Aplicando MDD al desarrollo de sistemas agropecuarios: Modelado de sistemas de control de calidad de granos almacenados en silobolsas

Director:

Dra. Claudia Pons
(claudia.Pons@lifia.info.unlp.edu.ar)

Alumnos:

Miguel Martinez -3828/8
(martinezmiguel80@gmail.com)

Sebastian Lavié 4296/3
(sebastianlavie@gmail.com)

Tabla de Contenidos

Parte IV. Definición del DSL para modelado de sistemas de control de granos almacenados en silobolsas

Introducción

Las silobolsas son un sistema de almacenamiento de granos que está siendo cada vez más adoptada por los productores en muchos países. El sistema incluye una bolsa de polímero especializado, carga de granos dedicados y equipo de extracción.

Las silobolsas son aún una tecnología en progreso, sin embargo ya se han convertido en una de las principales opciones de almacenamiento de grano en granja en los países de América Latina, especialmente en Argentina, donde más de 20 millones de toneladas por año fueron almacenadas en las temporadas 2005/06 y 2006/07.

El Instituto Nacional de Agricultura INTA [1] en la Argentina y el CSIRO [2] en Australia, entre otros, han llevado a cabo actividades de investigación para determinar los límites y los riesgos de la tecnología existente en las silobolsas. Esto incluye investigar el potencial de esta tecnología para mejorar o utilizarse de manera más eficaz para superar cualquier problema identificado.

La pérdida de la calidad del grano es altamente dependiente de la duración del almacenamiento y los niveles de humedad, temperatura y dióxido de carbono en la silobolsa. Los incrementos de los niveles de estas variables dentro de la bolsa es una amenaza muy peligrosa para la preservación del estado de granos

Las investigaciones confirman que las silobolsas, aunque tienen algunas limitaciones, ofrecen a los productores un medio relativamente barato y una solución fiable de almacenamiento de granos. Por lo tanto, es importante trabajar en pro de la mejora de la tecnología de las silobolsas. En particular, la incorporación de sistemas de software sería un activo valioso.

En esta tesis se aplicará una rama de la ingeniería de software denominada Desarrollo de Software Dirigido por Modelos (MDD por sus siglas en ingles: Model Driven software Development)

El enfoque de Desarrollo Dirigido por Modelos (MDD), surge como un cambio del paradigma de desarrollo de software centrado en el código al desarrollo basado en modelos. Este enfoque promueve la sistematización y la automatización de la construcción de artefactos de software. Los modelos son considerados constructores de primera clase en el desarrollo de software, y el conocimiento de los desarrolladores se encapsula a través transformaciones de modelos. La característica esencial de la MDD es que el el foco primario de desarrollo de software y los productos del trabajo, son modelos. Su principal ventaja es que los modelos se pueden expresar en los diferentes niveles de abstracción y, por lo tanto están menos vinculados a algún tipo de soporte tecnológico.

El objetivo de esta tesis es proveer soporte para la creación de modelos de software para sistemas de control de calidad de granos almacenados en silobolsas y sistemas similares en otras áreas de agro-tecnología.

Para cumplir con los objetivos planteados, se llevó adelante los siguientes desarrollos que tomarán un tiempo estimado de 6 meses:

1- Análisis del dominio de los sistemas de control de silobolsas.

2- Evaluación de formalismos de meta-modelado existentes.

3- Definición de la sintaxis abstracta y concreta del lenguaje de modelado de los sistemas de control de silobolsas.

4- Análisis de requerimientos para el editor grafico que permita la construcción de modelos propuestos.

6- Construcción del prototipo de editor gráfico.

7- Validación del prototipo de editor gráfico en la construcción de una aplicación real.

El resultado final de esta tesis es una herramienta de software que brinda soporte a la creación de modelos de software para sistemas de control de calidad de granos almacenados en silobolsas.

Parte I: Modelado Específico del Dominio

En esta sección se describirá la metodología de Modelado Específico de Dominio, sus características, diferencias con metodologías clásicas y otras metodologías similares que se pueden hallar en la actualidad, ventajas provistas y factores negativos.

Capitulo 1 – Introducción

El Modelado Específico de Dominio o Domain Specific Modeling (DSM) es una metodología de desarrollo de software en la cual se busca incrementar el grado de abstracción en la representación de las aplicaciones o sistemas más allá de los lenguajes de programación convencionales, utilizando conceptos y reglas tomados directamente del dominio del problema de software a resolver. La metodología propone comenzar el desarrollo con una definición formal de los conceptos y reglas que caracterizarán a las representaciones de alto nivel (modelos), las cuales serán el artefacto principal de especificación de software. La obtención de representaciones de un menor nivel de abstracción como, por ejemplo, código fuente, son logradas a través de un proceso de derivación, el cual toma como entrada los modelos formales confeccionados de acuerdo a los conceptos y reglas definidos. Esta derivación es posible gracias a la cota que se pone en la cantidad de aspectos considerados del problema de software a tratar, la cual conforma el "dominio específico" de los modelos y promueve un incremento en la semántica de los conceptos y reglas utilizadas en las descripciones. Tal incremento abre paso a una generación de código completa, en el sentido de que los desarrolladores no requieren proveer más información que los mismos modelos a fin de obtener la implementación final, a costas de acotar las soluciones que se pueden obtener a un dominio particular.

En contraste con otros procesos de desarrollo de software, la arquitectura de la metodología DSM requiere, por parte de unos pocos desarrolladores con experiencia en el dominio particular, la creación de 3 elementos principales que serán la herramienta fundamental del equipo que llevará a cabo la construcción del software:

1. Un lenguaje de dominio específico formal a través del cual se definen las soluciones.
2. Un generador de código para el lenguaje anteriormente especificado.
3. Un framework de dominio que sirva como base al código generado en el punto anterior para que la traducción sea más sencilla.

Un dominio, según la metodología, es definido como un área de interés en relación al desarrollo de software, y puede tratarse de situaciones comunes y acotadas a una problemática particular en dicho desarrollo como, por ejemplo, persistencia, diseño de interfaces de usuario, comunicación, etc., así como también puede referirse a un campo específico para el cual el mismo está destinado como telecomunicaciones, procesos de negocios, control de robots, manejo de ventas, etc. Cuanto más se acote el dominio, más posibilidades habrá de generar aplicaciones funcionales directamente de las representaciones de alto nivel, dado que mayor será el contenido semántico de los componentes y reglas que las conforman.

En el desarrollo basado en modelos de dominio específico, los modelos son los principales artefactos de especificación de software, permitiendo a través de los mismos incrementar la abstracción y ocultar la complejidad, obteniéndose las implementaciones finales a partir de ellos. Los elementos con los que se construyen los

modelos están definidos a través de un modelo particular denominado meta-modelo, necesitando éste a su vez ser definido usando un lenguaje particular. Este último es conocido como meta-metamodelo principales artefactos de especificación de software, permitiendo a través de los mismos incrementar la abstracción y ocultar la complejidad, obteniéndose las implementaciones finales a partir de ellos.

Capitulo 2 - Breve Historia

La historia del desarrollo de software ha mostrado que los incrementos en la productividad en el mismo han estado asociados mayormente a incrementos en la abstracción de las especificaciones utilizadas para describirlo. Sin embargo, a comparación de cómo la aparición de los lenguajes de tercera generación (3GLs) contribuyeron a incrementar la productividad hace algunas décadas, los lenguajes de modelado y programación tradicionales en la actualidad están contribuyendo relativamente poco en este campo [Kelly08]. Es en este punto en donde la metodología de Modelado Específico de Dominio intenta innovar, persiguiendo un incremento en la abstracción más allá del provisto por los lenguajes de programación y técnicas de modelado genéricas actuales, utilizando para describir la solución conceptos del dominio del problema de software a tratar.

DSM surge como una solución a la "desconexión" entre la etapa de modelado y de implementación de las metodologías actuales. En la primera, se generan un conjunto de modelos para abstraer aspectos de implementación que son complejos o muchas veces desconocidos en las momentos tempranos del desarrollo. Luego, en la etapa de implementación subsiguiente, los modelos son interpretados por los programadores, quienes manualmente convierten las descripciones abstractas en implementaciones concretas. Esta traducción manual e informal, es propensa a errores frecuentes en el desarrollo, los cuales resultan en una disminución de la calidad del software y en la productividad alcanzada durante su construcción.

La separación entre las etapas de modelado e implementación implica también el mantener la consistencia entre modelos e implementaciones: siendo manual la tarea de conversión entre ambos, ocurre que cualquier cambio en alguna de ellas implica una modificación en su contraparte, lo cual nuevamente deberá ser una tarea manual y, por lo tanto, propensa a errores. Dado que el costo de mantener los modelos actualizados es mayor que el beneficio inicial obtenido por los mismos, usualmente las actualizaciones no son efectuadas al cambiar aspectos en la implementación, lo cual a los torna inservibles como documentación.

Mantener actualizados los modelos y la implementación automáticamente rara vez es posible en su totalidad y el grado en que esto puede llevarse a cabo depende de cuán parecido sean los modelos e implementaciones entre sí. Dada la separación ya mencionada, ocurre que los modelos contienen aspectos demasiado abstractos que no pueden ser inferidos directamente de la implementación, teniendo esta última a su vez definiciones que no pueden ser descriptas en los modelos o pueden representarse con más de una estructura distinta en el mismo. Por otro lado, filtrar y organizar información de la implementación y mostrarla de una manera distinta (por ejemplo, gráficamente) no agrega ninguna semántica a la representación; en todo caso, la hace más amena a la vista.

En la Figura 2 se describen esquemáticamente diferentes maneras de relacionar el modelo con el código fuente y la aplicación final, incluyendo la metodología DSM. Las alternativas más usuales en la actualidad son:

No incluir ningún modelo, lo cual es razonable para software de tamaño reducido.

Se incluyen modelos que luego son descartados, debido a que el costo de mantenerlos actualizados es mayor a los beneficios que aportan.

Se utilizan modelos para visualizar el código mediante alguna técnica de ingeniería inversa, lo cual puede facilitar su comprensión, pero no agrega semántica alguna al mismo.

Ida y vuelta entre modelos y código, intentando actualizar los cambios producidos en cualquiera de los dos en su contraparte. Esto es posible de manera automática sólo cuando los formatos son estructuralmente parecidos o cuando se actualizaran los aspectos comunes entre ambas partes.

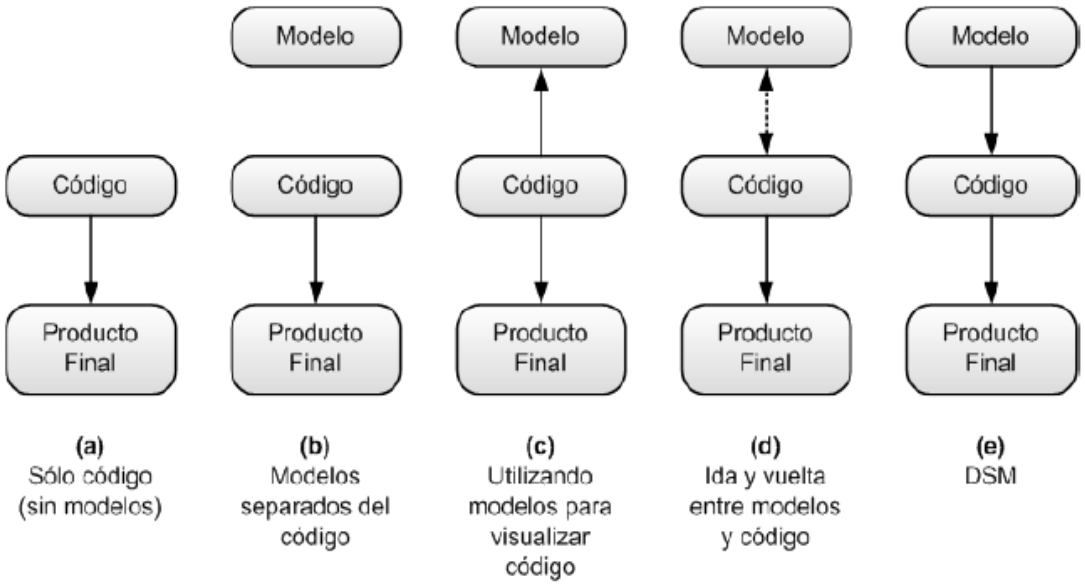

Figura 2. Distintas formas de relacionar el código con los modelos y su comparación con la metodología DSM

La metodología DSM (e), a diferencia de los casos anteriores, tiene a los modelos como la fuente de información fundamental para derivar el código, obteniéndose luego la implementación final a partir de este último. Al generarse el código fuente desde los modelos, los problemas anteriormente comentados a causa de la separación entre las etapas de modelado e implementación no se presentan.

En la actualidad, el hecho de que los ampliamente usados lenguajes de Tercera Generación permitan, a través de compiladores, generar código de bajo nivel confiable el cual no tiene que ser modificado por los desarrolladores, muestra que esta derivación es factible y además conlleva a un incremento considerable en la productividad.

Capitulo 3 - Características

A continuación, se enumerarán y describirán brevemente las características principales que definen a la metodología.

3.1. Dominio acotado

El hecho de que un lenguaje se base en representar soluciones con un conjunto acotado de conceptos, relaciones y reglas provenientes del dominio del problema, permite que especificaciones y dominio estén más próximos. Dada esta proximidad, se dispone de mucha más información acerca de los elementos y relaciones utilizados para describir la solución, lo cual incrementa la semántica de las representaciones de alto nivel concebidas.

Cuanto más acotado sea el dominio del lenguaje, mayor será la semántica que los modelos confeccionados con el mismo contendrán y, por ende, mejores posibilidades habrá de derivar implementaciones funcionales completas desde los mismos. Como contrapartida, a cota que debe ponerse en el dominio del problema a tratar hace difícil utilizar los lenguajes y generadores fuera del contexto de la compañía u organización para la cual se lleva a cabo el desarrollo. Sin embargo, aquellos meta-modelos orientados a expresar soluciones a problemáticas recurrentes en la construcción de software (por ejemplo, especificación de interfaces de usuario para aplicaciones de escritorio) brindan una mayor posibilidad de reuso en diferentes proyectos de desarrollo, estando igualmente acotados a un dominio particular.

La cercanía entre dominio y especificación también permite guiar a los desarrolladores y validar errores en la etapa de modelado, utilizando para esto reglas derivadas del mismo dominio. Estas reglas pueden ser expresadas directamente en el metamodelo, lo cual permitiría controlar su validez en tiempo de modelado. Alternativamente, las mismas pueden ser incluidas como parte de los generadores de código, los cuales corroborarán su cumplimiento en tiempo de derivación.

3.2 Alto nivel de abstracción

DSM incrementa el grado de abstracción en las representaciones de software, al especificar la solución al problema a tratar con conceptos tomados del dominio inherente al mismo. Este incremento de la abstraccion hacia arriba de la abstracción en general, conduce a un aumento de la productividad. Mejora de la productividad no se refiere sólo al tiempo y los recursos necesarios para hacer la especificación en primer lugar, sino también a los trabajos de mantenimiento.

Meta-modelos

Los meta-modelos especifican reglas sintácticas que deberán respetar los modelos, así como también definen aspectos semánticos, los cuales son cercanos al dominio y son representados por los elementos utilizados en los modelos y sus relaciones. En los modelos de dominio específico, la asociación entre conceptos del dominio del problema y conceptos del lenguaje confeccionado es sencilla y usualmente directa. Para representar todos los aspectos concernientes al software, los meta-modelos deben permitir especificar tanto características estáticas y estructurales como dinámicas, basadas estas últimas generalmente en algún modelo de cómputo

subyacente ya existente como, por ejemplo, Diagramas de Transición de Estados.

El lenguaje de modelado puede estar compuesto por dos o más lenguajes separados que consideren distintos aspectos del software a especificar. Los diferentes modelos generados con los distintos lenguajes son integrados al momento de la derivación. La vinculación entre modelos puede llevarse a cabo utilizando en cada uno conceptos definidos en otros modelos relacionados (el meta-modelo en este caso, aunque fragmentado en distintos aspectos, es único) o bien utilizando conceptos a modo de "proxy" o "puente". La fragmentación de distintos aspectos del lenguaje y su integración posterior facilita la modularización y el trabajo concurrente de los desarrolladores.

Los meta-modelos describen formalmente los conceptos que los modelos podrán incluir, sus relaciones, propiedades, jerarquías y reglas de corrección sociadas. Según ha sido probado, como se menciona en [Kelly08], al menos 4 niveles de instanciación son necesarios en el Modelado Específico de Dominio: el nivel más concreto es representado por la aplicación funcional corriente, la cual es una instancia de un modelo particular, el cual es definido a través de un meta-modelo, representando este último una instancia concreta de un meta-meta-modelo. Cada "instanciación" puede verse como la definición de un modelo que utiliza los conceptos y relaciones y respeta las reglas definidas por el modelo superior (su meta-modelo). En la Figura 3, pueden apreciarse gráficamente la relación entre estos 4 niveles de instanciación en meta-modelos y modelos particulares comúnmente utilizados. La filosofía de DSM requiere que los modelos presentes en cada uno de estos niveles sean formales, es decir, que estén definidos a través de algún meta-modelo formal. La existencia de herramientas que permitan el desarrollo de estos meta-modelos de manera sencilla y sin imponer restricciones particulares a algún dominio o área es también un requerimiento de base impuesto por la metodología. Estas herramientas son creadas y/o configuradas por expertos en el dominio y utilizadas por la mayor parte de los desarrolladores.

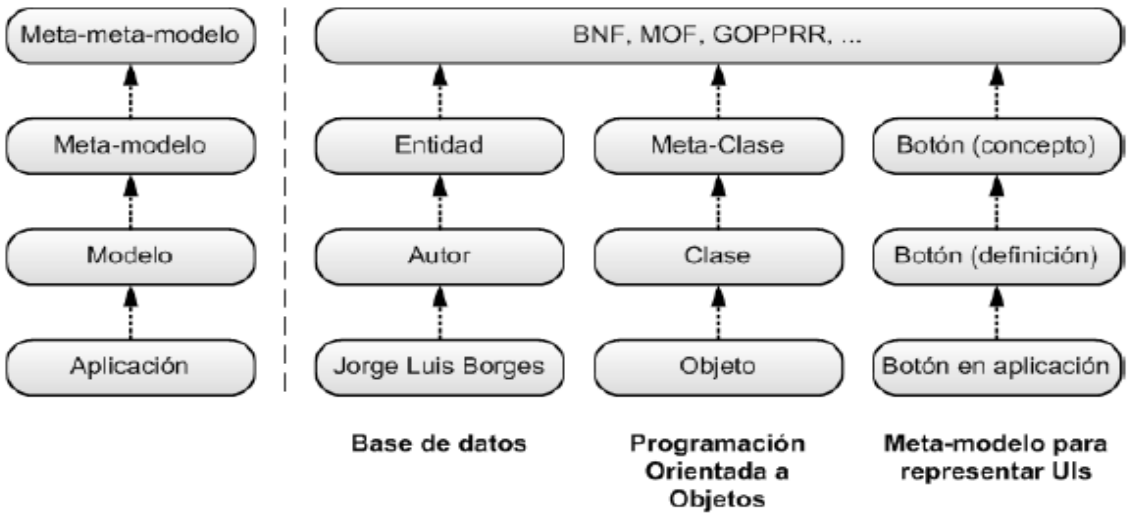

Figura 3. Los 4 niveles de instanciación, ejemplificados en distintas circunstancias. Las flechas punteadas indican una relación del tipo "es instancia de".

3.3 Generación de código

El objetivo de DSM es la obtención de la aplicación final completa a través de la generación de código, siendo la misma posible gracias al alto nivel de semántica que los modelos poseen. Sin embargo, esto no quiere decir que la escritura de código se evada en un ciento por ciento, sino que los desarrolladores deberán indefectiblemente codificar a fin de construir el framework que servirá de soporte al código generado.

La metodología no considera correcto el comprender y modificar el código obtenido desde los generadores, por los motivos ya comentados en relación a las dificultades asociadas a mantener la consistencia entre implementación derivada y modelos de alto nivel. El efectuar estos cambios sería análogo a modificar el código máquina o ensamblador obtenido desde compiladores en los lenguajes de Tercera Generación. Si el dominio es capturado de manera correcta y completa, y tanto generadores como framework de dominio son escritos adecuadamente, cualquier personalización extra puede ser conseguida a través de cambios sobre estos dos últimos componentes. Las modificaciones efectuadas serán plasmadas finalmente en la implementación al ser la misma regenerada desde los modelos que especifican al software.

Los generadores deben considerar tanto la producción de código estático o de estructura así como también de comportamiento, siendo este último el más difícil de derivar. Hacer portable un modelo a diferentes dominios de solución (por ejemplo, plataformas de ejecución) implica la escritura de distintos generadores, uno para cada dominio destino particular, la cual deberá estar a cargo de programadores expertos en el área. Hacer efectiva esta portabilidad usualmente implica la elaboración de un framework que facilite la ejecución y traducción de las soluciones generadas a partir de modelos para cada plataforma destino deseada.

De acuerdo a la metodología, los generadores deberán derivar código de producción, lo cual es posible gracias al ya comentado contenido semántico presente en las especificaciones, en conjunción con la gran experiencia en el dominio requerida por parte de los creadores de la solución DSM, la cual es volcada directamente en generadores y framework de dominio. Estos dos componentes, en conjunto con la plataforma de ejecución, son invisibles a los desarrolladores de igual manera en que lo es el código máquina generado por los compiladores en los 3GLs actuales.

El funcionamiento de los generadores consiste básicamente en acceder y navegar los modelos, extraer información de los mismos (transformándola de ser necesario) y producir una salida correspondiente. Si se utilizaran varios lenguajes (meta-modelos) que consideraran distintos aspectos del dominio en la especificación, la integración entre los diferentes modelos puede ser llevada a cabo en última instancia por los generadores.

Finalmente, los derivadores son capaces de generar no sólo representaciones de menor abstracción como, por ejemplo, código fuente, sino también pueden realizar variadas tareas relativas al desarrollo como creación de build scripts, deploying, invocación de compiladores, generación de archivos de configuración, packaging, toma de métricas, etc. Los chequeos de validez finales en los modelos también pueden ser tareas adicionales que los generadores pueden llevar a cabo. Si bien la validación en tiempo de modelado ahorra muchas de las tareas de prueba de la aplicación final, aún así puede derivarse código de testing si el mismo fuera necesario.

3.4. Representaciones no necesariamente textuales

Bajo la metodología de Modelado Específico de Dominio se consideran una variedad de representaciones posibles además de las textuales como, por ejemplo, diagramas, tablas y matrices, además de las textuales. Las primeras 3 pueden ser igual de formales que las representaciones en texto, dado que establecen conceptos y reglas concretos y no ambiguos para componer especificaciones, siendo usualmente más amenas para construir desde el punto de vista del desarrollador y brindando mayor protección frente a errores semánticos (por ejemplo, utilizando conexiones gráficas para referencias en lugar de hacer las mismas utilizando identificadores). Características de las representaciones gráficas como conexiones entre componentes, por ejemplo, permiten alcanzar especificaciones de mayor riqueza, haciendo que las mismas sean más seguras y llevando a cabo un mejor reuso de componentes. La representación a escoger es aquella que se considere más conveniente para el tipo de especificación que se busca definir.

3.5. Un conjunto mayor de potenciales usuarios

El alto nivel de abstracción que brindan los modelos para componer software amplía el conjunto de usuarios que pueden participar del desarrollo, incluyendo a personas que pueden no tener conocimientos específicos de programación como, por ejemplo ingenieros o directivos. Estos tipos de usuarios pueden utilizar lenguajes de modelado que consideren los elementos fundamentales de su tópico y reglas para relacionarlos y validar su consistencia. La posibilidad de que analistas y diseñadores puedan crear especificaciones completas de software facilita también la interacción entre ambos. En consecuencia, se obtiene una mejor intercomunicación entre los diferentes agentes involucrados en el desarrollo de la aplicación o sistema. Finalmente, los únicos pocos desarrolladores que necesitan conocer aspectos de bajo nivel relativos a generadores, plataforma y frameworks subyacente son los expertos en el dominio a los cuales se les asignan la tarea de crear la solución DSM.

Capitulo 4 - Ventajas de usar DSM

4.1. Introducción

La metodología DSM ofrece diferentes ventajas de acuerdo a cómo descompone y organiza el proceso de desarrollo de las aplicaciones. A continuación, se enumeran las mejoras que la filosofía de Modelado Específico de Dominio ofrece en distintas áreas y tareas del desarrollo.

4.2. Uso de modelos específicos

Dado que los modelos son un artefacto de especificación de software necesario e inevitable en la mayoría de los proyectos de desarrollo, la metodología DSM permite

evitar la necesidad de comprender la semántica particular de otros lenguajes de modelado (por ejemplo, UML, E-R, etc.), y su consecuente requerimiento de asociar conceptos del dominio del problema a conceptos particulares del lenguaje. La proximidad del lenguaje de modelado al dominio del problema hace que esta asociación sea natural, menos ambigua y más directa que en los lenguajes de modelado genéricos. Sin embargo, es pertinente mencionar que el diseñador del lenguaje de modelado tendrá a cargo la tarea de analizar y representar los conceptos del dominio, utilizando constructores del meta-lenguaje elegido para describir el meta-modelo.

Las reglas de validez estipuladas en los modelos provienen del dominio del problema a tratar y permiten detectar errores en la etapa de modelado, donde es más sencillo corregirlos. Estas reglas son independientes de los generadores, por lo que cualquier generador que se utilice recibirá como entrada modelos válidos, lo cual tiene como consecuencia una reducción de la complejidad y lógica necesaria en los mismos. A su vez, el hecho de que los generadores sean más sencillos facilita su confección para distintos propósitos durante el desarrollo como, por ejemplo, prototipado, en pos de alcanzar una mayor productividad. Las reglas gramáticas guían el modelado, no haciendo posible a los desarrolladores la construcción de modelos inválidos. El hecho de que estas reglas provengan directamente del dominio también facilita su comprensión.

4.3. Código fuente

Los generadores de código facilitan la ejecución de tareas rutinarias, de manera que los desarrolladores puedan focalizarse en aspectos más interesantes del desarrollo en lugar de en detalles de implementación. Los encargados de los aspectos de implementación serán los autores de los generadores de código, los cuales serán expertos en el área y trasladarán implícitamente sus conocimientos a toda la implementación final derivada por medio de los generadores.

Un alto nivel de abstracción en las especificaciones se traduce en menos esfuerzo para describir el software, aunque es necesario invertir trabajo extra en el desarrollo de los generadores de código, meta-modelos y frameworks. Este esfuerzo inicial puede ser amortizado considerablemente debido al incremento en la productividad en el desarrollo alcanzado [Kelly08].

4.4. Calidad, evolución y mantenimiento

Debido a que el código fuente es producido por los generadores elaborados por el equipo de algunos pocos expertos en el área, los cuales conocen tanto el dominio del problema como la tecnología en la que se implementará, el código derivado se supone libre de errores y eficiente en el uso de recursos de ejecución y memoria, por lo que no necesita ser modificado u optimizado. Adicionalmente, todo el código generado seguirá un único estilo de programación y diseño.

Es necesario menos testing, al contener los lenguajes definidos para describir el software reglas que realizan verificaciones antes de derivar implementaciones ejecutables. Estos controles aseguran una detección y solución temprana de errores, dado que son efectuadas en la etapa de modelado. Al utilizar lenguajes no textuales, se pueden evitar errores tipográficos, de referencias entre componentes, variables no

declaradas o inicializadas, etc. Los detalles de código fuente son responsabilidad de los pocos expertos involucrados en la construcción de la solución DSM. La documentación, código relativo a testing, debugging, etc., pueden derivarse como artefactos adicionales a partir de los modelos confeccionados y cumplirán con los preceptos de calidad buscados, por la misma razón que el código fuente lo hará.

Dada la cercanía de los modelos con el dominio del problema a resolver, la metodología DSM soporta naturalmente la evolución de la arquitectura subyacente (plataforma de ejecución y framework de dominio) sobre la cual se implementa el sistema, reutilizando en su totalidad o en gran parte los modelos generados. De este modo, el software especificado es atemporal en términos tecnológicos y las representaciones de alto nivel que lo definen siguen siendo válidas mientas el dominio del problema siga siendo el mismo.

La proximidad modelo-dominio también tiene como ventaja el hecho de que los modelos funcionen como artefactos tanto de captura de requerimientos como de especificación de software. En consecuencia, los clientes participan con mayor actividad en el desarrollo, pudiendo inclusive colaborar directamente en la construcción de las representaciones que lo definen. Esto último, en conjunto con el rápido prototipado a partir de modelos que la metodología facilita, contribuye de manera importante a asegurar que el software especificado sea válido en término de requerimientos.

4.5. Productividad

Un mayor grado de abstracción en la especificación del software conlleva a un incremento en la productividad durante su desarrollo. La metodología DSM intenta explotar al límite la misma productividad que ha proporcionado la abstracción presente en los lenguajes de Tercera Generación actuales frente a la programación directa en código máquina, haciendo que las especificaciones de software sean construidas con conceptos del dominio del problema a tratar. Dado el dominio acotado y el consecuente alto grado de semántica contenido en los modelos, se necesita menos cantidad de éstos para especificar software a comparación de aquellos confeccionados con lenguajes de modelado genéricos como UML. Estos lenguajes buscan ser aplicables a un amplio espectro de dominios, lo cual obliga a que la semántica de sus conceptos y reglas sean lo suficientemente débiles como para cubrir una gran cantidad de áreas de aplicación. Esta misma debilidad semántica tiene como resultado modelos que aportan poca información significante para los generadores de código, los cuales se ven limitados en su tarea y requieren que las representaciones de bajo nivel que derivan sean completadas manualmente por desarrolladores. Como resultado, los programadores tienen que interpretar y refinar código fuente que no fue escrito por ellos mismos y, además, deben mantener actualizados los modelos si las modificaciones en las representaciones de bajo nivel de abstracción que efectuaran implicaran cambios en éstos. En [Kelly08] se resumen distintos casos reales en donde fue posible la toma de métricas, los cuales reportan ganancias de entre 300% y 1000% en la productividad utilizando soluciones DSM en dominios específicos correctamente capturados.

A diferencia de otras metodologías, en el Modelado Específico de Dominio, la productividad que puede alcanzarse se incrementa durante el desarrollo, a medida que los desarrolladores se van familiarizando con las herramientas de modelado y los meta modelos. En las etapas finales, los bugs evitados de manera anticipada con las

validaciones llevadas a cabo en modelos y con la generación automática de código, la facilidad para implementar cambios de requerimientos con sólo modificar modelos existentes y la posibilidad de portar el software a distintas plataformas a través de diferentes generadores implican incrementos extra en la productividad en etapas ya avanzadas del desarrollo. Para que este incremento en la abstracción sea factible, es necesario que la definición de la arquitectura DSM (meta-modelos, herramientas y framework de dominio) sea llevada a cabo por expertos en el área, de manera de reducir al mínimo posible la necesidad de cambios estructurales o de gran envergadura durante la especificación.

La agilidad de la metodología, proveniente de la cercanía entre modelos y requerimientos, facilita la adaptabilidad del producto y la especificación de cambios en los requerimientos. El incremento en la productividad implica un tiempo de desarrollo menor, cuyas consecuencias son una reducción de costos y del ciclo de retroalimentación con clientes y una mejora en términos de competitividad, dada la posibilidad de lanzar productos al mercado más rápidamente.

Aspectos relativos a la calidad y mantenimiento del producto (ya descriptos en detalle en la sección "Calidad, evolución y mantenimiento") y a mejoras asociadas a los costos de capacitación necesarios y aprovechamiento del conocimiento de expertos en el área (enunciados en la siguiente sección, "Desarrolladores y experiencia"), tienen un impacto adicional en el incremento de la productividad.

4.6. Desarrolladores y experiencia

En la mayoría de los proyectos de desarrollo de software, la cantidad de desarrolladores no es directamente proporcional a la magnitud de productividad alcanzada debido a diversos factores. Un ejemplo usual de este tipo de factores es la necesidad de capacitación inicial a desarrolladores novatos en las tecnologías utilizadas en la construcción de la implementación y en aspectos internos concernientes a cómo el dominio es representado a través de las mismas. Bajo la filosofía impuesta por DSM, son los expertos en el área quienes definen los meta-modelos y construyen los derivadores que realizarán la asociación entre conceptos de dominio e implementación. Los meta-modelos obligan a los desarrolladores (quienes no necesariamente deben poseer experiencia en la plataforma o dominio) a seguir preceptos de diseño como guía en la construcción del software, y su semántica resulta más fácil de comprender que aquella presente lenguajes de modelado no específicos como UML. Los derivadores, por otro lado, generarán código eficiente y correcto que los mismos expertos especificarán. Por esto último, se tiene una mayor cantidad de influencia implícita de los expertos en el producto final en comparación a otras metodologías clásicas que requieren comunicación directa entre integrantes experimentados del equipo y desarrolladores en entrenamiento. El Modelado Específico de Dominio impulsa también una inserción y capacitación más rápida de los integrantes del equipo de desarrollo, dado que los desarrolladores tienen que involucrarse sólo con aspectos del dominio del problema a resolver y, además, son asistidos en sus tareas por las herramientas de modelado creadas por los autores de la solución DSM.

Al existir los modelos formales como medio de comunicación, la misma se ve mejorada y facilita la interacción entre diferentes agentes involucrados en la confección de la aplicación o sistema, lo cual también resulta en un incremento extra en la productividad.

4.7. Aspectos económicos

La decisión de utilizar la metodología DSM dentro de una organización involucra tanto aspectos económicos como tecnológicos. Las alternativas más usuales son el crear una solución DSM desde cero, o bien utilizar una ya existente y pública, adaptándola a las necesidades que se requieran si fuera necesario y posible.

A pesar de que las soluciones DSM son generalmente un capital importante de la organización que la concibe y, por lo tanto, no se busca que la misma adquiera carácter público, existen casos en los cuales su difusión es de utilidad o importancia. Algunos de estos casos son el facilitar la comunicación e intercambio de información entre diferentes sectores de la organización o con subsidiarias, o el simplificar el desarrollo de software bajo plataformas o HW particulares que la misma empresa produce. El uso de soluciones DSM existentes, aunque implica ahorro de costos en la creación de una solución "in-house", puede no llegar a brindar la flexibilidad necesaria que se requiere.

El introducir la metodología en una empresa u organización representa una inversión, dado que implica un costo inicial sin obtención de ganancia alguna en primera instancia, obteniéndose esta tiempo después a través de las ya mencionadas ventajas alcanzadas en términos de productividad, calidad del producto obtenido, comunicación, etc. El costo inicial a afrontarse es representado por la confección de los componentes de la arquitectura de DSM (meta-modelos, herramientas, generadores, framework de dominio, etc.).

La viabilidad de la introducción de una solución DSM en una empresa aumenta proporcionalmente a la cantidad de "repetición" que se tenga en los desarrollos de software llevados a cabo dentro de la misma, ya sea en términos de distintos productos con características en común o bien en versiones, configuraciones o funcionalidades similares de un producto en particular. Esta viabilidad proviene del hecho de que, mientras más tareas similares haya dentro del mismo dominio considerado por la solución DSM, más se aprovecharán sus ventajas y, por ende, más ganancias resultarán de la inversión inicial efectuada.

Dadas las mejoras que se pueden obtener en términos de comunicación y capacitación en el equipo de desarrollo, la existencia de gran cantidad de desarrolladores es también un incentivo extra para introducir la metodología. En estos casos, la distribución de trabajo puede ser distinta en comparación a metodologías clásicas, debido a que los desarrolladores no necesitan conocer aspectos técnicos del software, los cuales son debidamente considerados por los creadores de la solución DSM. La introducción de la solución puede ser incremental, intercalando la misma con otras metodologías asociadas a la programación manual, a fin de no incurrir en pérdidas relativas a la espera de que las herramientas adecuadas estén disponibles.

Según se menciona en [Kelly08], es usual que ocurran cambios en la misma solución DSM a medida que el desarrollo avanza. Sin embargo, los mismos son introducidos con mayor facilidad y productividad que en otras variantes de desarrollo, dado que involucran esfuerzo por parte de unos pocos desarrolladores (los expertos en el dominio), mientras que el resto puede continuar con sus tareas. Adicionalmente, dado que el mantenimiento es efectuado sobre la solución DSM, el mismo no depende de la cantidad de desarrolladores ni del tamaño del producto. No obstante, es pertinente aclarar que cambios de envergadura en el meta-modelo pueden dejar obsoletos gran parte de los modelos confeccionados hasta el momento, los cuales

deberán ser ajustados o redefinidos adecuadamente. Por esta razón, la metodología considera indispensable que la definición de los meta-modelos sea llevada a cabo por expertos en el dominio, de manera de asegurar la menor cantidad posible de cambios en los mismos a futuro.

Finalmente, si no se dispone de expertos en el área dentro de la organización, una opción viable es el contratar a un equipo de consultores que hagan un análisis del dominio y confeccionen la solución DSM adecuada. Sin embargo, por lo ya dicho, es preferible que los creadores de dicha solución sean especialistas en el dominio a tratar.

Capitulo 5 - Diferenciación con otras metodologías centradas en modelos

Existen en la actualidad una gran cantidad de lenguajes de modelado de propósito general los cuales han sido estandarizados y han logrado amplia difusión, siendo el más conocido UML. Lamentablemente, ninguno de estos lenguajes fueron concebidos para la generación automática de código o representaciones de más bajo nivel a partir de modelos, dado que no responden a un incremento suficiente en el grado de abstracción de las especificaciones que permiten obtener [Kelly08]. La causa de este bajo nivel de abstracción se debe a que sus conceptos y relaciones son demasiado generales, lo cual hace que su semántica sea débil.

El lenguaje de modelado UML surge como una manera de acordar conceptos de modelado, su notación y símbolos utilizados, por lo cual nunca estuvo en sus planes la generación automática de código ejecutable. Los conceptos como clases, métodos, atributos, etc., son tomados del dominio del código fuente, no del dominio del problema a resolver. Esto deriva en el hecho de que, aún usando por completo y adecuadamente los conceptos que UML provee, solo una porción muy pequeña del código puede ser generada a partir de los modelos confeccionados en este lenguaje. Existen también casos en los cuales se ha podido alcanzar una mejor generación de código a partir de UML, pero en estos la estructura del lenguaje (meta-modelo) y el significado de sus conceptos han tenido que ser modificados [Kelly08].

La especificación de lenguajes de modelado ejecutables también ha sido considerado desde la década pasada. En este tipo de meta-meta-modelos, se utiliza usualmente un lenguaje textual en conjunto con modelos específicos para describir restricciones, cambios de estado o acciones, incluyéndose en algunos casos el uso directo de un lenguaje de programación. Todo esto obliga a los desarrolladores a comprender no sólo el lenguaje de modelado, sino todos los adicionales, incluyendo alguno extra que permita especificar acciones con mayor grado de refinamiento, si con todo lo anteriormente mencionado no puede representarse lo que se desea. Finalmente, en estos casos la abstracción lograda por los modelos es pobre, debido a que los mismos deben incluir indefectiblemente aspectos menos abstractos y más relacionados con el código que con el dominio del problema de software a tratar.

En el año 2003, la OMG lanza la Arquitectura Dirigida por Modelos (Model-Driven Architecture o MDA), asumiendo directamente que la generación completa de código a partir de UML no es posible. MDA utiliza tres tipos de modelos: modelos independientes del cómputo (CIM), modelos independientes de la plataforma (PIM) y modelos específicos a una plataforma de ejecución (PSM). En su forma más básica, la metodología implica la transformación, posiblemente automática, de uno o más modelos UML a otros modelos UML, generando código a partir del último modelo obtenido en la secuencia de transformaciones (el menos abstracto). Sin embargo, la metodología usualmente implica el hecho de que en cada transformación sea necesario

la extensión de los modelos obtenidos por parte de los desarrolladores, lo cual atenta claramente contra la generación de código (o modelos intermedios) total y sus problemáticas asociadas. Como solución a este problema, se plantea el uso de un mismo lenguaje en todos los niveles de abstracción. Esto último corre con la desventaja de forzar el mismo grado de abstracción en cada uno de los distintos niveles, resultando en un decremento de la abstracción general que proporciona el lenguaje de meta-modelado.

Otra opción para definir meta-modelos bajo UML es enriquecer a este último a través de profiles que permitan clasificar con estereotipos y agregar nuevos tipos de atributos a los elementos de modelo, pudiéndose también especificar restricciones a través de algún lenguaje ideado a tal fin como, por ejemplo, OCL. Aunque esta posibilidad de enriquecimiento hace más ameno a UML para definir modelos, no permite la definición de nuevos tipos completamente diferentes al lenguaje. Debido a estas limitaciones, la OMG ha propuesto una nueva manera de generar modelos, a través de la introducción de un lenguaje de meta-modelado denominado MOF (Meta-Object Facility).

Finalmente, se hace notar que la estandarización de meta-modelos es usualmente de poca utilidad e incluso contradictoria con el propósito para el cual los lenguajes son creados. Esto se debe a que los lenguajes de modelado a partir de los cuales se derivan las implementaciones finales están adaptados a un problema de software particular de una organización o empresa. Sin embargo, en el contexto de las ventajas en cuanto a abstracción y facilidad para comunicar conceptos que el Modelado Específico de Dominio brinda, han surgido meta-modelos estándares aplicados a dominios particulares como por ejemplo AUTOSAR en la industria automotriz.

Capitulo 6 - Herramientas de meta-modelado: necesidad y características

Las herramientas de meta-modelado son un elemento fundamental para la metodología DSM, ya que hacen posible la definición formal de los meta-modelos y la automatización de la generación de código. Como requerimiento primario, las mismas deben proveer total libertad a los desarrolladores para definir meta-modelos, sin asunción o restricción alguna, permitiendo la definición de lenguajes que puedan ajustarse por completo a las necesidades del dominio. Para esto, debe brindarse un lenguaje de metamodelado que haga posible el definir componentes y sus relaciones y restricciones de manera totalmente abstracta.

La inclusión de lenguajes que no son extensibles o personalizables, característica presente en la mayor parte de las herramientas CASE existentes, no permiten la flexibilidad que la metodología DSM requiere, dado que no facilita el ajustar el lenguaje al dominio específico. La diferencia entre las herramientas CASE convencionales y las orientadas al meta-modelado se esquematizan en la Figura 3.1.

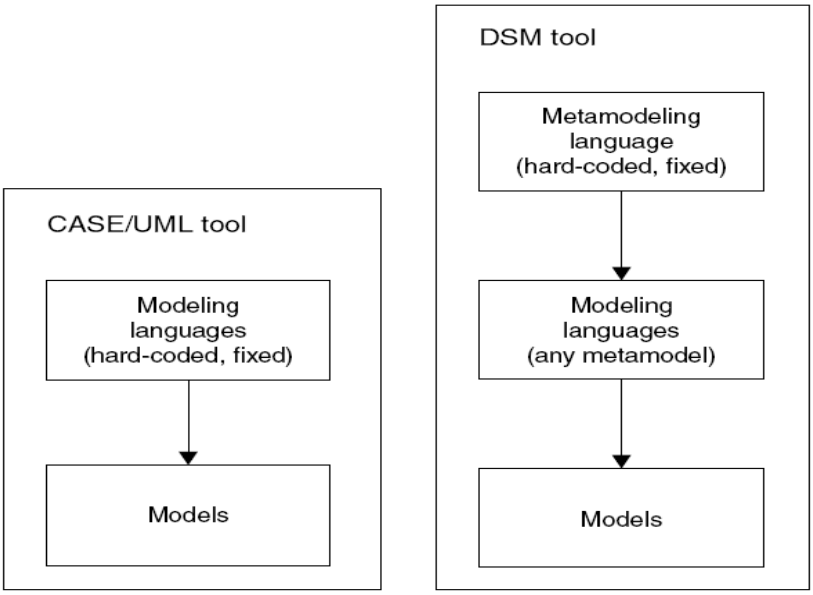

FIGURA 3.1 CASE or UML tool versus DSM tool

Adicionalmente, es deseable que las herramientas de meta-modelado sean seguras, es decir, que protejan la integridad de los modelos siempre que sea posible (por ejemplo, frente a modificaciones en el meta-modelo).

Parte II: La propuesta de Microsoft (DSL Tools)

En esta sección se describirá las Herramientas para creación de un Lenguaje específico de dominio provistas por Microsoft (DSL Tools). Estas herramientas forman parte del Visual Studio SDK y para dar soporte al modelado específico de dominio.

Capitulo 7 Framework de dominio y DSL Gráficos

7.1. Introducción

Como se describió anteriormente arquitectura de la medotologia DSM requiere la creación de tres elementos principales que serán las herramientas fundamentales del equipo que llevara a cabo la contruccion del software. Estos elementos son:

1. Un lenguaje de dominio específico formal a través del cual se definen las soluciones.
2. Un generador de código para el lenguaje anteriormente especificado.
3. Un framework de dominio que sirva como base al código generado en el punto anterior para que la traducción sea más sencilla.

En este capítulo se describen brevente los aspectos de los Frameworks de dominio y los DLS Graficos.

7.2. Framework de dominio

El Modelado Específico de Dominio es una metodología para resolver problemas recurrentes con aspectos simililares y que pueden ser resueltos de una vez y para siempre (Figura 7.1). Los aspectos variables del problema son representados con un lenguaje específico. De esta manera cada ocurrencia particular del problema se resuelve creando un modelo con el lenguaje especìfico e integrando luego el modelo a una solución general fija.

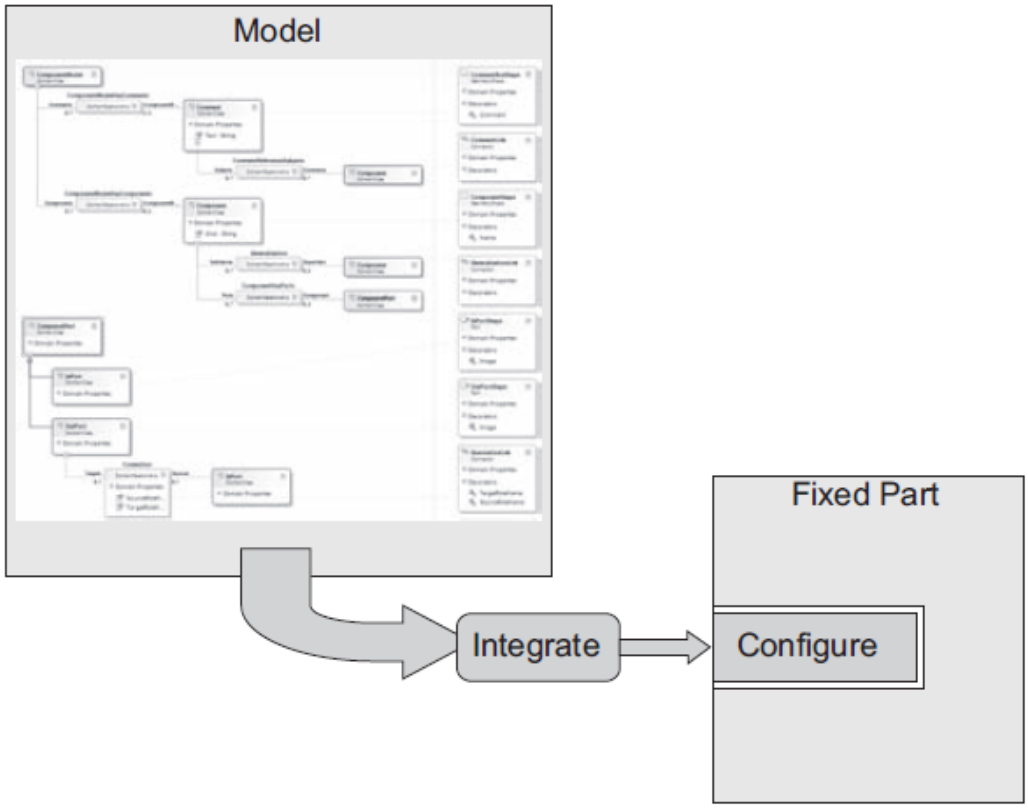

Figura 7-2: Domain-Specific Development

La parte fija de la solución se escribe usando las técnicas tradicionales de diseño, codificación y testing. Esta parte fija podrìa ser por ejemplo un framework. El producto final entonces es el resultado de la integración de la parte fija y la parte variable expresada por el modelo.

Para hacer esta integraciòn existen dos técnicas [Cook07] . El primero es a través de una interpretación, en la cual la parte fija contiene un intérprete del dsl usado para expresar la parte variable. Este método es flexible, pero tiene baja performance y dificulta la tarea de debbugging. En el segundo método la expresión particular o diagrama puede ser completamente transformada a código para ser luego compilado junto con el resto de la solución –método de generación de código-. Este provee las ventajas de buena performance y facilita la tarea de debugging.

7.3 DSLs Gráficos

Los lenguajes específicos pueden ser textuales o gráficos. Lenguajes gráficos tienen ventajas significativas sobre los lenguajes textuales, dado que ellos permiten que la solución pueda ser visualizada directamente como diagramas.

Los DSLs gráficos no solo son diagramas. Estos dsls permiten crear modelos que representan el sistema a construir, junto con una representación graficas de su contenido. Un modelo dado puede estar representado simultáneamente por más de un diagrama, y cada diagrama puede representar a aspecto particular del modelo como se muestra en la figura 7.3.

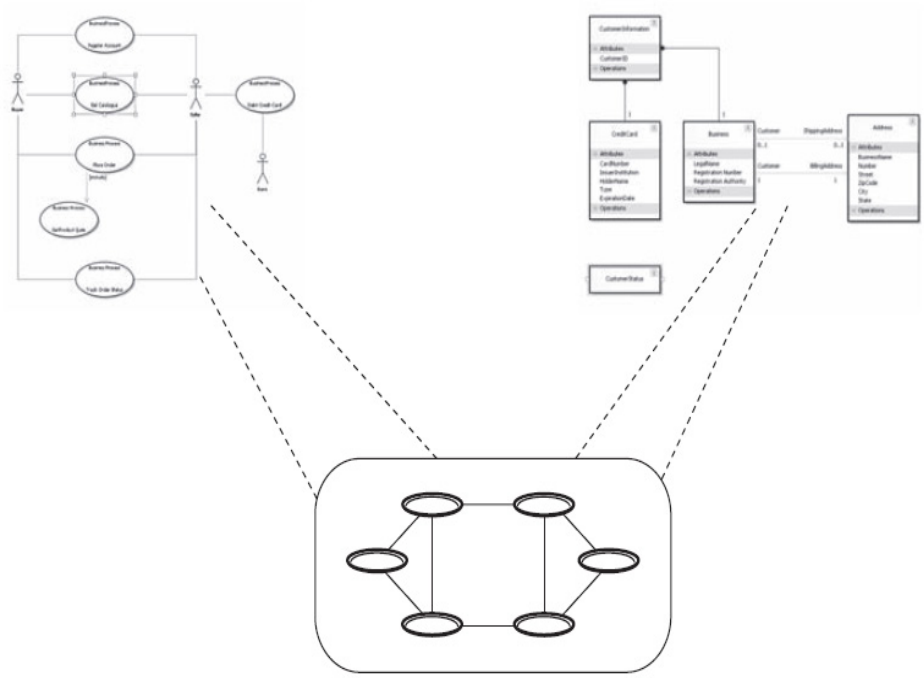

Figura 7.3

Características de los DSLs gráficos

Un DSL gráfico tiene varios aspectos importantes que deben ser definidos. Los más importantes de estos son la notación, el modelo de dominio, la generación de artefactos, la serialización y la integración de la herramienta [Cook07].

- **Notación:** Es el tipo de gráfico que el DSL soporta.
- **Modelo de dominio:** es un modelo de los conceptos descriptos por el lenguaje. EL modelo de dominio para un lenguaje gráfico juega un rol similar en su definición como el jugado por la gramática BNF para un lenguaje textual. Típicamente los conceptos del dominio son mapeados a figuras con el fin de poder representarlos en diagramas. Otro aspecto del modelo de dominio es la definición de restricciones, las cuales pueden ser definidas para verificar que los diagramas creados usando el lenguaje sea válido.
- **Generación de artefactos:** Una vez creado un modelo usando el lenguaje, es deseable poder crear algunos artefactos como código, datos, archivos de configuración u otro diagrama, o más aun alguna combinación de todo esto. Es

deseable poder regenerar todos artefactos eficientemente siempre que se cambie el diagrama.

- **Serialización:** Una vez creado un modelo, es deseable poder guardarlo, controlar estos con una herramientra de control de cambios, y recargarlos luego. La información a serializar incluye los detalles sobre las figuras y conectores en la area de diseño, dónde estos estan posicionados, de que colores son, y que conceptos de dominio representa cada figura.
- **Integracion de la herramienta:** Otro aspecto importa es como el dsl será mostrado en el ambiente de desarrollo en este caso con Visual Studio. Esto involucra aspectos como extensión de los archivos asociados al lenguaje, que iconos aparecerán en el toolbox cuando el diagrama es editado, que sucede si se hace doble clic en una figura o conector, etc.

Capitulo 8. Creando y usando un DSL

8.1 Introducción

El propósito de este capítulo es dar una rápida introducción a los principales aspectos de la definición de un lenguaje DSL usando Microsoft DSL Tools. Este capítulo tiene dos secciones. Primero se ven los pasos prácticos para la creación de un DSL y luego se describen los principales componentes de la arquitectura de DSL Tools.

8.2 Creaciòn de un DSL usando Visual Studio

Visual Studio provee varias plantillas o templates de proyectos y soluciones para la creación de DSLs. Por lo que para crear un DSL es posible usar alguno de estos templates el cual creará una solución (una solución es un conjunto de proyectos). Los detalles como el nombre del dsl, espacio de nombre del código fuente, extensión de los archivos para el DSL son configurados a través de un wizard. Finalmente el wizard crea una solución con dos proyectos dentro.
El código en los proyectos creados define tres cosas principales.

1. El DSL designer — el editor que los usuarios del DSL usarán para crear sus modelos.
2. Generadores de código.
3. Un serializador, que permite guardar los modelos en archivos con la extensión especificada, y permite cargar luego esto estos archivos.

Testeando la solución DSL

La definciòn del DSL se realiza dentro de una solution en el Visual Studio, al ejecutar esta solution, se crea una nueva instancia del Visual Studio. Esta nueva instancia muestra el DSL dentro de un Designer el cual permite usar el DSL. La figura 8.3 muetra un ejemplo del nuevo Visual Studio y el designer que se muestra.

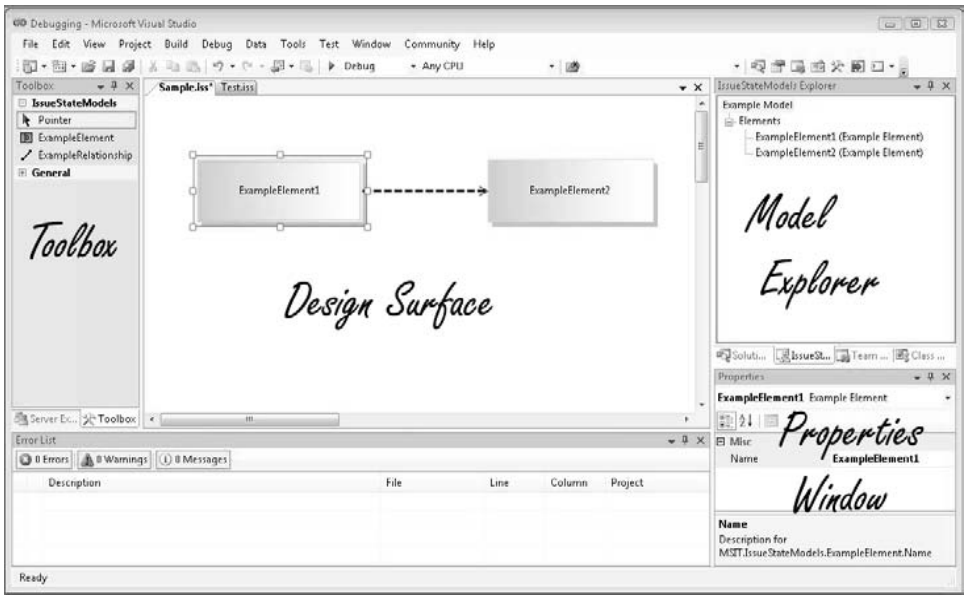

Figura 8-3 muestra las herramientas básicas que los usuarios del dsl podrán usar.

- La ventana más grande es el area llamada Design Surface que es la presentación principal de la instancia del DSL. Una vez terminado el DSL y desplegado, lo usuarios del DSL diseñaran sus diagramas en esta area.
- A la izquierda esta el Toolbox que contiene los elementos del DSL que se puede usar para el modelado.
- A la derecha el Model Explorer. En el se muestran los elementos yla relaciones del modelo en fomra de árbol. En un tab separado bajo el Model Explorer esta el Solution Explorer, el cual muestra el proyecto Debugging, en este se agregan los archivos para la generación de artefactos.
- Debajo esta la la Properties Window. En ella se muestran las propiedades de un elemento cuando este es seleccionado en la Design Surface o en el Model Explorer.

8.3 Arquitectura de DSL Tools

La solution para definir un DSL usando las DSL Tools contiene dos proyectos: el llamado Dsl y el llamado DslPackage. El proyecto Dsl provee el código que define el Designer para el DSL, contiene la definición de como el DSL es serializado y contiene la definición de las transformación para la generación de còdigo. Por otro lado el proyecto DslPackage provee el acoplamiento con Visual Studio para que las instancias del DSL puedan ser abiertas y guardadas como documentos.

Capas de la Arquitectura de DSL Tools

El código generado contiene las particularidades propias de cada dsl. Las características generales y comunes a todos los DSLs estan provistas por un conjunto de librerías o assemblies de las DSl Tools.

Hay tres capas principales en la arquitectura de un DSL [Cook07]: El framework compilado, el código generado desde la definición del DSL, y el código escrito a mano. Las principales partes se muestran en la figura 8.4.

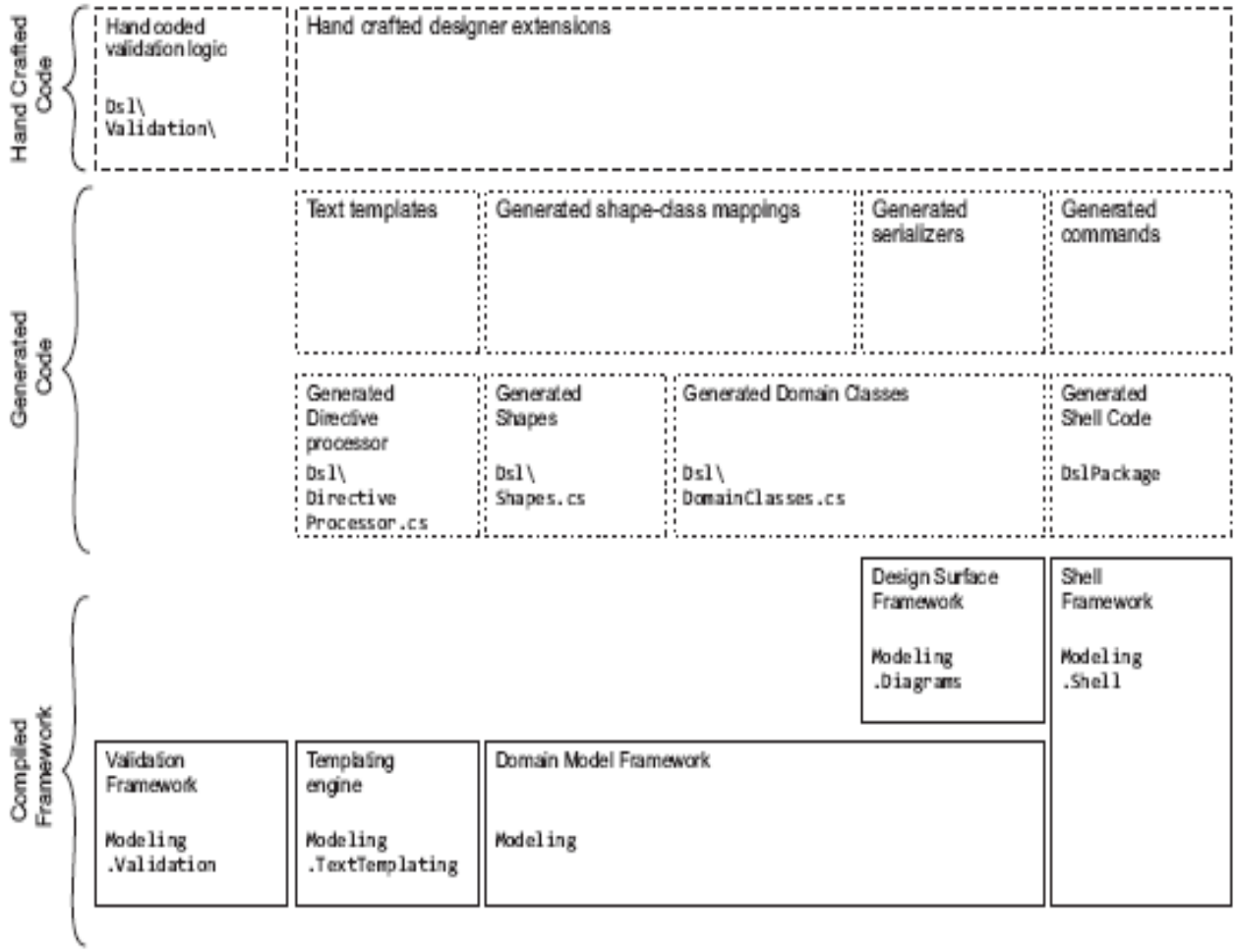

Figura 8.4 Arquitectura de DSL Tools

Los assemblies en DSL Tools

- **Microsoft.VisualStudio.Modeling**—El framework de modelo de dominio es el corazón del sistema, administra elementos y links, instancias de clases de dominio y sus relaciones. Este soporta transacciones, incluyendo undo y redo como propagación de cambies usando reglas y eventos. Por ejemplo, para mantener la pantalla de presentación sincronizada con el modelo interno.
- **Microsoft.VisualStudio.Modeling.Diagrams**—El diseño superficial del framework esta construido sobre el modelo de dominio del framework e interacciona con la notación gráfica manejando elementos visuales tales como diagramas, figuras, conectores, decoradores, etc. Estos elementos también son elementos de dominio, por lo tanto los servicios del framework pueden ser explotados para su manejo.

- **Microsoft.VisualStudio.Modeling.Validation**—El framework de validaciones maneja la ejecución de metodos de validación sobre elementos y links del modelo y crea tambien los objetos de error cuando las validaciones fallan. A su vez mantiene una interface con el shell para postear mensajes en la vista de errores de Visual Studio.
- **Microsoft.VisualStudio.Modeling.TextTemplating**—El motor de templateses usado para ejecutar templates de texto para la generación de código y otros artefactos. Este es un componente independiente que puede ser usado para ejecutar templates que obtienen sus formularios de entrada de otro lado que no sean DSLs.
- **Microsoft.VisualStudio.Modeling.Shell**—El shell maneja el hosting del diseñador en Visual Sturio, en particular, tratando con tools, menues de comandos, y abriendo y cerrando archivos.

Capitulo 9: Definición del modelo de dominio

9.1 Introducción

En este capítulo se verá lo que un modelo de dominio significa en términos de lo generado con las DSL Tools. Cada DSL tiene un modelo de dominio en su core. Éste define los conceptos representados por el lenguaje, sus propiedades, y las relaciones entre estos. El modelo del dominio es como una gramática para el DSL; este define los elementos que forman un modelo, y provee reglas para que lo elementas puedan ser conectados.

El modelo de dominio también provee la base para otros aspectos del lenguaje a construir. Las definiciones de la notación, el toolbox, el explorer, la properties windows, las validaciones, la serialización, y el despliegue son todos aspectos construídas en el modelo de dominio. El modelo de dominio es también usado para generar la API de programación, la cual puede usarse para personalizar y extender el lenguaje, y mediante la cual se accede desde los templates para generar código u otros artefactos textuales.

9.2 Designer del modelo de domino

Para la definición del modelo de dominio se utiliza un DSL Designer. En la Figura 9.1 se puede ver el DSL Designer, en la ventana principal el area titulada Classes and Relationships se modela el dominio. Como ejemplo se construíra un modelo de dominio llamdo IssueState.

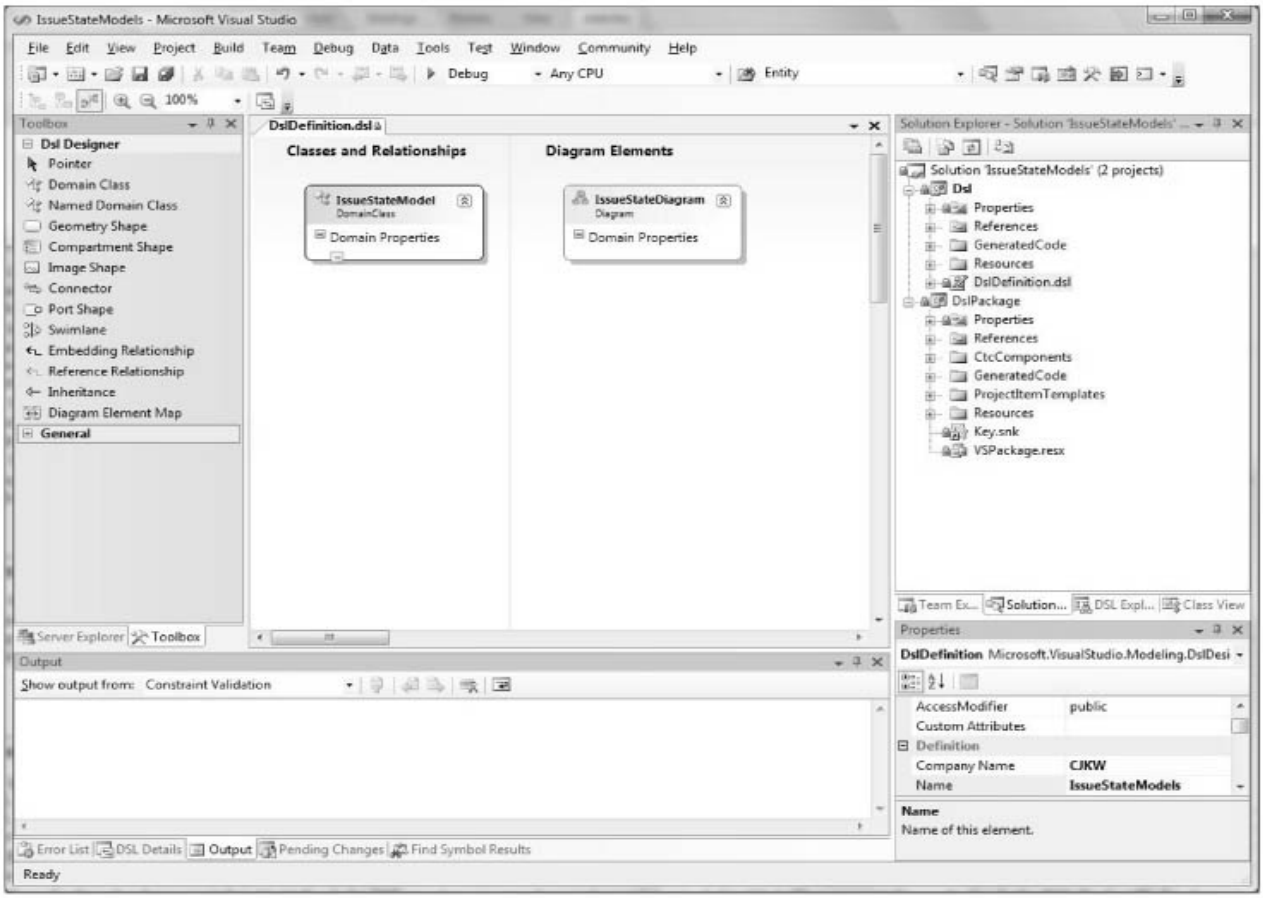

Figura 9.1: Smallest valid domain model

9.3. The In-Memory Store

Antes de continuar con la construcción de un modelo de dominio, es importante comprender lo que pasa dentro de las DSL Tools en tiempo de ejecución.

Dentro del core de una DSL Tool hay un conjunto de APIs llamadas In-Memory Store, las cuales están implementadas por una clase llamada Store.

Este Store da soporte a las operaciones de un DSL: creación, manipulación, borrado de los elementos y enlaces (links); transacciones, undo/redo, reglas, eventos; y acceso al modelo de dominio.

Cuando una herramienta de las DSL Tools, tal como el designer para el lenguaje, es ejecutado, se crea un nuevo store al cual se le especifica los modelos de dominio que consitituyen el DSL.

Las clases abstractas ModelElement y ElementLink dentro de la API de las DSL Tools, proporcionan acceso a toda la funcionalidad para la creacion, manipulación, y elimininación de elementos y enlances del modelo cargado en el Store.

El código generado para el modelo de dominio esta compuesto principalemente de clases que extienden a estas clases abstractas. Con el fin de abreviar, cuando se describen instancias de las clases derivadas de ModelElement, diremos simplemente como elementos del modelo o Mels; de igual manera para referirnos a instancias de clases derivadas de ElementLink diremos links.

9.4. Definición de las Clases de Dominio

Las clases de dominio son creadas en el DSL Designer mediante el arrastre de la herramienta "Domain Class" desde el toolbox y soltando esta en el sector llamado "Classes and Relationships" del area de diseño del DSL Designer.

Siempre que se crea un modelo existirá una clase por defecto llamada "nombreDelDsl**Model",** por ejemplo el IssueStateModel, la cual es la raíz de todas las clases de dominio y puede haber solo una en el modelo de dominio.

Usando el DSL Designer se pueden definir clases que representen conceptos del dominio tal como se muestra en las figuras 9-4:

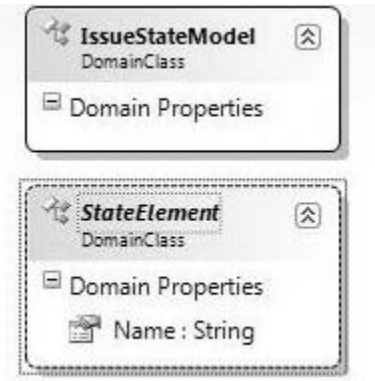

Figura 9-4: Creating the StateElement domain class

Es posible también la definición de jerarquías entre clases, y de esta manera las subclases hereradan las propiedas de la clase base de dominio.

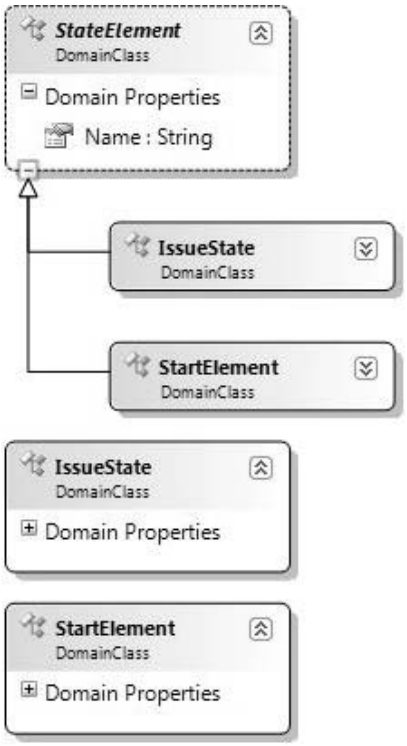

Figura 9-5 Creating an inheritance hierarchy

Cada clase de dominio puede tener cero o una clase de dominio base. Una clase de dominio puede ser marcada como abstract o sealed. Una clase de dominio abstract no podrá ser directamente instanciada, es decir deberá tener otra clase de dominio derivada. Una clase de dominio sealed no podrá tener clases que hereden de ella.

Propiedades de las clases

Es posible agregar propiedades a las clases de dominio. Cuando una clase de dominio es seleccionada sus propiedades son mostradas en la vista de propiedades Propertes View (Figura 9-5). Cada clase de dominio tiene una descripción, la cual será usada para generar comentarios dentro del código y en el esquema XML generado desde el modelo de dominio. En la categoría "Definition" de la vista de propiedades esta la configuración para la clase de la cual la clase de dominio hereda, el nombre de la clase de dominio, y el "Namespace" dentro del cual el código será generado.

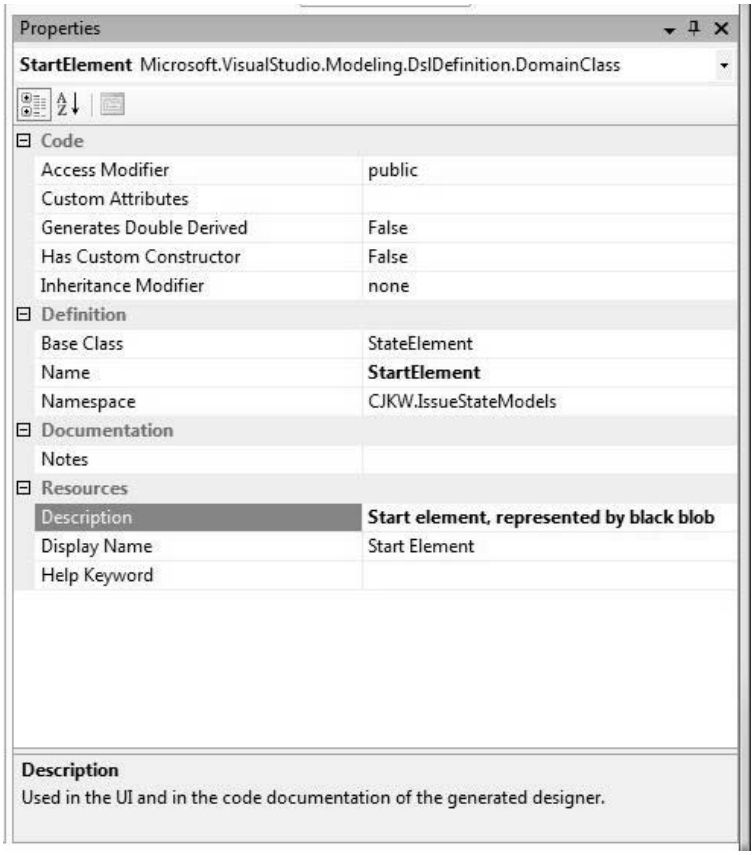

Figura 9-6: Properties window for domain class StartElement

Propiedades de una propiedad de clase

Cuando una propiedad de una clase de dominio es seleccionada sus propiedades son mostradas en la vista de propiedades (Figura 9-7). Cada propiedad tiene un tipo y puede también tener un valor por defecto, el cual será usado para inicializar su valor cuando una nueva instancia de la MEL es creada.

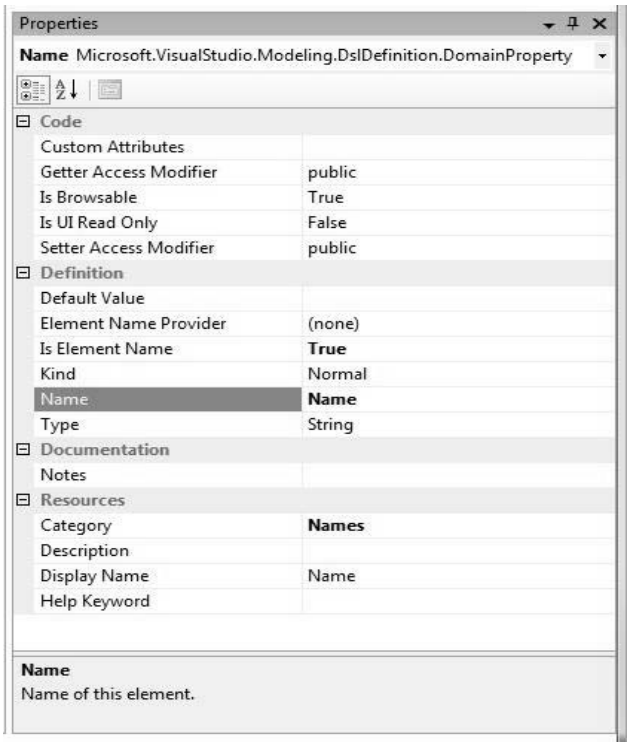

Figura 9-7: Properties window for the Name domain property

9.5 Definición de las relaciones de dominio

Cada relación tiene una dirección, de izquierda a derecha. El extremo izquierda es el origen y el derecho es el destino de la relación. Existen dos tipos de relaciones, llamadas embebidas (embedding) y referencias (references). Las relaciones embebidas están representadas por figuras conectadas mediante líneas solidas (solid), y las referencias están representadas mediante figuras conectadas con líneas punteadas. En la figura 9-8 se puede observar las partes que componen una relación. La relación misma tiene un nombre, CommentsReferToIssueStates en este caso. La linea en cada lado de la relación es llamda rol de dominio o simplemente rol. En este caso la relación es de tipo referencia que es indicado por las líneas punteadas. Cada rol conecta la relación a una clase de dominio, la cual es la que juega dicho rol.

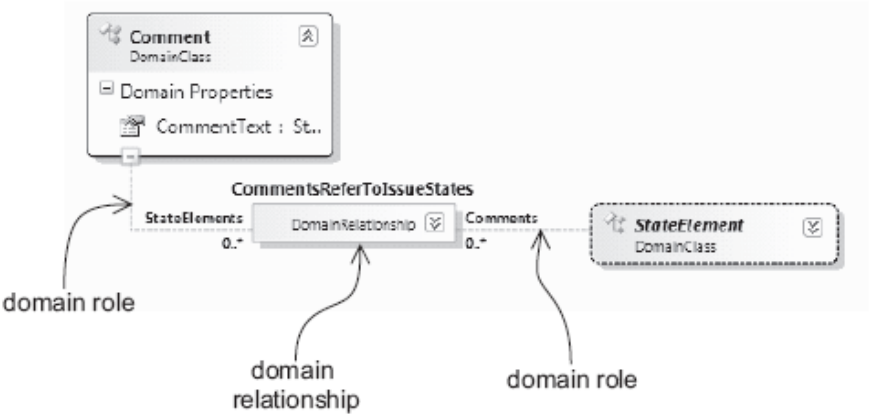

Figura 9-8

Un nombre se muetra seguido a cada rol. Este es el nombre para la propiedad. Los nombres de las propiedades por defecto son generados por el DSL Designer. Es posible cambiarlos para dar un nombre más descriptivo, por ejemplo subjects en vez de StateElements (Figura 9-9).

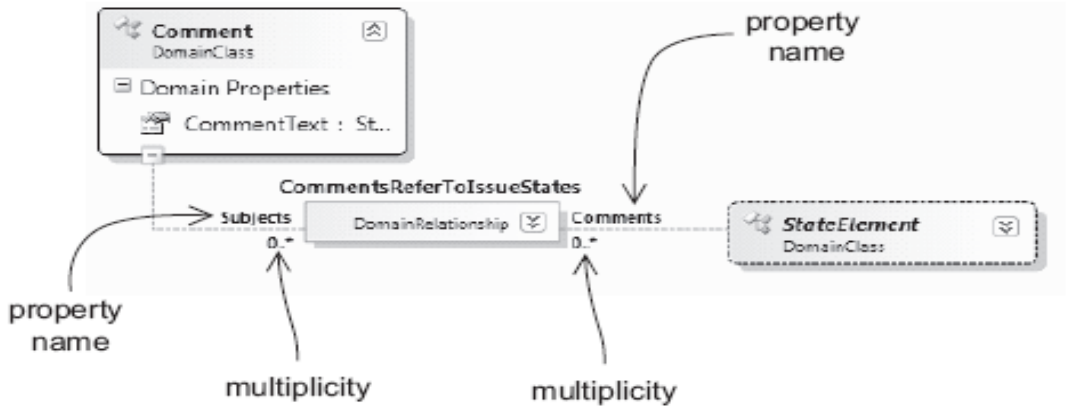

Figura 9-9

La figura Figura 9-9 tambien muestra que cada rol tiene una multiplicidad, que en este caso es cero o muchos (0..*).

Relaciones Embebidas (Embeddings)

Las relaciones embebidas proporcionan una manera de navegar el modelo como un árbol, esto es, como una estructura en la cual cada elemento (excepto para el único elemento en la raíz del árbol) tiene exactamente un elemento padre. Esto es importante por varias razones. En primer lugar porque los modelos son típicamente serializados en archivos XML, y la estructura de un documento XML es intrisicamente un árbol de elementos, empezando desde el elemento raíz. Las relaciones embebidas definidas en el modelo de dominio determinan la estructura de su árbol de elementos XML. En segundo lugar, el designer generado para el DSL tiene un explorador del modelo (model explorer). La estructura embebida determina la organización de este explorer, porque un explorer es también estructurado como un árbol. Y en tercer lugar, una estructura embebida proporciona un manera por defecto para el propagado del comportamiento de borrado y copiado a través de un modelo. Por defecto, el borrado del padre una relación embebida hará que se borren sus hijos.

Una relación embebida implica algunas restricciones especiales:

1. La multiplicidad en el rol destino debe ser uno o cero, porque un MEL puede estar embebido solo una vez.
2. Si una clase de dominio es un destino en mas de una relación embebida, la multiplicidad de todos estos roles destino deben ser cero o uno, porque un MEL de una clase puede solo estar embebido en una de estas relaciones a la vez.
3. En un modelo de dominio completo, cada clase de dominio excepto la raíz debe estar en el destino de al menos un relación embebida, porque sino sería imposible crear el árbol de MELs, y luego el model explorer y la serializacion no funcionarian.

Multiplicidad

La multiplicidad de un rol define, dado un MEL de una clase particular, cuantos enlaces pueden tener ese MEL jugando ese rol. Existen cuatro posibles valores para la multiplicidad.

- **One**: Cada MEL de esta clase (o clase derivada) debe jugar este rol exactamente una vez.
- **ZeroOne**: Un MEL de esta clase (o clase derivada) puede jugar este rol a lo sumo una vez. Es decir, puede estar enlazado a través de la relación con cero o un MEL.
- **ZeroMany**: Un MEL de esta clase (o clase derivada) puede jugar el rol cualquier número de veces.
- **OneMany**: Un MEL de esta clase (o clase derivada) debe jugar el rol al menos una vez.

Relaciones Referencias (References)

Las referencias no tienen restriccines como las relaciones embebidas. En general, los enlaces de estas relaciones pueden ir de cualquier MEL a cualquier otro MEL.

Capitulo 10 Presentación

10.1 Introduccción

En este capitulo se describe como definir el aspecto de la presentación. Hay tres tipos de ventanas en el UI donde se muestra la informaciòn: El area de diseño o Surface Design, el explorador del modelo o Model Explorer y la ventana de propiedades o Properties Window. La definición de la presentación del DSL por consiguiente involucra tres cosas: la definición de la notación gráfica usada en el area de diseño; personalizar la apariencia del explorador; y personalizar la apariencia de la ventana de propiedades.

La parte màs compleja es la definiciòn de la notacion gràfica, por lo que esta es la que se analizarà a continuación màs en detalle.

10.2 Notación Gráfica

Un diseñador gráfico presenta algunos elementos de un modelo sobre un area de diseño por medio de una notación gráfica compuesta de figuras y conectores (shapes y connectors). La figura 10-2-1 muestra un ejemplo. Las figuras en el diagrama estan mapeadas a los elementos en el modelo como muestra la figura 4-2.

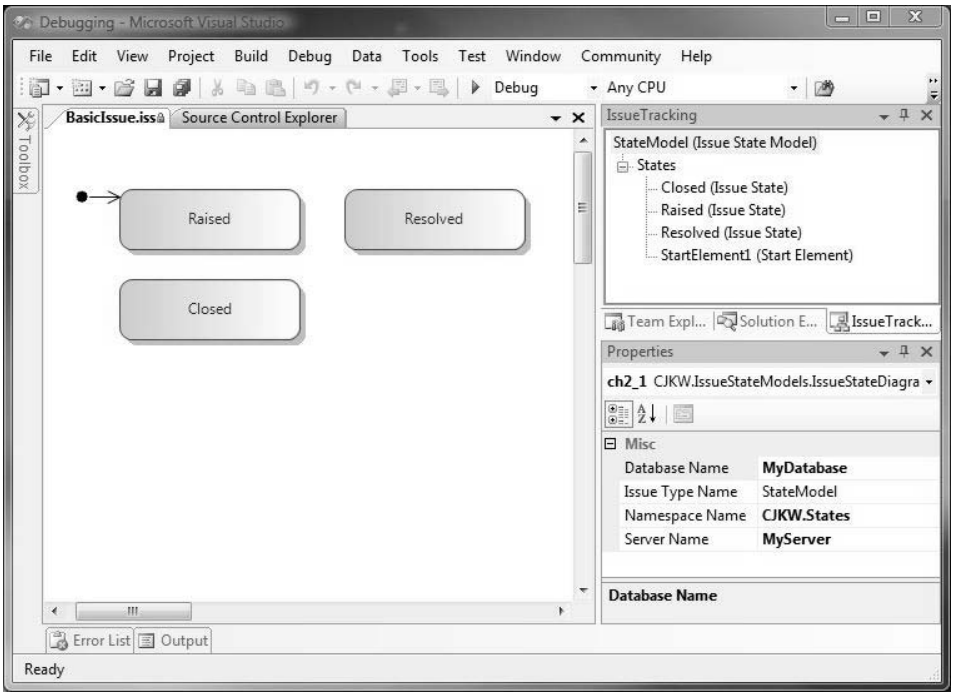

Figura 10-2-1: Designer para el Issue State que muestra la presentación de un modelo

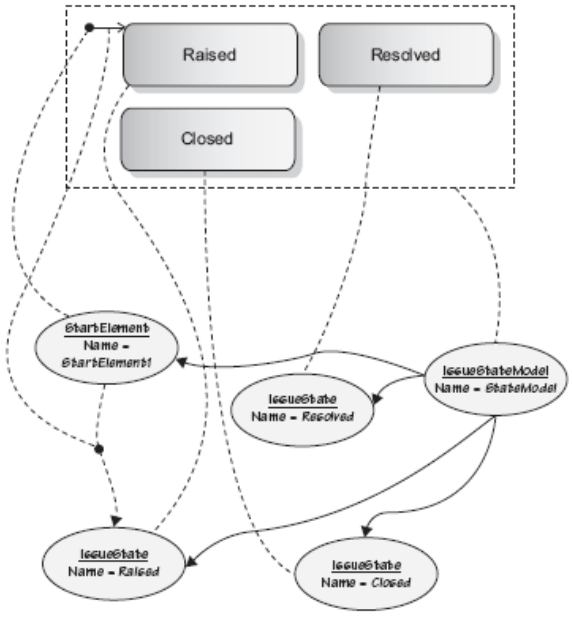

Figura 10-2-2: Maps between diagram elements and model elements

Tanto las figuras como los conectores en la definición del DSL tienen propiedades que permiten personalizar el color y estilo, el tamaño inicial cuando la figura es creada por primera vez, la forma geometríca de la figura, etc. Es posible también definir decoradores de texto e iconos para las figuras y conectores (shapes y connectors). Un

decorador de texto (text decorator) es usado para mostrar texto, usualmente el valor de alguna propiedad de dominio, dentro o adyacente a la figura (shape). Un decorador de icono es usado para mostrar una pequeña imagen la cual puede estar relacionada a una propiedad del dominio.

Para que los elementos del modelo de un DSL puedan ser visualizados en el area de diseño, es necesario definir:

a) el tipo y apariencia de las figuras que serán usadas para presentar estos elementos.
b) un mapeo de la definición de la figura a a la clase de dominio de los elementos, el cual dicta el comportamiento de la colocación de la figura y como la apariencia de los decoradores es afectada por los cambios en los datos.

Para que los enlaces puedan ser visualizados en el area del diagrama, es necesario definir:

a) la apariencia del conector que será usado para representar los enlaces.
b) un mapeo desde la definición del conector a la relación de dominio para los enlaces, las cuales dictan el comportamiento de la conexión y como la apariencia de los decoradores es afectada por el cambio en los datos.

10.3 Diagrama y Editor

La definición de la notación grafica es hecha en el contexto de un diagrama, que a su vez esta relacionado con la definición de un editor.

Diagrama

El diagrama es un contenedor para mapas de figuras y conectores. La figura 4-4 muestra la definición de un diagrama para a un DSL.

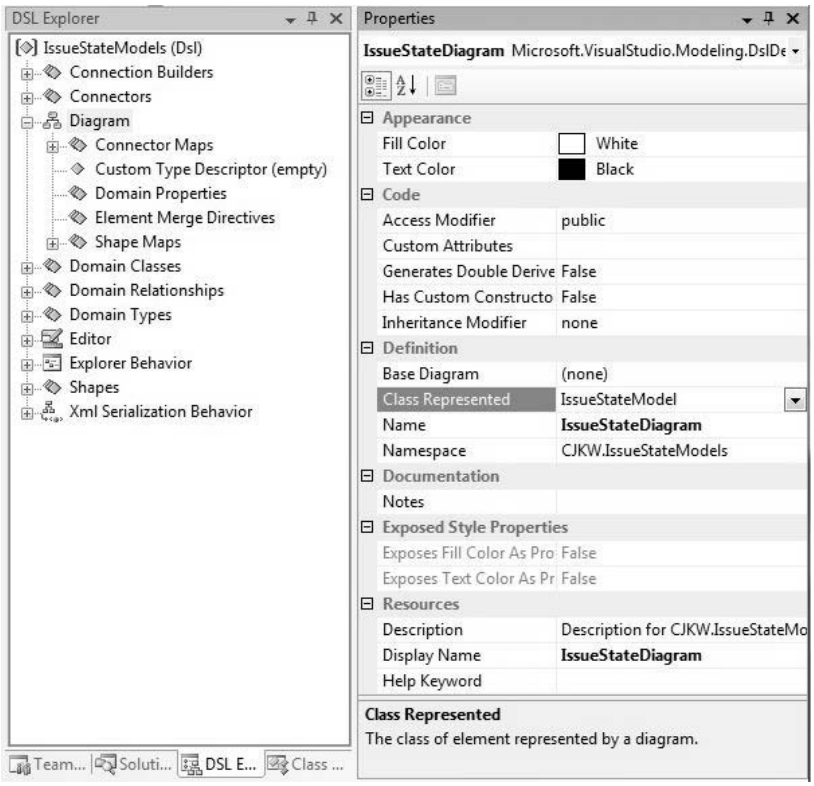

Figura 10-3-1: Definition of diagram for Issue State DSL

Categorias de las propiedades configurables del diagrama

- Appearance: permite configurar la apariencia visual del diagrama, por ejemplo el color de fondo.
- Code: permite hacer modificaciones al código generado para el diagrama.
- Definition**:** Estos settings son similares a aquellos para la clase de dominio (name, namespace, base diagram).
- Documentation: define una única propiedad que contiene notas sobre el diseño del DSL.
- Exposed Style Properties: Estas son todas las propiedades de solo lectura e indican si es posible que el usuario edite un estilo en la categoría appearance.
- Resources: Estas propiedades proveen los recursos de texto que serán usados dentro de la UI del designer destino.

Editor

La definición de un editor aparece debajo del nodo llamado "Editor" que esta dentro del DSL explorer. La información definida aquí es usada para generar la implementación de una clase EditorFactory, el Toolbox, y el comportamiento de las validaciónes. En figura 10-3-2 se puede ver la definición del Editor para el Issue State DSL.

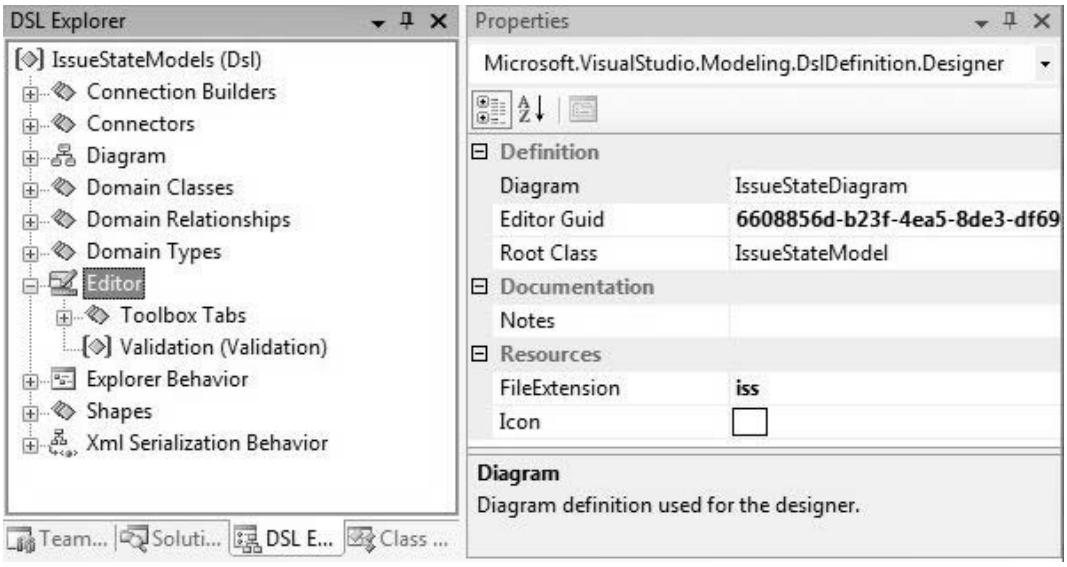

Figura 10-3-2: Definition of designer in Issue State DSL

Las propiedades del editor incluyen una referencia a la Clase Raiz (Root Class) asociada con el Designer. El "Editor Guid" es usado en el código para la generación de la clase EditorFactory. En la categoría Resources, se define la extensión de las archivos usados para almacenar el modelo para el DSL y se puede definir el icono asociado a estos archivos.

Designer

Cuando el editor es un Designer, hay un campo en la categoria Definition de la property Windows, que referencia a la clase del Diagrama que usara el Designer.

10.4 Figuras (Shapes)

Las Figuras son los nodos de una notación gráfica y son usadas para visualizar elementos de un modelo.

Categorias de las propiedades configurables de las figuras

- Appearance: es la configuración de la apariencia visual de la figura, tal como color, espesor de la linea, etc.
- Code: es la configuración del código generado para la figura.
- Definition**:** Estos settings son similiares a aquellos para la clase de dominio (name, namespace, figura geometrica base) con el agregado de la propiedad tooltip.
- Documentation: define una única propiedad que contiene notas sobre el diseño del DSL.
- Exposed Style Properties**:** Estas son todas las propiedades de solo lectura que indican si un estilo en la categoría apperearance, como color, puede ser seteada por el usuario del Designer Final en la vista de propiedades (properties window.)
- Layout**:** estas propiedades modifican el layout y tamaño de la figura.
- Resources**:** Estas propiedades proveen recoursos de texto e imagen, los cuales serán usados dentro de la UI del Desginer destino.

Tipos de Figuras

Hay cinco tipos de figuras diferentes: figuras geométricas, figuras compartimientos, figuras de imágenes, puertos y swimlanes.

Geometry Shapes

La figura 10-4-1 muestra la anotomía de una figura geométrica. Es posible configurarle las propiedades de ancho alto, color, entre otras.

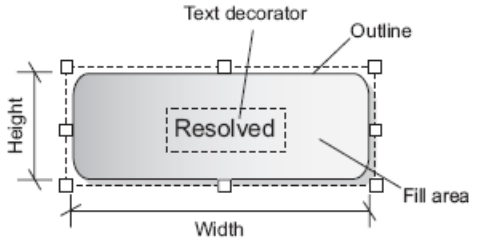

Figura 10-4-1: Anatomy of a geometry shape

Figura Compartimiento (compartment shape)

Una figura compartimiento es una figura geométrica con compartimientos. Un comportimiento es usado para visualizar una lista de elementos que pertenecen a un elemento mapeado a la instancia de la figura compartimiento. La figura 10-4-2 muestra la anotomía de la figura compartimiento.

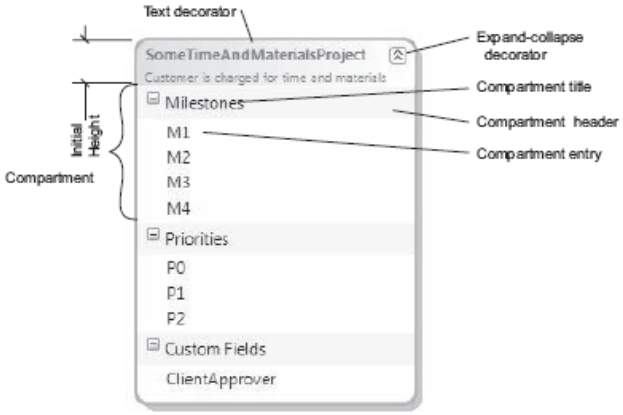

Figura 10-4-2: Anatomy of a compartment shape

Figura imagen (Image Shapes)

Una figura imagen es una figura que muestra una imagen. La figura 10-4-3 muestra la anatomía de una figura imagen.

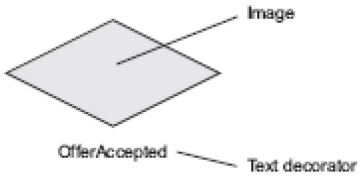

Figura 10-4-3 Anatomy of an image shape

Puertos (Ports)

Un puerto es una figura adjuntada al borde de otra figura y que solo puede ser movido alrededor del borde de esta, como muestra la figura 10-4-4.

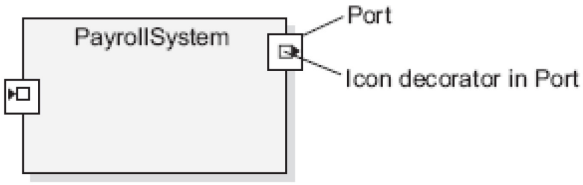

Figura 10-4-4: Geometry shape with ports attached

Swimlanes

Los swimlanes son usados para particionar un diagrama en filas o columnas. La anatomía de un swimlane vertical es mostrado en la figura 10-4-5, y la de un swimlane vertical es mostrada en la Figura 10-4-6

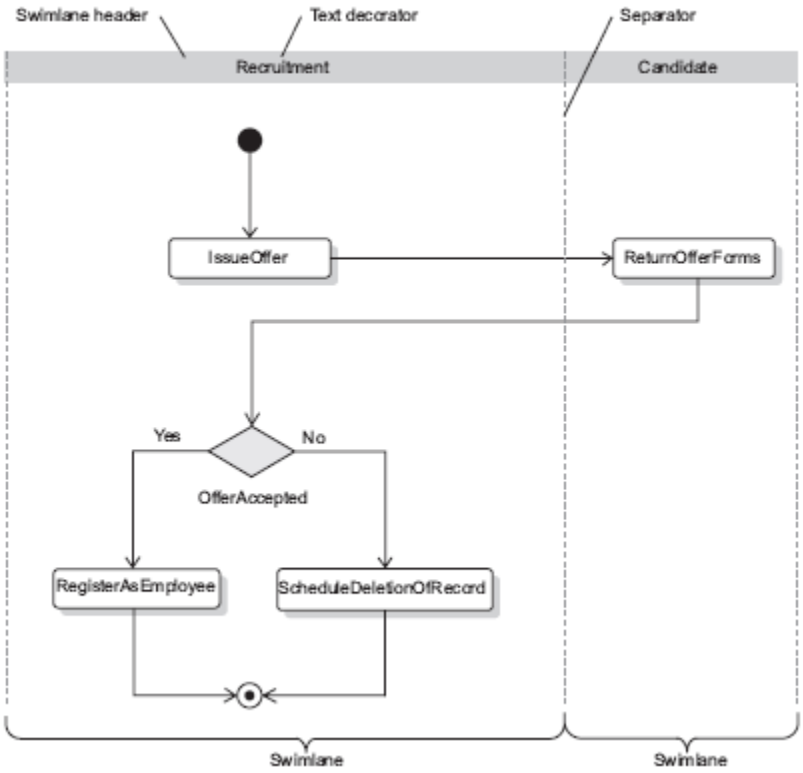

Figura 10-4-5: Anatomy of a vertical swimlane

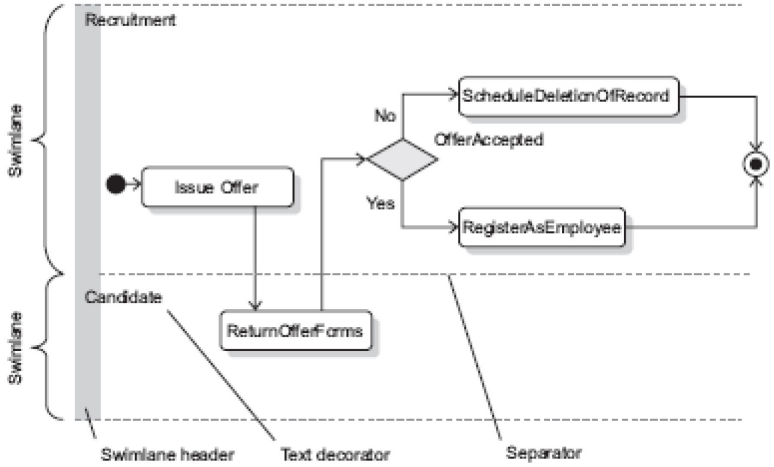

Figura 10-4-6: Anatomy of a horizontal swimlane

Mapeadores de Figuras (Shape Maps)

Los Mapeadores de figuras (Shape maps) son usados para asociar una clase de una figura con la clase de dominio que representa.

10.5 Conectores

De la misma manera que las figuras definen la apariencia de los nodos en una notación gráfica, los conectores definen la apariencia de los enlances.

Apariencia de la anatomia del conector

La anatomía del conector es mostrada en la figura 10-5-1. El conector usado en el ejemplo es el definido en el template del dsl de flujo de tareas (Task Flow DSL Template).

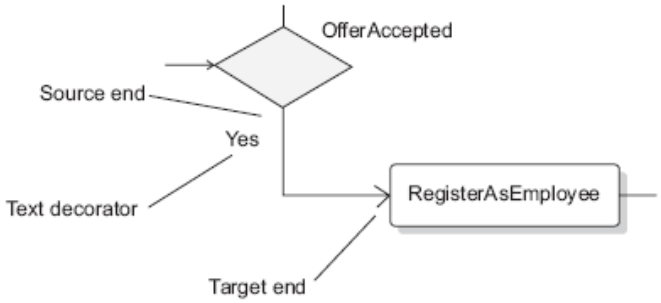

Figura 10-5-1: Anatomy of a connector

Un conector es direccional, tiene una fuente y un destino. Las categorías de las propiedades configurables que un conector tiene son las mismas que las de las figuras.

La configuración de las propiedades impactará en la apariencia de la línea usada para dibujar el conector y permite definir el estilo de los extremos para la fuente y el destino. Por ejemplo la propiedades configurables para el conector ilustrado anteriormente se muestrana en la Figura 10-5-2.

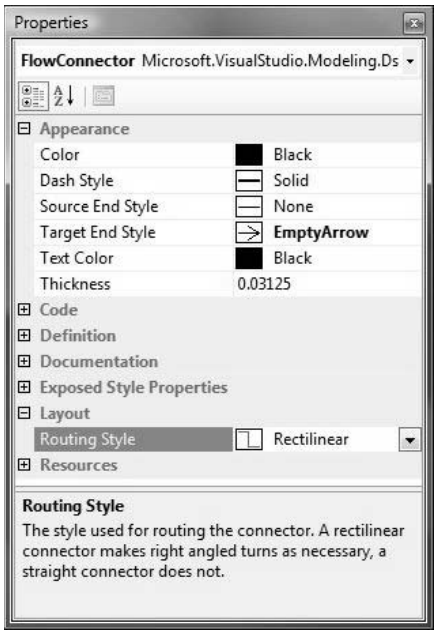

Figura 10-5-2: Connector settings

Connector Maps

Un mapeador de conector asocia un conector a una relación de dominio. La figura 10-5-3 muestra la definición de un mapa de conector para el **FlowConnector**.

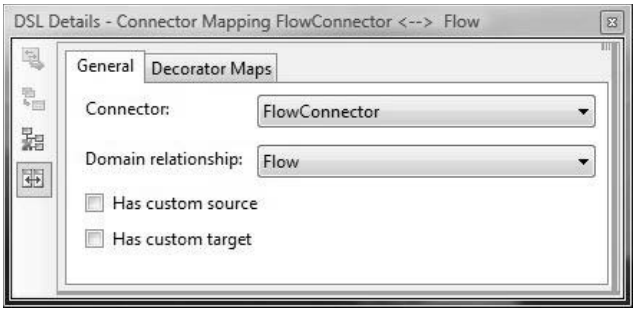

Figura 10-5-3: Definition of a connector map

10.6 Decoradores (Decorators)

Los decoradores adornan a las figuras y conectores con texto o imágenes.

Tipos de Decoradores

Hay tres tipos decoradores: texto, icono, y expand collapse. Un decorador texto es usado para adornar una figura con texto, y un decorador icono es usado para adornar una una figura o conector con una imagen. La figura 10-6-1 muestra la definición de una figura ProjectShape, la cual define un decorador texto y un decorador expand collapse. Tambien se define una figura puerto InPortShape, la cual tiene un decorador icono. Esto define como las instancias de estas figuras serán mostradas cuando se ejecute el designer.

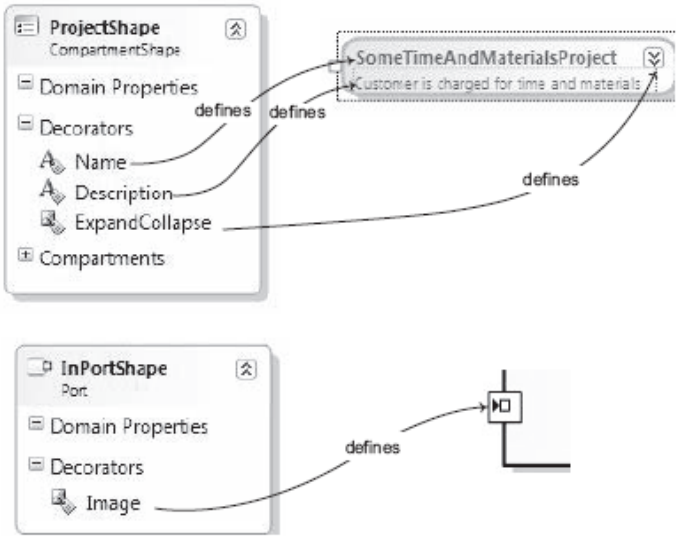

Figura 10-6-1: Different kinds of decorator

Mapeadores de Decoradores

La apariencia de un decorador en una instancia particular de una figura puede ser modificada dinámicamente en función de los cambios que se producen en el modelo. Este comportamiento se define en los mapeadore de decoradores, los cuales son parte de los mapeadores de figuras y los mapeadores de conecteores.

Capitulo 11 Creación, borrado y actualización del comportamiento

11.1 Introducción

Este capitulo se enfoca en como se define el comportamiento para la creación de elementos usando el toolbox y el explorer, la edición de elementos a través de la properties windows, y la eliminación de elementos.

11.2 Creación de Elementos

Cuando un nuevo elemento es creado en el Store, este debe estar unido al árbol embebido, esto es, debe estar enlazado por alguna relación embebida, y debe haber un camino desde el hacia el elemento root. De otra manera no puede estar en el diagrama o ser serializado.

Entonces cuando el usuario arrastra un element desde el toolbox al diagram suceden las siguientes cosas:

- Se crea un nuevo element de la clase determinada por la defincion de la tool.
- Se crea un o varios enlaces entre el nuevo element y los existents.
- Se disparana las reglas que actualizan el diagram para pode mostrar el nuevo element. Estas están determinadas por los mapeadores de figuras o shape maps descriptos anterioremente.

The Toolbox

Las Tools estan definidas en el DSL explorer bajo "Editor\ToolboxTabs\YourTab\Tools." Se debe definer una tool para cada item que que se quiera hacer aparecer en el toolbox (Figura 5-1).

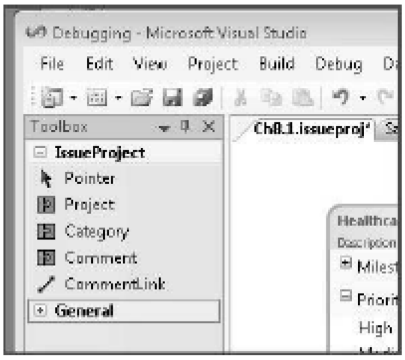

Figura 11.2-1: The toolbox for the Issue Tracking DSL

Hay dos tipos de tools: element tools y connection tools. Cuando se esta creando un modelo un element tool se utiliza arrastrando este desde el toolbox hacia el diagrama, y una connection tools se usa hacienda click en esta y luego arrastrando esta entre dos elementos que son los que se quieren conectar.

Cada element tool esta asociada con una única clase de dominio. A menos que se escriba código custom, la tool crea un único elemento de esta clase cada vez que se arrastre la tool desde el toolbox al diagrama.

Cada connection tools invoca un específico connection builder, el cual dicta que elementos pueden ser creados y el resultado de la creación de estos. Podrìa suponerse por analogía con las element tools, que cada connection tool esta definida para crear instancias de una particular clases de una relación del dominio, pero no es asi. Un connection builder puede ser definido para crear una o varias relaciones, dependiendo de las dos clases que el usuario quiera conectar.

Directivas para mezclar Elementos (Element Merge Directives)

Las directivas para mezclar elementos (EMDs por su sgilas element merge directivas) controlan que relaciones serán construidas cuando un elemento es mezclado dentro de otro. Una mezcla (merge) sucede cuando una de las siguientes cosas ocurre:

- El usuario arrastra un elmento desde el toolbox dentro del areda de diseño (o en alguna de las figuras en el).
- El usuario crea un elemento usando el menú “add” en el explorer del DSL.
- El usuario agrega un ítem en un compartment shape.
- El usuario mueve un ítem desde un swimlane a otro.
- Codigo custom invoca la directiva, por ejemplo, para implementar un operación de paste.

En cada uno de estos casos, hay dos elementos que serán conectados. Aunque puede haber más de un link entre ellos. La función de un EMD es determinar cuales enlaces serán construidos. Un EMD también tiene una función disconnect que es invocada cuando un elemento es movido entre padres, por ejemplo desde un swimlane a otro.

11.3 Connection Builders

Las connections tools en el toolbox del usuario funcionan de manera diferente de las elements tools. Mientras que un element tool crea elementos de una específica clase, una conection tool invoca un específico connection builder. Este controla que elementos el usuario puede seleccionar para unir y puede generar un link de una alguna de las varias de las clases de las relaciones de dominio, dependiendo de los elementos elegidos. Si se escribe custom code para esto, es posible realizar conecciones arbitrarias.

El conection builder es invocado tan pronto como su connection tool es cliqueada por el usuario. El connection builder dicta que elementos pueden ser conectados y exactamente que links serán creados.

The connection builder is invoked as soon as its connection tool is clicked by the user. (And of course you can also invoke a connection builder in custom code.) The connection builder governs what elements may be connected and exactly what links are made.

Cuando se mapea una relacion referencia a un connector en el DSL Designer, este automaticamente crea un connection builder.

11.4 Eliminación de un Elemento

Como el DSL se ejecuta en un Designer, las instancias pueden ser tanto creadas como eliminadas. En un designer, el usuario puede seleccionar una figura o un elemento en el explorer, y eliminarla. Cuando un elemento es eliminado, se ejecutan reglas que aseguran la consistencia del store, eliminando cualquier elemento dependiente al que se elimina.

Se dice que una egla de eliminación es inmutable si cuando al eliminar algún elemento, para cada link con el elemento destino o fuente también es eliminado. Lo que pasa luego depende de las reglas de propagación de los links. Es posible definir Reglas de eliminación extras para cada relación. Hay por defecto un conjunto de reglas, por lo cual no es necesario definirlas siempre, sino solo para caso especiales.

Reglas de propagacion de borrado por defecto

Hay algunas reglas por defecto, aunque es possible cambiarlas por otras.

- Si un link pretence a un relación embebida, entonces su elemento destino también es eliminado. Por lo tanto, eliminar un elemento elimina completamente su subárbol embebido.
- El elemento fuente de una relación embebida no es eliminado por defecto cuando se elimina su hijo.
- La eliminación de una relación referencia nunca afecta a los elementos asociados a la relación (rolplayers).

11.5 Serialización

11.5.1 Introdución

Cuando se define un DSL utilizando DSL Tools, se genera automáticamente un serializador específico de dominio dentro del proyecto que se encargará de guaradr y cargar modelos en formato XML. El usuario de la DSL puede modificar dicho modelo si desea, mediante opciones de customización provistas con tal propósito.

Una de las metas de las DSL Tools es permitir que el usuario entienda y edite directamente los estos archivos, ya que es posible crear un modelo de dominio sin generar un diagrama, usando archivos XML como entradas a templates que generen código y otros artefactos. Otra opción es transformar modelos ya persistidos por otras herramientas o procesos electrónicos usando XSLT.

11.5.2 Formato de archivos de modelo XML

Modelos y diagramas son guardados por defecto como documentos XML. Los archivos XML son una buena alternativa para este propósito ya que>

- Son legibles por humanos
- Es un estándar de la W3C (World Wide Web Consortium)
- Existen muchas librerías y herramientas para su procesamiento
- Su estructura interna se corresponde bien con estructuras embebidas en modelos y diagramas
- Documentos XML pueden ser validados por schemas

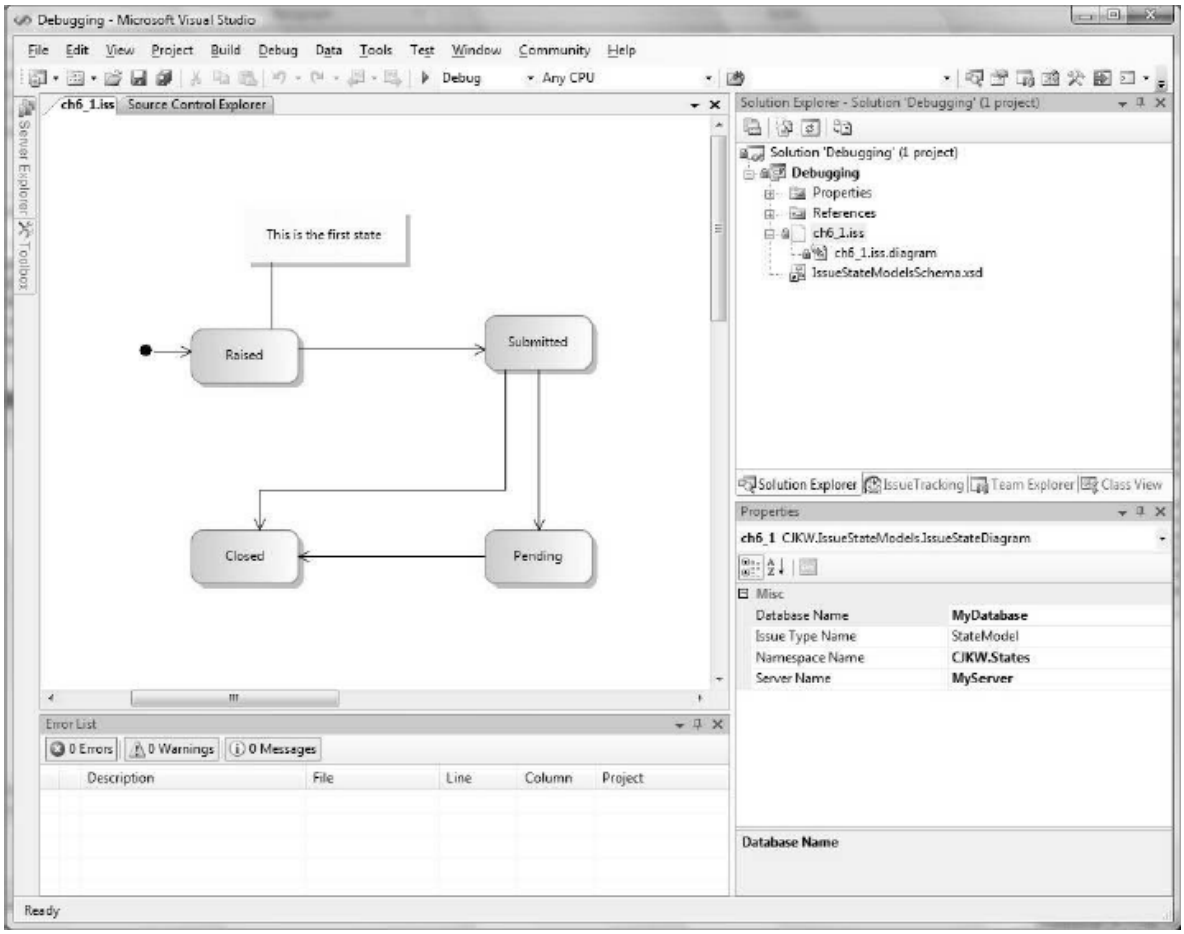

Figura 6-1: Issue State model in a solution

El siguiente es el XML generado para el modelo de la figura 6-1:

```
<?xml version=”1.0” encoding=”utf-8”?>
<issueStateModel dslVersion=”1.0.0.0
   namespaceName=”CJKW.States” issueTypeName=”StateModel”
   serverName=”MyServer” databaseName=”MyDatabase”
   xmlns=”http://schemas.cjkw.com/IssueStateModels”>
        <comments>
        <comment Id=”9f8bc7e9-579c-402a-a9a3-c854767161d1”>
   <commentText>This is the first state</commentText>
   <subjects>
        <issueStateMoniker name=”/CJKW.States/StateModel/Raised” />
        </subjects>
        </comment>
   </comments>
   <states>
        <issueState name=”Raised” icon=”raised.ico”
        description=”The issue has been raised”>
        <successors>
        <issueStateMoniker name=”StateModel/Submitted” />
        </successors>
   </issueState>
   <issueState name=”Submitted” icon=”submitted.ico”
```

```
        description="The issue has been submitted for assessment">
        <successors>
        <issueStateMoniker name="StateModel/Pending" />
        <issueStateMoniker name="StateModel/Closed" />
        </successors>
    </issueState>
    <issueState name="Pending" icon="pending.ico"
        description="The issue is pending resolution">
        <successors>
        <issueStateMoniker name="StateModel/Closed" />
        </successors>
    </issueState>
    <issueState name="Closed" icon="closed.ico"
        description="The issue is closed" />
        <startElement name="Start">
        <startState>
        <issueStateMoniker name="StateModel/Raised" />
        </startState>
        </startElement>
        </states>
</issueStateModel>
```

Al comienzo del document, podemos ver el prologo normal de cualquier CML, incluyento la version y el encoding.

Luego sigue el elemento root del documento, <issueStateModel>. El editor de DSL identifica solo un elemento root, el cual es serializado y mapeado a una clase de dominio dentro del diagrama, en este caso la clase **IssueStateModel**.

El elemento contiene atributos para persistir las propiedades de la clase, como por ejemplo **NamespaceName**, **IssueTypeName**, **ServerName** y **DatabaseName**.

Existen dos elementos internos a <issueStateModel>: <comments> y <states>. Estos corresponden a dos relaciones internas, **IssueStateModelHasComments** e **IssueStateModelHasStates**. <states> Representa una relación embebida mientras que <comments> representa relaciones mediante referencias. Cada elemento referenciado por una relación, debe tener una forma de identificarlo unívocamente de otros de la misma clase, mediante un atributo id.

11.5.3 Relaciones

La representación por defecto de una relación de dominio es un elemento XML que deriva su nombre a partir del PropertyName del source. Esta regla aplica tanto para relaciones embebidas como para referencias.

11.5.4 Formato de diagramas XML

La serialización de diagramas usa la misma técnica que la serialización de modelos. A medida que esta se va construyendo dentro de DSL Tools, también se puede ir generando desde un modelo de dominio de la misma forma. Por tal motivo, el formato del archivo de diagrama es tan simple de entender.

11.5 Versionado

Un issue importante a la hora de trabajar con DSLs es que pasa si existen modelos cuyas definisiones cambian. Los autores de la DSL pueden agregar, eliminar, renombrar propiedades, clases de dominio, relaciones, etc.

Para manejar este tipo de situaciones, los archivos pueden salvarse con un atributo del elemento de más alto nivel, llamado dslVersion.

De esta manera, es posible utilizar diferentes serializadores para el caso de tener diferentes versiones de un modelo, pudiendo referenciar versiones anteriores e ir migrando dicho modelo al mismo tiempo a versiones mas recientes del mismo.

Así, el File Reader puede tomar diferentes decisiones al encontrarse con archivos con versiones incorrectas. Por ejemplo:

- Pueden cambiar los órdenes de las propiedades, elementos, etc.
- Elementos inesperados pueden ser ignorados
- Atributos inesperados pueden ser ignorados
- Elementos y atributos perdidos pueden causar que el modelo sea populado de manera diferente, usando valores por defecto para los mismos

11.6 Schema XML

Un schema XML es un archivo que describe la estructura de documentos XML. La definición de un schema es un estándar de la W3C (World Wide Web Consortium).

Dado un schema, documentos XML pueden ser validados para determinar si su estructura es acorde.

En Visual Studio, si un schema esta disponible, se habilitan opciones del editor tales como autocompletado, chequeo de errores en tiempo real, etc.

Cuando un lenguaje es definido mediante DSL Tools, un schema es generado para el mismo, habilitando las opciones mensaionadas anteriormente para la edición de nuestro lenguaje.

Capitulo 12 Restricciones y Válidaciones

12.1 Introdución

Los lenguajes de modelado tienen una sintaxis la cual debe ser validada [Cook07]. Estos lenguajes generalmente tienen un sistema de tipos asociado y sus restricciones implícitas pueden ser aplicadas; por ejemplo, las propiedades ancho y alto de una clase que representa una Puerta deberán de ser tipo float. Ademàs los lenguajes de modelado pueden tener mecanismos o sublenguajes para expresar explícitamente restricciones definidas por el usuario. Las restricciones definidas por el usuario son

generalmente expresiones que se evaluan contra el modelo y producen un resultado boolean. Ejemplos de restricciones del modelo expresadas en castellando pueden ser:

- La propiedad Altura en un bombero es siempre mayor que 1.80 metros.
- Una casa siempre tiene al menos una casa.

Las restricciones pueden ser típicamente pensadas como invariantes, es decir, ellas seran siempre verdaderas durante toda la vida del modelo, o bien pueden ser pensadas como precondición o postcondicion de alguna operación, ya sea en tiempo de diseño (ejemplo, generación de código, guardado) o en tiempo de ejecución (por ejemplo, el debito en una cuenta bancaria). Las restricciones pueden ser vistas como un mecanismo para que los humanos evaluen el estado actual del modelo con respecto a algún criterio.

Las DSL Tools no proveen soporte explicito para modelar resctricciones de tiempo de ejecución. Sin embargo, es posible que parte de un DSL pueda ser el modelo de un sistema de restricciones y que el código generado por el Designer final complete un framework que soporte restricciones de tiempo de ejecucion para si mismo. Por ejemplo, una tool creada con las Dsl Tools, que permite diseñar interfaces de usuario, debe incorporar notaciòn para definer restricciones sobre los datos que pueden ser ingresados en los campos de texto en la interfaz. La herramienta para diseñar interfaces podría entonces generar código que complete un framework de validación de datos, el cual seria parte de la plataforma en la que se ensambla la interfaz.

Por otro lado Las DSL Tools proveen un framework con todas las características para la creación y evaluacion de restricciones sobre un DSL como parte de la definición del lenguaje.

12.2 Restricciones débilies vs Restricciones fuertes

En el contexto de un diseñador visual o una API, las restricciones pueden ser divididas en dos categorías [Cook07]:

- **Restricciones fuertes** son restricciones que la herramienta impede al usuario violar.
- **Restricciones débiles** son restricciones que el usuario puede vulnerar en algún punto del tiempo pero no en otros, por ejemplo todos los elementos de un modelo tiene un nombre único.

Cuando las restricciones del lenguaje son muy expresivas, calcular el alcance del grafo de dependencias de cada restricción es un problema difícil. Consecuentemente, cuando un modelo cambia, es costoso calcular cuales restricciones re evaluar.

Las DSL Tools aplican restricciones basadas en tipos sobre los valores como una restricción fuerte en el momento que el modelo cambia. Esta decisión es principalmente para asegurar el almacenamiento de datos con un tipo vàlido, lo cual trae gran rendimiento y robustes porque elimina muchas conversiones de tipos y chequeos de binding explícitos. Si una propiedad es declarada de tipo entero y luego se le asigna el valor de tipo String entonces la interfaz de usuario o la API causarà un error inmendiato.

12.3 Restricciones débiles en las DSL Tools

Las restricciones débiles son implementadas en las DSL Tools a través de la escritura de métodos extras en las clases de dominio. Esta decisión de usar simplemente código .Net en lugar de especificar un lenguaje de restricciones viene de la experiencia de los autores con el lenguaje de restricciones Object Constraint Language (OCL), el cual es para UML. OCL es un lenguaje textual capaz de proveer restricciones testeables contra cualquier modelo definido con UML.

Los observación de los autores fue que las habilidades requeridas para escribir de manera efectiva en un lenguaje de restricciones como OCL son muy similares a las requeridas para escribir en C#, además el pool deherramientas de C# es mucho màs amplio que cualquier lenguaje de restricciones actual. Hay muchos pros y contras para esta metodología, pero en cualquier caso, si fuera necesario un sistema de lenguaje de restricciones este puede ser agregado luego.

El hecho de que las restricciones no son contenidas en el modelo significa que ellas no están inmediatamente. Sin embargo, esto también implica que se editan con con toda la potencia y facilidad de un editor de C #.

Una gran ventaja de usar C# es que es muy fácil reusar lógica existente en otras clases del Framework .NET o de un librería propia. Por ejemplo si se desea verificar que una propiedad de tipo string es un identificador C# válido, es posible usar la clase Microsoft.CSharp.CSharpCodeProvider invocando su método IsValidIdentifier().

El siguiente punto a notar de las restricciones débiles en las DSL Tools es que ellas son wrapeadas dentro de un método llamado *validation method*. El concepto de validación expresa la observación de que en un entorno de herramientas, escribir restricciones resuelve la mitad del problema y, en muchos casos, toma mucho menos de la mitad del tiempo empleado. El tiempo restante se destina a la creación y parametrización de advertencias o mensajes de error destinados al usuario final para explicar la restricción en un lenguaje claro con el nivel de abstracción correcto.

Mètodos de Validaciòn

Los mètodos de validación pueden ser añadidos a cualquier subclase de ModelElement de las DSL Tools, incluyendo clases de dominio, relaciones de dominio, figuras y conectores. Estos mètodos son típicamante agregados escribiendo manualmente una nueva clase parcial (partial class) con un conjunto de métodos de validación.

Habilitando la validación

El frameowork de validación de las DSL Tools incluye un sistema que usa reflexion sobre los métodos de las clases del modelo, en busca los métodos de validación a invocar. Este sistema indentifica a los métodos de validacion por medio de la combinación de dos factores:

a. Cada clase de dominio que desea participar en la validación debe tener el atributo ValidationState con el parámetro Enabled.

```
[ValidationState(ValidationState.Enabled)]
partial class MyDomainClass
    {
    …
    }
```

Es posible agregar este atributo usando la propiedad CLRAttributes de una clase en el DSL designer y tener este generado en el código o bien, agregar este en una clase parcial escrita manualmente que contiene el metodo de validación.

b. Cada método de validación debe estar marcado con un atributo custom ValidationMethod.

```
[ValidationMethod(ValidationCategory.Open|ValidationCategory.Save)]
```

En este caso además se puede ver que el método indica que debe ser llamado en el momento en que el designer abre el archivo del modelo y en el momento que guarda este.

Estos dos factores para identificación de métodos son funcionalmente redundantes, porque claramente todas las clases que contienen de validación pueden ser identificadas cuando la primera validación ocurre. Sin embargo, para mejorar la performance, esta marca extra es usada para reducir el conjunto de clases a ser escaneadas.

Las validaciones estan puestas en categorías para determinar cuando serán evaluadas. Los literales del enumerativo ValidationCategory pueden ser combinados usando el operador logico OR con el fin de combinar los momentos en la que la validación es ejecutada.

12.4. Restricciones fuertes en las DSL Tools

Las restricciones fuertes pueden ser añadidas por medio de código custom en el Designer. Como lo que hace que una restriccion fuertes sea invàlida está íntimamente con la experiencia del usuario no hay una única manera standard para definirlas. Los cambios que invalidan una restricción fuerte pueden ser rechazados al nivel de la API disparando una excepción cuando un valor cambia en el modelo. Se verà luego las Rules el cual es un mecanismo para manejar cambios. Otro mecanismo para manejar cambios es sobreescribir el método OnAPropertyChanging() del manejador de valores para una propiedad de dominio dada. Por ejemplo, si la propiedad string Name de una clase de dominio NameElement nuca debe tener un valor nulo, el siguiente còdio es una ejemplo de cómo aplicar esta restricción:

```
/// <summary>
/// Add a hard constraint to NamedElement to prevent its
/// "Name" property from being empty.
/// </summary>
public partial class NamedElement
{
  /// <summary>
  /// Value handler for the NamedElement.Name domain property.
  /// </summary>
  internal sealed partial class NamePropertyHandler : DomainPropertyValueHandler<NamedElement,
  global::System.String>
  {
  protected override void OnValueChanging(NamedElement element, string oldValue,string newValue)
    {
       if (!element.Store.InUndoRedoOrRollback)
         {
           if (string.IsNullOrEmpty(newValue))
            {
              throw new ArgumentOutOfRangeException("Name", "Name cannot be empty or null.");
            }
         }
      base.OnValueChanging(element, oldValue, newValue);
    }
  }
 }
```

Rules

Las Rules en las DSL Tools proveen un método versatil para la implementación de comportamiento que depende de los cambios producidos en el modelo. Por consiguiente, las Rules pueden ser usadas como otro método para implementar un tipo de restricción.

Una Rule puede ser usada para levantar una excepción impidiendo un cambio, o puede ser usada para propagar un cambio a través del modelo, forzando a otras partes del modelo ajustarse al cambio.

Una Rule puede estar asociada con una clase de dominio (incluyendo relaciones y elementos del diagrama). Si una instancia de una clase cambia, la regla se ejecutará. Las reglas pueden configurarse para que se disparen cuando una propiedad de dominio cambia, cuando una instancia es agregada o eliminada, o en otras varias situciones más.

Capitulo 13 Generación de artefactos

13.1 Introducciòn

Históricamente, la generación de artefactos estuvo asociada con la generación de código. Sin embargo, la realidad actual es un mundo donde los sistemas de sofware están compuestos por muchos más grupos diversos de artefactos donde el código fuente es solo una parte. Otros artefactos pueden ser por ejemplo archivos de configuración para la aplicación o middleware, o contenido de una base de datos, tanto el esquema como los datos. En consecuencia, aunque los DSLs pueden ser vistos como una forma más abstracta de código fuente para un compilador que genera código personalizado para una solución, ellos tambien pueden ser usados para configurar

aplicaciones y middleware o tanto como para inicializar y llenar una base de datos. El beneficio de usar DSLs en estas situaciones es variado, un DSL gráfico trae orden y entendimiento visual a los datos que en otro caso necesitarían una interpretación experta [Cook07].

13.2 Estilos de generación de Artefactos

Existen tres técnicas para la genereación de artefactos [Cook07], a continuación se describirán los pros y contras de utilizar cada una de estas.

XSLT (Extensible Stylesheet Language Transformations)

Una técnica es simplemente transformar la representación persistida de un DSL directamente al artefacto deseado. Dado que los DSLs son típicamente persistidos como XML, la herramienta natural para esta tarea de transformación es XSLT.

Un aspecto importante de las DSL Tools es que el mecanismo por defecto de serialización usa archivos de tipo XML, Esto archivos están están especificados de manera natural y directa. Sin embargo, mientras esta especificación de dominio en formatos XML implica que las transformaciones usando XSLT son posibles con las DSL Tools, existen varios temas que hacen que eso no sea lo ideal. La primera cosa a notar es que hay mucho màs código generado que el relacionado solo con la sintaxis de serializacion del DSL; Aun para un pequeño framento de código a generar, el XSLT no es trivial y suele ser muy grande debido a que la generación de artefactos requiere ricas operaciones para la manipulación de strings. Ademàs si por ejemplo el DSL esta exparsido en varios archivos XML, el código XSLT rápidamente se vuelve duro de crear y mantener. Por otro lado una ventaja de usar transformaciónes XSLT es que son extremadamente rápidas de ejecutar.

Usando la API de Dominio Específico

Cuando se crea un lenguaue con las DSL Tools, automáticamente se provee con este una API específica de dominio que permite manipular las instancias del lenguaje en memoria. Esta API puede ser usada para la generación de artefactos puesto que nos permite recorrer el modelo de dominio [Cook07].

Utilizar esta técnica implica en primer lugar cargar el modelo en memoria. Como se mencionó las instancias del modelo viven en una instancia de un Store, por lo tanto es necesario inicializar un Store y luego cargar el modelo. A continuación se muestra un ejemplo de esto:

```
Store store = new Store(typeof(ProjectDefinitionDomainModel));
using (Transaction t = store.TransactionManager.BeginTransaction("Deserialize", true)
 {
   ProjectDefinition def = ProjectDefinitionSerializationHelper.LoadModel( store,
"MyProjectDefinition.pdef", null, null);
   t.Commit();
 }
```

En primer lugar, se inicializa un Store con el tipo del objeto DomainModel que queremos que el Store contenga. Luego se inicia una transacción en la cual se realizará la carga. Esta transacción es creada especificamente para deserializar el archivo del modelo. Finalmente, la clase estática ProjectDefinitionSerializationHelper se usa para

leer el archivo. Para generar código se debe recorrer el modelo cargado usando la referencia llamada "def".

Como esta tecnina usa el código standard .Net, tiene como ventaja que es posible utilizar métodos de una clase Helper por ejemplo para manipular Strings. Sin embargo tiene algunos defectos. El código que va a ser generado esta harcoded en strings. Con lo que para grandes bloques de código, el ruido visual que producen sentencias como Console.WriteLine y especialmente las comillas y paréntesis extras, genera la posibilidad de errores, más aun cuando son necesarios carateres de escape.

Transformaciones basadas en Templates

La forma general para generar código en las técnicas anteriores sigue un procedimiento simple: Se crea primero un ejemplo del código que se quiere generar y luego se crea un algoritmo para generar este. Existe un patrón bien conocido que soporta este tipo de proceso y que hace un buen uso de la API especifica de dominio, este patrón es llamado generación basada en templates parámetrizados. A continuación se muestra un fragmento de un template parametrizado:

```
public IssueId CreateIssue(ProjectMilestone issueMilestone,
ProjectUser creator, ProjectUser initialOwner
<#
foreach (CustomField field in

{
#>,<#=field.Type.FullName#> <#=StringHelper.ToCamelCase(field.Name)#>

<# } #>
)
{
// Some implementation

}
```

La clave de esta técnica es que los elementos fuera de las marcas de control (<# and #>) son escritas directamente en el archivo de salida, y el codigo dentro de estas marcas es evaluado y usado para agregar estructuras y comportamiento dinámico.

Este es la principal tecnología para la generación de artefactos usada por las DSL Tools, llamada también como *text templating*. Una gran ventaja de esta técnica es que puede ser un proceso gradual. Una minima parametrizacion puede usarse para proveer una estrategia de generación general.

La legibiliada de estos templates no es perfecta, pero esta mas cerca a la salida que se desea generar que las técnicas anteriores. La legibilidad de este código puede ser mejorada usando editores que color el texto en los templates de manera de indentificar las estructuras de control y los literales.

Parte III. Sistema de almacenamiento en silobolsas

En esta parte de describe una breve historia del sistema de almacenamiento en silobolsas en la argentina.Y luego se explican detalladamente los principios del este sistema de almacenamiento.

Capitulo 14 Historia de las silbolsas en argentina

La tecnología de almacenamiento en bolsas plásticas fue introducida en el país en el año 1994 y a partir del año 1995 se comenzaron a realizar, por el INTA, los primeros ensayos en la Argentina. Prácticamente, en ese entonces, los antecedentes de investigación en el mundo eran muy escasos, pero de mano del INTA se comenzó a desarrollar una serie de ensayos en ciertos cultivos que dieron una base para que esta tecnología también comenzara a difundirse lentamente en el país. Desde sus comienzos se usó con escasos conocimientos técnicos debido a la poca información a nivel local e internacional que se disponía. En ese entonces se producían pérdidas en cantidad y calidad dada la poca experiencia de los productores que usaban este sistema. Con el transcurrir del tiempo los productores agropecuarios comenzaron a descubrir una serie de ventajas técnico – económicas y la adopción de la tecnología fue creciendo rápidamente y la demanda de investigación y experimentación también fue creciendo. Ante este escenario, en el año 2004, se concreta un Convenio de Vinculación Tecnológica entre el INTA y las principales empresas fabricantes de bolsas plásticas en el país, con el objetivo de fortalecer el desarrollo de esta tecnología mediante la investigación y experimentación aplicada. Esta decisión de las empresas Industrias Plásticas por Extrusión S.A. (IPESA), Plastar San Luis S.A. (Plastar) y Venados Manufacturas Plásticas S.A. (Inplex Venados) fortaleció en forma enérgica la investigación en almacenamiento de granos en bolsas plásticas. Gracias a esta iniciativa, desde el año 2004, se formó una red de investigación y experimentación del INTA a lo largo de todo el país, incluyendo el Chaco (EEA Las Breñas), Salta (EEA Salta), Córdoba (EEA Manfredi), Entre Ríos (EEA C. del Uruguay), Corrientes (UNEECorr) y Buenos Aires (EEA Pergamino y EEA Balcarce), que se comportaron como verdaderos centros de desarrollo tecnológico.

El trabajo de los técnicos del INTA y la concreción de este Convenio de Asistencia Técnica, a través de los años, dieron como producto la determinación de la tecnología adecuada para este sistema de almacenamiento. Se trabajó en granos de los cultivos tradicionales (Maíz, Soja, Trigo, Girasol, Sorgo) y se incorporaron otros como la Cebada, Arroz, Poroto y Algodón. Toda esta experiencia lograda gracias a una fuerte integración entre el sector privado y el INTA, hace que hoy la Argentina tenga los avances más destacados del mundo en almacenamiento de granos en bolsas plásticas y lidera esta tecnología en otros países. Esto fomentó la exportación de máquinas, equipos y bolsas hacia otras partes del mundo y permitió que esta práctica, que se aplicaba solamente a nivel de productor agropecuario, también se aplique en los acopios, puertos y empresas industriales. La confiabilidad lograda en esta tecnología, incentivó el uso de este tipo de almacenamiento, llegando a la última campaña (07/08) a un volumen de granos en bolsas plásticas superior a los 35 millones de t.

Capitulo 15 Descripciòn del sistema de almacenaje en silobolsas

15.1 Introducción

En este capítulo, se describe el sistema de alamacenamientos en silobolsas y una descripcion general del uso de esta tecnología en Argentina. Ademas se describen las ventajas y desventajas del uso de silobolas.

15.2 Tipos de almacenamiento de granos

En general hay dos tipos de almacenamiento de granos: en atmósfera normal y en atmósfera modificada [Cardoso09].

- **Atmósfera normal**: es un almacenamiento donde el aire que rodea a los granos prácticamente tiene la misma composición de gases del aire atmosférico. Es el tipo de almacenamiento más difundido y que incluye a los silos de chapa, celdas de almacenamiento, silos de malla de alambre, galpones, etc. En este tipo instalaciones, debido a que el aire que los rodea es el aire normal que circula por el ambiente, tiene el riesgo que se pueden desarrollar insectos y hongos, para los cuales se necesita que permanentemente se realice un estricto control químico con insecticidas, secado de los granos y/o aireación entre otras tareas.
- **Atmósfera modificada**: consiste básicamente en generar condiciones de hermeticidad tales que se genera una modificación de gases de la atmósfera intergranaria, ocasionando una reducción de la concentración de Oxígeno y un aumento en la concentración de dióxido de Carbono, actuando como controladores de los procesos respiratorios de hongos e insectos. De esta forma se controla su desarrollo y se evita el daño de los granos. Este tema ha sido largamente estudiado en todo el mundo, lleva más de 100 años de análisis y se han encontrado muchas ventajas con respecto al almacenamiento en Atmósfera normal. Efectivamente los granos en ausencia de Oxígeno disminuyen su deterioro y mejora notablemente su conservación. Pero este sistema no se pudo poner en práctica hasta la aparición de las bolsas plásticas. Este sistema de almacenamiento es una tecnología que se ha impuesto no sólo en Argentina sino también en otros países del mundo. La tecnología actualmente empleada, es de origen argentino y se está difundiendo en esos países. Hasta el momento ha demostrado ser un sistema altamente eficiente, seguro y no contaminante de los granos. En el almacenamiento en bolsas plásticas no es usual el uso de insecticidas para controlar insectos y el riesgo de desarrollo de micotoxinas es muy bajo, si se mantiene la bolsa intacta. Esto significa que este sistema de almacenar los granos en bolsas plásticas se presenta como una gran alternativa para productores, acopiadores e industrias.

15.3 Almacenaje hermético

Para que un sistema de almacenaje sea exitoso es necesario que se creen dentro del granel condiciones desfavorables al desarrollo de insectos y hongos, y que además disminuya la propia actividad de los granos. El principio básico del almacenaje

hermético es la eliminación del oxígeno existente en el depósito hasta un nivel que suprima o inactive la capacidad de reproducción y/o desarrollo de insectos plagas y hongos. Los procesos respiratorios de los integrantes bióticos del granel (granos, insectos, hongos, etc.) consumen el oxígeno existente en el ambiente, produciendo dióxido de carbono. Como el almacenaje hermético impide el pasaje de aire y gases entre el interior y el exterior del recipiente, una vez que la atmósfera se modifica, no se vuelven a crear condiciones favorables para el desarrollo de plagas, asegurándose su conservación en el tiempo. La energía que necesitan los seres vivos para crecer y desarrollarse se obtiene del proceso respiratorio y conforma una serie compleja de reacciones químicas iniciadas por enzimas presentes en los propios organismos.

En ausencia de O2 algunos organismos, como levaduras y bacterias, pueden vivir y desarrollarse descomponiendo hidratos de carbono en forma incompleta produciendo ácido láctico, acético y alcoholes [Rodríguez09]. Esta reacción se llama fermentación, libera mucho menos calor que en presencia de aire y se produce en ambientes herméticos con un alto grado de humedad.

15.4 Efecto de la hermeticidad sobre la actividad de los insectos

La actividad respiratoria de los insectos y granos confinados provocan la caída en los niveles de O2 y el aumento de CO2 en el granel confinado en un ambiente hermético. Cuanto mayor es la actividad del granel, más rápido será el consumo de O2 y la generación de CO2. Oxley y Wickenden , citado por Bogliaccini en [Bogliaccini01], estudiaron el consumo de O2 y la generación de CO2 en trigo confinado infectado con 13 y 133 gorgojos (*Sitophilus granarius*) por kg. Ellos encuentran que en el trigo infectado con 13 gorgojos por Kg la producción de CO2 fue en incremento hasta los 20 días, donde se estabilizó en 14%, en tanto que aproximadamente el nivel de O2 disminuyó desde 21% a 2%. En el caso del trigo infectado con 133 gorgojos por Kg el consumo de O2 fue mucho más rápido, disminuyendo a 3% en solo 5 días y a casi 0% en 10 días.

La bibliografía referida al control de insectos con atmósferas modificadas es extensa y ha merecido importantes revisiones [Annis86]. Estos trabajos se basan en la modificación de la atmósfera a través de la adición de gases (N2 o CO2) para eliminar el oxígeno y crear un ambiente desfavorable al desarrollo de insectos y hongos. La literatura establece que concentraciones de CO2 y O2, tiempo de exposición, especie de insecto, estado de desarrollo (huevo-larva-pupa-adulto), temperatura y humedad relativa son los principales factores que influencian la mortalidad de los insectos en los tratamientos de control. Los estudios de control de insectos con atmósferas controladas o modificadas se pueden separar en: atmósferas con baja concentración de O2 y atmósferas enriquecidas con CO2.

Atmósferas con baja concentración de oxígeno: la mayoría de los trabajos se refieren a atmósferas con concentraciones de O2 menores a 1%. Estas atmósferas se logran agregando N2, CO2 o cualquier otro gas. La mayoría de las especies estudiadas mostraron una mortalidad de 95% o mayores durante 10 días de exposición, tanto en atmósferas con 0,1 o 1% de O2 [Rodríguez09].

Atmósferas enriquecidas con CO2: cuando la concentración de O2 es menor a 5% se observa un incremento en la mortalidad. Los datos de eficacia de control de insectos con atmósferas con menos de 20% de CO2 son confusos. No se sabe cual sería el tiempo de exposición requerido para lograr un control total, pero sería superior a los 25 días. En los tratamientos de fumigación con CO2, el producto de la concentración de CO2 y tiempo de exposición (de aquí en adelante llamada ct-producto) es utilizado para representar la dosis [Rodríguez09]. A una determinada temperatura y contenido de humedad, la mortalidad de los insectos es influenciada por la concentración del gas

y el tiempo de exposición. Para realizar un control total de la mayoría de las plagas de granos almacenados en atmósferas enriquecidas con CO2, Bank y Annis [Bank80], recomiendan una relación ct-producto de 12600%h, en tanto que Annis en [Annis86] recomienda elevar la dosis a 16000%h. En teoría esta dosis se podría cumplir con cualquier relación concentración/tiempo, pero la mayoría de los trabajos realizados parten de una dosis mínima de 40% de CO2. Bartosik [Bartosik01], encuentran que para una misma relación ct-producto, aquella conseguida con la menor dosis y mayor tiempo de exposición fue la más efectiva. Esta sería una situación favorable para las bolsas, ya que la concentración de CO2 lograda no sería muy elevada, pero el tiempo de exposición puede ser lo suficientemente prolongado como para realizar un buen control. La literatura demuestra que el control de insectos con CO2 a bajas dosis es igualmente efectivo [White93].

La humedad relativa del granel también tiene efecto sobre la actividad de los insectos. A muy bajas humedades relativas se produce una pérdida de agua a través de la cutícula, lo que causa el desecamiento y el aumento de la mortalidad de los insectos. Aunque existen especies que logran soportar humedades relativas del orden del 10%, la gran mayoría mueren.
La temperatura afecta no solo la actividad de los insectos, sino también la de todo el granel. Los insectos plagas de los granos son un gran problema en climas tropicales o subtropicales, no obstante pueden causar serios problemas en climas templados. El óptimo de desarrollo de los insectos de los granos se encuentra entre 25-30 °C [Yanucci96], pero con temperaturas por encima de 10°C algunas especies pueden causar problemas. La respiración del grano también está influenciada por la temperatura del granel [Rodríguez09].

Cuanto más baja es la temperatura del granel, menor es la actividad biológica en el mismo. A bajas temperaturas disminuye la actividad de los insectos (disminuye el riesgo de infección y el consumo de materia seca) y la de los propios granos, mejorando las condiciones de almacenamiento de los mismos.
El almacenaje en bolsas además de crear un ambiente poco favorable para el desarrollo de insectos en su interior, también reduce notablemente la posibilidad de contaminación del granel. Los insectos pueden infectar los graneles en tres diferentes vías: 1) a campo, 2) instalaciones contaminadas previo al ingreso del grano y 3) infestación posterior de granos ya almacenados. Con el almacenaje en bolsas plásticas la única vía posible de infestación es a campo. Si el grano viene con insectos desde el campo, estos van a ingresar a la bolsa junto con los granos. En cambio, la segunda vía no es factible debido a que las bolsas son descartables por lo que no hay posibilidad que estén contaminadas antes de su uso. Este es un aspecto muy importante porque esta segunda alternativa generalmente es la fuente más importante de contaminación del granel. La tercera vía también es eliminada, ya que la bolsa cerrada herméticamente constituye una barrera que impide la entrada de cualquier tipo de insectos.

15.5 Efecto de la hermeticidad sobre la actividad de los hongos

Los hongos necesitan humedades relativas por encima de 67% promedio para desarrollarse. Esa humedad relativa corresponde a un contenido de humedad de 13.6% en maíz, 13.7% en trigo y 12% en soja a 25°C [Rodríguez09]. Dentro de los daños que causan los hongos posiblemente el más importante es la producción de micotoxinas. No todas las colonias de hongos producen toxinas, debido a que su producción esta influenciada por el substrato, el pH, concentración de O2 y CO2 y estrés hídrico. Sin embargo, a medida que las condiciones de temperatura y humedad

sean las adecuadas, las especies fúngicas que acompañan a los granos almacenados se van a desarrollar, aumentando las posibilidades de producción de toxinas. Baran, encontraron que atmósferas enriquecidas con CO2 estabilizaron el crecimiento de hongos y retardaron la síntesis de micotoxinas en maíz contaminado con Aspergillus [Rodríguez09].

15.6 Almacenamiento de granos en bolsas plásticas

Principios básicos del almacenamiento de granos en bolsas plásticas

El principio básico de las bolsas plásticas, es similar a un almacenamiento hermético, donde se crea una atmósfera automodificada ya que se disminuye la concentración de Oxígeno y aumenta la concentración de anhidrido Carbónico. Esto es el resultado principalmente de la propia respiración de los granos. Esta modificación de la atmósfera interior de las bolsas plásticas crea situaciones muy diferentes de lo que ocurren en un almacenamiento tradicional. Al aumentar la concentración de anhidrido Carbónico se produce un control, en general, sobre los insectos y sobre los hongos. Cabe destacar que los hongos son los principales causantes del calentamiento de los granos cuando se almacenan con tenores de humedad superior a los valores de recibo. También al disminuir el porcentaje de Oxígeno, disminuye el riesgo de deterioro de los granos. Los insectos son los primeros que sufren el exceso de anhidrido Carbónico y falta de Oxígeno, controlándose primeramente los huevos, luego las larvas, los adultos y finalmente las pupas. Éstas últimas comienzan a controlarse con una concentración anhidrido Carbónico mayor al 15% en el aire interior de la bolsa plástica. La condición inicial influye en gran proporción en el comportamiento de los granos durante el almacenamiento. No se recomienda almacenar en este sistema granos húmedos y además que tengan mucho daño climático y/o mecánico.

Elementos que intervienen en el almacenamiento de granos en bolsas plásticas

Los elementos fundamentales que intervienen en esta tecnología son: la bolsa plástica, la máquina embolsadora, la tolva autodescargable y la máquina extractora.

- ***La bolsa plástica*** es un envase de polietileno de baja densidad, aproximadamente de 235 micrones de espesor, conformada por tres capas y fabricada por el proceso de extrusado. La capa exterior, es blanca y tiene aditivos, filtros de UV y (dióxido de Titanio) para reflejar los rayos solares. La del medio, es una capa neutra y la del interior tiene un aditivo (negro humo), que es protector de los rayos ultravioletas y evita la penetración de la luz. Son muy similares a los envases (sachets) que se usan para muchos tipos de alimentos fluidos (leche, jugos, etc.). Son fabricadas con una alta tecnología (máquinas extrusoras). La bolsa es un envase, cuyo tamaño puede ser de hasta 400 t de granos. Se presentan de 5, 6 y 9 10 y 12 pies de diámetro y con una longitud de 60 y 75 m.
- ***La máquina embolsadora*** es un implemento que se utiliza para cargar (depositar) el grano en la bolsa plástica. Consta de una tolva de recepción, un túnel donde se coloca la bolsa y un sistema de frenos, con los cuales se regula el llenado y estiramiento de la bolsa. Se activa por intermedio de la toma de fuerza del tractor, conectada a la embolsadora por intermedio de una barra

cardánica. Estas máquinas pueden embolsar aproximadamente 250 t de granos por hora.

- ***La máquina extractora*** es un implemento que se utiliza para vaciar la bolsa. Consta de una serie de tornillos sinfin, que tienen por misión tomar el grano de la bolsa y transportarlo hasta la tolva autodescargable. Estas máquinas son activadas por la toma de fuerza del tractor, conectada a la extractora por intermedio de una barra cardánica. Su capacidad de extracción es superior a las 150 t/hora, siendo el valor declarado por las fábricas de 180 t/hora.
- ***La tolva autodescargable*** es un carro con una gran tolva que se utiliza para llevar directamente el grano desde la cosechadora a la embolsadora. Esta tolva consta además, de un gran tornillo sinfin que transporta el grano desde este carro a la tolva de la embolsadora.

Capitulo 16 Aspectos económicos

En Argentina, fueron cosechadas 95 millones de toneladas de trigo, maíz, soja y girasol en 2006/07 [Bartosik10]. Al mismo tiempo, el total de la capacidad de almacenaje permanente del país fue estimada entre 65 y 70 millones de toneladas, resultando el déficit de la capacidad de almacenaje cerca de 25 a 30 millones de toneladas [Bartosik10]. Debido a esta insuficiente capacidad de almacenamiento, una parte importante de la producción de grano de la Argentina debe ser directamente entregada a los acopiadores y de allí a los puertos. Otra consecuencia es una insuficiente flota de camiones para transportar toda la cosecha desde el campo hasta los acopiadores y los puertos, aún trabajando a 100% de su capacidad. Aparte de eso, los camiones son usados como almacenamiento temporal en los puertos en largas filas. Como resultado, los productores de grano tienen que pagar precios más altos por transporte y servicio de acopio (secado, limpieza, etc.) durante el tiempo de cosecha que durante el resto del año. Esta ineficiencia en el sistema de poscosecha de grano ha influido en las tareas de cosecha y distribución, con aumentos substanciales en los costos de producción.

Para superar estas circunstancias desfavorables, los productores de granos comenzaron a aumentar la capacidad de almacenaje en campo teniendo la oportunidad de comercializar el grano después de la temporada de cosecha, cuando no sólo el precio es usualmente más alto, sino que también los costos de servicio son menores. Sin embargo, construir una nueva instalación de almacenamiento o ampliar una ya existente por lo general no es accesible para la mayoría de los productores argentinos. Las principales limitaciones son la elevada inversión inicial, alta tasa de interés y préstamos a corto plazo. Bajo estas circunstancias, el almacenaje en silobolsas ha ganado popularidad entre los productores. Esta técnica ha estado en el mercado por varios años para almacenar grano húmedo para alimentar animales (silaje de grano) y posteriormente se fue adaptando para almacenar grano seco.

Cada silobolsa puede contener aproximadamente 200 toneladas de grano, y con la maquinaria desarrollada en los últimos años las tareas de llenado son altamente eficientes.

Las empresas locales también han desarrollado mecanismos para la descarga de la bolsa y transferir el grano directamente a camiones, tolvas, etc. La nueva generación de cosechadoras de alta capacidad de trabajo encuentra en el sistema de bolsas pláticas el socio ideal, ya que la capacidad de carga de la máquina de embolsado es

básicamente limitada por la capacidad de transporte desde la cosechadora al lugar donde la bolsa es confeccionada. Otra ventaja de este sistema es que puede ser fácilmente incorporado en programas de identidad preservada de granos (PI). Las bolsas plásticas pueden ser confeccionadas fácilmente en el campo, reduciendo los riesgos de contaminación con otros granos. Muchos productores han encontrado en esta metodología de almacenamiento la herramienta ideal para separar las diferentes variedades de trigo u otras semillas directamente en el campo.

En el 2001, alrededor de 2 millones de toneladas de grano (maíz, trigo, soja y girasol) fueron almacenados con este sistema. Durante los últimos años, esta técnica de almacenaje se ha perfeccionado, y el sistema de "bolsas plásticas" ha ganado rápida adopción en muchos productores argentinos, a tal punto de que en el año 2007 alrededor de 22-25 millones de toneladas fueron almacenadas bajo esta metodología (cerca del 23 % del total de la producción), y el la campaña 2007-2008 mas de 35 millones de toneladas de grano fueron embolsadas.

Ventajas

En la actualidad, más del 20% de los granos producidos en el país se conservan en bolsas plásticas. Como ventajas del almacenamiento de granos en bolsas plásticas (conservación en atmósfera modificada),al compararlo con el almacenamiento en silos metálicos, celdas, silo-mallas de alambre (con-servación en atmósfera normal), se pueden citar que es un sistema económico y de baja inversión la práctica de embolsar los granos por un período de tres-cuatro meses genera un ahorro a los productores del 20-25% en trigo, 30-35% en maíz y 20-25% en soja dependiendo de la distancia a puerto, sistema de comercialización, etc., el almacenamiento de granos de manera diferenciada separando granos por calidad, variedad, etc., el almacenaje de los granos en el mismo lote de producción, haciendo más ágil la cosecha, cosechar en momentos en que no se puede sacar la producción del campo por falta de caminos, posibilidad de obtener créditos sobre la mercadería guardada, sistema flexible para los acopios que les permite incrementar su capacidad de almacenaje según las necesidades que tengan en un año en particular, la inactivación de desarrollo de hongos e insectos debido al almacenamiento en una atmósfera rica en CO y prácticamente nula en O2

Desventajas

La principal desventaja de este sistema de almacenamiento es la posibilidad de ruptura de la bolsa lo que ocasionaría el ingreso de O2 provocando el desarrollo de hongos, insectos, aumento de humedad y temperatura perjudicial para una buena conservación de los granos. Además, otras desventajas serían la alta superficie expuesta, los riesgos climáticos que pueden causar roturas de las bolsas, y la presencia de roedores, causantes de roturas de bolsa con el consecuente deterioro de la mercadería conservada por presencia de O2 y cambios bruscos de temperatura y humedad.

Capìtulo 17 Estudios realizados con silobolsas

17.1 Ensayos con maíz, trigo, girasol y soja

En las experiencias realizadas con silobolsas se remiten principalmente al INTA donde se comenzó a trabajar en el año 1995-1996 y en el año 2000 en la EEA Balcarce se realizaron una serie de ensayos con maíz, trigo, girasol y soja. Algunos de los resultados más relevantes son [Cardoso09]:

- Con granos secos (valores de humedad de recibo) no existen problemas de conservación, por lo tanto no hay un deterioro causado por el sistema de almacenamiento. Cuando se almacena grano hay una tendencia al deterioro de la calidad de los granos almacenados con este sistema en el tiempo (disminución de poder germinativo, calidad panadera, acidez de la materia grasa, etc.). A mayor humedad inicial, los tiempos de conservación se acortan.
- No hay modificación de la humedad inicial de los granos.
- Cuanto menor es la calidad inicial (daño mecánico, materias extrañas, etc.) del grano almacenado, mayores son los riesgos de pérdida de calidad durante el almacenamiento. En cuanto al daño mecánico, además de la cosecha también se puede producir daño mecánico en la descarga de los sinfines de los carros tolvas por excesiva velocidad (rpm.) de los mismos.
- En los ensayos realizados, la concentración de CO_2 alcanzada dentro de la silobolsa es lo suficientemente alta para lograr un control total de insectos.
- Existe, en general un importante control de hongos y no se detectó producción de micotoxinas en el interior de la bolsa.
- Si no existe daño en la silobolsa, no se produce aumento de temperatura en la misma por generación de calor propio del sistema, aun con granos muy húmedos.
- La variación de temperatura en el interior del silo acompaña la variación de la temperatura ambiente. En ensayos realizados midiendo la evolución de la temperatura en tres estratos de la silobolsa (superior, medio e inferior) se concluyó que las variaciones el estrato superior de la silobolsa responden a las oscilaciones diarias de la temperatura ambiente mientras que el estrato medio e inferior de la misma no presenta oscilaciones.
- La variación de la temperatura en la capa superior de granos produce la migración de humedad desde el interior de la silobolsa hacia la superficie. En algunos casos se observo condensación de humedad en la pared interior de la silobolsa. Esta última situación se acentúa en áreas donde existe una mayor amplitud térmica, manifestándose en mayor proporción en primavera y cuando existen zonas de la silobolsa que no poseen el estiramiento recomendado.

17.2 Almacenaje de Granos con destino a industria

Los granos cuyo destino es principalmente la industria (soja, maíz pisingallo, cebada cervecera, etc.), tienen requerimientos específicos de calidad y condición adicionales a las reglamentarias. Estos granos son generalmente almacenados en silos aunque últimamente también en silo-bolsas (cebada cervecera, soja).

En algunos granos la humedad es uno de los factores principales que influyen en el rendimiento industrial: el maíz pisingallo se comercializa en base al volumen de expansión, el cual depende básicamente del contenido de endosperma corneo del grano y de la humedad del mismo. Con humedades de 13 a 14,5 % se logra el máximo volumen de expansión [Cardoso09]. En soja la humedad del grano influye tanto en el rendimiento industrial, como en la puesta a punto del proceso de prensado, la humedad de óptima varía según la finalidad del subproducto (pellets, harina de alta proteína) pero como pauta general se recomienda 11 % de humedad.

Existe investigaciones que demuestran la existencia de una amplia variabilidad entre la humedad de los granos provenientes de distintos materiales, ambientes, en la misma planta e inclusive proveniente de la misma espiga [Cardoso09]. Según este autor, debido a el orden progresivo de formación de granos en la espiga, cuando en la punta de la espiga existe una variación a la humedad de cosecha un valor promedio de

18 % (valores extremos entre 13 y 25 %) en la base de la espiga una humedad promedio fue de 25 % (extremos 15.5 y 36.5). Esta dispersión no es mostrada por el humedímetro, el cual solo indica el valor promedio de la humedad los granos individuales de una muestra. Dicha variabilidad es máxima cuando el grano es recientemente cosechado, se mantiene aun si el grano es secado a alta temperatura, y solo puede ser reducida mediante un período de acondicionamiento [Cardoso09] para la industria.

Es necesario que la humedad óptima no sea un promedio de una gran disimilitud de humedades de granos individuales sino que exista una homogeneidad en la humedad de los mismos. Esto permitiría un óptimo rendimiento industrial en el caso de procesamiento de soja, molienda húmeda y seca del maíz y un máximo volumen de expansión en el caso del maíz pisingallo. Teniendo en cuenta en el sistema de atmósfera modificada no se puede acondicionar el grano almacenado mediante aireación, es de importancia monitorear la variabilidad de la humedad de los granos individuales en el tiempo para almacenamiento de grano con destino a industria.

17.3 Almacenamiento de forrajes en silobolsas

Una herramienta para optimizar los sistemas de almacenamiento de forrajes es la silobolsas. Actualmente se cuenta con la tecnología necesaria para realizar silaje de planta entera, considerada como la forma de conservación más adecuada para la preservación de grandes volúmenes de masa vegetal, manteniendo las características del forraje en condiciones parecidas a las originales [Marinissen].

El ensilaje de forraje verde es una técnica de conservación que se basa en procesos químicos y biológicos generados en los tejidos vegetales cuando éstos contienen suficiente cantidad de hidratos de carbono fermentables y se encuentran en un medio de anaerobiosis adecuada. La conservación se realiza en un medio húmedo y debido a la formación de ácidos, que actúan como agentes conservadores, es posible obtener un alimento suculento y con valor nutritivo muy similar al forraje original [Marinissen].

Una propiedad fundamental del material biológico almacenado dentro de la silobolsa es su característica de absorción de agua. Esta característica está determinada por condiciones ambientales muy particulares, debido a que la bolsa plástica que contiene el material se encuentra cerrada herméticamente y se produce en su interior una atmósfera modificada dentro de la cual los valores de temperatura y humedad relativa son significativamente diferentes a la atmósfera exterior. Como se sabe también el contenido de O2 y CO2 sufre variaciones importantes en la atmósfera interior.

Para poder diseñar sistemas eficientes de manipulación y almacenamiento de materiales higroscópicos dentro de bolsas plásticas cerradas herméticamente es necesario disponer de datos que relacionen el contenido de humedad en equilibrio del material (EMC siglas en inglés de Equilibrium Moisture Content) y la humedad relativa en equilibrio del ambiente (ERH siglas en inglés de Equilibrium Relative Humidity) [Marinissen].

Capitulo 18 Detección temprana de procesos de descomposición de granos en silobolsas

18.1 Introducción

El monitoreo periódico de la concentración de CO2 se puede utilizar como herramienta para detectar un aumento en la actividad biológica en bolsas y relacionarlo con procesos de descomposición del grano.

La concentración típica de CO2 para bolsas de trigo y soja con evidencia de grano afectado y con condiciones de almacenaje seguras fue determinada.

Se mide la concentración de CO2 en el la bolsa y se la compara con la concentración CO2 típica para las bolsas con condiciones de almacenaje seguras e inseguras. Por comparación, la condición de almacenaje de la bolsa se clasifica como segura, riesgosa o insegura.

La concentración de CO2 en bolsas plásticas de trigo en condiciones de almacenaje seguras aumenta con la humedad del grano (debajo de 5% de CO2 para 13% de humedad o menos, y hasta el 17% CO2 para 16% de humedad).

La humedad del grano de soja no afecta substancialmente la concentración CO2 de bolsas con condiciones de almacenaje seguras (en un rango de humedades del 11 a 15%). Así, cualquier medida de concentración de CO2 por debajo del 4% significa condiciones de almacenaje "segura", entre 4 y el 12% significa condiciones de almacenaje "riesgosa", y por encima de 14% significa condición de almacenaje "insegura".

18.2 Medición de la temperatura

La medición de la temperatura del grano es la principal herramienta usada para supervisar la condición del grano en sistemas de almacenaje tradicional (silos y celdas) por los establecimientos rurales, acopios comerciales y la industria, puesto que un incremento en temperatura en un sector del granel está altamente correlacionado con el aumento en la actividad biológica en dicha area. Desafortunadamente, esta tecnología no es útil para monitorear condiciones de almacenaje en bolsas plásticas. Se demostró que la temperatura del grano almacenado en las bolsas sigue el patrón de la temperatura media ambiental, variando con las estaciones. Esto se debe a que la bolsa tiene una mayor capacidad de intercambiar calor con el aire ambiente y con el suelo. La relación superficie/volumen de una bolsa de 180 toneladas es aproximadamente 1,42 mientras que para un silo estándar de metal de la misma capacidad (9 m de altura y 7 m de diámetro) el cociente es 44% menor (0,79). De esta manera la relación de la temperatura del grano con la actividad biológica se puede enmascarar por el efecto de la temperatura ambiente. A su vez, el ecosistema generado en un ambiente hermético tiene una tasa de respiración disminuida respecto de un ecosistema de un silo o celda convencional, por lo que la tasa de liberación de calor del grano en descomposición es menor.

18.3 Monitoreo del grano mediante calado

El monitoreo del grano almacenado en bolsas mediante el calado tradicional es un proceso bastante fácil de implementar. Sin embargo, cada perforación hecha en la cubierta plástica disturba la hermeticidad del sistema, lo cual limita el número de muestras que se pueden recoger de cada bolsa y la frecuencia de muestreo. Además, este monitoreo es útil para obtener una idea de la calidad total del grano almacenado en la bolsa (porcentaje de humedad, contenido proteico, falling number, etc), pero no es conveniente para detectar problemas tempranos de almacenaje (la mayor parte del proceso de descomposición del grano ocurre en lugares muy localizados de la masa del grano, típicamente en el fondo de la bolsa, donde la punta del calador tradicional no llega a recoger la muestra). Otra desventaja de esta metodología es la cantidad de mano de obra y tiempo implicado.

18.4 Monitoreo del grano mediante la medición de CO2

Las bolsas son impermeables y tienen un alto grado de hermeticidad a los gases (Oxígeno (O2) y dióxido de Carbono (CO2)). Consecuentemente, la respiración de los componentes bióticos del granel (granos, insectos y hongos) eleva la concentración de CO2 y reduce la de O2. Así, el nivel de modificación de los componentes gaseosos del aire intersticial se puede relacionar con la actividad biológica dentro de la bolsa, y utilizar dicha medición como herramienta de monitoreo para detectar problemas tempranos de granos dañados.

Cardoso y Rodríguez [Bartosik08] estudiaron los principales factores que afectaban la concentración de CO2 en trigo y soja almacenados en bolsas plásticas, estableciendo además los valores típicos de concentración para bolsas sin problemas de almacenamiento. En su estudio implementan dos metodologías de detección de problemas de almacenamiento de granos en bolsas plásticas basado en:

1. La medición frecuente en el tiempo de la concentración de CO2 (evolución en el tiempo)
2. Mediante la comparación de la medición de la concentración de CO2 de una bolsa particular con la concentración típica de CO2 de trigo y soja almacenados en bolsas sin problemas de almacenaje

18.5 Como se realizo un ensayo: Materiales y métodos

En las pruebas en REF, los ensayos para la medición de de CO2 las bolsas fueron llenadas con grano posteriormente a la cosecha. Generalmente las bolsas se instalaron en el mismo lote, o en plantas de acopio durante el periodo de cosecha. El experimento duro aproximadamente 5 meses hasta que las bolsas fueran abiertas para extraer el grano.

Para cada bolsa se establecieron tres lugares de muestreo. El procedimiento consistió en medir la concentración de CO2 mediante un analizador portátil de gases (PBI Dan Sensor, CheckPoint, Dinamarca (Figura 18.1), perforando la cubierta plástica con una aguja hipodérmica. La composición del gas fue analizada para tres niveles por

cada sitio de muestreo de la bolsa (superior, medio e inferior). En cada punto de muestreo el grano fue recolectado realizando un calado tradicional, a partir de tres diversos niveles (superior: 0,10 m de profundidad, medio: 0,75 m de profundidad, inferior: 1,6 m de profundidad. Altura total de la bolsa: 1,7 m). Una muestra de grano fue extraída de cada sitio y posteriormente remitida al laboratorio para la medición de humedad [Bartosik08]. Después del calado de la bolsa las aberturas fueron selladas con una cinta adhesiva especial para restituir la hermeticidad.

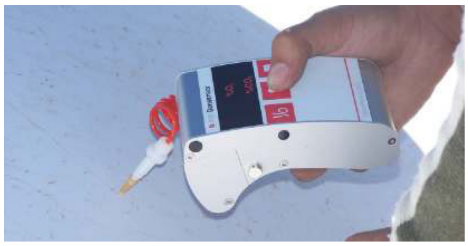

Figura 18.1. Muestreo del nivel de O y CO con un medidor portátil de gases.

Se repitió el monitoreo aproximadamente cada 15 días durante todo el período de almacenaje. Cuando finalmente las bolsas fueron vaciadas, el grano y la integridad física de la cubierta plástica fueron examinados para detectar si el grano se encontraba afectado. Se clasificó entonces a las bolsas plásticas como "sin evidencia de problemas del almacenaje" o "con evidencia de problemas del almacenaje" (Figura 18.2).

Figura 18.2 Grano afectado detectado durante el vaciado de la bolsa.

18.6 Resultados

Durante invierno (julio-agosto) la concentración de CO2 se encontró por debajo del 3% para todas las humedades analizadas. Comenzando la primavera (septiembre) la concentración de CO2 comenzó a aumentar llegando a 9 y 10%, para las bolsas con 12,9% y 11,5% de humedad, respectivamente.

Durante octubre, la concentración de CO2 incrementó hasta 16% y 18% para las mismas bolsas. Posteriormente, la concentración de CO2 disminuyó hasta 10 y 13% y se estabilizó hasta el último reporte (diciembre). La bolsa con soja húmeda (14,9%) tuvo una concentración de CO2 por debajo del 2% durante todo el período de almacenaje. Cuando las bolsas que presentaron alta concentración de CO2 fueron vaciadas, una cantidad significativa de grano con afecciones severas fue detectada. Se observó que al menos una capa de grano de 0,1 m de espesor se encontraba afectada producto de perforaciones en la cubierta plástica en la base de la bolsa. Las roturas en la bolsa permitieron la entrada de agua y O2, creando condiciones favorables para el

desarrollo de hongos cuando la temperatura del grano incrementó en la primavera temprana. Por otra parte, la bolsa con soja húmeda no presentó ninguna perforación significativa en la cubierta plástica, por lo que las condiciones de almacenaje seguras perduraron durante el período de almacenaje incluso durante el final de la primavera. Consecuentemente, el grano no demostró evidencia de deterioro cuando se examinó durante la extracción final, coincidiendo con la baja concentración de CO2 registrada.

Estos resultados demostraron que el monitoreo de la concentración de CO2 se puede usar como herramienta para la detección temprana de problemas de granos dañados en bolsas.

Aunque el monitoreo periódico de la concentración CO2 permite detectar un aumento de la actividad biológica en bolsas durante el almacenaje, a veces sólo es posible realizar una medición de la concentración CO2 de la bolsa. En este caso, no existe la posibilidad de comparar los resultados con valores anteriores para determinar si la actividad biológica se incrementó y la condición de almacenaje del grano se tornó insegura. Para estas situaciones otro tipo de aproximación es necesaria para relacionar la concentración CO2 con la condición de almacenaje del grano.

La concentración media de CO2 para bolsas de trigo en condiciones de almacenaje apropiadas fue significativamente más baja que la concentración media para las bolsas con evidencia de grano afectado. Con humedades inferiores a 13%, la diferencia entre las dos líneas fue cerca de 10 puntos porcentuales de CO2, mientras que para 16% de humedad la diferencia se redujo a 7,5 puntos porcentuales. Se pudo observar que en las bolsas de trigo que están por debajo de 16% de humedad, el grano afectado fue localizado en las capas inferiores del grano. En estas bolsas se observaron varias perforaciones en la cubierta plástica, lo cual permitió la entrada de humedad (por las precipitaciones) y de O2. Las perforaciones fueron causadas por los animales (peludos y otros), o a causa de que la bolsa fue armada sobre los residuos de la cosecha anterior (los vástagos perforan la bolsa si no se toman los cuidados apropiados durante la operación de armado y llenado de la bolsa). Otra causa de deterioro del grano estuvo relacionada a deficiencias en el cierre del extremo de la bolsa, lo cual permitió la entrada de humedad y O2 al sistema. Finalmente, algunas bolsas se armaron sobre terrenos bajos resultaron inundados temporalmente después de una precipitación intensa. En esta última situación una bolsa sana y bien cerrada es normalmente afectada.

Por otra parte, cuando el trigo se almacenó por encima de 18% de humedad el grano resultó afectado, aun cuando no se observó ninguna rotura o falla en la confección de la bolsa. El grano de trigo excesivamente húmedo dio lugar a alta actividad de hongos, causando un proceso de descomposición del grano generalizado.

La media de la concentración CO2 para las bolsas de soja con condiciones de almacenaje apropiadas fue significativamente más baja que la concentración media para las bolsas con evidencia de grano afectado. Las bolsas de soja bajo condiciones de almacenaje apropiadas presentaron siempre valores de CO2 por debajo del 4%, y no mostraron una tendencia de aumentar el CO2 con el incremento de la humedad. Por otra parte, las bolsas con evidencia de soja dañada dieron lugar a concentraciones de CO2 entre 6% y 18%, con un promedio entre 11,5 y el 14%. En contraste con los datos del trigo, en soja no se observó una correlación clara entre el aumento de la concentración de CO2 y el incremento de la humedad (tanto en bolsas con el grano en condiciones de almacenaje apropiadas como en bolsas donde se evidenció grano afectado). Las razones que causaron la descomposición del grano fueron similares a las descriptas para las bolsas de trigo (perforaciones en la cubierta plástica, el cierre incorrecto, o anegamientos temporales por causa de precipitaciones, etc).

De acuerdo a estos resultados, los autores proponen utilizar el monitoreo de la concentración de CO_2 para detectar problemas en bolsas en trigo y soja. En el caso de las bolsas de trigo, el operador debería medir la concentración de CO_2 y tomar también una muestra física de grano para determinar la humedad. Si la concentración CO_2 medida está por debajo de la concentración típica de CO_2, la condición de almacenaje se puede clasificar como "segura". Por otro lado, si la concentración de CO_2 medida está por encima de la concentración de CO_2 segura se clasifica la condición de almacenaje como "riesgosa" y el operador debe supervisar esta bolsa con mayor detalle. Finalmente, si la concentración de CO_2 medida es cercana o por encima de la concentración típica de CO_2 de una bolsa afectada, entonces la condición de almacenaje se clasifica como "insegura".

En el caso de las bolsas de soja la humedad de almacenaje del grano no es crítica para las condiciones del ensayo. Para condiciones de humedad del grano en un rango entre 11 y 15% y concentración CO_2 por encima de 4%, la condición de almacenaje de la bolsa plástica se debe clasificar como "riesgosa". En caso contrario la condición de almacenaje de la bolsa se debe clasificar como "segura" (concentración de CO_2 por debajo del 4%). Cuando la concentración de CO_2 está cercana o por encima de 11,5-14%, la condición de almacenaje de la bolsa se debe clasificar como "insegura".

Parte IV. Definición del DSL para sistemas de control de granos almacenados en silobolsas

En esta parte se describen todos las pasos para la definición y construcción del DSL de silobolsas. Estos pasos son: Identificación de los conceptos a Modelar; Defincón del modelo de dominio; Definición de la notación para el lenguaje; Definición de los generadores de código; Definición del Framework de dominio.

Capitulo 19 Definición del DSL

19.1 Identificación de los conceptos a Modelar

Luego del análisis de dominio hecho en la sección anterior se definió cuales conceptos del dominio se van a poder modelar con el DSL.

Información estática y dinámica

La información del dominio se clasificó en dos categorías, información estática e información dinámica:

- **Información estática**: dentro de esta categoría esta la información que define tipos de objetos. Por ejemplo los Tipos de sensores, Tipos de variables a monitorear (temperatura, humedad), Tipos de Contenidos.
- **Información dinámica**: dentro de esta categoría esta la información que define entidades únicas, las cuales son de algún tipo. Por ejemplo los sensores colocados en una silobolsa es informción dinámica. Estos sensores de un Tipo de sensor.

19.1.1 Conceptos a Modelar a partir de la información estática

Dentro de la información estática se indentificaron los siguientes conceptos que el DSL a crear debe permitir modelar:

Variables de medición

El DSL debe permitir modelar las distintas variables que serán monitoreadas por los sensores. Denominamos varible de medición a toda propiedad que sirve para monitorear el estado del contenido de una silbolsa. Por ejemplo el dióxido de carbono CO2 es una variable de medición.

Las principales variables de medición son:

- Temperatura
- CO2

- O2
- Humedad

Tipo de Sensores

El DSL debe permitir modelar los distintos tipos de sensores que son usados para medir las diferentes variables de medición. Además se debe poder especificar las siguientes características de los sensores:

- Marca
- Modelo
- Variable que mide
- Rangos de medición
- Descripcion general

Tipo de Contenido

EL DSL debe permitir modelar los distintos tipos de contenidos que puede ser almacenado dentro de una silobolsa. Existen distintos tipos, por ejemplo se puede almacenar grano y forraje. A su vez los granos existen distintos variedades. Al modelar el tipo de contenido se debe poder especificar las características del mismo:

- Tiempo de almacenamiento máximo recomendado
- Valores máximos y mínimos para las varibles de mediciones (Tempertatura, CO2, etc.).
- Tiempo máximo de almacenamiento recomendado.

19.1.1 Conceptos a Modelar a partir de la información dinámica

Dentro de la información dinámica se indentificaron los siguientes conceptos que el DSL deberá permitir modelar:

Silobolsas

El DSL debe pemitir modelar la silobolsa y sus características. Entre la información más importante se identificó:

- Fabricante de la silobolsa
- Modelo
- Material
- Espesor
- Tamaño largo y ancho
- Descripción general
- Red de sensores
- Contenido que almacena
- Posición geográfica
- Si esta cubierta por una media sombra

Contenido de una silobolsa

El DSL de permitir modelar el contenido almacenado en cada silobolsa, y la información del mismo:

- Tipo de contenido
- Fecha de almacenamiento
- Nivel inicial de temperatura
- Nivel inicial de humedad.
- Nivel inicial de dióxido de carbono (CO2)
- Nivel de oxígeno (O2)
- Porcentajes de granos dañados en el momento que se almacenó
- Porcentaje de materia extraña cuando se almacenó

Sensor

El DSL debe permitir modelar la red de sensores que tiene cada silobolsa, para esto se debe poder representar cada sensor de la silobolsa y la información asociada a el:

- Posición del sensor en la silobolsa
- Tipo de sensor
- Identificación del sensor
- Frecuencia de monitoreo
- Acciones a ejecutar en caso de que una medición supere un umbral.

Registro de Medición

Se debe poder modelar un registro de medición, el cual representa el resultado del sensor luego de tomar una medición. La información para este registro es:

- Sensor que genero el registro
- Variable que mide
- Fecha y hora de la medición
- Valor de la medición

19.2 Defínción del modelo de dominio

La definición del modelo de dominio es la definición del meta modelo del DSL, es decir es la definición del las variables antes identificadas usando las DSL Tools.

SilobolsaModel

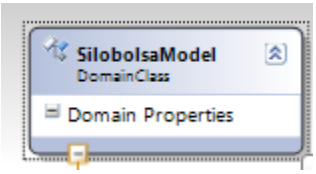

Figura 19.2.1

Lo primero que se define es la clase SiloBolsaModel, la cual es la raíz de el metamodelo, es decir todo elemento del modelo esta asociado a esta clase directa o indirectamente.

Un silobolsaModel tiene SiloBolsas y Tipos. Los tipos es una representación de toda la información estatica, por ejemplo Tipos de Sensores, Variables de medición, etc. La siguiente figura muestra como se modelo esto:

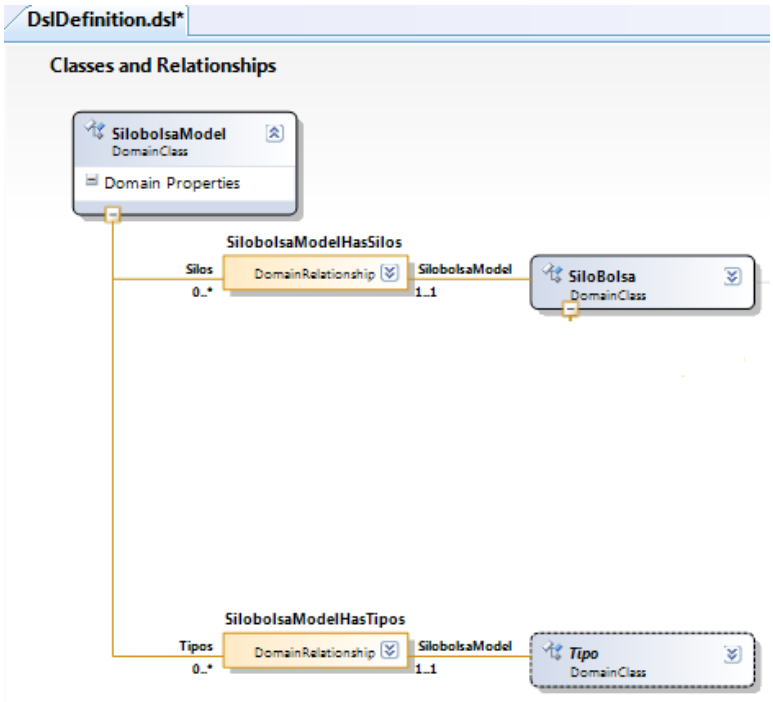

Figura 19.2.2

19.2.1 Modelado de la Información estática

En primer lugar se modelo toda la información estática, es decir los tipos. Todos los tipos fueron agrupados en una jerarquía.

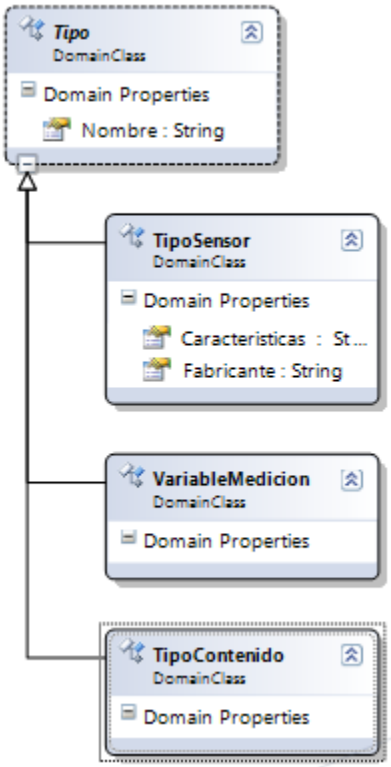

Figura 19.2.3

A continuación se describen los elementos de esta jerarquía

TipoContenido

Los distintos tipos de contenidos que pueden ser almacenado en una silobolsa fueron representados por una clase llamada TipoContenido.

VariableMedicion

Todas las variables de medición fueron representadas por una clase llamada VariableMedicion, a la cual se le debe especificar el nombre. Una variable de medición estará asociada a un sensor por una relación "sensor mide variable".

TipoSensor

La clase TipoSensor en el modelo representa los distintos tipos de sensores. Las propiedades que le fueron definidas son Caracteristicas y Fabricante.

MinMaxRango

Cada tipo de contenido (Soja, Maiz, etc.) tiene asociado un valor mínimo y máximo para cada tipo de variable de medición (CO2, Temperatura, etc.). Esta definicion es usada para dispar una alarma si un sensor detecta que una medición esta fuera del rango aceptable.

La clase MinMaxRango modela esta información

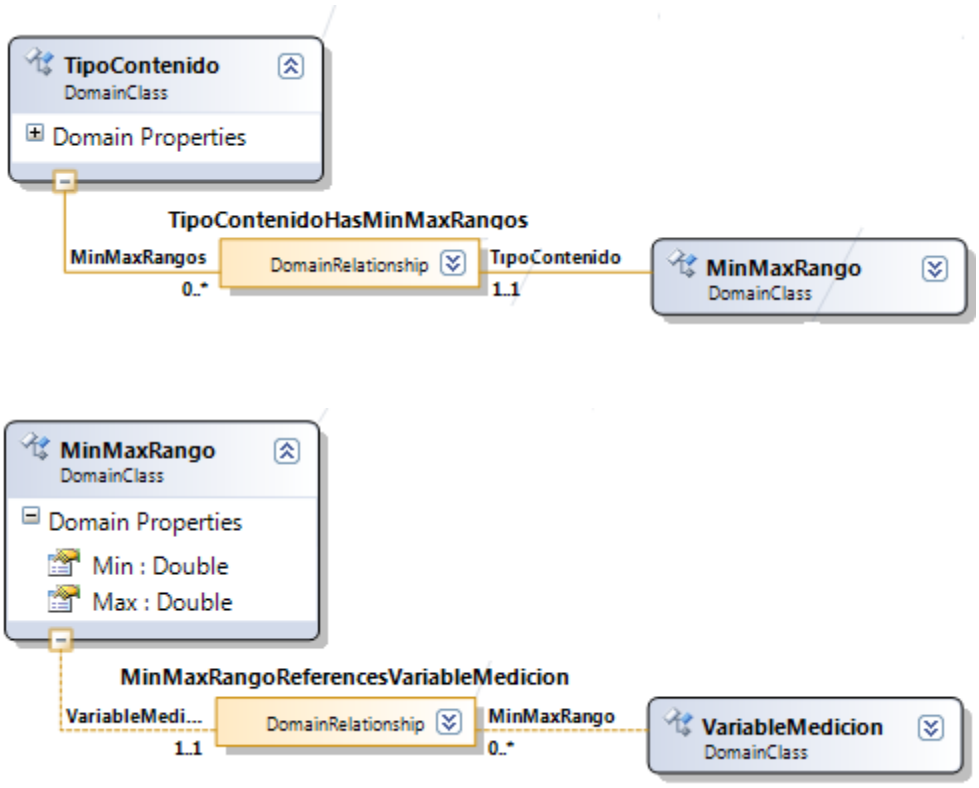

Figura 19.2.4

En la figura se puede ver que un tipo de contenido puede tener muchos MinMaxRango, esto es porque tiene uno por cada VariableDeMedicion. Además un MinMaxRango es configurado para solo un tipo de VariableDeMedicion

19.2.2 Modelado de la Información Dinámica

A continuación se muestra como se modeló la información dinámica: las silobolsas, sensores, contenido de las silobolsas, etc.

SiloBolsa

Las silobolsas y su información se representaron con la clase SiloBolsa

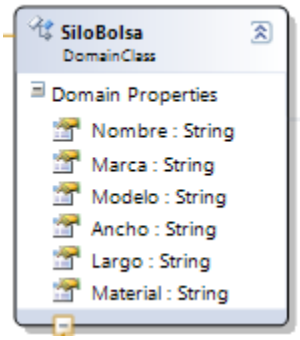

Figura 19.2.5

Una SiloBolsa tiene una lista de sensores, la cual representa la red de sensores asociada a la misma. Un Sensor puede estar asociado a una sola silobolsa y solo existe si esta existe. La relación "silobolsa tiene sensores" es de tipo embebida y esta representada en el modelo de la manera siguiente:

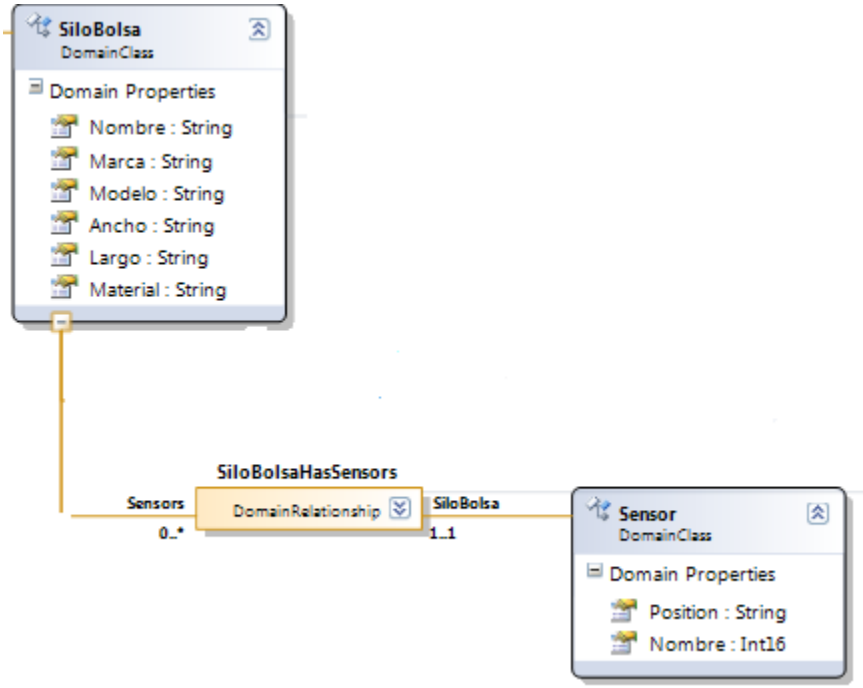

Figura 19.2.6

Además una silobolsa tiene un contenido, el contenido de la silbolsa pertenece solo a una silobolsa y no puede existir si no esta esta asociado a una silobolsa, esta relación es tipo embebida.

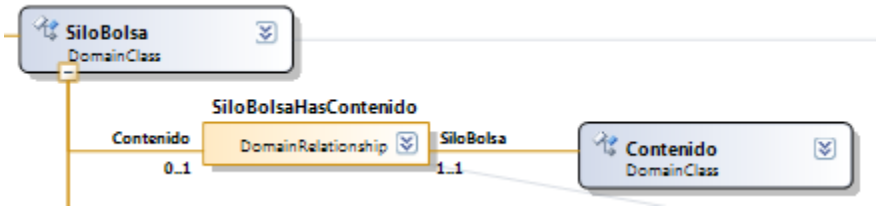

Figura 19.2.7

Finalmente una silobolsa tiene una lista de RegistroMedicion

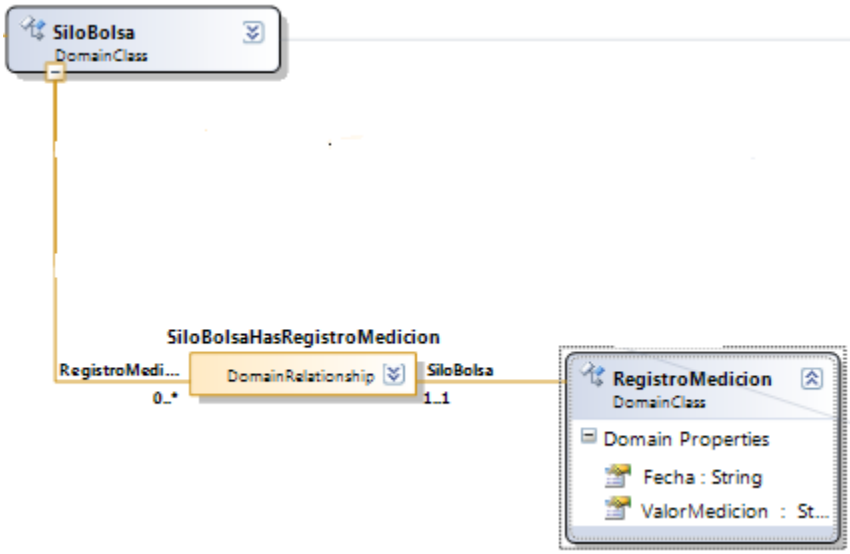

Figura 19.2.8

Los registros representan la información de medición que generan los sensores asociados a la bolsa.

Contenido

El contenido de la silobolsa y sus propiedades se modeló con la clase Contenido. Además un contenido tiene un tipo de contenido. Como el tipo de contenido puede existir por si solo aunque no haya un contenido que lo referencie, la relación "contenido es de tipo" es del tipo referencia.

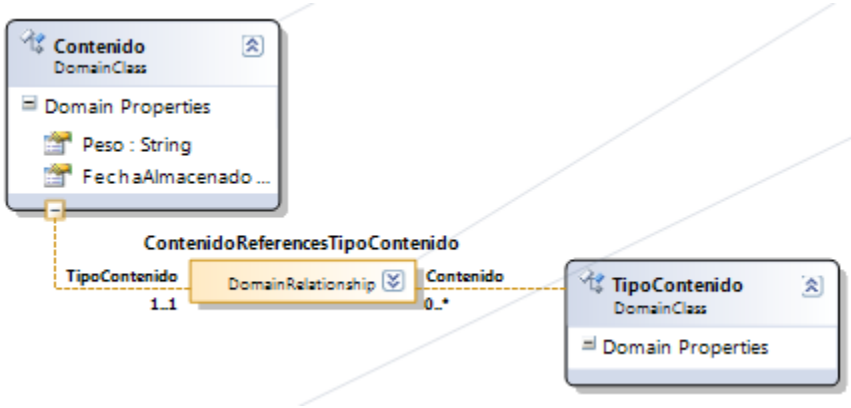

Figura 19.2.9

Sensor

Para modelar la red de sensores asociada a una silobolsa se definió la clase llamada Sensor. Un Sensor esta asociado a un Tipo de sensor mediante el cual se determina la variable que mide y sus características generales. Esta relación es de tipo referencia dado que un TipoSensor puede existir aunque no haya sensores de este tipo.

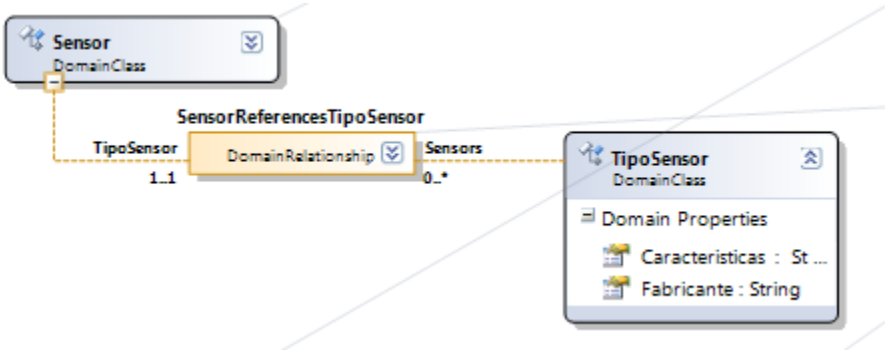

Figura 19.2.10

MediaSombra

Una silobolsa puede estar recubierta por una media sombra, la media sombra se modelo con la clase MediaSombra:

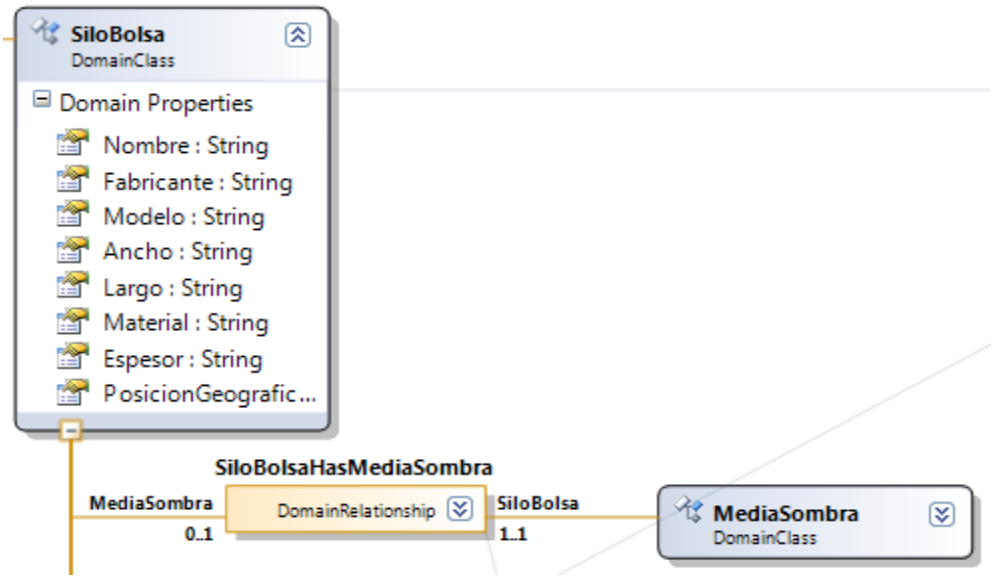

Figura 19.2.11

RegistroMedición

El registro de la medición que produce un sensor se modeló con la siguiente clase

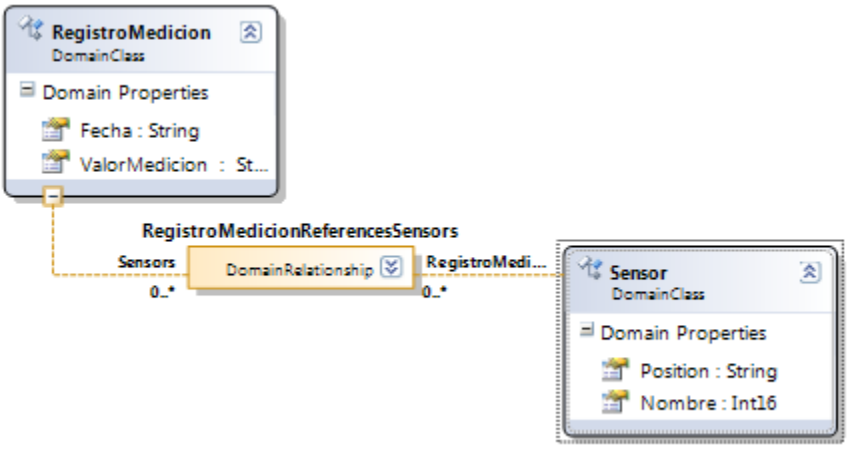

Figura 19.2.12

Como se ve en la figura anterior un registro de medición pertenece solo al un sensor que lo generó.

19.3 Definición de la notación para el lenguaje

La definición de la presentación o de la notación del DSL involucra la definición de varios aspectos: la definición de la notación gráfica de los elementos del DLS; La definición de los elementos que van a aparecer en el Toolboox; La customizacion de la apariencia de las properties Windows. La parte más importante es la definición de la notación gráfica del DSL.

19.3.1 Notación Gráfica del los elementos del DSL

La información del modelo de dominio y la de definicion del modelo de dominio se llevó a cabo siguiendo la categorización de la información en información estática e información dinámica. De igual manera la definicion de la notación gráfica se realizo teniendo esto en cuenta.

Como se describió anteoriormente existentes varias clases de figuras en las DSL Tools que pueden ser usadas para la definición de la notacionn gráfica: Geometric shapes, Image shape, etc. Para lo notación gráfica del DSL se usaron las GeometricShapes y las ImageShape. Todo elemento del DSL que corresponde a información dinámica se representó usando una figura tipo imagen y todo elemento que corresponde a información de tipo estática se represetó usando una figura tipo geométrica.

Modelo mínimo usando el DSL

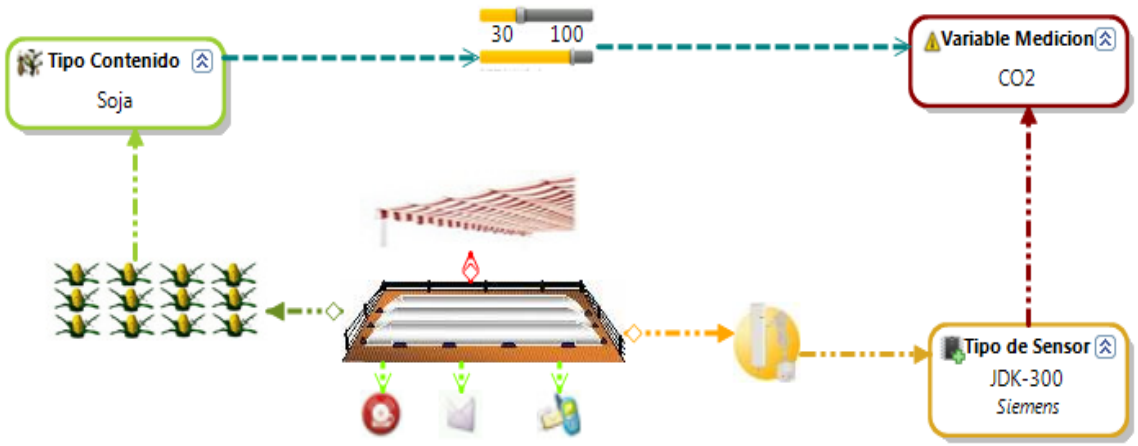

Figura 19.3.1

En la figura anterior se ve como un modelo y se puede ver la notación gráfica del DSL. A continuación se describe la notación gráfica para cada elemento.

Notación grafica para la información dinámica

Los elementos que que pertenecen a información dinámica (Silobolsa, Contenido, Sensor) se representaron gráficamente usando ImageShapes como se muestra a continuación:

Silobolsa

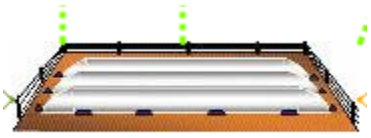

Figura 19.3.2

Contenido

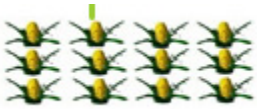

Figura 19.3.3

Sensor

Figura 19.3.4

MediaSombra

Figura 19.3.5

Notación para la información estática

Los elementos que pertenecen a información estátic (TipoContenido, TipoSensor, VariableMedición) se representaron gráficamente con usando GeometricShape como se muestra a continuación:

TipoSensor

Figura 19.3.6

En la notación gráfica para el TipoSensor se muestra el modelo y la marca del mismo, en este ejemplo es un modelo JDK-300 de la marca Siemens.

TipoContenido

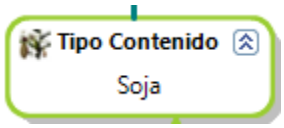

Figura 19.3.7

En la notación gráfica para el TipoContenido se muestra el nombre del contenido, en este ejemplo Soja.

VariableMedicion

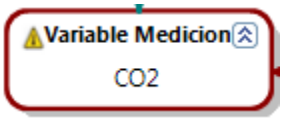

Figura 19.3.8

En la notación gráfica para la VariableMedición se muestra el nombre de la variable, en este ejemplo CO2.

MinMaxRango

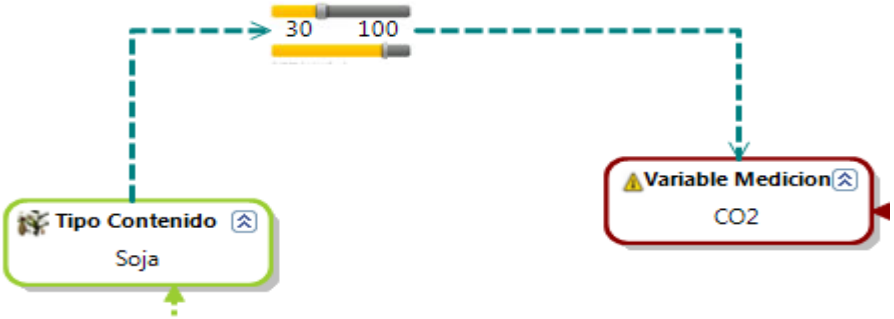

Figura 19.3.9

En la notación gráfica para el MinMaxRango se muestran los valores mínimo y máximo. En el ejemplo 30 sería el mínimo y 100 el máximo.

19.3.2 ToolBox y Area de diseño

A continuación se muestran el aspecto gráfico de todos los elementos en el ToolBox

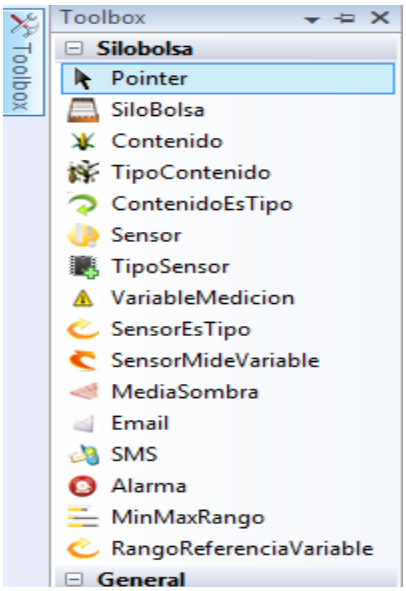

Figura 19.3.10

Capitulo 20 Definición del generador de código

20.1 Introducción

En esta sección se define cual es el código a generar a partir del modelo creado usando el dsl y como es realizada la generación. Además se describen las distintas técnicas para generación de código junto con sus pros y contras de cada una.

20.2 Artefactos a generar

A partil del modelo diseñado usando el dsl, se desea generar tres cosas:

- **Clases C#:** se deben generar las clases necesarias para el representar el modelo.
- **Grafo de objetos:** se deben generar las instancias de todas las clases generadas anteriormente y relacionar estos objetos entre si para reprensentar el modelo creado con el DSL.
- **Loader del Modelo:** se necesita definir un objeto que ofrezca una API que permita poder trabajar con el grafo de objetos antes mencionado. Esta API será particularmente usada por el Framework de Dominio.

Para la generación de estos artefactos se deben definir en archivos templates los algoritmos que recorran el modelo y generen los artefactos.

A continuación se describe con un ejemplo cuales son los artefactos a generar.

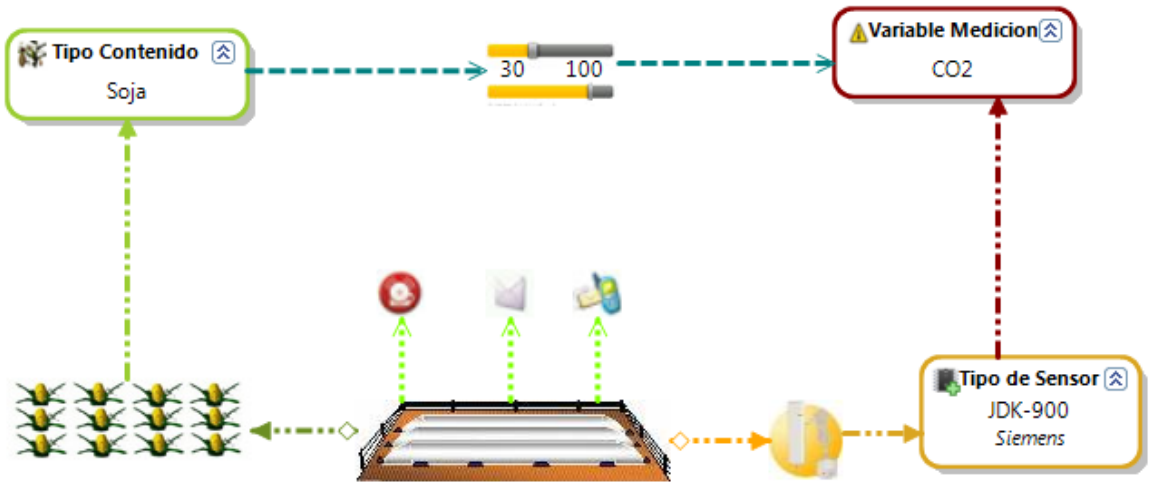

Figura 20.1

Este modelo representa una Silobolsa con contenido de tipo Soja. Este contenido es monitoreado por un sensor de marca Siemens el cual permite permite tomar los valorores de dióxido de carbono (CO2).

A partir de este modelo se debe generar los tres artefactos mencionados en código C#.

Generación de Clases

En primer lugar se deben crear las clases C# para representar cada concepto de nuestro modelo, a continuacion se da un ejemplo de cómo seria la clase de la Silobolsa a generar:

```
	/// </summary>
	/// Double-derived base class for SiloBolsa
	/// </summary>
	public abstract partial class SiloBolsaBase
	{
		public virtual string Nombre { get; set; }

		public virtual string Marca { get; set; }

		public virtual string Modelo { get; set; }

		public virtual int Ancho { get; set; }

		public virtual int Largo { get; set; }

		public virtual string Material { get; set; }

		public virtual Contenido Contenido { get; set; }

		public virtual System.Collections.Generic.IList<Sensor>
SensorList { get; set; }

		public virtual
System.Collections.Generic.IList<RegistroMedicion> RegistroMedicionList {
get; set; }

	}
```

Esta clase Silobolsa contiene las variables de instancias necesarias para representar las propiedades de las silobolsas. De manera similar se debe generar el código del resto de las clases (Contenido, Sensor, etc.).

Generación del grafo de objetos

Una vez creada todas las clases, el segundo paso es crear todas las instancias de estas clases para representar la información. Por ejemplo a continuación se muestra como debe ser el còdigo para la instancion de una silobola y asignacion de valores a sus variables de intancia:

```
SiloBolsa aSiloBolsa = null;
aSiloBolsa = new SiloBolsa();
aSiloBolsa.Nombre = "SiloBolsa1";
aSiloBolsa.Fabricante = "plastar";
aSiloBolsa.Modelo = "f100";
aSiloBolsa.Ancho = 100;
aSiloBolsa.Largo = 200;
aSiloBolsa.Material = "polietileno";
```

Por simplicidad, en este ejemplo solo se asignan valores a variables de instancias con tipos simples. Sin embargo en la generación real de código para instancion de las clases se deben inicializar todas las relaciones para generar un grafo de objetos consistente con el modelo.

Generación del Loader

El tercer elemento a generar es un objecto que contenga el grafo de objetos y que brinde una API para acceder a este grafo.

El código a generar debe ser similar al siguiente:

```
public class SilobolsasSystem
{
    private static SilobolsasSystem aSilobolsasSystem;
    public IList<SiloBolsa> Silos { get; set; }
    public IList<VariableMedicion> VariableMedicionList { get; set; }
    public IList<TipoContenido> TipoContenidoList { get; set; }
    public IList<TipoSensor> TipoSensorList { get; set; }

    public static SilobolsasSystem GetSilobolsasSystem()
    {
        aSilobolsasSystem = new SilobolsasSystem()
        {

            VariableMedicionList = new List<VariableMedicion>(),
            TipoContenidoList = new List<TipoContenido>(),
            TipoSensorList = new List<TipoSensor>(),
            Silos = new List<SiloBolsa>()

        };

        //…
```

La clase SilbolsasSystem tiene las variables de instancias en las cuales almacenará todas las intancias creadas, además tiene un método getSiloBolsasSystem() el cual es un método de clase que funcióna de builder. Este método crea una instancia del SilbolsasSystem y le carga el grafo de objetos. Esta instancia de SilbolsasSystem será usada luego por el Framework de Dominio para acceder al grafo de objetos.

20.3 Técnicas para la generación de código

Existen tres diferentes técnicas para la generación de código usando archivos .tt. Se describiran los principios de cada y sus ventajas.

Double-derived pattern

Este patrón (tambien conocido como el patron "Generatoin Gap") permite agregar cambias al codigo generado, sin perder estos cambios cuando el código es regenerado.

Los principios son:

- Se genera una clase base que contiene una implementación por defecto de propiedades y métodos.
- Se genera también una clase concreta parcial que extiende la clase anterior. Esta clase no contiene ningún método ni propiedad.
- Cualquier cambio en el código se hace creando otra clase partial de la concreta subclase en un nuevo archivo.

Cuando el código es regenerado, los cambios agregados en el nuevo archivo no son sobreescritos.

- Transformación desestructurada

Todo el código necesario para generar la salida está contenido en este template. Dicho código trabaja directamente contra los elementos del modelo, por ejemplo recorriendo todas las ModelClasses partiendo desde el elemento raiz y generando código para cada una. Luego itera las interfases embevidas en el modelo.
Sin embargo, la estructura de control lógica es dificil de ver, ya que está entremezclada con el código que genera la salida. Por esta misma razón es dificil diferenciar en C# entre el código template y el código que genera la salida.
Algunas funciones "helper" son definidas en bloques de clase al final del template (por ejemplo, el código contenido entre las marcas <#+ .... #>).
El código usa las funciones "PushIndent(...)" y "PopIndent()" para ayudar el manejo del layout de la salida generada. Esto es una alternativa a embever caracteres vacios directamente en el template.
Los templates hacen un uso extensivo de LINQ para navegar el modelo de objetos.

- Transformacion usando includes

En lugar de tener todo el código del template en un solo archivo, este template lo desglosa en multiples templates. El template principal ("TransformationUsingIncludes.tt") tiene directivas "<#@ include ... >" para importar los demás templates en tiempo de compilación.
Como resultado, el template principal es mucho mas corto, y contiene solo lógica de control. Esta estrategia es mucho más declarativa y simple de seguir.

Todos los templates incluyen bloques de clase que contienen funciones "helper".
Esta metodología simplifica el mantenimiento ya que todos los templates son cortos y su código está lógicamente agrupado. Por ejemplo, todo el código relacionado a la generación de clases está contenido en el archivo "Classes.tt".
Además, el template hace uso de la extensión "ForEach" definida en el DSL para simplificar la iteración de colecciones y ejecutar operaciones sobre sus elementos.
Sin embargo, este enfoque continua teniendo problemas. En primer lugar, agregar código relacionado al código de generación del DSL no es ideal, ya que esto significa que habrá que recompilar y redeployar el código si el código generado tiene cambios.
Por otro lado, estos templates aún contienen código complejo de leer y entender, llamadas a funciones y referencias al modelo DSL.

- Transformación usando una libreria externa

Este ejemplo demuesrta una referencia de ensamblado separada que contiene rutinas de generación. Este ensamblado separado contiene un objeto Model que encapsula la API normal generada para el diagrama de clases DSL y provee metodos adicionales y propiedades que simplifican la generación de código.
Por ejemplo, cada modelo del DSL es "wrappeado" como una instancia de CodeClass. Instancias de CodeClass son creadas mediante la invocación de "CodeGenUtilties.GetCodeClass(ModelClass)".
CodeClass tiene propiedades adicionales específicas de generación de código que retornan el string modificador de la instancia, y el nombre calculado de la clase base.
Estp permite que algunas funciones puedan quitarse de lso templates y encapsularlas dentro de herlpers, simplificando el debug. Esto también simplifica el codigo del template, lo cual significa que el codigo del template puede reducirse a:

```
public<#= codeClass.InheritanceModifierString #> partial class <#=
codeClass.Name #> : <#= codeClass.BaseClassName #>
```

Separar la funcionalidad de generación de codigo del código del modelo es conveniente, ya que el código puede ser modificado facilmente sin tener que redeployar el modelo ya ensamblado. Además, esta metodología facilita el soporte multiple de juegos de códigos diferentes que sean generados a partir de un mismo modelo, posiblemente en diferentes lenguajes. Tener todo esto en un mismo lugar podría dificultar el mantenimiendo y deploy.
languages. Placing all of this code in one assembly would make it much harder to maintain and deploy.

Para la generacion de artefactos se elijió usar la tercer técnica. A continuacion se describe la librería a para hacer el doble-derived pattern como tambien los archivos .tt usadas para la generacion de artefactos.

20.4 Defínción de Librería Helper

En primer lugar se agregó a la solution un proyecto de tipo assembly que contiene los wrappers de API generada para el diagrama de clases del DSL. Estos wrappers proveen métodos y propiedades adicionales que facilitan la generación de código desde los archivos templates.

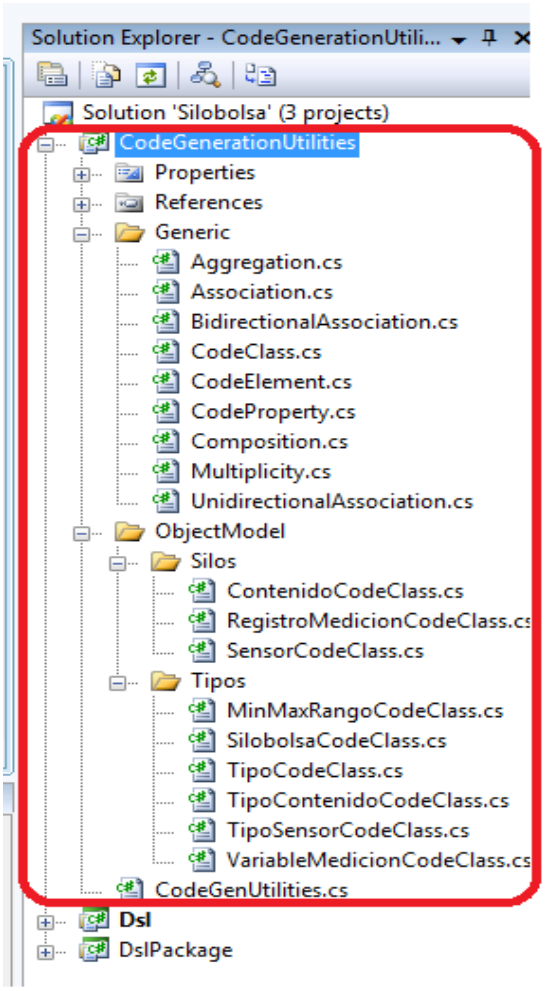

Figura 20.2

Como se ve la figura el assembly se llama CodeGenerationUtilities y contiene dos carpetas principales llamadas Generic y ObjetModel, a continuación se describe el contenido de cada una:

- Carpeta Generic: para generar código de clases con sus propiedades y métodos se creara abstracciones de estos conceptos. Esta carpeta contiene estas abstracciones.
- Carpeta ObjectoModel: esta carpeta contiene una clase por cada elemento del modelo. Estas Clases extienden CodeClass y definenProperties y Associations, etc. Estas claes se utilizan en los templates, para generación de código.

CodeGenUtilities: Esta clase proporciona la API principal que se usará desde los archivos .tt en la generación de código.

Ejemplo de clase wrapper

Como ejemplo tomaremos a SiloBolsaCodeClass

En primer lugar esta clase extiene a CodeClass

```
public class SilobolsaCodeClass : CodeClass
```

CodeClass es una clase base que define en forma genérica la información para la generación de clases. Por ejemplo define información para la generación de asociaciones:

```
    public abstract class CodeClass : CodeElement
    {
        public IList<Association> Associations { get; set; }
```

Toda subclase de CodeClass define información para la generación de código de propiedades y relaciones. En el ejemplo de SiloBolsaCodeClass la defición sería como la siguiente:

```
internal override void Build()
        {
            this.AddProperty(
                  new CodeProperty()
                  {
                      Name = "Nombre",
                      Type = "string",
                      IsIEnumerable = false,
                  });
            this.AddProperty(
                  new CodeProperty()
                  {
                      Name = "Fabricante",
                      Type = "string",
                      IsIEnumerable = false,
                  });

...
            this.AddProperty(
                  new CodeProperty()
                  {
                      Name = "Material",
                      Type = "string",
                      IsIEnumerable = false,
                  });
        }
```

En el ejemplo se define un objeto CodeProperty por cada propiedad de una Silbobolsa, este objeto tiene información para la generación del código de las variables de instancia de una clase Silbolsa: nombre de propiedad y tipo de la propiedad.

El código para la definición de relaciones es el siguiente

```
            UnidirectionalAssociation association = new
UnidirectionalAssociation();
            association.Target = contenidoCodeClass;
            association.TargetMultiplicity = Multiplicity.One;
            association.TargetRoleName = contenidoCodeClass.Name;
            association.Source = this;
            this.AddAssociation(association);

            association = new UnidirectionalAssociation();
            association.Target = sensorCodeClass;
            association.TargetMultiplicity = Multiplicity.ZeroMany;
            association.TargetRoleName = sensorCodeClass.Name;
            association.Source = this;

            this.AddAssociation(association);

            association = new UnidirectionalAssociation();
            association.Target = registroMedicionCodeClass;
            association.TargetMultiplicity = Multiplicity.ZeroMany;
            association.TargetRoleName = registroMedicionCodeClass.Name;
            association.Source = this;

            this.AddAssociation(association);
```

En este ejemplo se crean objetos de tipo UnidirectionalAssociation con la información necesaria para la generación de código correspondiente a asociaciones (multiplicidad, nombre de los roles etc.). Estas asociaciones son agregadas a la lista de asociaciones de la SiloBolsaCodeClass en la línea de la

```
            this.AddAssociation(association);
```

Toda esta inicialización se lleva a cabo dentro del método build(), este método es llamado por CodeGenUtilities, cuando es usado desde los archivos .tt.

20.5 Defínción de los archivos templates .tt (voy por acá)

Para la genración de código apartir del modelo se deben definir archivos templates.
Los templates a definir deben realizar las siguientes acciones:

- Recorrer el modelo para generar el código de las clases
- Recorrer el modelo para generar el grafo para el código con la instanciacio clases anteriores.
- Generar el código del Loader que provee la API para manipular los objetos.

Para lograr esto se definieron los siguientes templates .tt:

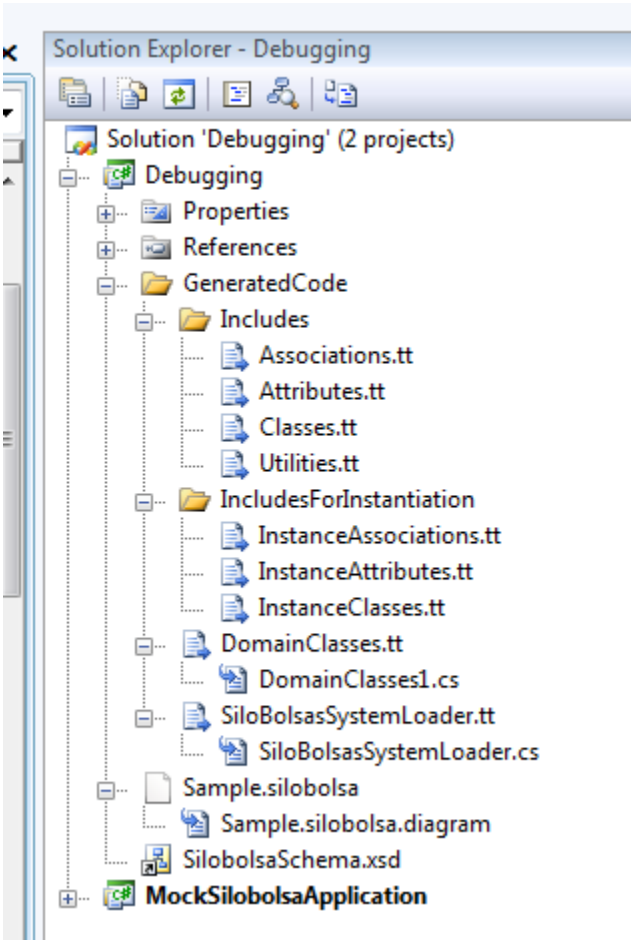

Figura 20.3

Como se puede ver hay varios archivos .tt, sin embargo los dos principales son: DomainClasses.tt y SiloBolsasSystemLoader.tt

20.5.1 DomainClasses.tt

Este es el template encargado de recorrer el modelo y generar las Clases C#. Esto es, definir las estructar de las clases, definir sus propiedas y definir sus relaciones. Para Esto Colabora con la librería externa e utiliza funciones definidas en los templates que están dentro de la carpeta Includes. A continuación se muestra el código de este template.

```
<#@ include file="Includes/Utilities.tt" #>
<#@ include file="Includes/Classes.tt" #>
<#@ include file="Includes/Associations.tt" #>
<#@ include file="Includes/Attributes.tt" #>
<# //*****************************************************#>
<# // Add in the references to the external code library and namespace #>
<#@ Assembly Name="Tesis.Silobolsas.CodeGeneration.dll" #>
<#@ import namespace="Tesis.Silobolsas.CodeGeneration" #>
<# //*****************************************************#>

<#

GenerateHeader();
#>

<#

OpenNamespace("Tesis.Silobolsa.Model", true);
IList<CodeClass> TiposCodeClass=
CodeGenUtilities.GetTiposCodeClass(this.SilobolsaModel.Tipos);
foreach(CodeClass codeClass in TiposCodeClass){
GenerateClass(codeClass);
}

IList<CodeClass> SiloCodeClasses=
CodeGenUtilities.GetSiloCodeClasses(this.SilobolsaModel.Silos);

foreach(CodeClass codeClass in SiloCodeClasses){
GenerateClass(codeClass);
}

#>

<#
CloseNamespace();
#>
```

El template tiene dos partes importantes, una parte es donde se hace los includes de los templates con las funciones para la generación de código, y la la importación de la librería helper `CodeGenUtilities`

La segunda parte es donde se define el código para la generación de las clases. Como se puede ver hay dos foreach. Estos foreaches están relacionados con la clasifiación que se hizo de la información en estatica y dinámica. En primer lugar se recuperan todas las CodeClass correspondientes a las clases asociadas a la información estatica (TipoContenido, VariableMedicion, etc.) Luego se recorren todas estas codeClasses y para cada una se genera el código C# correspondiente. Esto se hace en la siguiente invocación dentro del loop:

```
GenerateClass(codeClass);
```

Èn el segundo foreach se genera el código de las clases para la información que denominamos dinámica. Para esto se le pide a la librería `CodeGenUtilities` que nos de todas las CodeClases asociadas con la informaicon dinámica (SiloBolsa, Sensor,

Contenido, etc.), luego se genera el código de las clases como antes invocando al método

```
GenerateClass(codeClass);
```

La función GenerateClass esta definida en el template Classes.tt que esta dentro de la carpeta Includes. Esta funcion basicamente es una abstraction para generar codigo de clases tomando informacion de una instancia de CodeClass, la definicion de esta funcion es la siguiente:

```
<#+
private void DoGenerateClass(CodeClass codeClass)
{     PushIndent("\t");
      GenerateSummary(codeClass.Summary);
      //GenerateComments(codeClass.Comments);
#>
public partial class <#= codeClass.Name #> : <#= codeClass.BaseClassName
#>
{
}

<#+
      GenerateSummary(string.Concat("Double-derived base class for ",
codeClass.Name));
#>
public <#= CodeGenUtilities.AbstractClassModifier #> partial class <#=
codeClass.BaseClassName #>
{
<#+
      GenerateAttributes(codeClass.Properties);
      GenerateAssociations(codeClass);
#>
}

<#+
      PopIndent();
}

#>
```

Como se puede ver, genera el código con la estructura de la clase, y además invoca a dos funciones

```
      GenerateAttributes(codeClass.Properties);
      GenerateAssociations(codeClass);
```

Estas funciones estan definidas en los templates Attributes.tt y Associations.tt respectivamente. Estan funciónes generan el código para representar una variables de instancia de un tipo simple o una varible de instancia que es una asociación.

Todas las clases generadas están un el archico llamado DomainClasses1.cs.

20.5.2 SiloBolsasSystemLoader.tt

Este template es el encargado de recorrer el modelo y generar todas las instancias de las clases generadas con el template anterior. Además también genera el código de la clase Loader que provee acceso al modelo de objetos.

La lógica de este template es un poco más complicada que el del anterior, dado que no solo tiene que recorrer el modelo y generar el código con la instancion de las clases, sino que también debe asocias todas estas intancias en forma consistente, para arma el grafo de objetos que representan la información del modelo.

Generación de objetos para representar Información Estática

Para esto en primer lugar se crean las instancias de la información estática, es decir (TipoContenido, TipoSensor, etc.), ya que entre ellas no hay dependencias:

```
<#
foreach(Tipo tipo in this.SilobolsaModel.Tipos)
 {

   GenerateInstanceOfTipo(tipoCodeClass, tipo);

     }
   }

}
```

Básicamente se recorre todos los tipos del modelo y se crean las instancias para cada uno de ellos invocando la siguiente función

```
   GenerateInstanceOfTipo(tipoCodeClass, tipo);
```

Esta función está definida en el template InstanceClasses.tt dentro de la carpeta llamada IncludesForInstantiaon.

Como se puede ver esta función recibe dos parámetros, tipo es el elemento de DSL y tipoCodeClass es el wrapper asociado con este tipo. Dentro de esta función se inicializaran las propiedades de la instancia.

Generación de objetos para representar Información Dinámica

Una vez creadas todas las instancias necesarias para representar la información estática, hay que crear todas las instancias para representar la información dinámica (Silobolsas, contenidos, sensores, etc.). Todas estas instancias tienen una estructura de árbol, donde la raíz es la silobolsa. Por esto para la generación de instancias se una silobolsa y se recorren todas sus asociaciones como un árbol y a medida que se avanza se crea una instancia por cada asociación de la silobolsa (Sensor, Contenido, etc.).

```
foreach(SiloBolsa siloBolsa in this.SilobolsaModel.Silos)
   {

      GenerateInstanceOfSilobolsa(siloBolsa,
siloCodeClass,ContenidoCodeClass,sensorCodeClass);

   }

 #>
```

Como se ve en el código se recorren todas las silobolsas del modelo y se generan las instancias y la de las sus relaciones a partir de la invocación de la siguiente función:

```
GenerateInstanceOfSilobolsa(siloBolsa,
siloCodeClass,ContenidoCodeClass,sensorCodeClass);
```

Esta función también esta definida dentro del template InstancesClasses.tt.

Generación SiloBolsasSystemLoader

Además el template SiloBolsasSystemLoader.tt genera el código del loader que contiene todas estas instancias y provee acceso al grafo de objetos generados.

Todos estos templates son ejecutados cuando se presiona el botón Tranform all Templates.

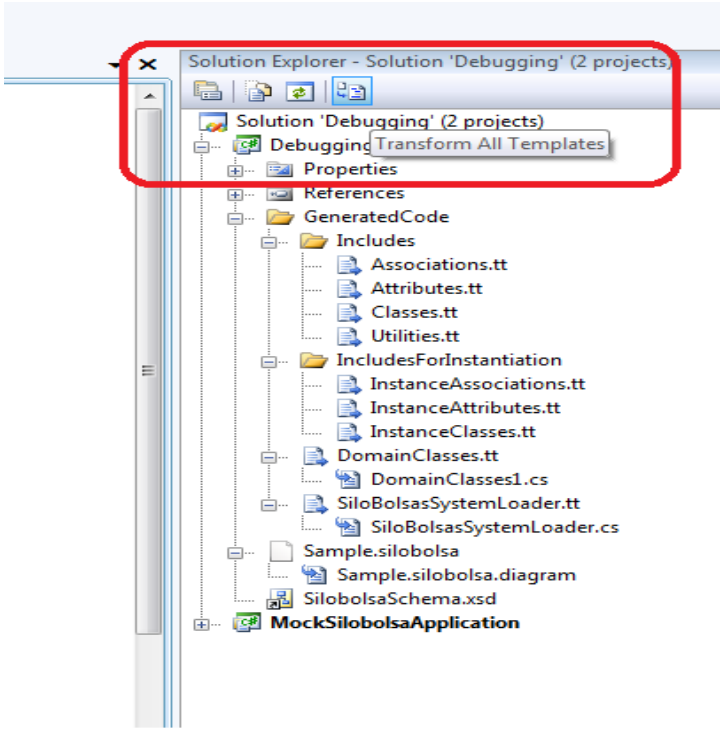

Figura 20.4

Capitulo 21 Definición del Framework de Dominio

21.1 Aplicación para las silobolsas

Se definió una aplicación simple de manera de poder mostrar el concepto de integrar los artefactos generados con una framework. Esta aplicación provee la siguiente funcionalidad:

- Listado en pantalla de todas las silobolsas del sistema
- Detalle de las propiedades de una silobolsa
- Listado en pantalla de los sensores de una silobolsa
- Listado de todas las mediciones hechas por los sensores de una silobolsa
- Indicar a los sensores que registren las mediciones
- Exportación a Excel

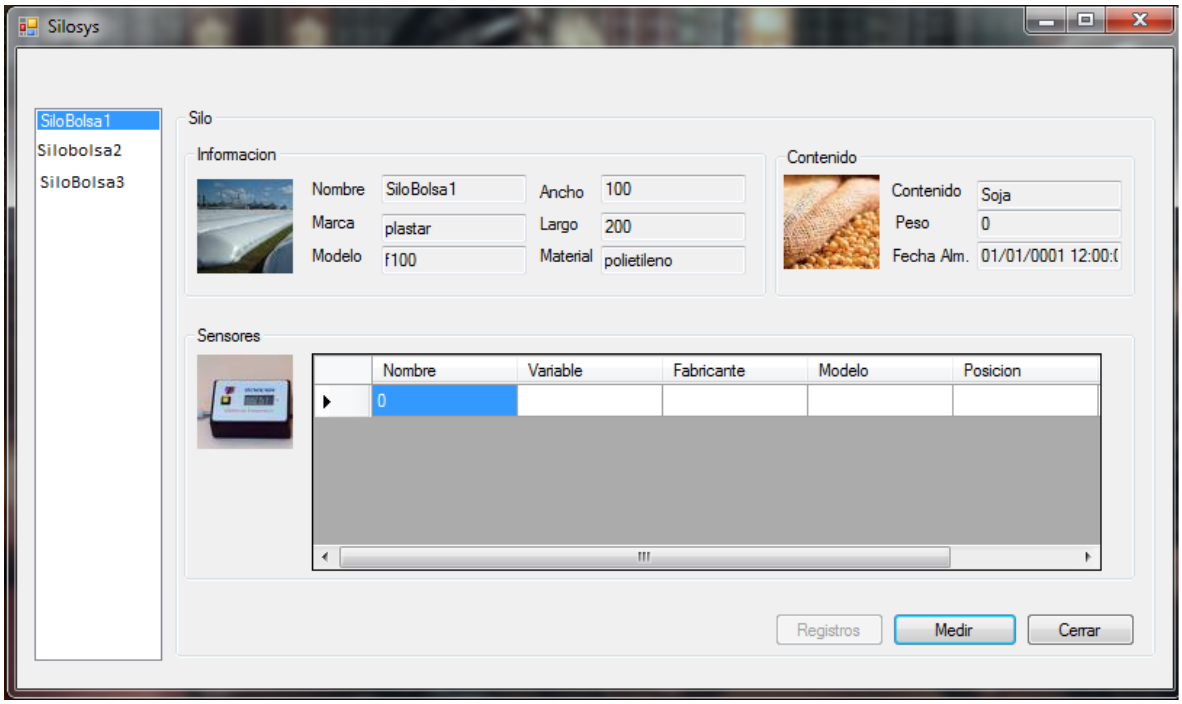

Figura 21.1

Esta es la pantalla principal de la aplicación donde a la izquierda se listan todas las silobolsas. Cuando una silobolsa es selecciona su información es mostrada.

El botón Medir hace que los sensores de la silobolsa seleccionada tomen mediciones y generen registros.

Si una silobolsa posee registros de medición presionando el botón Registros se despliega una nueva pantalla mostrando la información de los registros

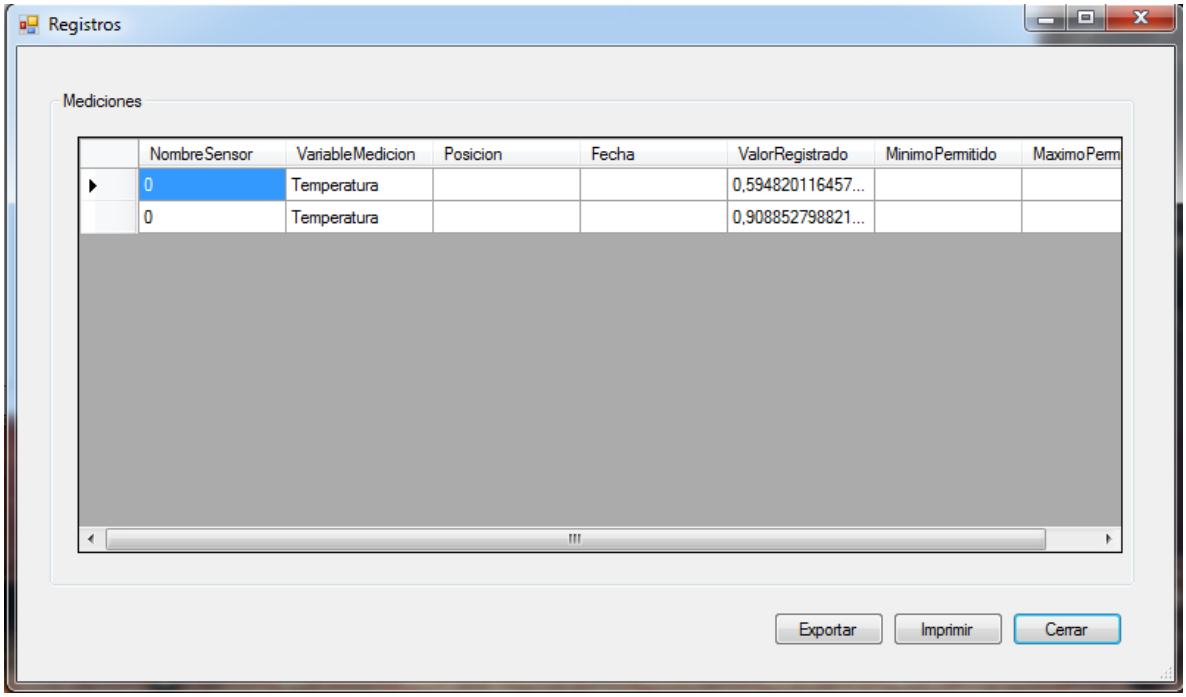

Figura 21.2

Esta nueva pantalla tiene un botón Exportar por el cual se permite generar un archivo Excel con la información de los registros.

21.2 Importando el SilobolsaSystem

La utilización desde la aplicación de las instancias generadas mediante el loader, se realiza simplemente importando el loader en MockSiloBolsaApplication

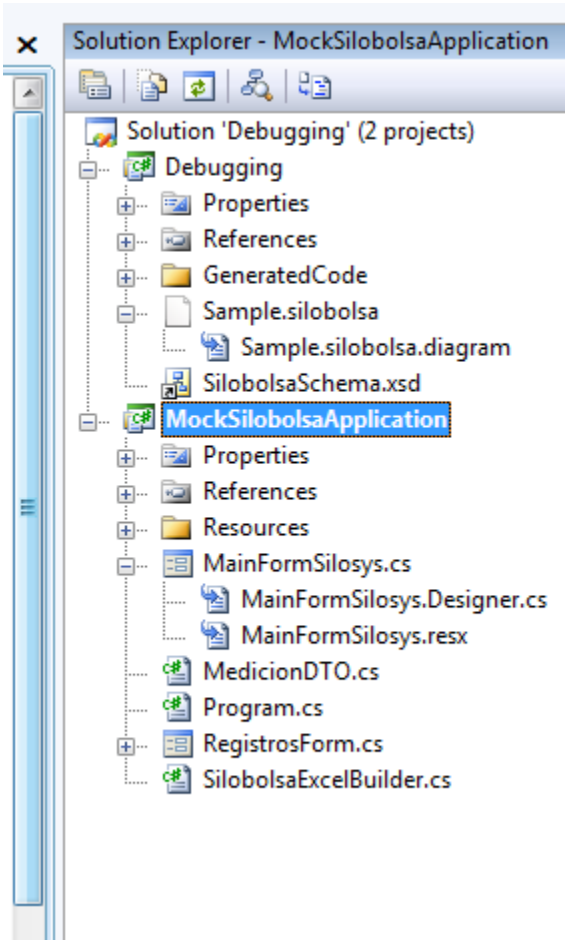

Figura 21.2

Esta aplicación tiene una clase principal MockSilobolsaApplication.cs en la cual se importa el loader con las silobolsas.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using WindowsFormsSilosys;
using Tesis.Silobolsa.Model;

namespace MockSilobolsaApplication
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {

            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
```

```
            MainFormSilosys MainFormSilosys = new
MainFormSilosys(SilobolsasSystem.GetSilobolsasSystem());
            Application.Run(MainFormSilosys);
        }
    }
}
```

Capitulo 22 – Conclusiones

Las silobolsas son un sistema de almacenamiento de granos que está siendo cada vez más adoptada por los productores en muchos países.

La pérdida de la calidad del grano es altamente dependiente de la duración del almacenamiento y los niveles de humedad, temperatura y dióxido de carbono en la silobolsa.

Las investigaciones confirman que las silobolsas, aunque tienen algunas limitaciones, ofrecen a los productores un medio relativamente barato y una solución fiable de almacenamiento de granos. Por lo tanto, es importante trabajar en pro de la mejora de la tecnología de las silobolsas. En particular, la incorporación de sistemas de software sería un activo valioso.

En esta tesis se demostró que es posible construir un sistema de red de sensores que permita monitorear el contenido en silobolsas mediante la utilización de la metodología DSM de manera efectiva. Ya que dada la independencia entre los modelos y el código, la metodología DSM soporta naturalmente la evolución de la arquitectura subyacente (plataforma de ejecución y framework de dominio) sobre la cual se implementa el sistema, reutilizando en su totalidad o en gran parte los modelos generados. De este modo, el software especificado es atemporal en términos tecnológicos y las representaciones de alto nivel que lo definen siguen siendo válidas mientras el dominio del problema siga siendo el mismo. La proximidad modelo-dominio también tiene como ventaja el hecho de que los modelos funcionen como artefactos tanto de captura de requerimientos como de especificación de software. En consecuencia, los clientes (productores rurales) pueden participar con mayor actividad en el desarrollo.
Debido a que el código fuente es producido por los generadores automáticamente, dicho código se supone libre de errores y eficiente en el uso de recursos de ejecución y memoria, por lo que no necesita ser modificado u optimizado. Adicionalmente, todo el código generado seguirá un único estilo de programación y diseño. La faceta desfavorable de esta propuesta consiste en que es necesario invertir trabajo extra en el desarrollo de los componentes de la arquitectura de DSM (meta-modelos, herramientas, generadores, framework de dominio, etc.). Este esfuerzo inicial puede ser amortizado considerablemente debido al incremento en la productividad en el desarrollo alcanzado y aumenta proporcionalmente a la cantidad de "repeticiones" que se tenga en los desarrollos de software llevados a cabo dentro del mismo dominio.

Uno de los aportes de la presente tesis, es detallar como la tecnología DSL Tools da soporte a DSM y como esta puede ser utilizada para construir un DSL en un caso concreto.

Las actividades para la definición de un DSL incluyen:

- Definición del DSL
  - Identificación de los conceptos a Modelar
  - Definición del modelo de dominio
  - Definición de la notación para el lenguaje
  - Definición de las reglas del lenguaje
- Definición del generador de código
- Definición del Framework de Dominio

Las DSL Tools conforman una tecnología poderosa y eficaz que permite definir lenguajes específicos de dominio propios y crear herramientas de modelado basadas en esos lenguajes.

De esta manera es posible:

- Definir fácilmente un modelo de dominio que determina que combinaciones entre elementos y relaciones son validas.
- Realizar transformaciones sobre el modelo y generar automáticamente código fuente especifico del mismo.
- Desarrollar un lenguaje de notación gráfica con un conjunto de elementos que representan los elementos de nuestro dominio, conectores y formas en el diagrama.

Los aspectos más importantes que del dominio de silobolsas para la construcción del DSL son:

- Materiales utilizados en la construcción de silobolsas: cómo influyen respecto a factores ambientales y climáticos internos y externos de las mismas.
- Características de las silobolsas: tamaño en largo, ancho y altura, capacidad en toneladas dependiendo del tipo de cereal o forraje, etc.
- Forma de almacenamiento de granos en silobolsas.
- Valores de medición: dependiendo del tipo de grano almacenado, existen diferentes rangos de medición que denotan la correctitud del proceso de almacenamiento.
- Ubicación de sensores en silobolsas: estos pueden ubicarse a diferentes alturas y distancias a lo largo de la silobolsa, habiendo puntos estratégicos que ayudan a efectuar una buena y correcta toma de valores de medición.
- Características generales de sensores medición: variable de medición (humedad, temperatura, etc.), configuración de los mismos.

El artefacto final creado en esta tesis es un lenguaje de notación gráfica simple de usar por un operador con escasos conocimientos técnicos, el cual brinda un gran aporte a la industria agropecuaria, permitiendo sistematizar los controles realizados sobre el almacenamiento de granos en silobolsas, permitiendo al usuario armar su red de silobolsas y sensores especifica de manera simple e intuitiva, configurar en la misma las características relevantes que sean necesarias y realizar reportes estadísticos necesarios a la hora de tomar decisiones sobre el contenido almacenado.

References

[Kelly08] Steven Kelly, Juha-Pekka Tolvanen. Domain-Specific Modeling. John Wiley & Sons, Inc. 2008.

[Cook07] Steve Cook, Gareth Jones, Stuart Kent, Alan Cameron Wills.Domain Specific Development with Visual Studio DSL Tools. Addison Wesley. 2007.

[Marinissen] J.Marinissen , C. Pons , J. Pons , S. Orionte , Características Higroscópicas de Forrajes Almacenados en Bolsas Plásticas. Extraído de www.inta.gov.ar en octubre de 2009.

[Cardoso09] Marcelo L. Cardoso, Agr. Ricardo E. Bartosik, Juan C. Rodríguez, "Estudio de la Evolución de la Humedad de los Granos Individuales en Silo-bolsas de Maíz y Soja", Extraído de www.inta.gov.ar en octubre de 2009.

[Rodríguez09] Rodríguez, J. C., Bartosik, R. E. Malinarich H.D. EEA INTA Balcarce, "Almacenaje de Granos en Bolsas Plásticas: Sistema Silobag Informe Final de Trigo", Extraído de www.inta.gov.ar en julio de 2009.

[Bartosik10] Bartosik, Ricardo, Juan Carlos Rodríguez, Hector Malinarich y Leandro Cardoso, INTA EEA Balcarce, “Almacenaje de maíz, trigo, soja y girasol en bolsas plásticas herméticas”. Extraído de www.inta.gov.ar en octubre de 2009.

[Bartosik08] Ricardo Bartosik, Leandro Cardoso, Darío Ochandio y Diego Croce, “Detección temprana de procesos de descomposición de granos almacenados en bolsas de plástico herméticas mediante la medición de CO2”, Extraído de www.inta.gov.ar en julio de 2009.

[Bartosik01]Bartosik, R.E., Maier, D.J. y Rodríguez, J.C. 2001. Effects of CO2 Dosage and Exposure Time on the Mortality of Adult and Immature Stages of Sitophilus oryzae. Enviado al congreso de ASAE 2001. Paper Nº 01-6110.

[White93]White, N.D.G. y Jayas D.S. 1993. Effectiveness of carbon dioxide in compressed gas or solid formulation for the control of insects and mites in stored wheat and barley. Phytoprotection 74:101-111.

Bibliografía

Jack Greenfield, "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools", Microsoft Corporation
Noviembre 2004

Gunther Lenz and Christoph Wienands, Practical Software Factories in .NET, Apress
Julio 2006

National Institute of Agriculture (INTA) http://www.inta.gov.ar. Argentina.
DSLTools http://msdn.microsoft.com/en-us/vstudio/cc677260.aspx