=EString '}'
116 ;
117
118 TAssociationEnd returns textRam::TAssociationEnd:
119   {textRam::TAssociationEnd}
120     (lowerBound=ElementBound
121     '..')?
122     (upperBound=ElementBound)?
123     classReference=[textRam::TClass|EString]
124 ;
125
126 ElementBound returns ecore::EInt:
127   '*' | INT;
128
129 TInstantiationHeader returns textRam::TInstantiationHeader:
130   type = InstantiationType
131   externalAspects+=[textRam::TAspect|EString] ("," externalAspects+=[textRam::TAspect|
          EString])*
132 ;
133
134 Instantiation returns Instantiation:
135     externalAspect=[Aspect|EString]
136     ('{' mappings+=TClassifierMapping ( ";"? mappings+=TClassifierMapping)* '}')?
137 ;
138
139 AbstractClassifierMapping returns ClassifierMapping:
140     TClassifierMapping
141 ;
142
143 TClassifierMapping returns textRam::TClassifierMapping:
144     (partialFromElement?='|')? fromElement=[Classifier|EString] ('<' fromMembers+=[
            textRam::TClassMember|EString] ("," fromMembers+=[textRam::TClassMember|EString
            ])* '>')?
145     '->'
```

```
146       (partialToElement?='|')? toElement=[Classifier|EString] ('<' toMembers+=[textRam::
              TClassMember|EString] ("," toMembers+=[textRam::TClassMember|EString])* '>')?
147   ;
148
149   TAbstractMessages returns textRam::TAbstractMessages:
150     TMessageView | TAspectMessageView
151   ;
152
153   TMessageView returns textRam::TMessageView:
154     {textRam::TMessageView}
155     'messageView' (create?='new')? (partialClass?='|')? class=[textRam::TClass|EString]
            '.' (partialOperation?='|')? specifies=[textRam::TOperation|EString]
156     '('
157         ( arguments+=[textRam::TParameter|EString] ( "," arguments+=[textRam::TParameter|
                EString])* )?
158       ')'
159       ('affectedBy' affectedBy+=[textRam::TAspectMessageView|EString] (',' affectedBy+=[
              textRam::TAspectMessageView|EString]* )? )?
160     ('{'
161       interactionMessages+=TInteraction (';'? interactionMessages+=TInteraction)*
162     '}')?
163   ;
164
165   TAspectMessageView returns textRam::TAspectMessageView:
166     {textRam::TAspectMessageView}
167     'aspectMessageView'
168       name=EString '{'
169       'pointcut' (create?='new')?  ( (partialClass?='|')? class=[textRam::TClass|EString]
              '.')? (partialOperation?='|')? specifies=[textRam::TOperation|EString]
170       '(' ( arguments+=[textRam::TParameter|EString] (',' arguments+=[textRam::TParameter
              |EString] )*  )? ')'
171       'advice' '{'
172         interactionMessages+=TInteraction (';'? interactionMessages+=TInteraction)*
173       '}'
174     '}'
175   ;
176
177   TLifeline returns textRam::TLifeline:
178     (static?='static')?
179     referenceType=TLifelineReferenceType
180     name=EString ':' (represents=[textRam::TTypedElement|EString])?
181     ('{'
182       localProperties+=TTemporaryProperty (";"? localProperties+=TTemporaryProperty)*
183     '}')?
184   ;
185
186   TValueSpecification returns textRam::TValueSpecification:
187     TTemporaryProperty | TParameter | TLifeline | TDummyValueSpecification
188   ;
189
190   TDummyValueSpecification returns textRam::TDummyValueSpecification:
191     dummy?='true'
192   ;
193
194   TTemporaryProperty returns textRam::TTemporaryProperty:
195     TReference | TLocalAttribute
196   ;
```

```
197
198 TReference returns textRam::TReference:
199   name=EString ':' (partialClass?='|')? reference=[textRam::TClass|EString]
200 ;
201
202 TLocalAttribute returns textRam::TLocalAttribute:
203   type=[PrimitiveType|EString]
204   name=EString
205   ('=' value=EString)?
206 ;
207
208 TInteraction returns textRam::TInteraction:
209   TInteractionMessage | TCombinedFragment | TOcurrence | TReturnInteraction
210 ;
211
212 TOcurrence returns textRam::TOcurrence:
213   leftLifeline=[textRam::TLifeline|EString] '=>' ocurrenceType=OcurrenceType
214 ;
215
216 TInteractionMessage returns textRam::TInteractionMessage:
217   leftLifeline=[textRam::TLifeline|EString] '=>' (create?='new')? rightLifeline=[
        textRam::TLifeline|EString]  ('{' message=TMessage '}')?
218 ;
219
220 TCombinedFragment returns textRam::TCombinedFragment:
221   interactionOperator=InteractionOperatorKind '[' interactionConstraint=EString ']' '{'
222     containers+=TInteraction ( containers+=TInteraction )*
223   '}'
224   ('else' '{'
225     elseContainers+=TInteraction ( elseContainers+=TInteraction )*
226   '}')?
227 ;
228
229 TReturnInteraction returns textRam::TReturnInteraction:
230   'return' return=[textRam::TValueSpecification|EString]
231 ;
232
233 TMessage returns textRam::TMessage:
234   (assignTo=[textRam::TMessageAssignableFeature|EString] ':=')?
235   (partialOperation?='|')? signature=[textRam::TOperation|EString] '(' ( arguments+=[
        textRam::TValueSpecification|EString] (',' arguments+=[textRam::
        TValueSpecification|EString] )*  )? ')'
236 ;
237
238 //TODO: It should be StructuralFeature | Parameter
239 TTypedElement returns textRam::TTypedElement:
240   TAssociation | TAttribute | TClass
241 ;
242
243 //AssignableFeature
244 TMessageAssignableFeature returns textRam::TMessageAssignableFeature:
245   TAssociation | TTemporaryProperty | TLifeline
246 ;
247
248 LiteralString: {LiteralString};
249
250 enum Visibility returns Visibility:
```

```
251   public = '+' | private = '-' | protected = '#' | package = "~"
252 ;
253
254 enum InstantiationType returns InstantiationType:
255     Depends = 'dependsOn' | Extends = 'extends';
256
257 enum MessageSort returns MessageSort:
258     synchCall = 'synchCall' | createMessage = 'createMessage' | deleteMessage = '
          deleteMessage' | reply = 'reply'
259 ;
260
261 enum ReferenceType returns ReferenceType:
262   Composition = '*' | Aggregation='o'
263 ;
264
265 enum AssociationDirectionMultplicity returns textRam::AssociationDirectionMultiplicity:
266   unidirectional = '->' | bidirectional = '&'
267 ;
268
269 enum InteractionOperatorKind returns InteractionOperatorKind:
270   alt = 'alt' | opt = 'opt' | loop = 'loop'
271 ;
272
273 enum OcurrenceType returns textRam::OcurrenceType:
274   original='*' | destruction='X'
275 ;
276
277 enum TLifelineReferenceType returns textRam::TLifelineReferenceType:
278   reference='ref' | association='assoc' | attribute = 'attr'
279 ;
```

Anexo B
Hoja de ayuda

Cheatography

TextRAM Cheat Sheet

by mjorod via cheatography.com/17938/cs/1710/

Aspect's sections

```
aspect Observer {
structure { ... }
instantiations { ... }
messages { ... }
}
```

Structural View

structural view	structure { ... }
class	class Observer { ... }
partial class	class |Observer { ... }
attribute	int myAttribute
public operation	+ startObserving(|Subject)
private operation	- privateOperation()
protected operation	~ |update()
associations section	associations { ... }
association named mySubject	|Observer -> 0..1 |Subject { mySubject }
unidirectional association	|Observer -> |Subject { mySubject }
bidirectional association	|Observer & |Subject { mySubject }
super type	class Derived : Supertype { ... }
abstract class	abstract class AbstractClass { ... }

Instantiations

instantiations section	instantiations { ... }
depends on	dependsOn ZeroToMany { ... }
extends	extends ZeroToMany { ... }
class mapping	|Data -> |Subject
operation mapping	|Data<|modify> -> |InterceptStateChange<|modifyData>
attribute mapping	|Data<sourceAttribute> -> |InterceptStateChange<targetAttribute>

Message View Headers

message view declaration	messageView |Data.add(|Associated a) { ... }
aspect message view	aspectMessageView initializationAssociation { ... }
message view reference	mesageView |Data.|create() affectedBy initializeAssociation

Control Flow in Messages View

alt combined fragment	alt [" i > 5 "] { ... }
opt combined fragment	opt [" i > 5 "] { ... } else { ... }
loop combined fragment	loop ["i = 0; i < 10; i++"] { ... }

Message View Body

lifelines declaration	lifelines { ... }
reference class lifeline	ref target:StockGUI
association lifeline	assoc mySet:Set
attribute lifeline	attr s:size
reference local variable	ref target:StockGUI { window:Window }
attribute local variable	ref target:Map { int size }
attribute with value local variable	ref s:Socket { String ip = "localhost" }
message ocurrence	add(parameter)
message ocurrence with assignment	result := remove(a)
return message	return result
interaction	target => mySet { result := remove(a) }
original behaviour ocurrence	target => *
destruction ocurrence	socket => X

Message View Aspects

pointcut	pointcut create(mySocket)
advice	advice { ... }

By **mjorod**
cheatography.com/mjorod/

Published 10th February, 2014.
Last updated 1st November, 2014.

Sponsored by **Readability-Score.com**
Measure your website readability!
https://readability-score.com

Anexo C
Proceso de instalación

La instalación de TextRAM exige el seguimiento estricto de las siguientes instrucciones. Las instrucciones están proporcionadas en inglés:

Getting Started

To get your IDE set up, the code and run TextRAM the first time, please follow all of these instructions.

Installing Java JDK

It is highly recommended that you use the Java JDK. The Java Runtime Environment is not enough. The JDK contains documentation and the Java source, which will help you when developing and debugging.

1. Install the Java JDK if you don't have it installed yet. You can get it from Oracle.

- **Mac OS X**: As of right now, TextRAM doesn't run on Java 7 (provided by Oracle). Go to Apple Developer and download the latest Java for OS X [...] Developer Package. You need an Apple ID (free) for that.

Installing Eclipse

While you may use your existing Eclipse installation, it is recommended to start off with the Pre-configured Eclipse with Xtext Xtext pre-configured. If you feel comfortable enough, you can add these to your existing installation.

1. Choose Help -> Install New Software... from the menu bar and Add... Insert the adequate URL from Xtext. This site aggregates all the necessary and optional components and dependencies of Xtext.

2. Select the Xtext SDK from the category Xtext {version} and complete the wizard by clicking the Next button until you can click Finish.

3. After a quick download and a restart of Eclipse, Xtext is ready to use.

Note: Xtext relies on the Antlr generator library. Xtext will try to download it on demand. To avoid this, you can manually install the feature Xtext Antlr-2.0.0 from the itemis update site (http://download.itemis.de/updates/).

Installing required plug-ins and setting up

The following plug-ins are required to be installed. Eclipse gets already shipped with Git support (EGit)

OCL Samples and Editors

The metamodel of RAM makes use of OCL for derivation, operation bodies and constraints.

1. Go to Help > Install New Software.
2. Select the release update site of the current Eclipse version (e.g., Kepler - [...] for Eclipse Kepler).
3. Choose OCL Samples and Editors from the Modeling category.
4. Follow the instructions.

That's it. Now you can now configure Git before you get the code.

Configure Git

The code is maintained using the distributed version control system Git. You can either use your local Git client or Eclipse, which by default has Git support (EGit). We recommend using Eclipse.

Now you can get the code.

Get the Code

GitHub repository provides you with the repository URL (https://github.com/mjorod/textram.git) when you select Clone on the repository.

Using Eclipse

1. Open the Git Repository Exploring perspective.
2. Clone the repository.
3. Import projects from the cloned repository.

Using the command-line

1. Clone the repository to a local location of your choice.
2. Import the projects into your workspace in Eclipse.

Once that is done you can run TextRAM.

Run TextRAM

1. You must make sure that the launch configuration has enough PermGen size. You need to specify this VM argument in your launch configuration: -XX:MaxPermSize=256m
2. Xtext relies on MWE2 to generate the language infrastructure of a DSL. Therefore whether you are installing TextRAM for the first time or you changed the DSL's grammar definition, you have to run the artefact generator. Locate cl.uchile.pleiad.textram/src/GenerateTextRAM.mwe2 and from its context menu choose Run As ->MWE2 Workflow
3. In order to run TextRAM, right-click on the project cl.uchile.pleiad.textram and navigate to Run As | Run As Configurations...; in the dialog you will see Launch Runtime Eclipse under Eclipse Application; Select that and click Run.

4. A new Eclipse instance will be run and a new workbench will appear. Let's create a new General project and create a new file with the name of the aspect that you want to model. The extension of the file must be .xram. As soon as the file is created it will also be opened in a text editor and you will be asked to add the Xtext nature to your project. You should accept that to make your TextRAM editor work correctly in Eclipse.

5. To be able to see RAM's generated models you have to create a folder named src-gen in the project you are working on. The model aspects will be saved on src-gen/aspects.

Anexo D
Modelos SM de TextRAM

Instancia RAM de ZeroToMany-Ordered

```
1  aspect Betting extends Game {
2    structure {
3          class Machine {
4            int credits
5            + void withDrawCredits(int amount)
6            + void depositCredits (int amount)
7            + void initialize()
8            + void play()
9            + Reel getReelAtPosition(int index)
10           + void addCredits(int credit)
11         }
12
13         class Reel {
14           + Symbol getCurrentSymbol()
15         }
16
17         class Symbol {}
18
19         class PayTable {
20           + PayTable create(String fileName)
21           + int payout(Symbol first , Symbol second , Symbol third , Symbol fourth , Symbol
                 fifth)
22         }
23
24         associations {
25           Machine -> 0 .. 1 PayTable { myPayTable }
26         }
27       }
28
29       messages {
30         lifelines {
31           ref target:Machine { String default = "default"; int credits; r1:Reel; int
                 index; s1:Symbol; s2:Symbol; s3:Symbol; s4:Symbol; s5:Symbol; int win }
32           ref myPayTable:PayTable
33           ref r:Reel
34         }
35
36         aspectMessageView loadPayTable {
```

```
37            pointcut Machine.initialize()
38            advice {
39              target =>*
40              target => new myPayTable { myPayTable := create(fileName) }
41            }
42          }
43
44          aspectMessageView playWithMoney {
45            pointcut Machine.play()
46            advice {
47              opt ["credits >= 5"] {
48                target => target     { withDrawCredits(credits) }
49                target =>*
50                target => target     { r1 := getReelAtPosition(index) }
51                target => r          { s1 := getCurrentSymbol() }
52                target => myPayTable { win := payout(s1, s2, s3, s4, s5) }
53                target => target     { addCredits(win) }
54              }
55            }
56
57          }
58
59          messageView Machine.initialize() affectedBy loadPayTable
60          messageView Machine.play() affectedBy playWithMoney
61
62
63
64        }
65  }
```

```
1   aspect Blockable {
2     /* Should suspend callers that call |blockableOperation
3       until continue or skip is called.
4       If continue is called, the operation call proceeds.
5       If skip is called, the operation is not called
6       and instead the value given as the parameter to skip is returned.
7     */
8     structure {
9
10      class |Blockable {
11        boolean blocked
12        + boolean skip()
13        + void skip (|ReturnType value)
14          + |ReturnType |blockableOperation()
15        + void continue()
16        + void block()
17      }
18
19      class |ReturnType {}
20    }
21  }
```

```
1   aspect CriticalError extends Betting, dependsOn Named, Blockable, Singleton {
2     structure {
3       class ArrayOfInt {}
4
5       class Symbol {}
```

```
6
7       class RandomReels {
8         + ArrayOfInt spinReels()
9         + void block()
10        + void continue( )
11      }
12
13      class CriticalError {
14        + CriticalError create (String errorName)
15      }
16
17      class Machine {
18        + void withDrawAllCredits()
19        + void ceasePaying()
20        + void unblockMachine()
21        + void blockMachine(CriticalError e)
22        + Machine getMachine()
23        + |ReportingInterface getReportingInterface()
24        + int currentCredits()
25        + void withDrawCredits(int credits)
26
27      }
28
29      class |ReportingInterface {
30        + void clearReport()
31        + void reportCritical(CriticalError e)
32      }
33
34      class PayTable {
35        + int payout(Symbol first, Symbol second, Symbol third, Symbol fourth, Symbol
              fifth)
36        + void block()
37        + void continue( )
38        + void skip()
39      }
40
41      associations {
42        Machine -> ReportingInterface { myReportingInterface }
43      }
44    }
45
46    instantiations {
47      Named {
48        |Named -> CriticalError
49      }
50
51      Blockable {
52        |Blockable <blockableOperation> -> RandomReels < spinReels >;
53        |ReturnType -> ArrayOfInt;
54        |Blockable <blockableOperation> -> PayTable < payout >
55      }
56
57      Singleton {
58        |Singleton <getInstance> -> Machine <getMachine>
59      }
60    }
61
```

```
62    messages {
63      lifelines {
64        ref target:CriticalError
65        ref machine:Machine {  ri:|ReportingInterface; int c }
66        ref myRandom:RandomReels
67        ref myPayTable:PayTable
68        ref myRI:ReportingInterface { err:CriticalError }
69      }
70
71      messageView CriticalError.create(errorName) {
72        target => machine { blockMachine(target) }
73      }
74
75      /*
76       * Does not include user interface lockup screen
77       */
78      messageView Machine.blockMachine(e) {
79        machine => myRandom   { block() }
80        machine => myPayTable { block()  }
81        machine => machine    { ri := getReportingInterface() }
82        machine => myRI       { reportCritical(e) }
83      }
84
85      messageView Machine.unblockMachine() {
86        machine => myRandom   { continue()  }
87        machine => myPayTable { continue() }
88        machine => machine    { ri := getReportingInterface() }
89        machine => myRI       { clearReport() }
90      }
91
92      messageView Machine.ceasePaying() {
93        machine => myRandom   { continue() }
94        machine => machine    { withDrawAllCredits() }
95        machine => myPayTable { skip() }
96        machine => machine    { ri := getReportingInterface() }
97        machine => myRI       { clearReport() }
98      }
99
100     messageView Machine.withDrawAllCredits() {
101       machine => machine { c:= currentCredits() }
102       machine => machine { withDrawCredits(c) }
103     }
104   }
105 }
```

```
1   aspect Demo extends Betting, dependsOn Singleton {
2
3     structure {
4       class ArrayOfInt {}
5       class Symbol {}
6
7       class RandomReels {
8         + ArrayOfInt spinOfReels()
9         + ReelConfiguration getMyReelConfiguration()
10      }
11
12      class Machine {
```

```
13        boolean demoMode
14      + void setDemo(boolean on)
15      + ArrayOfInt getNextOutCome(ArrayOfInt Symbol)
16      + void setNextOutCome(Symbol s)
17      + Machine getMachine()
18      + int currentCredits()
19      + int savedCredits()
20    }
21
22    class ReelConfiguration {
23      + int getPositionForSymbol(int reel, Symbol sym)
24    }
25  }
26
27  instantiations {
28    Singleton {
29      |Singleton <getInstance> -> Machine <getMachine>
30    }
31  }
32
33  messages {
34    lifelines {
35      ref target:Machine { int savedCredits; int credits; null:Symbol }
36      ref randomReels:RandomReels { result:ArrayOfInt; outcome:ArrayOfInt;
            rc:ReelConfiguration }
37      ref rc:ReelConfiguration { int i; outcomeItem:Symbol }
38    }
39
40    messageView Machine.setDemo(on) {
41      alt [ "on" ] {
42        target => target { savedCredits := currentCredits() }
43      }
44      else {
45        target => target { credits := savedCredits() }
46      }
47      target => target { setNextOutCome(null) } //setNextOutCome(null)
48    }
49
50    aspectMessageView determineNextOutcome {
51      pointcut RandomReels.spinOfReels() // result := spinOfReels
52      advice {
53        randomReels => target { outcome := getNextOutCome(result) }
54        alt [ "outcome != null" ] {
55          randomReels => target { setNextOutCome(null) } //setNextOutCome(null)
56
57        randomReels => randomReels { rc := getMyReelConfiguration() }
58
59          loop [ "int = 0; i < 5; i++" ] {
60            randomReels => rc { getPositionForSymbol (i, outcomeItem) } //
                  getPositionForSymbol(i, outcome[0])
61          }
62          }
63          else {
64            randomReels =>*
65          }
66
67          return outcome
```

```
68
69            }
70          }
71
72        messageView RandomReels.spinOfReels() affectedBy determineNextOutcome
73
74
75    }
76  }
```

```
1   aspect DemoNetworkResolution {
2     structure {
3       class Result {}
4
5       class OutOfServiceResult : Result {
6         + OutOfServiceResult create()
7       }
8
9     }
10
11  }
```

```
1   aspect Game dependsOn ZeroToManyOrdered, Random {
2     structure {
3
4       class ArrayOfInt {}
5       class Symbol {}
6
7       class RandomReels {
8         - int spinReel()
9         + RandomReels create()
10        + ArrayOfInt spinReels()
11      }
12
13      class Machine {
14        + void initialize()
15        + void play()
16        + void addReelAtPosition(int index, Reel a)
17        + Reel getReelAtPosition(int index)
18      }
19
20      class Reel {
21        int position
22        + int getCurrentPosition()
23        + Symbol getCurrentSymbol()
24        + void setCurrentPosition(int newPosition)
25        + void addSymbol(Symbol a)
26        + Symbol getSymbol(int position)
27        + Reel create()
28        + Symbol getSymbolAtPosition(int position)
29        + void addSymbolAtPosition(int position, Symbol s)
30      }
31
32      class ReelConfiguration {
33        - ReelConfiguration create(String filename)
34        - Symbol getSymbol(int reel, int position)
35      }
```

```
36
37      associations {
38        Machine -> 1 RandomReels      { myRandom }
39        Machine -> ReelConfiguration { myRC }
40      }
41    }
42
43    instantiations {
44      ZeroToManyOrdered {
45        |Data <add, get> -> Machine <addReelAtPosition, getReelAtPosition>;
46        |Associated -> Reel
47        |Data <add, get> -> Reel <addSymbol, getSymbol>
48        |Associated -> Symbol
49      }
50
51      Random {
52        RandomGenerator <nextRandom> -> RandomReels <spinReels>
53      }
54    }
55
56    messages {
57      lifelines {
58        ref target:Machine { s:Symbol; result:ArrayOfInt; r:Reel; int i; int j; int
              intValueFromResult }
59        ref myRandom:RandomReels
60        ref myRC:ReelConfiguration { String default = "default" }
61        ref newReel:Reel { symbolResult:Symbol; int position }
62      }
63
64      messageView Machine.initialize() {
65        target => new myRandom { myRandom := create() }
66        target => new myRC     { myRC := create(default) }
67
68        loop ["int␣i␣=␣0;␣i␣<␣5;␣i++"] {
69          target => new newReel { newReel := create() }
70          target => target { addReelAtPosition(i, newReel) }
71
72          loop ["int␣j␣=␣0;␣j␣>␣60;␣j++"] {
73            target => myRC   { s:= getSymbol(i, j) }
74            target => newReel{ addSymbolAtPosition(j, s)   }
75          }
76        }
77      }
78
79      messageView Machine.play() {
80        target => myRandom { result := spinReels() }
81        loop["int␣i␣=␣0;␣i␣<␣result;␣i++"] {
82          target => target  { r := getReelAtPosition(i) }
83          target => newReel { setCurrentPosition(intValueFromResult) }
84        }
85      }
86
87      messageView Reel.getCurrentSymbol() {
88        newReel => newReel { symbolResult := getSymbolAtPosition(position) }
89        return symbolResult
90      }
91    }
```

```
}
```

```
aspect InterceptReturnsMetering extends Metering {

  structure {
    class |InterceptReturns {
      int data
      ~ int getData()
      ~ void setData(int data)
      + void interceptedMethod() //  + *|interceptedMethod(..)
    }
  }

  instantiations {
    Metering {
      |InterceptStateChange <modifyData> -> |InterceptReturns <setData>
    }
  }

  messages {
    lifelines {
      ref target:InterceptReturns  { int data }
    }

    aspectMessageView interceptedMethod {
      pointcut  |InterceptReturns.interceptedMethod()
      advice {
        target =>*
        target => target { setData(data) } //setData(returnValue)
//        target => <<
      }
    }
  }

}
```

```
aspect Map {

  structure {

    class |DataMap {
      + |DataMap create()
      + void add(|Key key, |Value value)
      + void remove(|Key key)
      + |Value getValue(|Key key)
      + Set<Key> getKeys()
      + Set<Value> getValues()
    }

    class Map {
      ~ Map create()
      ~ void put(|Key k, |Value v)
      ~ void remove(|Key k)
      ~ |Value get(|Key k)
      ~ Set<Key> keySet()
      ~ Set<Value> getValues()
      ~ void destroy()
```

```
22      }
23
24      class |Key {}
25      class |Value {}
26
27      associations {
28        Map o-> 0 .. * Value { myValue }
29        DataMap *-> 1 Map { myMap }
30      }
31    }
32
33    messages {
34      lifelines {
35        ref data:DataMap
36        ref key:Key
37        ref val:Value
38
39        // associations
40        assoc myMap:myMap { result:Map }
41      }
42
43      aspectMessageView initializeAssociation {
44        pointcut new |DataMap.create()
45
46        advice {
47          data => new myMap { myMap := create() }
48          data =>*
49  //          >> => data
50        }
51      }
52
53      messageView |DataMap.add(key, value) {
54        data => myMap { put(key, val) }
55      }
56
57      messageView |DataMap.remove(key) {
58        data => myMap { remove(key) }
59      }
60
61      messageView |DataMap.getKeys() {
62        data  => myMap { keySet() } // result := keySet()
63  //      myMap => <<     //{  result }
64      }
65
66      messageView |DataMap.getValue(key) { // result := getValues()
67        data  => myMap { get(key) }
68  //      myMap => <<     //{  result }
69      }
70
71  //    messageView |Data.getValues(key) { // result := getValues()
72  //      data  => myMap { get(key) }
73  ////        myMap => <<     //{  result }
74  //    }
75
76      messageView |DataMap.create() affectedBy initializeAssociation
77    }
78  }
```

```
1  aspect Metering dependsOn Observer, Named {
2    structure {
3      class |InterceptStateChange {
4        int data
5        + int getData()
6        //+ * |ModifyData
7        + void |modifyData(int value)
8      }
9
10     class Meter {
11       int value
12       + Meter create(String name, int initialValue)
13       + void setValue(int newValue)
14       + void attachMeter(|InterceptStateChange s)
15       + void notifyChange(|InterceptStateChange s)
16       ~ void updateMeter(|InterceptStateChange s)
17     }
18
19     class StandardMeter : Meter {
20       + void updateMeter(|InterceptStateChange s)
21     }
22
23     class AdditiveMeter : Meter {
24       ~ void updateMeter(|InterceptStateChange s)
25     }
26
27     class CountingMeter : Meter {
28       ~ void updateMeter(|InterceptStateChange s)
29     }
30   }
31
32   instantiations {
33     Observer {
34       |Subject < modify > -> |InterceptStateChange < modifyData >
35       |Observer < update, startObserving > -> Meter < notifyChange, attachMeter >
36     }
37
38     Named {
39       |Named -> Meter
40     }
41   }
42
43   messages {
44     lifelines {
45       ref target:Meter { int v }
46       ref interceptpStateChange:InterceptStateChange
47       ref standardMeter:StandardMeter { int n }
48     }
49
50     messageView Meter.notifyChange(s) {
51       target => interceptpStateChange { v:= getData() }
52       target => target { updateMeter(interceptpStateChange) }
53     }
54
55     messageView StandardMeter.updateMeter(s)  {
56       standardMeter => standardMeter { updateMeter(s) }
57     }
```

```
58    }
59  }
```

```
1   aspect NetErrorsArbiter extends SMNetwork, NetworkedError, dependsOn Map, Named,
        Singleton {
2
3     structure {
4
5       class NetworkCommandExecution {
6         + NetworkCommandExecution getInstance()
7         + void sendMessage(NetworkCommandExecution executor)
8       }
9
10      class NetErrorsArbiter : NetworkCommandExecution {
11        + void sendMessage(ErrorResult err)
12        + NetworkCommandExecution getValue()
13        + Arbiter getArbiter()
14      }
15
16      class ErrorResult {
17        // not originally defined
18        + StringClass getName()
19        + NetworkCommandExecution getValue()
20
21      }
22
23      class StringClass {}
24      class BooleanClass {}
25      class Arbiter {}
26    }
27
28    instantiations {
29
30
31      Named {
32        |Named -> ErrorResult
33      }
34
35      Singleton {
36        |Singleton <getInstance> -> NetworkCommandExecution <getInstance> // Arbiter <
              getInstance >
37      }
38
39      Map {
40        |Key -> StringClass
41        |Value -> BooleanClass
42        |DataMap  -> NetErrorsArbiter
43      }
44    }
45
46    messages {
47      lifelines {
48        ref nce:NetworkCommandExecution { result:Arbiter }
49        ref nea:NetErrorsArbiter { name:StringClass; executor:NetworkCommandExecution }
50        ref res:ErrorResult
51      }
52
```

```
53      aspectMessageView arbitExce {
54        pointcut NetworkCommandExecution.getInstance()
55        advice {
56          nce => nea { result:= getArbiter() }
57
58          return result
59        }
60      }
61
62      messageView NetErrorsArbiter.sendMessage(err) {
63        nea => res { name:= getName() }
64        nea => nea { executor := getValue() }
65        nea => nce { sendMessage(executor) }
66      }
67    }
68  }
```

```
1   aspect NetworkedError extends CriticalError, NormalError, SMNetwork, dependsOn
        Singleton {
2
3     structure {
4       class |ReportingInterface {
5         + void reportCritical(CriticalError err)
6         + void reportError(NormalError err)
7       }
8
9       class NetworkCommandExecutor {
10        + void sendMessage(Result res)
11        + NetworkCommandExecutor getInstance()
12      }
13
14      class ErrorResult : Result {
15        + ErrorResult create(CriticalError err)
16        + ErrorResult create(NormalError err)
17      }
18
19      class CriticalError {}
20      class NormalError {}
21      class Result {}
22    }
23
24    instantiations {
25      Singleton {
26        |Singleton <getInstance> -> NetworkCommandExecutor <getInstance>
27      }
28    }
29
30    messages {
31      lifelines {
32        ref ri:ReportingInterface { res:Result }
33        ref nce:NetworkCommandExecutor
34        ref er:ErrorResult { err:CriticalError }
35      }
36
37      aspectMessageView sendCritical {
38        pointcut |ReportingInterface.reportCritical(err)
39        advice {
```

```
40          ri => er  { res := create(err)  }
41          ri => nce { sendMessage(res) }
42        }
43      }
44
45      aspectMessageView sendNormal {
46        pointcut reportError(err)
47        advice {
48          ri => er  { res := create(err) }
49          ri => nce { sendMessage(res) }
50        }
51      }
52    }
53  }
```

```
1   aspect NormalError dependsOn Named, Singleton {
2
3     structure {
4
5       class NormalError {
6         + NormalError create(String errorName)
7       }
8
9       class Machine {
10        + void reportError(NormalError err)
11        /*
12         * Should be called by user interface
13         */
14        + void clearError()
15        + Machine getMachine()
16        + ReportingInterface getMyReporingInterface()
17      }
18
19      class ReportingInterface {
20        + void reportError(NormalError err)
21        + void clearReport()
22      }
23    }
24
25    instantiations {
26      Named {
27        |Named -> NormalError
28      }
29
30      Singleton {
31        |Singleton <getInstance> -> Machine <getMachine>
32      }
33    }
34
35    messages {
36      lifelines {
37        ref normalError:NormalError { e:NormalError }
38        ref machine:Machine { ri1:ReportingInterface }
39        ref ri:ReportingInterface
40      }
41
42      messageView NormalError.create(errorName) {
```

```
43        normalError => machine { reportError(e) }
44      }
45
46      messageView Machine.reportError(err) {
47        machine => machine { getMyReporingInterface() }
48        machine => ri { reportError(normalError) }
49      }
50
51      messageView ReportingInterface.clearReport() {
52        machine => machine { ri1 := getMyReporingInterface() }
53        machine => ri      { clearReport() }
54      }
55    }
56  }
```

```
1   aspect ReturningCommand {
2
3     structure {
4       class |Command {
5         + void execute()
6       }
7     }
8
9   }
```

```
1   aspect SlotMachineMetering dependsOn Metering, InterceptReturnsMetering, CriticalError,
        NormalError, WeakError {
2     structure {
3
4       class Machine {
5         int credits
6         + void play(int i)
7         ~ int getCredits()
8         + void depositCredits(int amount)
9         + void initialize()
10        + void initializeSlotMachineMeters()
11        + void attachMeterToPlay()
12        + void attachMeterToDepositCredits()
13      }
14
15      class Symbol {}
16
17      class PayTable {
18        + int payout(Symbol first, Symbol second, Symbol third, Symbol fourth, Symbol
              fifth)
19      }
20
21      class InterceptSM {
22        + void create(String tag, int credit)
23        + void attachMeterToPayout(Machine m)
24      }
25
26      class StandardMeter {
27        + void create(String tag, int credit)
28        + void attacheMeterToPlay(Machine m)
29      }
30
```

```
31      class CountingMeter {
32        + void create(String tag, int credit)
33        + void attachMeterToDepositCredits(Machine m)
34      }
35    }
36
37    instantiations {
38      Metering {
39        |InterceptStateChange <getData, modifyData> -> Machine <getCredits, play>
40        //Meter <attachMeter> -> Machine <attachMeterToPlay>
41      }
42      Metering {
43        |InterceptStateChange <getData, modifyData> -> Machine <credits, depositCredits>
44        //Meter <attachMeter> -> Meter <attachMeterToDepositCredits>
45      }
46      InterceptReturnsMetering {
47        |InterceptReturns <interceptedMethod> -> PayTable <payout>
48        //Meter<attachMeter> -> Meter<attachMeterToPayout>
49        StandardMeter -> InterceptSM
50      }
51    }
52
53    messages {
54
55      lifelines {
56        ref target:Machine
57        ref sm:StandardMeter  { String s = "CurrentCredits"; int c }
58        ref cm:CountingMeter  { String s1 = "NumberOfDeposits"; int c1 }
59        ref cm2:CountingMeter { String s2 = "NumberOfPlays"; int c2 }
60        ref im:InterceptSM    { String s3 = "LastPayout"; int c3 }
61      }
62
63      aspectMessageView setupMeters {
64        pointcut Machine.initialize()
65        advice {
66          target =>*
67          target => target { initializeSlotMachineMeters() }
68        }
69      }
70
71      messageView Machine.initializeSlotMachineMeters() {
72        target => new sm { sm := create(s, c) }
73        target => sm     { attacheMeterToPlay(target) }
74
75        target => new cm { cm := create(s1, c1) }
76        target => cm     { attachMeterToDepositCredits(target) }
77
78        target => new cm2 { cm2 := create(s2, c2) }
79        target => cm2     { attachMeterToDepositCredits(target) }
80
81        target => new im { im := create(s3, c3) }
82        target => im     { attachMeterToPayout(target) }
83      }
84
85    }
86  }
```

```
1  aspect SMNetwork extends Betting dependsOn Named, ReturningCommand {
2
3    structure {
4
5      class |Result {}
6
7      class |MeterResult : |Result {
8        + |MeterResult create(int value)
9      }
10
11     class MeterRepository {
12       + int getValueOfMeter(String name)
13     }
14
15     class |SMCommand {
16       ~ |Result execute()
17     }
18
19     class |ReadMeterCommand {
20       ~ |Result execute()
21       + String getMeterName()
22     }
23
24     class NetworkCommandExecutor {
25       + void receiveCommand(|SMCommand com)
26       + void sendMessage(|Result res)
27     }
28   }
29
30   instantiations {
31     ReturningCommand {
32       |Command <execute> -> |SMCommand <execute>
33     }
34     Named {
35       |Named <getName> -> |ReadMeterCommand <getMeterName>
36     }
37   }
38
39   messages {
40     lifelines {
41       ref target:NetworkCommandExecutor { result:|Result }
42       ref com:SMCommand
43       ref rmc:ReadMeterCommand { String name; int value; res:|MeterResult }
44       ref mr:MeterRepository
45       ref mres:MeterResult
46     }
47
48     messageView NetworkCommandExecutor.receiveCommand(com) {
49       target => com { result := execute() }
50
51       opt [ "result != null" ] {
52         target => target { sendMessage(result) }
53       }
54     }
55
56     messageView |ReadMeterCommand.execute() {
57       rmc => rmc  { name  := getMeterName() }
```

```
58        rmc => mr   { value := getValueOfMeter(name) }
59        rmc => mres { res   := create(value) }
60
61        return res
62      }
63    }
64  }
```

```
1   aspect SMNetworkArbiter extends SMNetwork, dependsOn Map, Named, Singleton {
2
3     structure {
4       class Arbiter {
5         ~ Result notAllowed()
6         + Arbiter getArbiter()
7         + boolean getValue()
8
9       }
10
11      class Command {
12        + Result execute()
13        + String getName()
14        + boolean getValue()
15      }
16
17      class Result {
18      }
19
20      class NotAllowed : Result {
21        + NotAllowed create()
22      }
23
24      class StringType {}
25      class BooleanType {}
26    }
27
28    /*
29     * Incomplete in that the map is not filled in.
30     * It should be done at instantiation time or configuration time
31     *
32     */
33    instantiations {
34      Map {
35        |Key     -> StringType
36        |Value   -> BooleanType
37        |DataMap -> Arbiter
38      }
39      Named {
40        |Named   -> Command
41      }
42      Singleton {
43        |Singleton <getInstance> -> Arbiter <getArbiter>
44      }
45
46    }
47
48    messages {
49      lifelines {
```

```
50        ref com:Command { String name; boolean allowed; ret:NotAllowed }
51        ref arb:Arbiter
52        ref na:NotAllowed
53      }
54
55      /**
56       * Independent of actual protocols being used.
57       * Only assumes that the commands are named differently according
58       * to the protocol being used,
59       * such that they can be configured unambiguously.
60       *
61       */
62      aspectMessageView arbitExce {
63        pointcut execute()
64        advice {
65          com => com { name := getName() }
66          com => arb { allowed := getValue() }
67
68          alt [ "allowed" ] {
69            com =>*
70          }
71          else {
72            com => new na { ret:=create() }
73            return ret
74          }
75        }
76      }
77
78      messageView Command.execute() affectedBy arbitExce
79    }
80  }
```