

**ESPECIFICACIÓN FORMAL EN OCL DE REGLAS DE CONSISTENCIA ENTRE  
EL MODELO DE CLASES Y DE CASOS DE USO DE UML**

Guillermo González Calderón  
Ingeniero de Sistemas e Informática

Director:  
Carlos Mario Zapata Jaramillo, M.Sc, Ph.D(C)  
Profesor Escuela de Sistemas

Codirector:  
Fernando Arango Isaza, Ph.D  
Profesor Escuela de Sistemas



Universidad Nacional de Colombia  
Facultad de Minas - Escuela de Sistemas  
Maestría en Ingeniería de Sistemas - Área de Ingeniería de Software  
Medellín, Colombia  
Marzo de 2007

## Resumen

En el ciclo de vida del software, durante las fases de definición y análisis se realiza una especificación de los requisitos. Para ello, es necesario realizar un proceso de captura de las necesidades y expectativas de los interesados, que se traduce posteriormente en un conjunto de modelos que representan tanto el problema como su solución. Por lo general, la mayoría de esos modelos se expresan en el Lenguaje Unificado de Modelamiento (UML), dirigido por el Grupo de Gestión de Objetos (Object Management Group, OMG). UML define un conjunto de artefactos que permiten especificar los requisitos del software, los cuales deberían guardar consistencia cuando se traten del mismo modelo, ya que están definidos bajo las reglas de buena formación (Well-Formedness Rules, WFR), las cuales se encuentran definidas dentro de la especificación de UML en el Lenguaje de Restricciones de Objetos (Object Constraint Language, OCL). La consistencia interna de cada artefacto está por tanto definida en la especificación de UML y algunas de las herramientas CASE (Computer-Aided Software Engineering) utilizan esas reglas intramodelo para validar este tipo de consistencia. Sin embargo la consistencia entre diferentes artefactos no se encuentra definida en la especificación de UML y poco se ha trabajado con este tipo de consistencia; además, los trabajos que se han realizado en este tema se identifican por:

- No se suele definir de manera formal la consistencia entre los diferentes artefactos.
- Por lo general se estudia sólo la consistencia entre alguno de los artefactos y el código resultante.
- Algunos establecen reglas formales de consistencia, pero únicamente a nivel de intramodelo.

Entre los artefactos más utilizados para especificar una pieza de software se encuentran el diagrama de clases y el diagrama de casos de uso, los cuales suministran dos perspectivas diferentes del desarrollo de software: por un lado el diagrama de clases, de tipo estructural, muestra los objetos del mundo y sus relaciones; por otro lado, el diagrama de casos de uso, de

tipo comportamental, se concentra en las funciones que realizan los actores del mundo para lograr un resultado en el software. Estos dos puntos de vista deberían ser complementarios y, por ello, tener información común, susceptible de ser sometida a un análisis de consistencia.

En esta Tesis se propone un método para verificar la consistencia entre el diagrama de clases y el diagrama de casos de uso de UML de una manera formal. Dicho proceso se lleva a cabo evaluando una serie de reglas definidas en OCL que se deben cumplir para garantizar que la información brindada por dichos modelos sea consistente. Como se reconoce la participación de los dos diagramas en la elaboración de las Interfaces Gráficas de Usuario (Graphical User Interfaces, GUI), se define adicionalmente la consistencia con este artefacto. Las reglas se implementaron en XQuery, utilizando como diagramas de entrada representaciones en XML generadas con la herramienta CASE ArgoUML® (en el caso del diagrama de clases y de casos de uso) y con Microsoft Visio® (en el caso de las GUI). Finalmente, se muestra un caso de estudio donde se aplican estas reglas y se muestran los posibles errores y advertencias que se tienen entre los elementos de tales artefactos.

# Abstract

Requirements specification is made throughout the definition and the analysis phases of software development lifecycle. For completing this task, a capture process of the stakeholder's needs and expectations is needed; the results of this process are then translated into representative diagrams, for modeling both the problem domain and the solution. Commonly, most of these diagrams are expressed in the Unified Modelling Language (UML), a directed standard of the Object Management Group (OMG). UML defines a set of artifacts for specifying the software requirements; the UML artifacts are defined by means of Well-Formedness Rules (WFR) in the Object Constraint Language (OCL) and, consequently, must be consistent with each other. The intra-model consistency of each artifact is guaranteed by the UML specification, and some of the CASE (Computer-Aided Software Engineering) tools use intra-model rules for consistency checking. However, consistency among different UML artifacts is not defined by the UML specification, and the research about inter-model consistency is still immature. Some of the issues for this research topic are:

- Inter-model consistency is not usually defined in a formal way.
- In most cases, only consistency between one artifact and the resultant code is studied.
- Some of the works in this area establish formal consistency rules, but only in the intra-model level.

Two of the most useful artifacts for making a specification of a software piece are class and use case diagram. These diagrams present two different viewpoints of software development: firstly, class diagram gives a structural viewpoint, and it can represent the objects of the world and its relationships; secondly, the use case diagram gives a behavioural viewpoint, and it concentrates in the functions that the actors of the world carry out for achieving a result in the software piece. These two viewpoints should be complementary and, as a consequence, they should contain information in common. This information is susceptible of being checked by means of a consistency analysis.

We propose in this Thesis a method for checking the consistency of UML class diagram against use case diagram in a formal way. For completing this task, we define a set of rules in the OCL, which must be fulfilled in order to guarantee that the offered information of those models is consistent. The two mentioned diagrams contribute in the elaboration of the Graphical User Interfaces (GUI); for this reason, we define the consistency with this artifact too. The rules were implemented in XQuery, using XML representations as input diagrams (class and use case diagrams), which are generated by the ArgoUML® CASE tool; also, we use XML code generated by Microsoft Visio® for the GUIs. Finally, we present a case study in order to apply these rules, and to exhibit the possible errors and warnings obtained for the elements of such artifacts.

# Agradecimientos

Gracias principalmente a Dios por permitirme llegar hasta este punto de mi vida, por darme la oportunidad de estudiar la Maestría en Ingeniería de Sistemas, y por haberme puesto en el camino personas tan valiosas para mi formación personal y profesional.

Mis más sinceros agradecimientos al que más que un Director, ha sido mi amigo, Carlos Mario Zapata, por su apoyo y por su invaluable aporte en el desarrollo de esta tesis.

A mi familia, por su contribución en mi formación personal y por su continuo apoyo, no solo económico sino también moral, durante estos años de estudio y especialmente durante la realización de esta Tesis de Maestría.

Gracias también a los profesores Fernando Arango, Juan David Velásquez, Francisco Moreno, Santiago Montoya (Q.E.P.D.) y John William Branch por todo el apoyo que me brindaron durante este tiempo.

Finalmente quiero agradecer también a mis amigos y compañeros: Roberto, Alexander, Sandra, Jorge, Germán, Elizabeth, Miguel y Carlos.

# Índice General

<b>INTRODUCCIÓN</b> .....	<b>1</b>
<b>MARCO TEÓRICO</b> .....	<b>6</b>
<b>2.1 MODELO DE CASOS DE USO</b> .....	<b>6</b>
<b>2.1.1. Elementos de un diagrama de Casos de Uso</b> .....	<b>7</b>
2.1.1.1. Actores .....	7
2.1.1.2. Caso de Uso.....	8
<b>2.1.2. Pasos para definir un Caso de Uso</b> .....	<b>8</b>
<b>2.1.3 Diagramas de Casos de Uso: Notación UML</b> .....	<b>10</b>
2.1.3.1 Herencia .....	10
2.1.3.2 Inclusión.....	11
2.1.3.3 Extensión .....	11
<b>2.2 DIAGRAMA DE CLASES</b> .....	<b>12</b>
<b>2.2.1. Elementos de un Diagrama de Clases</b> .....	<b>13</b>
2.2.1.1. Clases .....	13
2.2.1.1.1. Atributos.....	14
2.2.1.1.2. Operaciones .....	14
2.2.1.1.3. Notación UML para las clases.....	15
2.2.1.2. Relaciones .....	16
<b>2.2.2. Diagrama de Clases para el Sistema de Puntos de Venta</b> .....	<b>17</b>
<b>2.3 INTERFACES</b> .....	<b>18</b>
<b>2.3.1 Interfaces Gráficas de Usuario</b> .....	<b>18</b>
2.3.1.1 Botones de comando .....	20
2.3.1.1.1 Botón por defecto .....	20
2.3.1.1.2 Botón de cancelación.....	20
2.3.1.2 CheckBox (Listas de chequeo) .....	21
2.3.1.3 Radio Button (Listas de opciones).....	21
2.3.1.4 Combo Box (Lista Combinada).....	22
<b>2.3.2 Componentes para texto</b> .....	<b>22</b>
2.3.2.1 Label (Etiqueta) .....	22
2.3.2.2 Text Field (Cuadro de Texto) .....	23
<b>2.3.3 Modelo de Interfaces</b> .....	<b>23</b>
<b>2.4 OCL</b> .....	<b>25</b>
2.4.1 Lenguaje de expresión .....	25
2.4.2 Lenguaje de modelamiento .....	25
2.4.3 Lenguaje Formal .....	26
2.4.5 En dónde usar OCL.....	26
2.4.6 Conexión con el metamodelo UML.....	28
2.4.6.1 self .....	28
2.4.6.2 Características Estáticas.....	28
2.4.6.2.1 Pre y Post-condiciones .....	28
<b>2.4.7 Objetos y propiedades</b> .....	<b>29</b>
2.4.7.1 Propiedades .....	29
2.4.7.1.1 Atributos.....	30
2.4.7.1.2 Operaciones.....	30
2.4.7.1.3 Extremo de asociación y navegación .....	30

2.4.7.2 Navegación hacia tipo asociación.....	31
2.4.7.3 Navegación desde la asociación como clase.....	32
2.4.7.4 Expresiones generales.....	32
2.4.7.5 Tipos predefinidos en OCL.....	32
2.4.7.5.1 Collection.....	33
2.5 XQUERY.....	33
2.5.1. Descripción del Lenguaje.....	33
2.5.2 XPath.....	34
2.5.3 Modelo de Datos.....	36
2.5.4 Expresiones FLWOR.....	38
2.5.4.1 For.....	38
2.5.4.2 Let.....	38
2.5.4.3 Where.....	38
2.5.4.4 Order by.....	38
2.5.4.5 Return.....	38
2.6. METAMODELOS.....	39
2.6.1. Metamodelo del diagrama de Espacios de Nombres del paquete Kernel.....	40
2.6.2. Metamodelo del diagrama de Clasificadores del paquete Kernel.....	40
2.6.3 Metamodelo de diagrama de clases.....	43
2.6.4. Metamodelo de Diagrama de Casos de Uso.....	43
2.6.5. Metamodelo del Modelo de Interfaces.....	45
<b>PROBLEMÁTICA GENERAL DE INVESTIGACIÓN.....</b>	<b>48</b>
3.1 LIMITACIONES ENCONTRADAS.....	48
3.2 DEFINICIÓN DEL PROBLEMA.....	49
3.3 OBJETIVOS.....	50
3.3.1 Objetivo General.....	50
3.3.2 Objetivos Específicos.....	50
<b>REVISIÓN DE LA LITERATURA.....</b>	<b>51</b>
<b>MÉTODO PROPUESTO.....</b>	<b>60</b>
5.1. DEFINICIÓN DE REGLAS DE CONSISTENCIA.....	60
5.1.1 Regla 1.....	61
5.1.2 Regla 2.....	62
5.1.3 Regla 3.....	63
5.1.4 Regla 4.....	64
5.1.5 Regla 5.....	65
5.1.6 Regla 6.....	66
5.1.7 Regla 7.....	67
5.1.8 Regla 8.....	68
5.2 VALIDACIÓN DE REGLAS DE CONSISTENCIA.....	69
5.2.1 Integración de sinónimos.....	69
5.2.2 Consulta Xquery.....	70
5.3 COMPARACIÓN CON OTROS TRABAJOS.....	74
<b>RESULTADOS.....</b>	<b>76</b>
6.1 CASOS DE USO PARA EL SISTEMA DE RESERVACIONES DE VUELO.....	78
6.2 DIAGRAMA DE CLASES PARA EL SISTEMA DE RESERVACIONES DE VUELO.....	80
6.3 INTERFACES PARA EL SISTEMA DE RESERVACIONES DE VUELO.....	85
6.4 VALIDACIÓN DE REGLAS DE CONSISTENCIA.....	89
6.4.1 Análisis de Resultados.....	92
6.4.1.1 Regla 1.....	92

6.4.1.2 Regla 2.....	92
6.4.1.3 Regla 3.....	92
6.4.1.4 Regla 4.....	92
6.4.1.5 Regla 5.....	92
6.4.1.6 Regla 6.....	93
6.4.1.7 Regla 7.....	93
6.4.1.8 Regla 8.....	93
<b>CONCLUSIONES.....</b>	<b>95</b>
7.1 CONTRIBUCIONES PRINCIPALES .....	95
7.1.1 <i>Definición de reglas Intermodelos</i> .....	95
7.1.2 <i>Especificación Formal</i> .....	96
7.1.3 <i>Integración con Interfaces Gráficas de Usuario</i> .....	96
7.1.4 <i>Integración con Xquery</i> .....	97
7.1.4 <i>Transformación de OCL a XQuery</i> .....	97
7.1.4 <i>Utilización de diccionario de sinónimos</i> .....	97
7.1.5 <i>Resultados Vía Web</i> .....	98
7.2 LIMITACIONES DEL MÉTODO PROPUESTO .....	98
7.2.1 <i>Include y Extend</i> .....	98
7.2.2 <i>Herencia</i> .....	98
7.3 TRABAJO FUTURO.....	98
7.3.1 <i>Extender el método a otros Modelos</i> .....	99
7.3.2 <i>Integración de OCL en UN-Método</i> .....	99
7.3.3 <i>Corpus</i> .....	99
<b>BIBLIOGRAFÍA.....</b>	<b>100</b>

# Índice de Figuras

<i>Figura 2. 1 Representación de un caso de Uso.....</i>	<i>10</i>
<i>Figura 2. 2 Representación de la relación de herencia entre casos de Uso .....</i>	<i>10</i>
<i>Figura 2. 3 Representación de la relación de inclusión entre casos de Uso .....</i>	<i>11</i>
<i>Figura 2. 4 Representación de la relación de extensión entre casos de Uso .....</i>	<i>11</i>
<i>Figura 2. 5 Ejemplo de relaciones de extensión e inclusión entre casos de uso.....</i>	<i>12</i>
<i>Figura 2. 6 Representación de una clase en UML.....</i>	<i>16</i>
<i>Figura 2. 7 Diagrama de Clases del Sistema de Puntos de Venta (PDV).....</i>	<i>17</i>
<i>Figura 2.8. Ejemplo de Botón de Comando.....</i>	<i>20</i>
<i>Figura 2.9. Ejemplo del comando Checkbox.....</i>	<i>21</i>
<i>Figura 2.10. Ejemplo de Radio Buttons. ....</i>	<i>22</i>
<i>Figura 2.11. Ejemplo de combo box. ....</i>	<i>22</i>
<i>Figura 2.12. Ejemplos de labels y text fields. ....</i>	<i>23</i>
<i>Figura 2.13 Interfaz para Procesar Venta.....</i>	<i>24</i>
<i>Figura 2. 14 Diagrama de Clases para el juego PACMAN.....</i>	<i>27</i>
<i>Figura 2.15 Metamodelo del elemento Namespace.....</i>	<i>41</i>
<i>Figura 2.16 Metamodelo del elemento Classifier.....</i>	<i>42</i>
<i>Figura 2.17 Metamodelo del Diagrama de Clases.....</i>	<i>44</i>
<i>Figura 2.18 Metamodelo de Casos de Uso.....</i>	<i>45</i>
<i>Figura 2.19 Metamodelo para el modelo de Interfaces.....</i>	<i>47</i>
<i>Figura 4. 1 Ejemplo de conjunto de documentos .....</i>	<i>52</i>
<i>Figura 4. 2 Restricción de ejemplo .....</i>	<i>52</i>
<i>Figura 4. 3 Hipervínculos resultantes.....</i>	<i>53</i>
<i>Figura 4. 4 Explorador de proyectos, panel de salida y un diagrama de clases. ....</i>	<i>55</i>
<i>Figura 4. 5 Dos conflictos de nombre en el Namespace. ....</i>	<i>56</i>
<i>Figura 5.1. Descripción gráfica de la Regla 1. ....</i>	<i>61</i>
<i>Figura 5.2. Descripción gráfica de la Regla 2. ....</i>	<i>62</i>
<i>Figura 5.3. Descripción gráfica de la Regla 3. ....</i>	<i>63</i>
<i>Figura 5.4. Descripción gráfica de la Regla 4. ....</i>	<i>64</i>
<i>Figura 5.5. Descripción gráfica de la Regla 5. ....</i>	<i>65</i>
<i>Figura 5.6. Descripción gráfica de la Regla 6. ....</i>	<i>66</i>
<i>Figura 5.7. Descripción gráfica de la Regla 7. ....</i>	<i>67</i>
<i>Figura 5.8. Descripción gráfica de la Regla 8. ....</i>	<i>68</i>
<i>Figura 6.1. Casos de uso para el sistema de reservaciones de vuelo .....</i>	<i>78</i>
<i>Figura 6.2 Diagrama de Clases para el sistema de reservaciones de vuelo propuesto por Weitzenfeld .....</i>	<i>81</i>
<i>Figura 6.3 Diagrama de Clases para el sistema de reservaciones de vuelo (modificado).....</i>	<i>82</i>
<i>Figura 6.4. Pantalla de Menú de Servicios.....</i>	<i>85</i>
<i>Figura 6.5. Pantalla de Registro de Usuario por Primera Vez.....</i>	<i>86</i>
<i>Figura 6.6. Pantalla de Registro de Tarjeta por Primera Vez.....</i>	<i>86</i>
<i>Figura 6.7 Reglas de Consistencia entre el diagrama de clases y el diagrama de casos de uso de UML .....</i>	<i>94</i>

# Índice de Tablas

<i>Tabla 2. 1 Eventos Externos, Actores y Objetivos del Sistema.</i> .....	9
<i>Tabla 2. 2 Tipos predefinidos en OCL</i> .....	32
<i>Tabla 2. 3 Ejemplos de colecciones de OCL</i> .....	33
<i>Tabla 5.1 Comparación de trabajos afines al método propuesto (parte 1 de 2)</i> .....	74
<i>Tabla 5.2 Comparación de trabajos afines al método propuesto (parte 2 de 2)</i> .....	75

# Capítulo 1

## Introducción

Un proceso de desarrollo de software tiene como propósito la producción eficaz y eficiente de un producto software que cumpla con las necesidades y expectativas de los interesados o participantes (*stakeholders* es el término en inglés). Este proceso empieza típicamente cuando se identifica un problema que puede requerir una solución computarizada, cuyo desarrollo requiere una especificación de requisitos que se logra durante las fases de definición y análisis. Esta especificación es a menudo informal y posiblemente vaga, lo que se suele denominar un “bosquejo áspero” (Jackson, 1995). Los ingenieros de requisitos necesitan examinar esta representación escrita, que es frecuentemente incompleta e inconsistente, basados en la información disponible y en su experiencia previa, para transformar este “bosquejo áspero” en una especificación correcta de requisitos. Luego, se presentan dichos requisitos a los interesados para su validación. Como resultado, se identifican nuevos requisitos para ser agregados a la especificación, o algunos de los previamente identificados podrían eliminarse para mejorarla; por tanto, en cada paso del desarrollo de una pieza de software, la especificación puede perder o ganar requisitos. Una de las tareas críticas de los ingenieros de requisitos en este proceso es asegurar que la especificación de requisitos permanezca correcta en cada paso, o que por lo menos los errores se encuentren lo más rápido posible y que sean identificadas sus fuentes para su revisión (Zowghi, *et al.*, 2002).

En general, los métodos de desarrollo de software (por ejemplo el *Unified Process* (Jacobson, *et al.*, 1999)), son iterativos e incrementales, repitiendo una serie de iteraciones sobre el ciclo de vida de un sistema. Cada iteración consiste de un paso a través de las etapas de requerimientos, análisis, diseño, implementación y prueba. El resultado de cada iteración representa un incremento sobre cada uno de los modelos construidos en las etapas anteriores. Durante estos ciclos de desarrollo, los modelos se construyen sucesivamente en un nivel más y más bajo de abstracción hasta que el nivel requerido para la codificación es alcanzado. El

costo de los errores encontrados al principio en el ciclo de desarrollo es muy inferior al costo de los errores encontrados en los últimos pasos tales como codificación o pruebas, por lo que se debe tratar de hacer un buen análisis desde el principio del proceso, especialmente en la especificación de requisitos, para evitar problemas futuros que requerirían incluso un nuevo diseño, perdiendo así gran parte del trabajo realizado hasta ese momento.

Es frecuente el caso en que, en una tentativa por mantener la consistencia dentro de los requisitos, se eliminen uno o más de la especificación y no se pueda preservar su integridad. Inversamente, cuando se agregan nuevos requisitos a la especificación para hacerla más completa, es posible introducir inconsistencias en ella (Zowghi, *et al.*, 2002). Dado que la especificación de los requisitos se logra con un conjunto de diagramas, que representan vistas interconectadas del modelo del problema, las inconsistencias pueden surgir entre cualquier par de diagramas, y se pueden acentuar en tanto esos artefactos se usan con mayor frecuencia, como es el caso de los diagramas de casos de uso y de clases.

A medida que se utilizan más artefactos, el problema de la consistencia entre ellos aumenta. Muy pocas técnicas de modelamiento de requisitos tienen una aproximación sistemática para tratar el problema de la consistencia intermodelos (Egyed, 2001). El problema de la consistencia es simplemente ignorado (y dejado a los ingenieros de requisitos y a los interesados, que tienen que validar la especificación de estos con procesos muchas veces manuales).

Después de haber reunido los requisitos, se debe hacer un modelamiento del problema para obtener una visión más detallada del sistema y así poder solventar por medio del computador las diferentes situaciones que se presentan en el problema a resolver. En este proceso, es preferible trabajar con un estándar de modelamiento, que sea compartido y comprendido por los analistas de diferentes sitios. Utilizando las reglas proporcionadas por el estándar, los Ingenieros de Software pueden crear modelos concretos y sin ambigüedades. Booch, Rumbaugh y Jacobson han definido un estándar de modelado denominado UML (Unified Modeling Language) (OMG, 2007). UML está conformado por un conjunto de artefactos que permiten especificar los requisitos del software. La forma de representar a UML es conocida

como metamodelo de UML, y está disponible al público junto con la definición del estándar en inglés (OMG, 2007). El metamodelo de UML sirve para que los Ingenieros de Software puedan verificar la corrección de sus modelos, ya que tiene la estructura y las reglas que definen las interacciones entre los diferentes diagramas.

UML propone, entre otras cosas, un conjunto de diagramas que representan un software desde distintos puntos de vista. Los diagramas UML son independientes pero se conectan; su metamodelo los describe bajo el mismo techo (OMG, 2007). Por ejemplo, un diagrama de casos de uso (Jacobson, 1992) muestra la funcionalidad que ofrece el sistema futuro desde la perspectiva de los usuarios externos al mismo, en tanto que un diagrama de clases (Jacobson, 1992) representa la estructura estática que las clases poseen y un diagrama de interacción (Jacobson, 1992) representa cómo los objetos intercambian mensajes.

Si bien no pertenece al estándar de UML, el modelo de interfaces describe la presentación de información entre los actores y el sistema (Weitzenfeld, 2005) y es complementario con la información que se presenta en los diagramas de clases y casos de uso. En este modelo, se especifica en detalle cómo se verán las interfaces gráficas de usuario al ejecutar cada uno de los casos de uso. Normalmente, un prototipo funcional de requisitos que muestre la manera en que funcionan las interfaces gráficas de usuario, posibilita la comprensión del software futuro por parte de los interesados, quienes pueden incluso validar si la información que allí se presenta es correcta y adecuada. Esto ayuda al interesado a visualizar los casos de uso según se mostrarán en el software por construir y permite eliminar muchos posibles malos entendidos. Cuando se diseñan las interfaces gráficas de usuario, es esencial involucrar a los interesados, y reflejar la visión lógica del sistema a través de las interfaces; por ello, debe haber consistencia entre los modelos conceptuales elaborados por los analistas y el comportamiento real del software por construir.

Sin embargo, un aspecto que no ha sido tratado muy frecuentemente es cómo estos diagramas se integran (Bustos, 2002), es decir, cuáles son las relaciones explícitas posibles entre estos diagramas cuando están describiendo un mismo modelo.

En algunas de las herramientas comerciales para el apoyo a las actividades y procesos de la Ingeniería de Software (denominadas *Computer-Aided Software Engineering*) se realizan actualmente chequeos de la consistencia interna de los diagramas, pero se realizan pocas revisiones sobre la consistencia entre los diferentes diagramas o artefactos (Chiorean, *et al.*, 2003), la cual debe ser analizada manualmente y de manera exhaustiva por los analistas y diseñadores del proyecto.

Los artefactos definidos por UML deberían guardar consistencia cuando se traten del mismo modelo. La consistencia interna de cada artefacto está definida en la especificación de UML, pero la consistencia entre diferentes artefactos poco se ha trabajado; además, aún subsisten problemas:

- La consistencia intermodelos no se especifica formalmente (Glinz, 2000). Una especificación formal de este tipo de consistencia facilita la comprensión de la información común que deben poseer los diferentes artefactos y posibilita su posterior implementación en herramientas computacionales.
- Sólo se realizan algunos chequeos de consistencia de manera automática entre alguno de los artefactos y el código resultante (Gryce, *et al.*, 2002). El código del software se elabora en una etapa posterior al análisis y el diseño; si el chequeo de la consistencia se realiza sobre el código del software, es probable que algunas de las inconsistencias previas hayan sobrevivido hasta la fase de implementación y, como se dijo previamente, es preferible encontrar este tipo de defectos en las fases iniciales del desarrollo.
- En algunos trabajos se definen reglas de consistencia con reglas de manera formal pero sólo a nivel de intramodelo (Chiorean, *et al.*, 2003), (OMG, 2007). Los modelos no se suelen construir con únicamente un diagrama; se requiere la participación de diferentes puntos de vista que suelen ser expresados en diferentes diagramas. Si esos diagramas se deben referir a la misma información, las reglas de consistencia intramodelo tienen un restringido campo de acción y pueden permitir que subsistan errores de consistencia entre los diferentes diagramas.

En esta Tesis se propone un método para verificar la consistencia entre el diagrama de clases y el diagrama de casos de uso de UML de una manera formal, evaluando una serie de reglas definidas en OCL (OMG, 2007), las cuales se deben cumplir para garantizar que la información brindada por dichos modelos sea consistente. Se define, adicionalmente, la consistencia con las Interfaces Gráficas de Usuario, ya que éstas son construidas con gran participación de ambos diagramas.

Esta Tesis está organizada de la siguiente manera: en el Capítulo 2, se presenta el marco teórico del trabajo, en el cual se caracterizan los principales conceptos necesarios para la comprensión de la solución planteada; en el Capítulo 3 se muestra la problemática general que motiva este trabajo de investigación; en el Capítulo 4 se presenta la revisión de la literatura especializada que se pudo recopilar alrededor del tema en estudio; en el Capítulo 5 se expone el método relacionado con la construcción de las reglas de consistencia y se describe el método para la validación de éstas; en el Capítulo 6, se muestra un caso de estudio, sobre el cual se aplican las reglas de consistencia definidas. Finalmente, en el Capítulo 7 se exponen las conclusiones y el trabajo futuro que se puede derivar de esta Tesis.

# Capítulo 2

## Marco Teórico

### 2.1 Modelo de Casos de Uso

En la fase de análisis del ciclo de vida de un producto de software, se hace uso de este diagrama para reconocer y describir los requisitos funcionales del sistema que se pretende construir. Este diagrama permite reconocer claramente los límites del sistema y las relaciones entre el sistema y su entorno; es decir, a través de la elaboración de este diagrama se identifica la funcionalidad del sistema independientemente de las herramientas o lenguajes de programación que vayan a ser utilizados para su implementación. Además, este diagrama permite dividir el conjunto de necesidades de acuerdo con los tipos de usuarios que interactuarán con el producto de software (OMG, 2007).

Otra de las ventajas del diagrama de casos de uso la constituye el hecho de que es un modelo basado en lenguaje natural, lo cual facilita su entendimiento por parte de los interesados y su refinamiento mediante la interacción con los mismos. A continuación se presenta, a modo de ejemplo, la descripción verbal de un caso de uso para un Sistema de Punto de Venta, que consiste en el software que se encuentra en las cajas de los grandes almacenes de cadena para la venta de productos. Este ejemplo se utilizará para describir los tres artefactos que se conectarán mediante reglas de consistencia intermodelos.

*Procesar Venta:* Un cliente llega a una caja con artículos para comprar. El cajero utiliza el sistema de Punto de Venta (PDV) para registrar cada artículo comprado. El sistema presenta una suma parcial y detalles de cada línea de venta. El cliente introduce los datos del pago, que el sistema valida y registra. El sistema actualiza el inventario. El cliente pide un recibo y luego se va con los artículos.

Esta es una primera aproximación a la descripción de un caso de uso, que busca comprender la funcionalidad básica del sistema. En las siguientes secciones se mostrará el formato que se sigue para su descripción detallada así como la manera de construir el Diagrama de casos de uso, que consiste en la representación grafica que acompaña su descripción verbal.

### **2.1.1. Elementos de un diagrama de Casos de Uso**

En un diagrama de casos de uso se reconocen dos elementos fundamentales: los *actores* que representan los roles que los diferentes usuarios pueden desempeñar, y los *casos de uso* que representan lo que deben estar en capacidad de hacer los usuarios con el sistema. Otro término importante son los *escenarios*, que corresponden a secuencias específicas de acciones e interacciones entre los actores y el sistema, y también se denominan instancias de un caso de uso. Por ejemplo: dentro del caso de uso *procesar venta* puede haber un escenario que corresponde a los clientes que cancelan sus artículos en efectivo. Otro escenario puede estar representado por los clientes que realizan sus pagos con tarjeta de crédito. Como se puede apreciar, los escenarios corresponden simplemente a situaciones particulares dentro un caso de uso (Fowler, 2004).

#### **2.1.1.1. Actores**

- En este punto del ciclo de vida del producto de software ya se deben haber identificado los actores principales del Sistema a través del análisis del *Modelo Verbal* del problema. Para ello, se debieron determinar los responsables de la realización de los procesos. Los actores son modelos de los usuarios potenciales del sistema y realizan un intercambio de información con éste. Es importante tener en cuenta que los actores pueden ser usuarios humanos, pero también pueden corresponder a otros sistemas que se comunican con el sistema que se pretende construir. En el ejemplo del Sistema de puntos de Venta se tienen como actores al cajero y al cliente que se acerca a la tienda a comprar productos. Para identificar los

actores de un sistema se deben plantear interrogantes como: ¿para qué fue diseñado el sistema? o ¿a quiénes se supone que el sistema debe ayudar?

### **2.1.1.2. Caso de Uso**

Un caso de uso es un camino específico de utilización del sistema donde se muestra una parte de su funcionalidad. Así, cada caso de uso constituye un curso completo de eventos iniciados por un actor y que especifican la forma en que interactúan dicho actor y el software por construir. Los casos de uso del sistema se identifican a través de los actores; para cada curso completo de eventos inicializados por un actor se identifica un caso de uso. Analizando cada actor, se identifican cuáles casos de uso se le deben asociar. Las siguientes preguntas pueden ayudar a la identificación de casos de uso:

- ¿Cuáles son las tareas principales de cada actor?
- ¿El actor necesita o modifica información del sistema?
- ¿El actor tiene que informar al sistema sobre cambios externos?
- ¿El actor debería estar informado sobre cambios inesperados dentro del sistema?

### **2.1.2. Pasos para definir un Caso de Uso**

El procedimiento básico a seguir para la definición de los casos de uso es el siguiente:

(1) *Elegir los límites del Sistema:* Este paso se lleva a cabo previamente a través del análisis del modelo verbal y de la determinación del vocabulario que hace parte del dominio del problema.

(2) *Identificar los actores principales del Sistema y las tareas que realizan.* Algunas preguntas útiles pueden ser: ¿Quién arranca y detiene el sistema? ¿Quién se encarga de la administración del sistema? ¿Quién gestiona los usuarios? ¿Quién evalúa la actividad o la seguridad y el rendimiento del Sistema? ¿Cómo se gestionan las actualizaciones de Software?

Otro mecanismo de ayuda en la búsqueda de actores y casos de uso consiste en identificar los eventos externos. Las preguntas que se deben responder son: ¿Cuáles son, de dónde proceden y por qué?

Se construye una Tabla como la que se muestra a continuación (véase Tabla 2.1) en la que se ilustran algunos eventos externos para el Sistema PDV, con qué actor están relacionados y de cuál objetivo forman parte. Por objetivo se puede entender la tarea que se quiere cumplir; en este caso es procesar una venta y de este objetivo se desprende el caso de uso específico. Para este sistema se tienen como actores principales al cliente y al cajero.

<b>Evento Externo</b>	<b>Actor</b>	<b>Objetivo</b>
Introducir línea de venta	Cajero	Procesar una venta
Introducir el pago	Cliente o Cajero	Procesar una venta
....	....	....

**Tabla 2. 1 Eventos Externos, Actores y Objetivos del Sistema.**

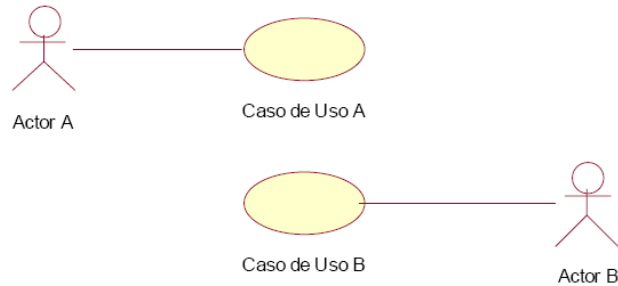
(3) *Escribir los casos de uso.* Si inicialmente se definieron objetivos de usuario, para cada uno de ellos se establece un caso de uso que se nombra de manera similar al objetivo asociado. Por ejemplo:

- Objetivo: procesar una venta.
- Caso de Uso: Procesar Venta.

Como dato adicional, se debe tener presente que los nombres de los casos de uso generalmente comienzan con un verbo. Además, esos casos de uso se deben ligar con los actores que realizan la acción definida por dicho verbo.

### 2.1.3 Diagramas de Casos de Uso: Notación UML

En un diagrama de casos de uso se representan básicamente los actores y los casos de uso relacionados con ellos, tal como se muestra en la Figura 2.1.



**Figura 2. 1 Representación de un caso de Uso**

En un diagrama de casos de uso se representan todos los casos de uso identificados dentro de un sistema así como todos los actores involucrados. Existen varios tipos de relaciones entre casos de uso, como son:

#### 2.1.3.1 Herencia

El caso de uso origen hereda la especificación del caso de uso destino y posiblemente la modifica y/o amplía (véase Figura 2.2).



**Figura 2. 2 Representación de la relación de herencia entre casos de Uso**

### 2.1.3.2 Inclusión

Una relación de inclusión entre casos de uso implica que el caso de uso base u origen incluye el comportamiento descrito por el caso de uso destino. Este caso de uso destino nunca ocurre por sí sólo, es decir, es instanciado dentro de un caso de uso mucho más amplio que lo contiene (caso de uso origen). Dicho de otra forma, el caso de uso origen “usa” al caso de uso destino (véase Figura 2.3).



**Figura 2. 3 Representación de la relación de inclusión entre casos de Uso**

Este tipo de relación entre casos de uso se utiliza para evitar describir el mismo flujo de eventos varias veces, extrayendo el comportamiento común dentro de un nuevo caso de uso.

### 2.1.3.3 Extensión

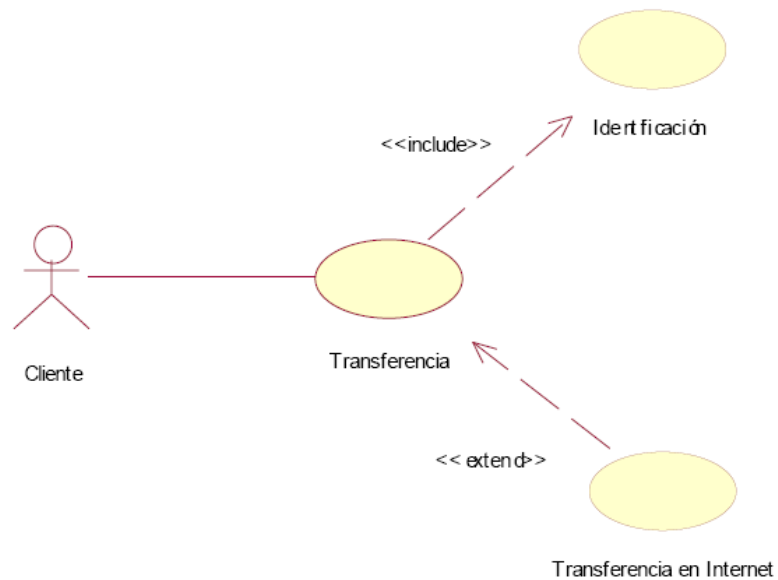
En la relación de extensión entre dos casos de uso, el caso de uso origen extiende o amplía el comportamiento del caso de uso destino, es decir, el caso de uso destino puede existir por sí sólo, pero bajo ciertas condiciones su comportamiento será el que se especifica en el caso de uso origen.



**Figura 2. 4 Representación de la relación de extensión entre casos de Uso**

Se puede usar esta relación de extensión para modelar parte de un caso de uso que puede ser visto como comportamiento opcional del sistema. En este caso, se separa el comportamiento

opcional de un caso de uso del estrictamente obligatorio. Un ejemplo del uso de las relaciones especiales entre casos de uso (extensión e inclusión), se puede apreciar en la Figura 2.5.



**Figura 2. 5 Ejemplo de relaciones de extensión e inclusión entre casos de uso**

## 2.2 Diagrama de Clases

En el proceso de construcción de un producto de software resulta fundamental identificar los elementos básicos que hacen parte del dominio y la manera como estos interactúan entre sí. El diagrama de clases permite visualizar las propiedades relevantes de dichos elementos y sus relaciones, así como especificar algunos detalles para su posterior construcción en un lenguaje de implementación de alto nivel.

Los diagramas de clases se utilizan para modelar el conjunto de entidades o clases necesarias para soportar los requerimientos funcionales del sistema. En términos generales, un diagrama de clases se construye para cumplir alguno de los siguientes objetivos:

- *Modelar el vocabulario de un sistema:* implica tomar decisiones acerca de cuáles aspectos del sistema hacen parte de él y cuáles están por fuera de sus límites. En un diagrama de clases se refleja qué entidades deben ser tenidas en cuenta en la construcción del sistema y sus responsabilidades.
- *Modelar Colaboraciones Simples:* Una colaboración podría definirse como una sociedad de clases que trabajan juntas para desempeñar una funcionalidad mayor de la que cada una ofrecería individualmente. Dentro de un diagrama de clases es posible especificar las colaboraciones entre clases. Esta funcionalidad se logra a través de transacciones dentro del sistema que permiten el ingreso, almacenamiento y cálculo de la información necesaria para llevar a cabo una tarea particular dentro del mismo.
- *Modelar Datos:* Se presenta una definición detallada de la información necesaria dentro del sistema, con una representación gráfica de su estructura en el diagrama de clases.

### **2.2.1. Elementos de un Diagrama de Clases**

Dentro de un diagrama de clases se encuentran dos elementos básicos: *Clases* y *Relaciones* establecidas entre dichas clases (OMG, 2007).

#### **2.2.1.1. Clases**

Una clase es una colección de objetos sobre los cuales se debe guardar o mantener información. En general, una clase es una descripción de uno o más objetos que presentan el mismo conjunto de atributos o propiedades y exhiben un comportamiento similar definido a través de sus operaciones.

Retomando el ejemplo del sistema de Puntos de Venta para la compra de artículos en las cajas de almacenes de cadena, las posibles clases son: *Venta*, *Registro del artículo*, *Catálogo de Productos*, entre otras. Obsérvese que todas las clases mencionadas corresponden a

entidades del sistema de las que se necesita almacenar información para que el Sistema pueda ofrecer toda la funcionalidad solicitada por el usuario.

Cada clase se distingue por:

- Su nombre
- Un conjunto de atributos o propiedades.
- Un conjunto de operaciones o servicios ofrecidos por la clase.

#### **2.2.1.1.1. Atributos**

Corresponden a las propiedades o características relevantes de una clase. Dicho de otra forma, un atributo es cualquier dato (información estática) para la que cada objeto o instancia de clase tiene su propio valor. Entiéndase por objeto cada ejemplificación particular de una clase; por ejemplo, para la clase artículo, que tiene como atributos nombre, código asociado y valor unitario, un objeto o instancia de esta clase sería una camisa, un par de zapatos o cualquier ítem que se pueda adquirir dentro de la tienda. Una vez se hayan identificado las clases, se entra a decidir qué atributos se deben tener en cuenta para cada una. Buscando facilitar esta labor para cada clase se deben responder interrogantes como:

- ¿Qué información de la clase se necesita conocer?
- ¿Qué información se necesita recordar y almacenar a través del tiempo?

#### **2.2.1.1.2. Operaciones**

Las operaciones de una clase corresponden a un conjunto de tareas realizadas por esa clase para dar funcionalidad al sistema. Estas operaciones las posee la clase para modificar sus atributos o para recibir o enviar información y, de esta manera, interactuar con otras clases. Las operaciones reflejan el comportamiento que debe presentar una clase para que el sistema pueda ofrecer la funcionalidad previamente establecida, es decir, al momento de definir las operaciones de una clase se debe tener en cuenta:

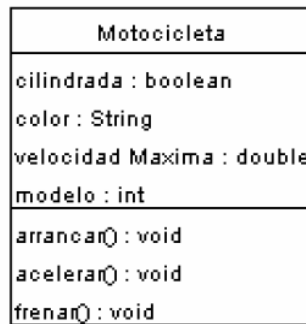
*Qué datos de entrada relacionados con esa clase requiere el sistema;* para el caso de la clase Venta, el cajero es quien ingresa la fecha y la hora en la que se realiza una venta, por tanto esta clase podría poseer dos operaciones que reciban como parámetro dichos datos llamados *ingresarFecha* e *ingresarHora* respectivamente. Igualmente, se puede tener una sola operación que permita ingresar el valor de ambos atributos. Esta decisión de diseño debe ser tomada por el Ingeniero de Software de acuerdo a las necesidades particulares de la aplicación (tiempos de respuesta, facilidad del manejo para el usuario o facilidad de implementación, entre otros).

*Qué datos de salida se requieren del sistema y cuáles están relacionados con cada clase.* Para el Sistema de Puntos de Venta (PDV), es importante obtener el valor total de la compra realizada, que corresponde al valor de un atributo calculado de dicha clase conocido como total. Por tanto, se espera que la clase Venta posea una operación que permita obtener dicho valor, el cual se puede denominar *obtenerTotal*.

Una clase posee operaciones no sólo para cambiar los valores de sus atributos u obtener su valor actual para un objeto o instancia de clase cualquiera, sino que puede tener asociadas operaciones que se encargan de entregar datos a otras instancias de clase del sistema, o que incluso sólo realizan modificaciones parciales sobre los datos de entrada. Por ejemplo, para el caso de la cantidad total asociada a una venta, así como se tiene una operación de la clase Venta que permite obtener este valor (*obtenerTotal*); también, puede existir otra operación que reciba los datos necesarios para calcular dicho valor, denominada *calcularTotal*, que simplemente actualiza el atributo total de la venta particular (instancia de clase) sobre la que se aplica.

### **2.2.1.1.3. Notación UML para las clases**

En UML, una clase se representa con un rectángulo con tres compartimientos: en el primero se coloca el nombre de la clase, en el segundo se enumeran sus atributos y en el inferior se listan las operaciones ofrecidas por dicha clase (Fowler, 2004), tal como se muestra en la Figura 2.6.



**Figura 2. 6 Representación de una clase en UML.**

Obsérvese en la Figura 2.6 que a los atributos de la clase Motocicleta se les indica el tipo de dato que les corresponde, mientras que en las operaciones se especifica que no devuelven ningún valor. Igualmente podría decirse qué argumentos reciben dichas operaciones de manera similar a como ocurre en una función en un lenguaje de programación, pero en la fase de análisis estos detalles no son indispensables, porque están más orientados a la implementación.

### **2.2.1.2. Relaciones**

El diagrama de clases se vale de relaciones para asociar de manera significativa dos o más clases. Las relaciones entre clases se representan por medio de líneas conectivas entre las clases que participan en la relación. Existen tres tipos básicos de relaciones entre clases: la generalización, denotada por una línea termina en un triángulo, la agregación, también una línea cuya terminación es un rombo, y la asociación, que es una línea sencilla que puede tener o no una punta de flecha en su terminación. La generalización expresa relaciones *es\_un* o relaciones de herencia entre las clases, en las cuales una clase puede heredar los atributos y operaciones de otra; la agregación es una relación *todo\_parte*, en la cual un conjunto de elementos conforma otra clase; la asociación es una relación de carácter temporal, puesto que dura mientras una clase ejecuta las operaciones incluidas en otra clase. Por simplicidad, en el Sistema de Puntos de Venta sólo se dibujan relaciones de asociación, pero los metamodelos de la sección 2.6 incluyen todos los tipos de relaciones.

## 2.2.2. Diagrama de Clases para el Sistema de Puntos de Venta

Para el Sistema de Puntos de Venta se extraen como clases las siguientes:

- Venta
- Tienda
- Pago
- Línea de Venta
- CatalogoDeArticulos
- Articulo

Recuérdese que, para el reconocimiento de las clases, se debe hacer un análisis detallado de la descripción verbal del sistema, buscando qué ítemes tienen asociada información necesaria para el adecuado funcionamiento del sistema en construcción. En la figura 2.7 se presenta una aproximación al modelo de Clases para el Sistema de Puntos de Venta, obviando algunos atributos y operaciones para facilitar su comprensión global.

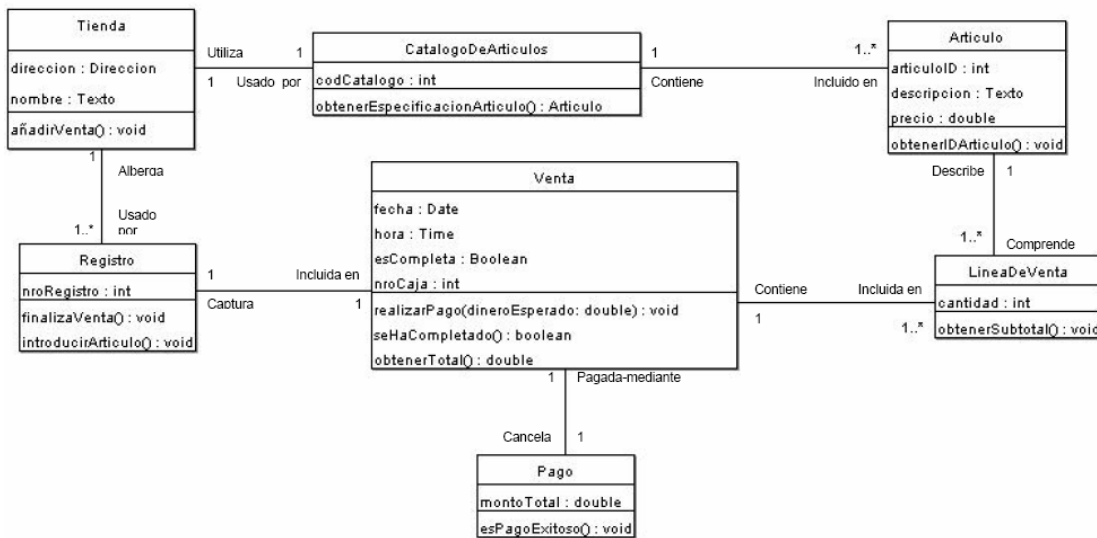


Figura 2. 7 Diagrama de Clases del Sistema de Puntos de Venta (PDV)

## 2.3 Interfaces

### 2.3.1 Interfaces Gráficas de Usuario

La interfaz gráfica de usuario (IGU), es el artefacto tecnológico de un sistema interactivo que posibilita, a través del uso y la representación del lenguaje visual, una interacción amigable con un sistema informático. La interfaz gráfica de usuario (en inglés *Graphical User Interface*, GUI) es un tipo de interfaz de usuario que utiliza un conjunto de imágenes y objetos gráficos (íconos, ventanas, tipografía) para representar la información y acciones disponibles en la interfaz. Habitualmente las acciones se realizan mediante manipulación directa, para facilitar la interacción del usuario con el computador.

Siguiendo la definición de Mandel (1997), una GUI es “la representación gráfica de programas, datos y objetos en la pantalla del ordenador y la interacción con ellos”. Este autor presenta una lista de los elementos que conforman la GUI:

- Pantalla de alta resolución.
- Dispositivos de entrada: teclado y ratón (u otros con los que sea posible apuntar un objeto). El usuario decidirá cuándo usar uno u otro.
- Coherencia entre los distintos programas.
- Visualización en pantalla de los objetos tal y como quedarían impresos.
- Interacción objeto-acción en lugar de acción-objeto. Esto significa que el usuario debe poder seleccionar primero un objeto y después elegir la acción que desea ejecutar en él; la otra forma mencionada es la que seguían las interfaces con línea de comandos o con menús.
- Posibilidad de transferir información entre programas.
- Manipulación directa de la información que se muestra en pantalla.
- Iconos y ventanas.
- Retroalimentación visual de las acciones que ejecuta el usuario.
- Representación visual de las acciones y modos que ejecutan el usuario y el sistema.
- Controles gráficos que puedan ser seleccionados.
- Posibilidad de que el usuario personalice la interfaz y las interacciones.

A pesar de ser interfaces que permiten la manipulación directa y que representan los objetos con iconos, los usuarios deben adquirir algunos conocimientos para poder manejarlas con agilidad. Entre otras cosas, los usuarios deben saber qué hardware están utilizando y cuál es la configuración de su software, así como comprender la estructura jerárquica de los directorios y las diferencias entre los distintos tipos de archivos; también deben conocer lo que representan los iconos que ven en su pantalla y cómo se interactúa con ellos, cuáles son los elementos básicos de una ventana (cómo cambiar el tamaño, cómo usar las barras de desplazamiento, maximizar, minimizar, restaurar, cerrar, etc.) y la manera de usar los controles para seleccionar y ejecutar archivos o para navegar por las ventanas.

La interfaz gráfica no es sólo el conjunto de elementos mencionados, sino que necesita estar realmente dirigida al usuario. La interfaz se podría asimilar a un iceberg: un 10% es lo que se ve, un 30% lo que el usuario siente y un 60% cómo interactúa el usuario a través del diseño; el diseño de la interfaz debería tomar en consideración de manera fundamental la opinión del usuario. Una interfaz bien diseñada debe reducir la carga de memorización por parte del usuario así como proveerle de claves visuales que le hagan identificar las opciones disponibles para seleccionar en cada momento, sin obligarle a recordar y a teclear las opciones.

El diseño de una interfaz gráfica debe considerar los siguientes aspectos recogidos por Marcus (1995, p. 425):

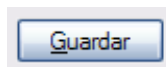
- Una imagen mental comprensible (una metáfora).
- Una organización apropiada de los datos, funciones, tareas y roles.
- Un esquema de navegación eficiente entre los datos y las funciones, las tareas y los roles.
- Una secuencia efectiva de interacción.

Este autor propone el uso de “lenguajes visibles” como técnica de diseño gráfico que facilite la comunicación de los contenidos que se quieren transmitir. El término incluye la disposición de elementos (layout), la tipografía, el color y la textura, las imágenes, la animación, las secuencias, el sonido y las reglas que den al diseño una consistencia.

Una aplicación, al estar conformada por muchos formularios, debe mantener entre éstos, uniformidad en el diseño así como en su organización. Si la aplicación tiene formularios cuyo diseño varía considerablemente, esto puede causar una mala impresión al usuario. Es una buena práctica establecer los lineamientos generales a nivel de diseño y organización de formularios antes de su construcción. Por otro lado, se debe tener especial cuidado en los íconos que se asocian con los botones para no dar una idea equivocada al usuario acerca de su función. A continuación se presentan los controles básicos de una GUI:

### **2.3.1.1 Botones de comando**

Son botones que pueden contener texto, gráficos o ambos. Generalmente se emplea una única palabra para identificar la acción que representa el botón (Véase la Figura 2.8).



**Figura 2.8. Ejemplo de Botón de Comando.**

#### **2.3.1.1.1 Botón por defecto**

- Aparece en los cuadros de diálogo
- Se activa cuando el usuario presiona Enter, y desencadena la ejecución de las acciones asociadas a dicho botón (las realizadas más a menudo).
- Una opción no segura (que ocasione la pérdida de datos) nunca puede ser el botón por defecto.

#### **2.3.1.1.2 Botón de cancelación**

- Se activa al pulsar la tecla ESCAPE y provoca la ejecución de las acciones asociadas al botón identificado como de cancelación.

- A diferencia del anterior, es necesario implementar este comportamiento, es decir, al pulsar la tecla Escape no se ejecuta automáticamente el código asociado al botón de cancelación.

#### ***Botones Cerrar, Cancelar y Salir:***

A menudo tienden a confundirse los papeles que cumplen los botones de comando (Cerrar, Cancelar y Salir). Las siguientes son las funciones que cada uno cumple en una aplicación.

- *Cerrar*: para cerrar el formulario, independientemente del trabajo llevado a cabo anteriormente en él.
- *Cancelar*: para cancelar las tareas llevadas a cabo en el formulario. Todos los cambios efectuados quedan sin efecto y el formulario se cierra.
- *Salir*: utilizado cuando la aplicación tiene un formulario principal y carece de una opción de menú para salir de ella. Termina la ejecución de la aplicación.

### **2.3.1.2 CheckBox (Listas de chequeo)**

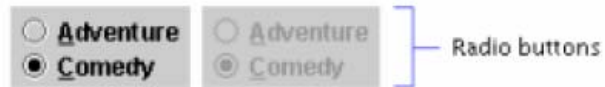
Es un control que representa dos estados: on y off. Se pueden seleccionar varias opciones a la vez (Véase la Figura 2.9).



**Figura 2.9. Ejemplo del comando Checkbox**

### **2.3.1.3 Radio Button (Listas de opciones)**

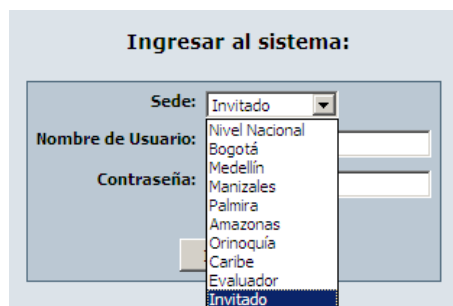
Permite seleccionar una única opción dentro de un conjunto de opciones relacionadas. Sólo una opción puede estar seleccionada en cada momento (Véase la Figura 2.10).



**Figura 2.10. Ejemplo de Radio Buttons.**

### 2.3.1.4 Combo Box (Lista Combinada)

Es un componente con una flecha que, al hacer click sobre ella, permite seleccionar entre un conjunto de opciones mutuamente exclusivas, que están en una lista desplegable (Véase la Figura 2.11).



**Figura 2.11. Ejemplo de combo box.**

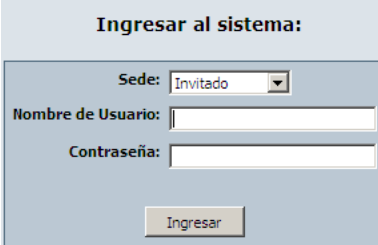
## 2.3.2 Componentes para texto

Permiten a los usuarios ver y editar texto en una aplicación, ingresando información requerida en los formularios para ser procesada posteriormente.

### 2.3.2.1 Label (Etiqueta)

Una etiqueta muestra texto, gráficos o ambos, pero de sólo lectura (Véase la Figura 2.12). Tiene dos funciones en una aplicación:

- Identificar componentes.
- Comunicar el estado o dar instrucciones a los usuarios.



The image shows a login form with a light blue background. At the top, it says "Ingresar al sistema:". Below that, there is a dropdown menu labeled "Sede:" with "Invitado" selected. Underneath are two text input fields: "Nombre de Usuario:" and "Contraseña:". At the bottom center is a button labeled "Ingresar".

**Figura 2.12. Ejemplos de labels y text fields.**

### **2.3.2.2 Text Field (Cuadro de Texto)**

Un campo de texto es utilizado para capturar información requerida en el formulario, la cual es digitada por el usuario (Véase la Figura 2.12).

### **2.3.3 Modelo de Interfaces**

El modelo de interfaces describe la presentación de información entre los actores y el sistema. Se especifica en detalle cómo se verán las interfaces de usuario al ejecutar cada uno de los casos de uso. Si se trata de Interacción Humano-Computador (HCI—*Human Computer Interaction*) se pueden usar esquemas de cómo vería el interesado las pantallas cuando se ejecuta cada caso de uso. También se puede generar una simulación más sofisticada usando un Sistema Manejador de Interfaces de Usuario (UIMS—*User Interface Management System*) (Weitzenfeld, 2005).

Normalmente, un prototipo funcional de requisitos que muestre las interfaces de usuario es una estrategia importante. Esto ayuda al interesado a visualizar los casos de uso según serán mostrados por el sistema a ser construido, y de esta manera eliminar gran cantidad de malos entendimientos. Cuando se diseñan las interfaces de usuario, es esencial involucrar a los

interesados, siendo esencial que las interfaces reflejen la visión lógica del sistema. Esto es realmente uno de los principios fundamentales del diseño de interfaces humanas, donde debe existir consistencia entre la imagen conceptual del usuario y el comportamiento real del sistema. Si las interfaces son protocolos de hardware, se pueden referir a los diferentes estándares, como protocolos de comunicación. Estas descripciones de interfaces son por lo tanto partes esenciales de las descripciones de los casos de uso y las deben acompañar. En estas etapas iniciales del desarrollo, el diseño gráfico de las interfaces (colores, formas, figuras, etc.) no es tan importante como el manejo de información que se ofrece, el cual debe corresponder a las necesidades de cada caso de uso, algo que se mostrará más adelante como ejemplo para un Sistema de Reservas de Vuelos (Weitzenfeld, 2005).

### 2.3.4 Interfaz para el Sistema de Puntos de Venta

En la Figura 2.13 se presenta una posible interfaz para el caso de uso *Procesar Venta*, para el Sistema de Puntos de Venta, PDV.

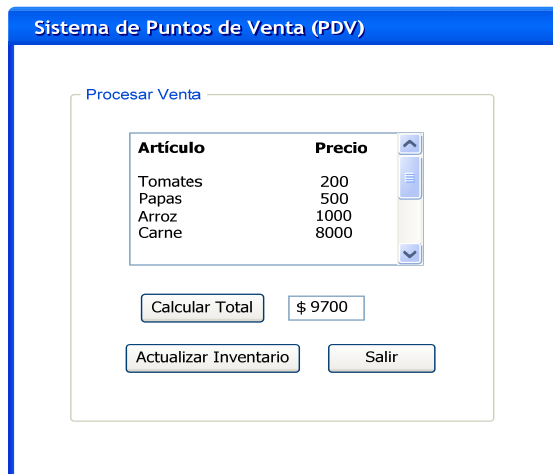


Figura 2.13 Interfaz para *Procesar Venta*

## **2.4 OCL**

El Lenguaje de Especificación de Objetos (*Object Constraint Language*) (OMG, 2007) es un lenguaje formal para expresar restricciones libres de efectos colaterales. Los usuarios del Lenguaje Unificado para Modelado (UML) y de otros lenguajes, pueden usar el OCL para especificar restricciones y otras expresiones incluidas en sus modelos. El OCL tiene características de un lenguaje de expresión, de un lenguaje de modelado y de un lenguaje formal.

### **2.4.1 Lenguaje de expresión**

El OCL es un lenguaje de expresión puro. Por lo tanto, garantiza que una expresión OCL no tendrá efectos colaterales; esto implica que es un lenguaje que declara expresiones, pero no puede cambiar nada en el modelo: el estado del sistema no cambiará nunca como consecuencia de una expresión OCL, aun cuando una expresión OCL puede usarse para especificar un cambio de estado, por ejemplo, en una postcondición. Así, cuando una expresión OCL es evaluada, simplemente devuelve un valor, no realiza cambios sobre el modelo.

### **2.4.2 Lenguaje de modelamiento**

El OCL no es un lenguaje de programación, por lo tanto, no es posible escribir lógica de programa o flujo de control en OCL. No es posible invocar procesos o activar operaciones que no sean consultas en OCL. Dado que el OCL es un lenguaje de modelamiento en primer lugar, es posible que haya cosas en él que no sean directamente ejecutables.

Como el OCL es un lenguaje de modelamiento, toda consideración de implementación está fuera de su alcance, y no puede ser expresada en el lenguaje OCL. Conceptualmente, cada

expresión OCL es atómica. El estado de los objetos en el sistema no puede variar durante la evaluación.

### **2.4.3 Lenguaje Formal**

OCL es un lenguaje formal donde todos los constructores tienen un significado formalmente definido; la especificación del OCL es parte del UML. El OCL no pretende reemplazar lenguajes formales existentes como VDM y Z.

En el modelamiento orientado a objetos, un diagrama como el de clases no es suficiente para lograr una especificación precisa y no ambigua. Existe la necesidad de describir características adicionales sobre los objetos del modelo. Muchas veces estas características se describen en lenguaje natural. La práctica ha revelado que muy frecuentemente esto produce ambigüedades. Para escribir sin ambigüedad se han desarrollado los lenguajes formales.

La desventaja de los lenguajes formales tradicionales es que son adecuados para personas con una fuerte formación matemática, pero difíciles para el modelador de sistemas. El OCL ha sido desarrollado para cubrir esa brecha. Es un lenguaje formal, fácil de leer y escribir. Ha sido desarrollado como un lenguaje de modelamiento para negocios dentro de la división Seguros de IBM, y tiene sus raíces en el método Syntropy.

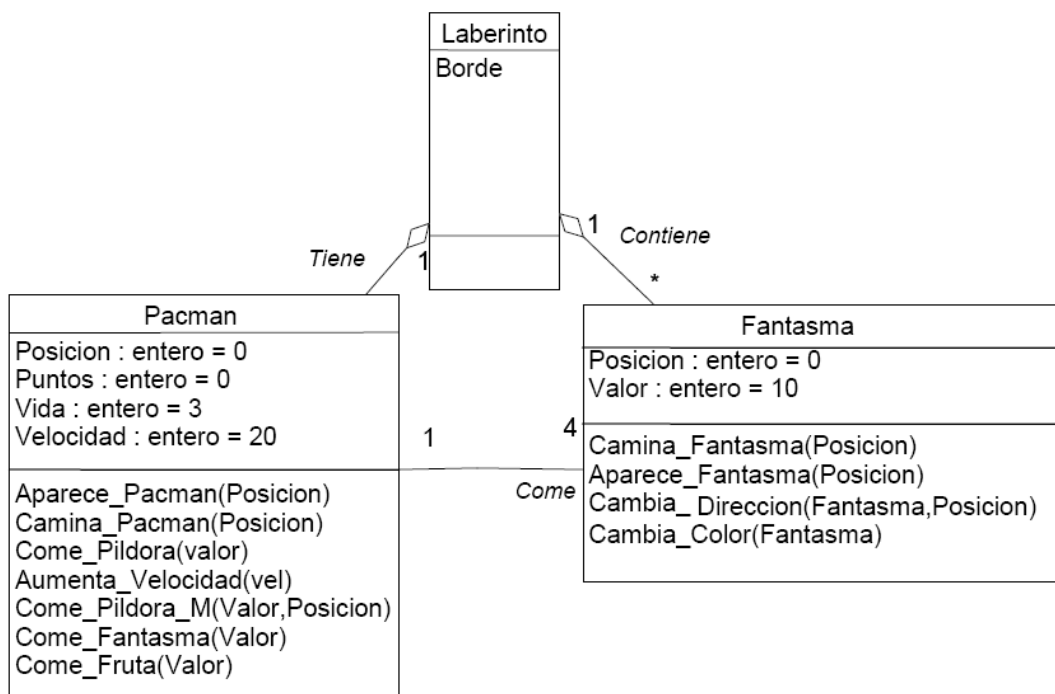
### **2.4.5 En dónde usar OCL**

El OCL puede ser usado con distintos propósitos:

- Para especificar características estáticas sobre clases y tipos en un modelo de clases.
- Para especificar características estáticas de tipo para Estereotipos.
- Para especificar pre y postcondiciones sobre Operaciones y Métodos.
- Como lenguaje de navegación.

- Para especificar restricciones sobre operaciones: Dentro del documento Semántica del UML, el OCL es usado en la sección reglas bien formuladas, como constantes estáticas sobre las metaclases en la sintaxis abstracta. En varios lugares también es usado para definir operaciones ‘adicionales’, que son tomadas en cuenta en la formación de reglas.

El diagrama de la Figura 2.14 servirá como ejemplo para las siguientes explicaciones. En dicha Figura se muestra el diagrama de clases correspondiente al popular juego de video *Pacman*.



**Figura 2. 14 Diagrama de Clases para el juego PACMAN**

## 2.4.6 Conexión con el metamodelo UML

### 2.4.6.1 self

Cada expresión OCL está escrita en el contexto de una instancia de un tipo en particular. En una expresión OCL el nombre *self* es usado para referirse a dicha instancia.

### 2.4.6.2 Características Estáticas

Una expresión OCL puede ser parte de una característica estática, que es una Restricción estereotipada con «*invariant*». Cuando una Característica estática está asociada a un Clasificador, la expresión es una invariante del tipo, y debe ser verdadera para todas las instancias de ese tipo en todo momento. Si el contexto es Pacman, entonces *self* se refiere a una instancia de Pacman en la expresión *self.Posicion*

#### 2.4.6.2.1 Pre y Post-condiciones

Las expresiones OCL pueden ser parte de precondiciones o postcondiciones, que son Restricciones estereotipadas con «*pre-condition*» y «*post-condition*» respectivamente. Las precondiciones o postcondiciones se aplican tanto a Métodos como a Operaciones. En este caso la instancia que da contexto es del tipo al que pertenece la operación. La notación usada en este documento es subrayar el tipo y la declaración de la operación, y poner la frase ‘pre:’ y ‘post:’ antes de las precondiciones y postcondiciones. La sintaxis que se asocia a una expresión OCL es la siguiente:

```
nombreDeTipo::nombreDeOperacion(parametro1 : Tipo1, ... ): TipoDevolucion  
pre : parámetro1 > ...  
post: resultado = ...
```

## 2.4.7 Objetos y propiedades

Las expresiones OCL pueden referirse a tipos, clases, interfaces, asociaciones (actuando como tipos) y tipos de datos. También todos los atributos, extremos de asociación, métodos y operaciones que no tengan efectos colaterales y estén definidos en esos tipos pueden ser usados. En un diagrama de clases, una operación o método no tiene efectos colaterales si el atributo *isQuery* de la operación es verdadero. Dentro de este documento, se va a referir a los atributos, extremos de asociación, métodos y operaciones libres de efectos colaterales como propiedades. Por consiguiente una propiedad puede ser una de las siguientes opciones:

- un Atributo,
- un Extremo de Asociación,
- una Operación con *isQuery* en verdadero,
- un Método con *isQuery* en verdadero

### 2.4.7.1 Propiedades

El valor de una propiedad para un objeto que está definido en un diagrama de clases se especifica con un punto seguido del nombre de la propiedad, como se muestra seguidamente:

UnTipo

self.propiedad

Si *self* es una referencia a un objeto, entonces *self.propiedad* es el valor de la propiedad *propiedad* para *self*.

### 2.4.7.1.1 Atributos

Por ejemplo, la vida de un Pacman se escribe como:

```
Pacman  
self.vida
```

El valor de esta expresión es el valor del atributo vida para Pacman. El tipo de esta expresión es el tipo del atributo vida, que es del tipo básico Entero.

### 2.4.7.1.2 Operaciones

Las operaciones pueden tener parámetros. Por ejemplo, en el diagrama mostrado anteriormente, un objeto Pacman tiene una función Camina\_Pacman. Esta operación puede ser expresada como sigue, para una instancia Pacman: unPacman y Posición:

```
unPacman.Camina_Pacman(Posición)
```

### 2.4.7.1.3 Extremo de asociación y navegación

Comenzando desde un objeto en particular, es posible navegar a través de una asociación en un diagrama de clases con el fin de hacer referencia a otro objeto y a sus propiedades. Para hacerlo, se navega la asociación usando su extremo opuesto:

```
objeto.nombreDeRol
```

El valor de esta expresión es el conjunto de objetos que están del otro lado de la asociación nombreDeRol. Si la multiplicidad del extremo de la asociación tiene un máximo de uno (“0..1” o “1”), entonces el valor de la expresión es un objeto. En el diagrama de clases del ejemplo, si se comienza en el contexto de Pacman (*self* es una instancia de Pacman), es posible escribir:

### Pacman

`self.Come -- es de tipo Set(Fantasma)`

La evaluación de la expresión resultará en un conjunto de Fantasmas. Por defecto, la navegación resultará en un conjunto. Cuando la asociación, en el diagrama de clases, está complementada por *{ordered}*, la navegación resultará en una Secuencia (*Sequence*). Las Colecciones, así como los *Set*, las *Bag* y las *Sequence*, son tipos predefinidos en OCL. Tienen un gran número de operaciones predefinidas. Una propiedad de la colección en sí es accedida utilizando la flecha, ‘->’ seguida del nombre de la propiedad. El siguiente ejemplo es en el contexto de Pacman:

### Pacman

`self.Come->tamaño`

El ejemplo aplica la propiedad tamaño al Set *self.Come*, que da como resultado el número de Fantasmas que come Pacman.`self`

### Pacman

`self.Come->esVacio`

Este ejemplo aplica la propiedad esVacio al Set *self.Come*. El resultado es verdadero si el conjunto de Fantasmas es vacío, y falso si no lo es.

## **2.4.7.2 Navegación hacia tipo asociación**

Para especificar la navegación en asociaciones como clase, OCL usa el punto y el nombre de la asociación como clase comenzando con minúscula. Las asociaciones como clase no tienen un nombre de rol específico en el diagrama de clases.

### 2.4.7.3 Navegación desde la asociación como clase

Se puede navegar desde la asociación como clase hacia el objeto que participa en la asociación. Esto se hace utilizando el punto y el nombre de rol del extremo de la asociación. La navegación desde una asociación como clase a uno de los objetos de la asociación siempre entregará exactamente un objeto. Este es el resultado de la definición de una asociación como clase. Por lo tanto, el resultado de esta navegación es exactamente un objeto, aunque también puede usarse como un Conjunto utilizando la flecha (->).

### 2.4.7.4 Expresiones generales

Cualquier expresión OCL puede ser usada como valor para un atributo en una expresión de una clase del UML o sus subtipos. En este caso, el documento Semántica del UML describe el significado de la expresión.

### 2.4.7.5 Tipos predefinidos en OCL

Se definen dentro del OCL los tipos de datos básicos, así como las operaciones que pueden realizar, los cuales se muestran en la Tabla 2.2:

<b>TIPO</b>	<b>VALOR</b>	<b>OPERACIÓN</b>
Bolean (Booleano)	True, False	And, or, xor, not
Integer (Entero)	1,2,34,2656...	*,+,/, abs
Real (Flotante)	1.5, 3.14, 6.78...	*,+,-/, floor(base)
String (Cadena)	To be, or, not	toUpper, concat

**Tabla 2. 2 Tipos predefinidos en OCL**

Dentro de los tipos predefinidos para OCL también se encuentran las Colecciones que juegan un rol muy importante en las expresiones OCL debido a que una navegación frecuente da como resultado una Colección.

### 2.4.7.5.1 Collection

Es un tipo abstracto con subtipos concretos, se distinguen 3 tipos diferentes de colecciones:

- *Set*: Conjunto matemático, que no contiene elementos duplicados
- *Bag*: Conjunto el cual puede contener elementos duplicados. El mismo elemento puede estar en la bolsa 2 o más veces.
- *Sequence*: Es como una bolsa, la diferencia es que los elementos están en forma ordenada.

Algunos ejemplos de las colecciones de OCL se pueden apreciar en la Tabla 2.3.

TIPO	EJEMPLO
Set	Set {1,2,5,8}
Sequence	Sequence {1,3,45,2,3}
Bag	Bag {1,3,4,2,1}

**Tabla 2. 3 Ejemplos de colecciones de OCL**

## 2.5 XQuery

### 2.5.1. Descripción del Lenguaje

XML ha significado mucho para el desarrollo de sistemas; aspectos tales como la posibilidad de comunicar de manera transparente sistemas pertenecientes a distintas plataformas que eran

de difícil solución en otros tiempos, han permitido que paulatinamente el lenguaje XML se haya posicionado como un estándar de comunicación entre aplicaciones. Aunque XML es un paso importante, por sí solo no es de gran utilidad; lo que realmente hace potente a esta tecnología es el conjunto de estándares que se han desarrollado (y los que aun están en desarrollo) en torno a la misma (Minoli, 2004).

Uno de esos estándares es el XQuery (W3C, 2007), que es un lenguaje de consulta para documentos XML. Este lenguaje provee mecanismos para extraer información de bases de datos XML nativas. XQuery se presenta como un lenguaje funcional; por ello, en vez de ejecutar comandos como lo haría un lenguaje procedural, cada consulta es una expresión a ser evaluada. Las expresiones se pueden combinar para hallar nuevas expresiones (Minoli, 2004).

## 2.5.2 XPath

XQuery hace un uso intensivo de XPath (W3C, 2007), que es un lenguaje utilizado para seleccionar porciones de XML; de hecho algunos ven a XQuery como un superconjunto de XPath (Minoli, 2004). Para observar la sintaxis básica de XPath, a continuación se presenta un ejemplo (tomado de (Minoli, 2004)) donde se muestran los datos de la descripción de un producto en formato XML:

```
<ProductDescription ProductModelID="19" ProductModelName="Mountain 100">
  <Summary>
    Our top-of-the-line competition mountain bike.
    Performance-enhancing options include the innovative HL Frame,
    super-smooth front suspension, and traction for all terrain.
  </Summary>
  <Manufacturer>
    <Name>AdventureWorks</Name>
    <ProductURL>HTTP://www.Adventure-works.com</ProductURL>
  </Manufacturer>
</ProductDescription>
```

```

</Manufacturer>
<Features>
  <wm:Warranty>
    <wm:WarrantyPeriod>3 years</wm:WarrantyPeriod>
    <wm:Description>parts and labor</wm:Description>
  </wm:Warranty>
  <wm:Maintenance>
    <wm:NoOfYears>10 years</wm:NoOfYears>
    <wm:Description>maintenance contract available trough...</wm:Description>
  </wm:Maintenance>
</Features>
<Picture>
  <Angle>front</Angle>
  <Size>small</Size>
  <ProductPhotoID>31</ProductPhotoID>
</Picture>
</ProductDescription>

```

Dados los datos origen, estos son algunos ejemplos de consultas XPath:

```
/ProductDescription/Summary
```

Selecciona todos los elementos <Summary> que son hijos del elemento <ProductDescription>, que es el elemento raíz del documento.

```
//Summary
```

Selecciona todos los elementos <Summary> que se encuentran dentro del documento. La doble barra indica una profundidad arbitraria del elemento summary

```
count(//Summary)
```

Retorna el número de elementos <Summary> que aparecen en el documento.

`//Picture[Size = "small"]`

Retorna todos los elementos <Picture>, de profundidad arbitraria, que tienen un hijo cuyo valor es "small".

`//ProductDescription[@ProductModelID=19]`

Retorna todos los elementos ProductDescription que contienen un atributo ProductModelID y su valor es 19. El símbolo @ indica que ProductModelID es un atributo.

`//ProductDescription[@ProductModelID]`

Retorna todos los elementos ProductDescription que contienen un atributo ProductModelID, independientemente del valor que contengan.

`//ProductDescription/@ProductModelID`

Retorna los valores del atributo ProductModelID.

`//Size[1]`

Retorna el primer nodo <Size> que encuentra en el documento (Minoli, 2004).

### **2.5.3 Modelo de Datos**

XQuery está definido en términos de un modelo formal abstracto, no en términos de texto XML. Los términos formales están definidos en el documento "XQuery 1.0 and XPath 2.0 Data Model" (W3C, 2007). Cada entrada a una consulta es una instancia de un modelo de datos, y la salida de una consulta también. En torno a este enfoque existen disputas entre los que provienen del "mundo de los documentos" (la comunidad XML) y los que provienen del "mundo de las bases de datos". La comunidad de bases de datos defiende la importancia de tener un modelo de datos; de hecho, este es el enfoque adoptado por el comité del W3C. Se intenta lograr un lenguaje cerrado con respecto al modelo de datos. Se dice que un lenguaje

es cerrado con respecto a un modelo de datos si se puede garantizar que el valor de cada expresión en el lenguaje se encuentra dentro del modelo.

En el modelo de datos XQuery, cada documento es representado como un árbol de nodos. Los tipos de nodos posibles son:

- Document
- Element
- Attribute
- Text
- Namespace
- Processing Instruction
- Comment

Cada nodo en el modelo de datos es único e idéntico a sí mismo, y diferente a todos los demás. Esto no implica que no puedan tener valores iguales, sino que conceptualmente se los debe tomar como entidades diferentes. Podría hacerse una analogía con el principio de identidad existente en la teoría de objetos.

Además de los nodos, el modelo de datos permite valores atómicos (*atomic values*), que son valores simples que se corresponden con los valores simples (*simple types*) definidos en la recomendación XML Schema, Parte 2 del *World Wide Web Consortium* (W3C, 2007). Estos pueden ser *string*, *boolean*, *decimal*, *integer*, *float*, *double* y *date*.

Un ítem es nodo simple o valor atómico. Una serie de ítems es conocida como *sequence* (secuencia). En XQuery cada valor es una secuencia, y no hay distinción entre un ítem simple y una secuencia de un solo ítem. Las secuencias sólo pueden contener nodos o valores atómicos, no pueden contener otras secuencias. El primer nodo en cualquier documento es el "nodo documento" (*document node*). El nodo documento no se corresponde con nada visible en el documento; éste representa el mismo documento.

Los nodos conectados forman un árbol, que consiste en un nodo "*root*" (raíz) y todos los nodos que se desprenden de él. Un árbol cuya raíz es un nodo documento se denomina árbol documento, todos los demás son denominados fragmentos (Minoli, 2004).

## **2.5.4 Expresiones FLWOR**

Las expresiones FLWOR (que suele pronunciarse "flower") son al XQuery lo que las distintas cláusulas dentro de una sentencia Select (select, from, where, etc) son al SQL. Es decir, son sus bloques principales. El nombre viene de For, Let, Where, Order by y Return:

### **2.5.4.1 For**

Asocia una o más variables a expresiones, creando un conjunto de tuplas en el cual cada tupla vincula una variable dada a uno de los ítems a los cuales está asociada la expresión evaluada.

### **2.5.4.2 Let**

Vincula las variables al resultado de una expresión, agregando estos vínculos a las tuplas generadas por la cláusula FOR.

### **2.5.4.3 Where**

Filtra tuplas, quedando sólo aquellas que cumplen con una condición dada.

### **2.5.4.4 Order by**

Ordena las tuplas en el conjunto de tuplas resultado.

### **2.5.4.5 Return**

Construye el resultado de la expresión FLWOR para una tupla dada.

Las cláusulas FOR y LET construyen el conjunto de tuplas sobre el cual se evaluará la sentencia FLWOR; al menos una de estas cláusulas tiene que existir en una consulta. Con estas sentencias se consigue buena parte de la funcionalidad que diferencia a XQuery de XPath. Entre otras cosas permite construir el documento que será la salida de la sentencia por medio de la cláusula RETURN.

## 2.6. Metamodelos

UML es el lenguaje unificado de modelamiento propuesto por el OMG para la especificación de sistemas informáticos (OMG, 2007). Este lenguaje de propósito general, que puede ser utilizado para la descripción de sistemas de diversos dominios, provee simultáneamente mecanismos de extensión, para que pueda adaptarse de manera óptima a un área de aplicación particular. Tal es la capacidad de adaptación del UML que la descripción de su especificación utiliza la simbología del diagrama de clases para caracterizar todos los elementos de UML.

Ahora bien, los metamodelos son modelos que especifican la estructura, semántica y restricciones de una familia de modelos (Mellor, *et al.*, 2002). Por ejemplo, el metamodelo del diagrama de clases UML especifica todos los elementos que puede contener este diagrama (clases, operaciones, clases de asociación, etc.) y las relaciones que se pueden establecer entre ellos (asociaciones, agregaciones y generalizaciones, entre otras). El Metamodelo de UML está especificado en MOF (*Meta Object Facility*) (OMG, 2007), que corresponde al lenguaje de metamodelamiento estándar del OMG. Este metamodelo provee mecanismos de extensión que permiten añadir elementos (como clases y relaciones) si se considera necesario; además, esos mecanismos permiten el uso de perfiles UML que brindan la posibilidad de extender un metamodelo existente con construcciones propias de un dominio particular sin modificar el metamodelo original de UML (Fuentes, 2002). La especificación de la versión más reciente de UML (versión 2.1) incluye el metamodelo correspondiente a los 13 diagramas que se pueden representar con este lenguaje (OMG, 2007).

Para generar las reglas de consistencia entre el diagrama de clases y el diagrama de casos de uso de UML de una manera formal, se hace necesario tener como base los metamodelos de dichos diagramas para poder generalizar y garantizar que las reglas sirvan para todos los modelos, teniendo en cuenta su estructura y las reglas de buena formación (WFR) definidas en la especificación de UML. Además, es necesario contar con el metamodelo de las interfaces gráficas de usuario; todos estos elementos se muestran en las secciones siguientes.

### **2.6.1. Metamodelo del diagrama de Espacios de Nombres del paquete Kernel.**

En la Figura 2.15 se presenta uno de los metamodelos de alto nivel incluido en MOF y que permite definir los *Namespace* como clases que heredan de los *NamedElement* y finalmente de los *Element*. Los *Namespace* son importantes porque los elementos de la mayoría de los diagramas de UML son de este tipo.

### **2.6.2. Metamodelo del diagrama de Clasificadores del paquete Kernel.**

En la Figura 2.16 se presenta el metamodelo del elemento *Classifier*, donde se muestra que este elemento hereda directamente de *Namespace*. Se puede apreciar también una relación de composición entre *Classifier* y *Generalization*, que permite expresar posteriormente las relaciones de generalización entre clases del diagrama de clases. Este metamodelo y el de la Sección 2.6.1. sirven de base para presentar los metamodelos de los diagramas de clases y casos de uso.

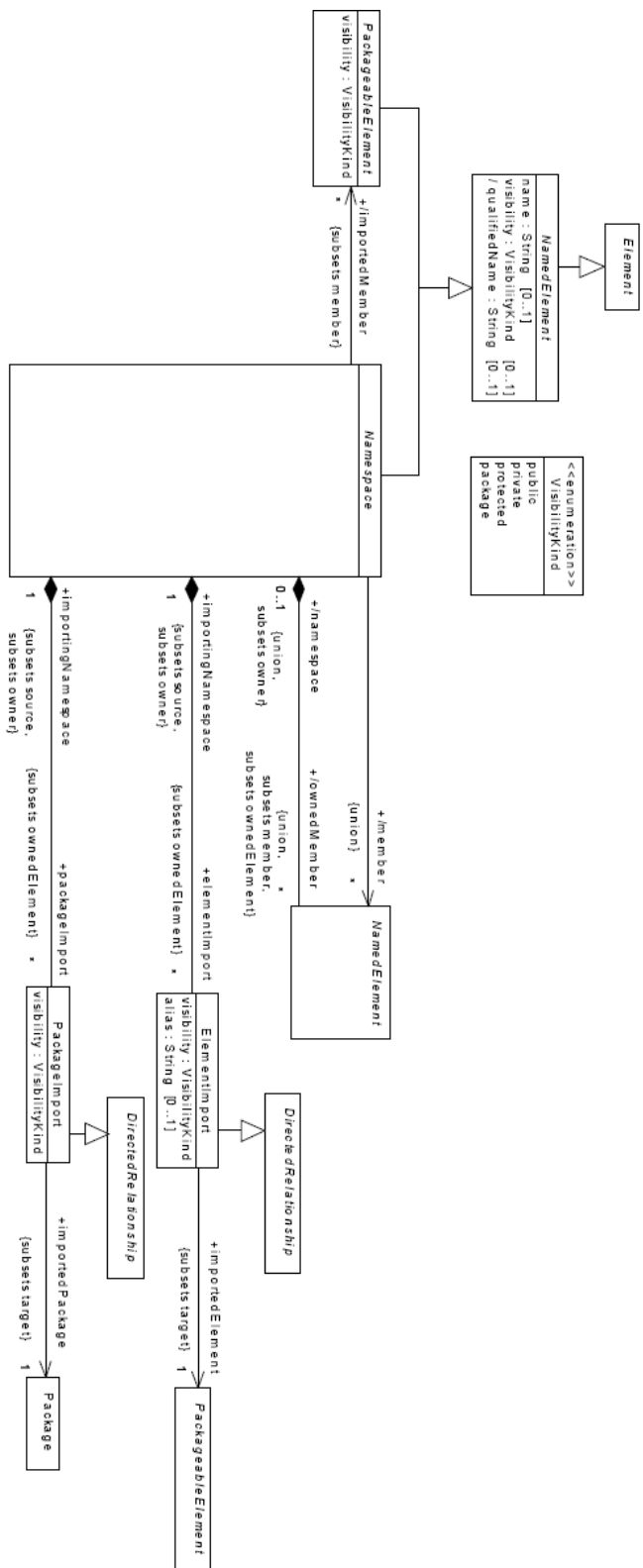


Figura 2.15 Metamodelo del elemento *Namespace*.



### 2.6.3 Metamodelo de diagrama de clases.

En la Figura 2.17 se puede apreciar el metamodelo del diagrama de clases. Los elementos que se presentan en la Sección 2.2.1 se pueden identificar en este metamodelo. Por ejemplo, la clase *Class* tiene el mismo significado que se anota para el diagrama de clases, la clase *Property* se asocia con los atributos, la clase *Association* permite definir las relaciones de asociación y la relación derivada */superClass* permite expresar la relación de generalización.

Nótese de la Figura 2.17 que la clase *Class* hereda del elemento *Classifier* (Véase Figura 2.10), el cual a su vez hereda del elemento *Namespace* (Véase Figura 2.15), que a su vez hereda del *NamedElement* (Véase Figura 2.15), de donde obtiene el atributo *name*.

### 2.6.4. Metamodelo de Diagrama de Casos de Uso.

En la Figura 2.18 se presenta el metamodelo en MOF del diagrama de casos de uso. Al igual que con el diagrama de clases, los elementos identificados en la Sección 2.1.3 se pueden identificar en este metamodelo. En este caso, los elementos tienen nombres similares a los definidos, así: *UseCase* en MOF es un caso de uso, *Actor* es un actor y las relaciones de extensión e inclusión se denominan respectivamente *extend* e *include*. Las relaciones de herencia entre actores y entre casos de uso se obtienen implícitamente del hecho de que ambos heredan de *Classifier*, para el cual ya se había definido la relación *Generalization*.

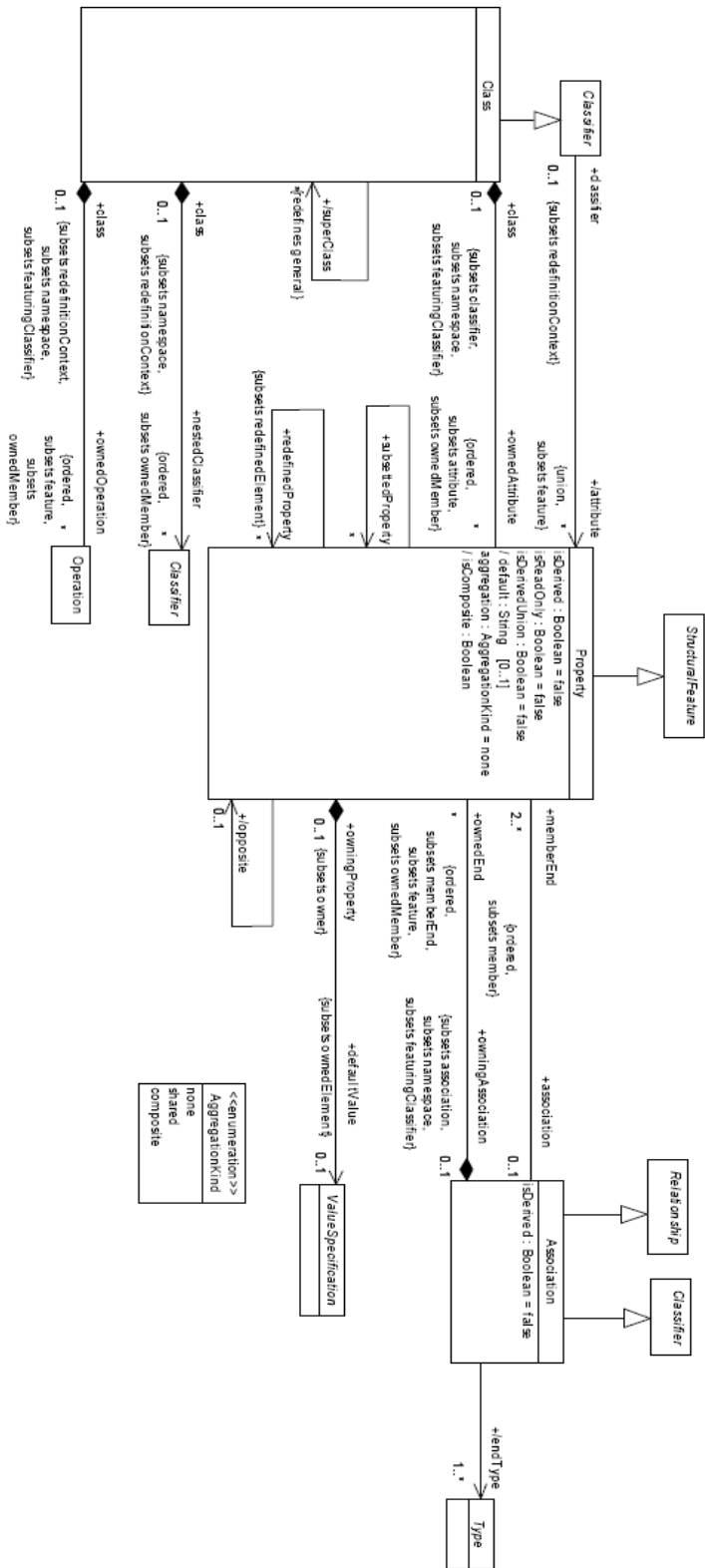
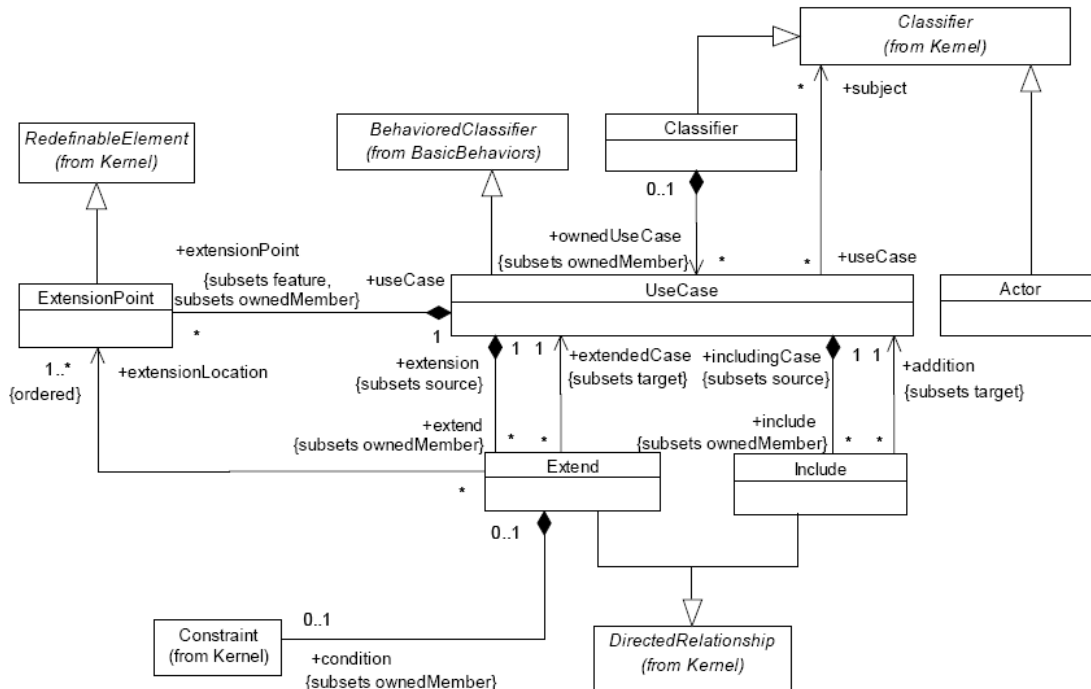


Figura 2.17 Metamodelo del Diagrama de Clases.



**Figura 2.18 Metamodelo de Casos de Uso**

Nótese de la Figura 2.18 que la clase *UseCase* hereda del elemento *Classifier* (Véase Figura 2.16), el cual a su vez hereda del elemento *Namespace* (Véase Figura 2.15), que a su vez hereda del *NamedElement* (Véase Figura 2.15), de donde obtiene el atributo *name*.

### 2.6.5. Metamodelo del Modelo de Interfaces.

Este modelo se aleja del estándar definido por MOF, puesto que no se trata de un elemento de UML; sin embargo, la simbología que se emplea para la definición del metamodelo es similar a la definida por MOF. En este caso, el *DiagramElement* es un conjunto de *Property*, y este elemento es el que define cuál de los controles de la interfaz gráfica de usuario se está utilizando, dependiendo de su atributo *Key*. Así, si *Key=1*, el elemento se trata de un título (*Title*), en tanto que si *Key=4*, el elemento se trata de un cuadro de texto (*TextBox*). En

cualquiera de los casos, se dispone de un *LeafElement* (que hereda de *DiagramElement*), el cual permite definir las características gráficas del elemento que se está incorporando en una determinada interfaz. De *LeafElement* heredan *textElement*, *Image* y *GraphicPrimitive*, que terminan de puntualizar las características del elemento de interfaz.

Tanto los elementos definidos en MOF para los diagramas de clases y casos de uso, como los elementos correspondientes a las interfaces gráficas de usuario definidas en el metamodelo que usa el diagrama de clases como lenguaje de metamodelamiento, se emplean en el Capítulo 5 para expresar las reglas de consistencia entre estos tres artefactos en OCL. Por ello, los elementos nombrados en ese Capítulo podrán ser consultados en los metamodelos aquí definidos.

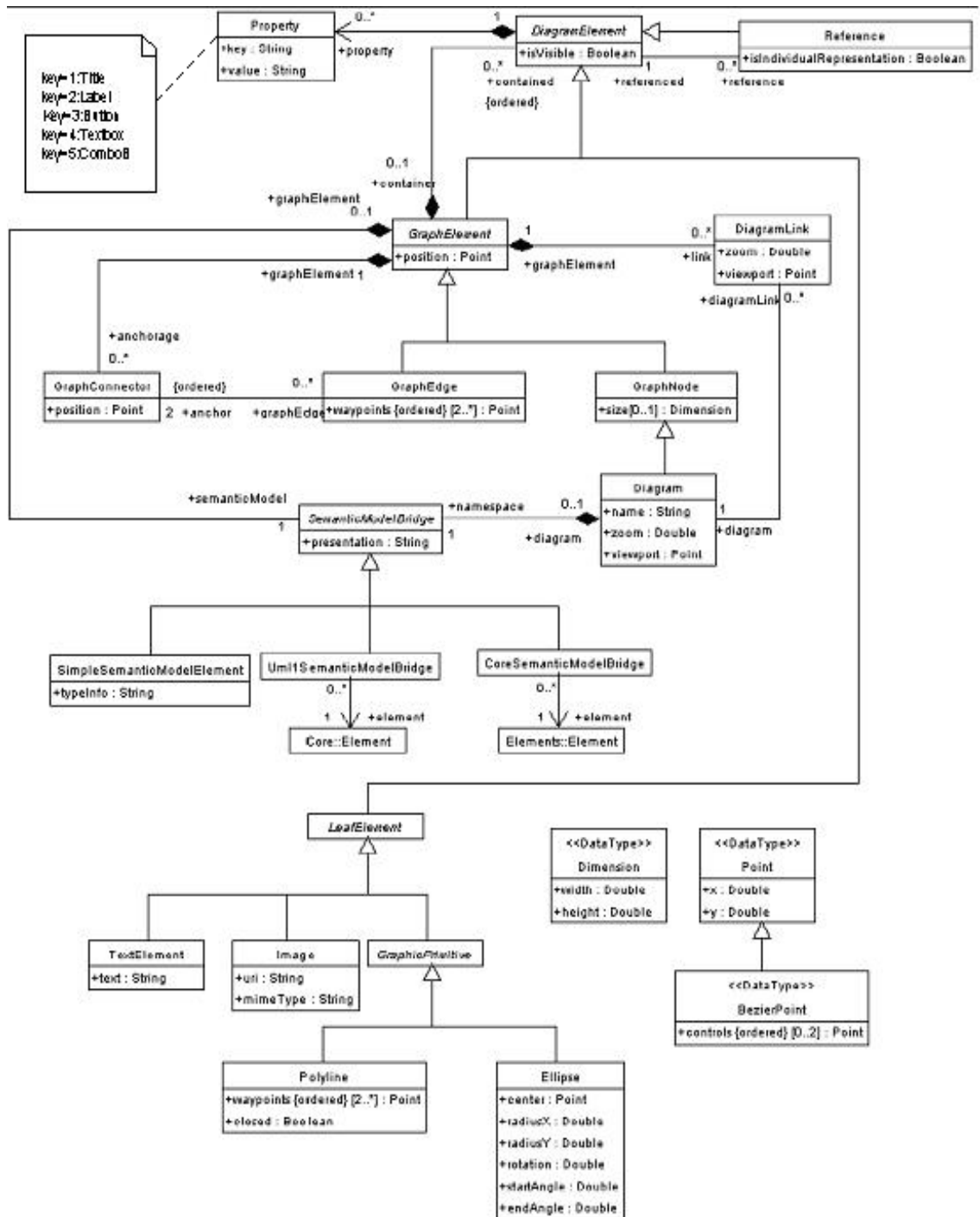


Figura 2.19 Metamodelo para el modelo de Interfaces (Blankenhorn,2004)

## Capítulo 3

### Problemática General de Investigación

#### 3.1 Limitaciones Encontradas

Existe gran cantidad de trabajos que abordan el problema de la consistencia a nivel de intramodelo tanto de manera formal como informal; sin embargo, son pocos los autores que han trabajado la consistencia entre modelos, la cual tampoco se ha expresado en lenguajes formales. Otros trabajos realizan consistencia entre diagramas y código resultante, lo que implica no poder comprobar reglas de consistencia antes de la implementación y las pruebas del software.

Para el caso específico, la mayoría de los trabajos que tratan el tema de la consistencia entre el modelo de clases y el modelo de casos de uso de UML lo hacen de una manera informal, en la que recurren al procesamiento de lenguaje natural para garantizar la correspondencia y completitud de los elementos de ambos diagramas, utilizando principalmente los escenarios y la descripción de los casos de uso para esto.

Por otro lado, ninguno de los trabajos recopilados tiene en cuenta las interfaces gráficas de usuario para el manejo de la consistencia entre estos modelos, aún a sabiendas de la alta relación que existe entre éstas y ambos diagramas. Por lo tanto, se plantea como una solución a las limitaciones anotadas el planteamiento de un conjunto de reglas de consistencia entre el diagrama de casos de uso y el diagrama de clases de UML teniendo en cuenta las interfaces gráficas de usuario, así como la definición de una especificación formal de estas reglas de consistencia; de esta manera, además de permitir un chequeo de la consistencia entre los dos diagramas, se puede involucrar la validación que de ellos puedan realizar los interesados por medio de las interfaces gráficas de usuario.

## 3.2 Definición del Problema

El tema de la consistencia se ha trabajado más a nivel intramodelo, donde se verifica que todos los elementos de un mismo diagrama o artefacto sean consistentes entre sí, pero sin tener en cuenta las relaciones con elementos de otros artefactos que pertenecen al modelo. En la parte de intermodelos, el trabajo desarrollado en consistencia ha sido relativamente poco, donde se muestran propuestas de métodos aislados para llevar a asegurar consistencia entre diferentes modelos, pero en general no se muestra un mecanismo formal que pueda ser aplicado por medio de un lenguaje de especificación como el OCL. En general, la falta de formalismo en la definición de las reglas puede conducir a problemas de ambigüedad en la interpretación de tales reglas, y finalmente a una implementación errónea de las mismas.

Se necesita, por tanto, una especificación formal de reglas de consistencia entre el modelo de casos de uso y el modelo de clases, empleando un lenguaje de especificación (que en este caso será OCL); estas reglas se podrían aplicar desde las fases de definición y análisis, del ciclo de vida del software, donde se tienen versiones preliminares de ambos diagramas, o incluso en fases posteriores de diseño previas a la implementación, donde los diagramas son más refinados por la cantidad de detalles que se deben precisar en ese proceso. Con ello, se busca que los analistas no inviertan demasiado tiempo y dinero en la revisión exhaustiva de los diagramas en busca de su consistencia, y que se detecten los errores potenciales en fases relativamente tempranas del desarrollo.

Las reglas que se planteen deberán tomar en consideración no sólo los errores que se pueden cometer al trabajar con puntos de vista independientes, sino también advertencias de posibles errores que pueden llegar a ser nocivos para el modelamiento que se está realizando del producto software. De esta manera, podría ser posible advertir a los analistas de los errores reales y potenciales que están cometiendo en la generación de los diagramas, como una especie de extensión de las reglas de buena formación de los mismos, pero tomando en consideración la interrelación con otros diagramas.

Un método como el que se plantea deberá resolver las siguientes preguntas de investigación:

- ¿Qué reglas se deben cumplir en el diagrama de clases y el de casos de uso de UML para garantizar la consistencia entre ellos?
- ¿Dada una especificación formal de estas reglas, cómo implementar un mecanismo que permita aplicarlas?
- ¿Cuáles aspectos del diagrama de casos de uso tienen relación directa con el diagrama de clases?
- ¿Cuáles son los factores que no se tratan en las herramientas CASE convencionales para el manejo de consistencia entre estos diagramas?

## **3.3 Objetivos**

### **3.3.1 Objetivo General**

Proponer una especificación formal de reglas de consistencia en OCL entre el modelo de clases y de casos de uso para el tratamiento adecuado de la consistencia entre estos modelos y ensayar dicha especificación en una herramienta MetaCASE.

### **3.3.2 Objetivos Específicos**

- Definir en lenguaje natural las reglas de consistencia entre el diagrama de clases y el diagrama de casos de uso (UML).
- Formalizar en OCL las reglas definidas.
- Ensayar, a modo de prototipo, las reglas formalizadas en una herramienta MetaCASE.
- Proponer y ejecutar un mecanismo de validación de las reglas definidas y formalizadas, con un caso de estudio.

## Capítulo 4

### Revisión de la Literatura

La principal fuente de reglas de consistencia para los diagramas de UML es la especificación misma emanada por el OMG (OMG, 2007). En dicha especificación, se incluyen algunas reglas de consistencia intramodelos, ya sea de los diagramas de casos de uso o de los diagramas de clases, pero no se definen reglas de consistencia intermodelos. Esas reglas intramodelos se encuentran plenamente definidas tanto en lenguaje natural como en OCL y algunas de ellas se encuentran incorporadas en algunas herramientas CASE convencionales, como es el caso de ArgoUML®.

Para el manejo de estas reglas de consistencia entre modelos UML se han trabajado básicamente algunos métodos:

- Xlinkit (Gryce, *et al.*, 2002) es un entorno para chequear la consistencia de documentos heterogéneos distribuidos. En esta propuesta se hace uso de XML (W3C, 2007), Xpath (W3C, 2007), Xlink (W3C, 2007) y DOM (W3C, 2007), y está conformada por un lenguaje basado en lógica de primer orden, que permite expresar restricciones entre documentos, un sistema de manejo de documentos y un motor que chequea las restricciones contra los documentos. Para explicar la operación de Xlinkit, se muestra en un ejemplo el caso de dos desarrolladores trabajando independientemente en el mismo sistema, con su información guardada en diferentes máquinas. Uno está especificando un modelo UML y el otro trabajando en una implementación en Java (SUN, 2007). El objetivo es chequear una restricción simple, que cada clase en el modelo UML tenga que ser implementada como una clase en Java. Esta restricción se debe satisfacer en el modelo y en la implementación para que sean consistentes. Xlinkit provee “conjuntos de documentos” para incluir los documentos y “conjuntos de reglas” para seleccionar las restricciones. La Figura 4.1 muestra el conjunto de documentos para el ejemplo, que consta de un documento de

UML (UMLmodel.xml), un documento Java (Main.java) y un documento para la definición de las reglas (ClassSet.xml). En la Figura 4.2 se muestra una restricción descrita en la codificación XML de Xlinkit, que establece que por cada clase que se encuentre en el documento UML debe existir una clase en el documento Java.

```
<DocumentSet name="UMLandJava">
  <Description>
    A UML model and some Java files
  </Description>

  <Document href="http://host1/UMLmodel.xml"/>
  <Document href="http://host2/Main.java" fetcher="JavaFetcher"/>
  <Set href="http://host2/ClassSet.xml"/>
</DocumentSet>
```

**Figura 4. 1 Ejemplo de conjunto de documentos**

```
<consistencyrule id="r1">
  <forall var="c" in="//UML:Class">
    <exists var="j" in="/java/class">
      <equal op1="$c/@name" op2="$j/@name"/>
    </exists>
  </forall>
</consistencyrule>
```

**Figura 4. 2 Restricción de ejemplo**

En tiempo de ejecución, Xlinkit aplica las expresiones XPath en la restricción a todos los documentos del conjunto, y así construye un conjunto de nodos para ser chequeados. La Figura 4.3 muestra 2 hipervínculos en una base de vínculos Xlinkit que ha sido generada de la regla de consistencia. Se han generado “vínculos consistentes” entre elementos consistentes, y “vínculos inconsistentes” entre elementos no consistentes.

```

<xlinkit:LinkBase docSet="DocSet.xml" ruleSet="RuleSet.xml">
  <xlinkit:ConsistencyLink ruleid="rule.xml#id('r1')">
    <xlinkit:State>consistent</xlinkit:State>
    <xlinkit:Locator xlink:href="http://host1/UMLmodel.xml#/UML:Class[1]"/>
    <xlinkit:Locator xlink:href="http://host2/Main.java#/java/class" fetcher="JavaFetcher"/>
  </xlinkit:ConsistencyLink>
  <xlinkit:ConsistencyLink ruleid="rule.xml#id('r2')">
    <xlinkit:State>inconsistent</xlinkit:State>
    <xlinkit:Locator xlink:href="http://host1/UMLmodel.xml#/UML:Class[2]"/>
  </xlinkit:ConsistencyLink>
</xlinkit:LinkBase>

```

**Figura 4. 3 Hipervínculos resultantes.**

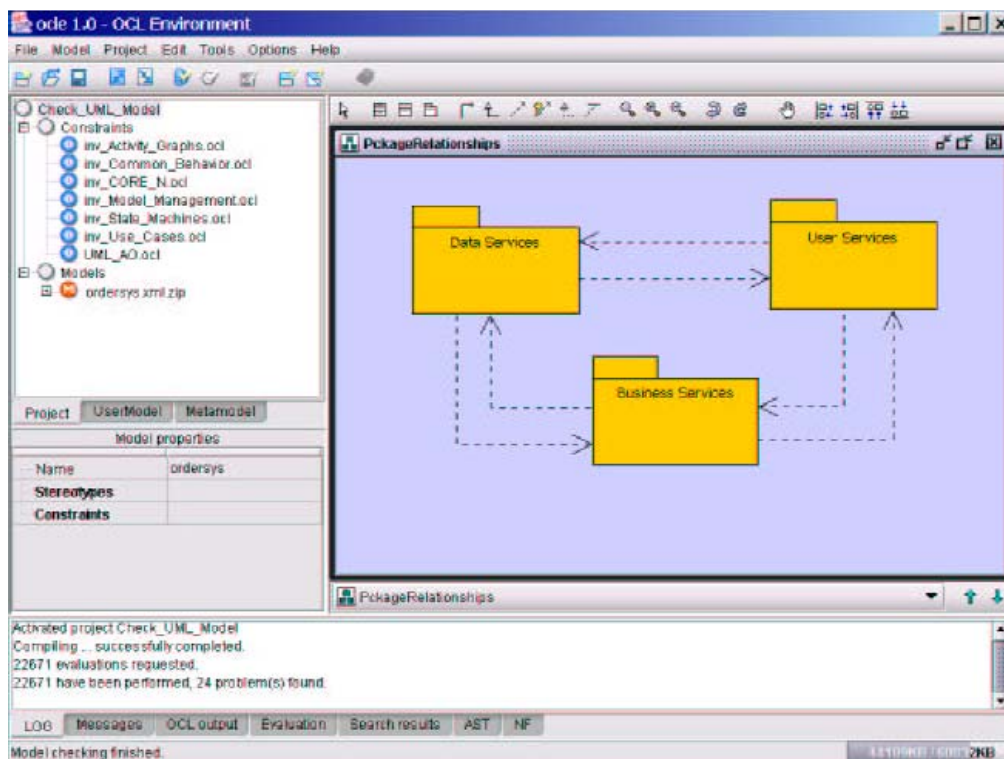
Se puede observar que la clase UML ha sido vinculada a la clase Java que conforma, y que la clase UML inconsistente se ha identificado, pero no ha sido vinculada a algo, porque no tiene clase Java correspondiente.

Xlinkit soporta el chequeo de documentos UML contra las reglas de buena formación para los elementos estáticos de UML; sin embargo, no incluye las reglas para los elementos dinámicos del metamodelo, lo que es necesario para una buena revisión de la consistencia entre diferentes tipos de modelos. Además, las reglas usadas para probar los modelos UML fueron traducidas de las reglas de buena formación del OMG. Ya que esas reglas tienen varias desventajas, los resultados obtenidos en el chequeo de las Reglas de Buena Formación de los modelos UML no son siempre correctos. Los resultados se dan como un reporte, que menciona sólo si una regla fue evaluada a falso o a verdadero. Por lo tanto, esta información no es útil en la identificación de causas de la falla de alguna regla.

- En el trabajo de Dan Chiorean (Chiorean, *et al.*, 2003) se trabaja con OCL (Booch, *et al.*, 1997) para chequear la consistencia de modelos UML. Se trabaja con una herramienta CASE UML, Object Constraint Language Environment (OCLE) (OCLE, 2002), la cual es totalmente compatible con OCL y soporta nivel de modelo y

metamodelo. Esta herramienta es compatible con XMI (OMG, 2007). Puede trabajar con modelos UML generados por las principales herramientas CASE que están disponibles actualmente (Together, Rational Rose, MagicDraw, Poseidon, ArgoUML) (Chiorean, *et al.*, 2003). La herramienta exporta también diagramas UML en formato XMI para que después puedan ser importados y modificados en cualquier herramienta que soporte XMI. OCLE tiene la habilidad de desarrollar diferentes tipos de chequeo de los modelos UML y corregir los errores identificados.

Como ejemplo para chequear la consistencia UML del modelo contra las WFR se trabaja con el “ordersys”. Es una aplicación de sistema de pedidos desarrollada para una compañía distribuidora de comida de mar. Este modelo incluye diferentes librerías de Visual Basic. El ejemplo contiene el modelo de aplicación y el proyecto de Visual Basic asociado. El modelo fue exportado en formato XMI desde Rational Rose e importado en OCLE. El proyecto OCLE contiene el modelo UML y el conjunto de reglas OCL guardado en varios archivos:



#### Figura 4. 4 Explorador de proyectos, panel de salida y un diagrama de clases.

Las reglas usadas en el ejemplo son las WFR que definen la semántica estática de UML. Considerando que el proyecto de Visual Basic asociado es ejecutable, se busca comprobar si la información del diseño del modelo es consistente con la información de la implantación del modelo. El resultado de la primera evaluación se muestra en el panel de salida (Figura 4.4): se encontraron 24 problemas de 22671 evaluaciones realizadas.

La primera regla que se consideró es definida en el contexto de la metaclass *Namespace*:

```
context Namespace

inv WFR_1_Namespace:
-- [1] If a contained element, which is not an Association or Generalization
-- has a name, then the name must be unique in the Namespace.

let noe: Set(ModelElement) = self.ownedElement->reject(e |
    e.oclIsKindOf(Association) or e.oclIsKindOf(Generalization)) in
if noe->reject(e | e.name='')->isUnique(e | e.name)
then true
else noe->select(e | noe->exists(ae | ae <> e and e.name =
    ae.name))->sortedBy (e | e.name)->isEmpty
endif
```

Una traducción de la especificación informal es más simple:

```
self.ownedElement->reject(e | e.oclIsKindOf(Association) or
e.oclIsKindOf(Generalization))->isUnique(e | e.name)
```

Los elementos del modelo sin nombre fueron rechazados porque, aparte de las asociaciones y generalizaciones hay otros elementos del modelo (como dependencias, instancias, etc.) cuyos nombres no se pueden definir en la mayoría de las herramientas CASE actuales (Chiorean, *et al.*, 2003).

Los elementos que causaron la falla fueron cuatro roles clasificadores, dos de ellos llamados *NewRow* y los otros llamados *dlg\_Order* (ver figura 4.5). Cambiar los nombres de esos elementos resolverá el problema.

Los resultados obtenidos en este y otros ejemplos (Chiorean, *et al.*, 2003) demuestran que usar OCL en el chequeo de reglas de consistencia del modelo UML es un muy buen acercamiento, ya que define todo lo concerniente a las reglas de consistencia de modelos UML en el nivel del metamodelo, por lo que esas reglas son independientes del modelador, soportando su reutilización en cualquier modelo UML. Esta aproximación sólo soporta la validación de la semántica de diagramas individuales de UML, manejando así la consistencia a nivel intramodelos.

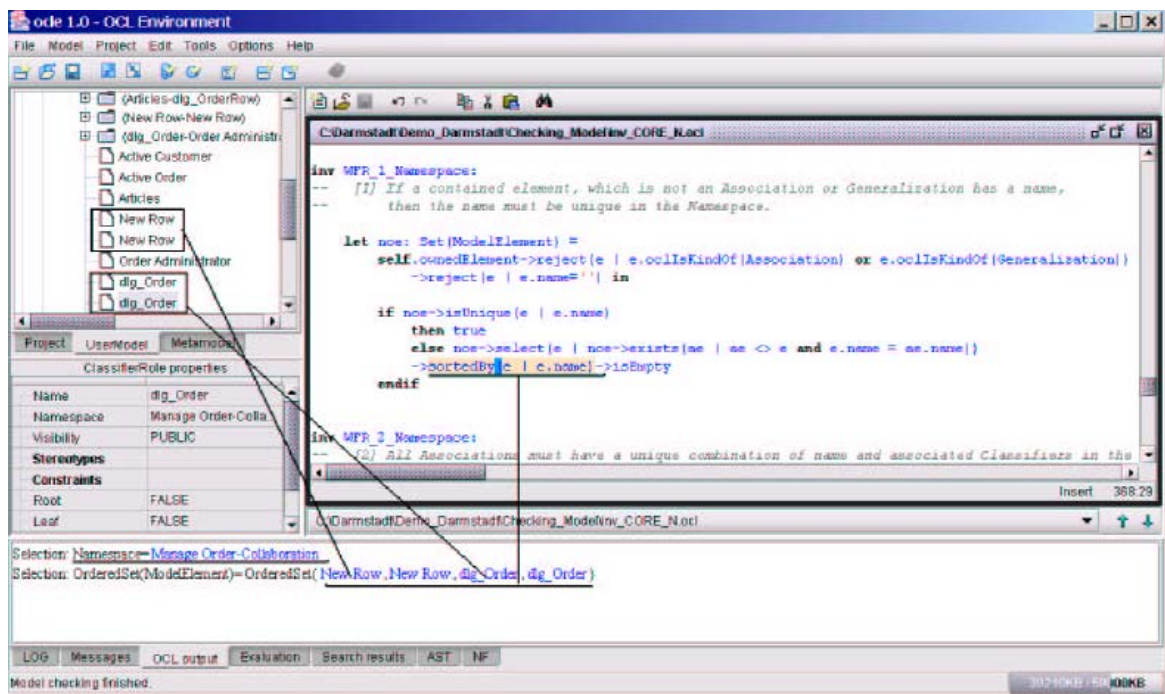


Figura 4. 5 Dos conflictos de nombre en el *Namespace*.

En los acercamientos que tratan la consistencia específicamente entre el diagrama de casos de uso y el diagrama de clases se encuentran los siguientes trabajos:

- En el trabajo de Kösters, Pagel y Winter (Kösters, *et al.*, 1997) se asocian el diagrama de casos de uso y el diagrama de clases al derivar el modelo de clases de los casos de uso, y anotando una especificación gráfica de casos de uso con los nombres de las clases que implementan ese caso de uso. Esta es una vista orientada al diseño donde el usuario procede del modelo de casos de uso al modelo de clases, y del modelo de clases a la implementación, sin embargo no es una aproximación formal.
- Un trabajo similar se encuentra en el trabajo de Liu (Liu, *et al.*, 2003), quienes automatizan el proceso de generación del diagrama de clases tomando como punto de partida las especificaciones textuales de los casos de uso, definiendo para ello unas plantillas especiales que capturan la información. En este caso, no se define la consistencia entre los dos tipos de diagrama, sino que se usa una descripción en un lenguaje controlado y de allí se obtiene el diagrama de clases.
- De manera análoga, Shiskov (Shiskov, *et al.*, 2002), procuran el mismo fin, pero partiendo de lenguaje natural generan el diagrama de clases utilizando como punto intermedio el diagrama de casos de uso. Para lograr ese fin, emplean los denominados análisis de normas, un conjunto de reglas heurísticas que examinan plantillas de información para transferirlas a los dos diagramas. Por el hecho de generar ambos diagramas desde la misma especificación textual, la consistencia queda garantizada, pero no se definen las reglas de consistencia que deberían existir entre los dos diagramas una vez se modifiquen.
- El trabajo de Glinz (Glinz, 2000) también es similar, pues trabaja el diagrama de casos de uso y el diagrama de clases como complementos de una misma especificación de requisitos, lo cual ninguno de los acercamientos mencionados lo hace; sin embargo, el punto de partida es, nuevamente, una especificación textual de los casos de uso en un formato especial. Además, el análisis que se realiza requiere una alta participación del analista en el proceso de integración de ambos artefactos, lo cual hace que su automatización se dificulte.

- Buhr (Buhr, 1998) mira los casos de uso como caminos causales que van hacia un modelo de objetos jerárquico. Los puntos de responsabilidad unen segmentos de un camino a elementos del modelo objetual. El modelo objetual y los caminos del diagrama de casos de uso se visualizan juntos en los llamados mapeos de casos de uso. La aproximación de Buhr es orientada al diseño también. Además el camino a través del modelo objetual es todo lo que se conoce del caso de uso; no hay especificación independiente de los casos de uso. Así, la importancia de la separación de lo orientado al usuario se pierde, lo que es una de las fortalezas de la especificación basada en casos de uso. A diferencia de (Kösters, *et al.*, 1997), quienes se enfocan en la transición de los casos de uso al diagrama de clases, Buhr usa los casos de uso para visualizar la dinámica del modelo objetual (Buhr, 1998).
- El trabajo de Grundy, Hosking y Mugridge (Grundy, *et al.*, 1998) también maneja la consistencia intermodelos, pero difiere del de Glinz en que se enfocan en herramientas de entorno de desarrollo de software, manejando inconsistencias en múltiples vistas que son derivadas de un repositorio común. Los artefactos en el repositorio se representan formalmente por una clase especial de gráfico, permitiendo así detección automática de inconsistencia.
- Sunetnanta y Finkelstein (Sunetnanta, *et al.*, 2003) presentan un enfoque para el chequeo de consistencia intermodelos basado en la conversión de los diferentes diagramas en grafos conceptuales y la definición de las reglas de consistencia en estos mismos grafos. Los grafos conceptuales no pueden ser considerados como un enfoque formal para la elaboración de este tipo de especificaciones, sino más bien un enfoque semiformal. Además, definen reglas entre los diagramas de casos de uso y colaboración (el de la versión 1.4 de UML, que actualmente se denomina diagrama de comunicación en la versión 2.0), y no definen las reglas de consistencia entre diagramas estructurales y dinámicos.

Un aspecto a considerar es que todos estos trabajos no incluyen las interfaces de usuario como medio complementario y necesario para validar la consistencia entre dichos modelos, sabiendo que hay una alta correlación entre el modelo de casos de uso e Interfaces, ya que el modelo de casos de uso está motivado y enfocado principalmente hacia los sistemas de información donde los usuarios juegan un papel primordial, por lo que es importante relacionarse con las interfaces a ser diseñadas en el sistema (Weitzenfeld, 2005).

El trabajo de Glinz (Glinz, 2000) se diferencia específicamente del planteado en esta Tesis en que no hace uso de un lenguaje formal para generar reglas de consistencia, sino que lo hace de modo informal al completar la documentación de los casos de uso con el mayor detalle posible, y como se mencionó antes, tampoco hace uso de las interfaces para tratar el problema de la consistencia, aspecto que debe ser tenido en cuenta, dado que estas interfaces sirven para apoyar de mejor manera la descripción de los casos de uso además de servir de base para prototipos iniciales.

# Capítulo 5

## Método Propuesto

### 5.1. Definición de reglas de consistencia.

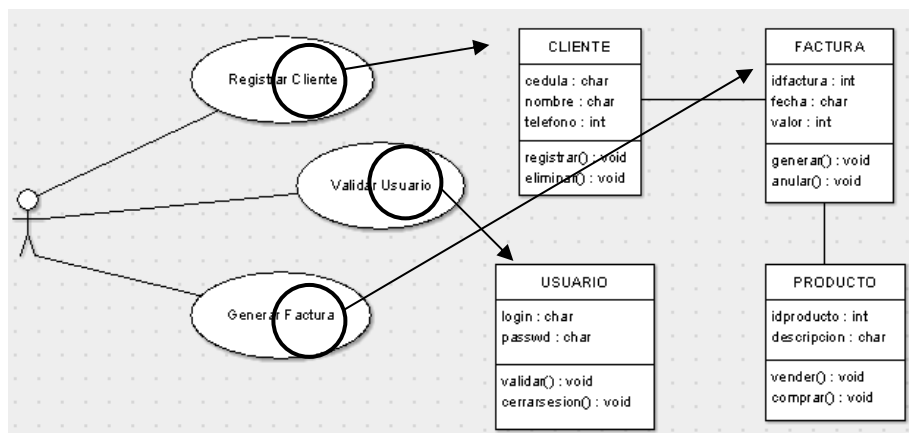
Para definir las reglas de consistencia entre el diagrama de casos de uso y el diagrama de clases UML, se debe definir también el alcance del problema particular, que es tratado bajo las siguientes premisas:

- Cada clase se distingue por su nombre, un conjunto de atributos o propiedades y un conjunto de operaciones ofrecidas por la clase.
- El modelo de casos de uso consta de actores que representan los roles que los diferentes usuarios pueden desempeñar, y los casos de uso que representan lo que deben estar en capacidad de hacer los usuarios con el software por construir. Otro término importante son los escenarios, que corresponden a secuencias específicas de acciones e interacciones entre los actores y el sistema; los escenarios, sin embargo, no serán tenidos en cuenta para esta propuesta, debido a que son una especificación no formal de los pasos llevados a cabo en cada caso de uso y tendrían que ser tratados por procesamiento de lenguaje natural, por lo que están fuera del alcance de esta Tesis, en la cual se busca proponer una especificación formal.
- El modelo de interfaces especifica cómo interactúa el software por construir con actores externos al ejecutar los casos de uso; en particular, en los sistemas de información ricos en interacción con el usuario, especifica cómo se visualizarán las interfaces gráficas de usuario y qué funcionalidad ofrecerá cada una de ellas (Weitzenfeld, 2005). Para el trabajo propuesto, una interfaz común consta de un título, etiquetas, campos de texto, campos de selección, uno o varios botones de enviar (submit) y un botón de cancelar o salir. Se suponen interfaces sencillas que sólo involucran elementos de una clase.

Para la elaboración de este trabajo, se toma en consideración la relación existente entre el diagrama de clases (OMG, 2007), el diagrama de casos de uso (Jacobson, 1992) y las interfaces gráficas de usuario (Weitzenfeld, 2005). Además, se considera el análisis heurístico que se puede obtener a partir de la experiencia por parte de analistas de software y, tomando como base la estructura de los metamodelos de esos diagramas, los cuales se presentaron en la Sección 2.6, se proponen 8 reglas de consistencia, que se presentan seguidamente.

### 5.1.1 Regla 1

El nombre de un caso de uso debe incluir un verbo y un sustantivo. El sustantivo debe corresponder al nombre de una clase del diagrama de clases. En otras palabras, para todo caso de uso U del diagrama de casos de uso, debe existir una clase C perteneciente al diagrama de clases, tal que U.name contenga a C.name. La expresión gráfica de esta regla se puede apreciar en la Figura 5.1.



**Figura 5.1. Descripción gráfica de la Regla 1.**

La expresión en OCL que representa esta regla es la siguiente:

Classifier

*self.UseCase->exists(us: Usecase, c: Class, x: Integer, y: Integer | y > x and us.name.toUpper.substring(x,y)=c.name.toUpper)*

### 5.1.2 Regla 2

Como se especificó en la sección 5.1.1, el nombre de un caso de uso debe incluir un verbo y un sustantivo. El verbo debe corresponder a una operación de una clase del diagrama de clases que se identificó en la Regla 1. En otras palabras, para todo Caso de Uso U debe existir una Clase C que contenga una operación Operation<sub>x</sub> tal que U.name contenga a C.Operation<sub>x</sub>. La expresión gráfica de esta regla se puede apreciar en la Figura 5.2.

La expresión en OCL que representa esta regla es la siguiente:

Classifier

*self.UseCase->exists(us: Usecase, c: Class, x: Integer, y: Integer | y > x and us.name.toUpper.substring(x,y)=c.operation.toUpper)*

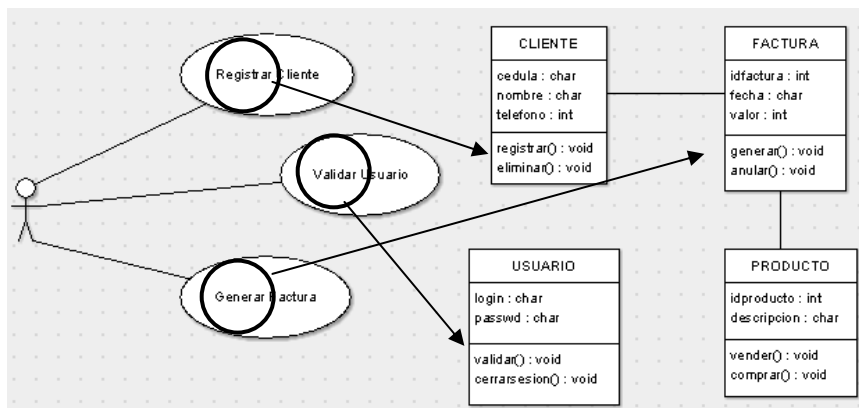


Figura 5.2. Descripción gráfica de la Regla 2.

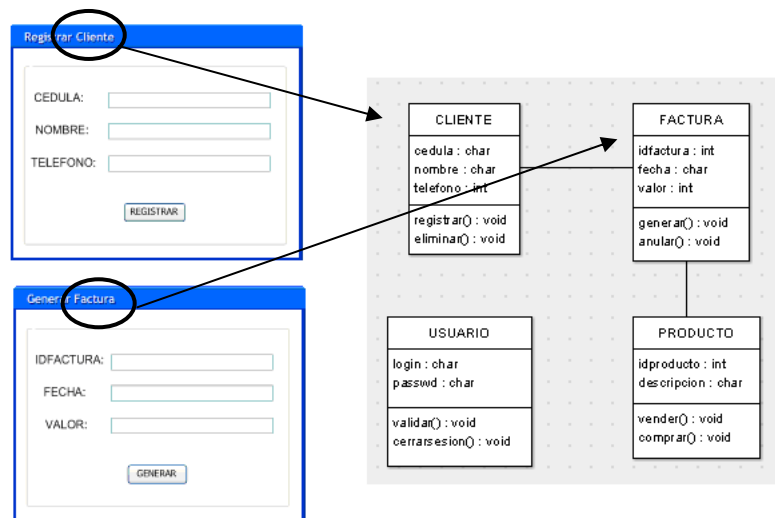
### 5.1.3 Regla 3

En el título de cada interfaz gráfica de usuario debe ir un verbo y un sustantivo. El sustantivo debe corresponder al nombre de una clase que se encuentre en el diagrama de clases. Dicho de otra forma, para toda interfaz de usuario I debe existir una clase C tal que  $I.key=1$  (título) y que  $I.value$  contenga a  $C.name$ . La expresión gráfica de esta regla se puede apreciar en la Figura 5.3.

La expresión en OCL que representa esta regla es la siguiente:

#### Classifier

*self.DiagramElement->exists(de: DiagramElement, c: Class, x: Integer, y: Integer |  $y > x$  and  $de.key=1$  and  $de.value.toUpper.substring(x,y)=c.name.toUpper$ )*



**Figura 5.3. Descripción gráfica de la Regla 3.**

### 5.1.4 Regla 4

Como se especificó en la sección 5.1.3, una Interfaz gráfica de usuario tiene en su título un verbo y un sustantivo. Dicho verbo debe corresponder a una operación de la clase identificada en la Regla 4; dicho de otra forma, para toda interfaz de usuario I debe existir una clase C que contenga una operación  $Operation_x$  en la que  $I.key=1$  (título) y que  $I.value$  contenga a  $C.Operation_x$ . La expresión gráfica de esta regla se puede apreciar en la Figura 5.4.

La expresión en OCL que representa esta regla es la siguiente:

Classifier

*self.DiagramElement->exists(de: DiagramElement, c: Class, x: Integer, y: Integer /  $y > x$  and  $de.key=1$  and  $de.value.toUpperCase.substring(x,y)=c.operation.toUpperCase$ )*

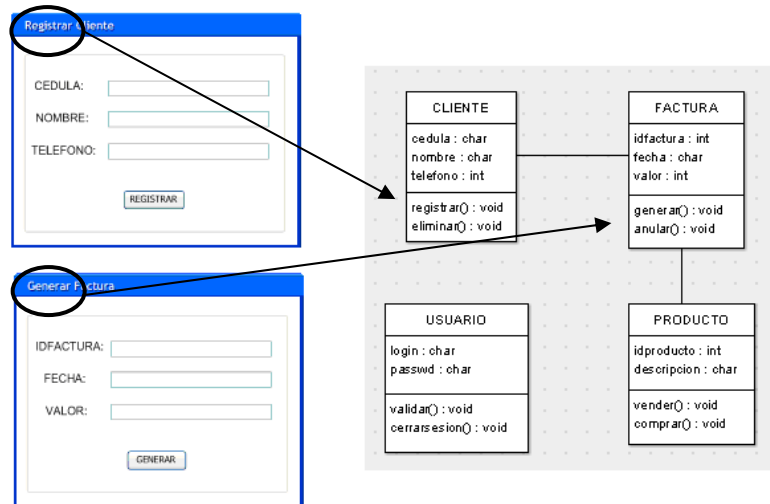


Figura 5.4. Descripción gráfica de la Regla 4.

### 5.1.5 Regla 5

En una interfaz gráfica de usuario debe existir un formulario con un botón para aplicar cambios o enviar información a otro formulario. Dicho botón tiene en su etiqueta un verbo. Dicho verbo debe corresponder a una operación de una clase. Dicho de otra manera, para toda interfaz de usuario I en la que exista un botón de enviar (submit) B debe existir una clase C que contenga una operación  $Operation_x$  tal que  $I.key=3$  (button) y  $I.value$  (label) contenga a  $C.Operation_x$ . La expresión gráfica de esta regla se puede apreciar en la Figura 5.5.

La expresión en OCL que representa esta regla es la siguiente:

Classifier

*self.DiagramElement->exists(de: DiagramElement, c: Class, x: Integer, y: Integer |  $y > x$  and  $de.key=3$  and  $de.value.toUpperCase().substring(x,y)=c.operation.toUpperCase()$ )*

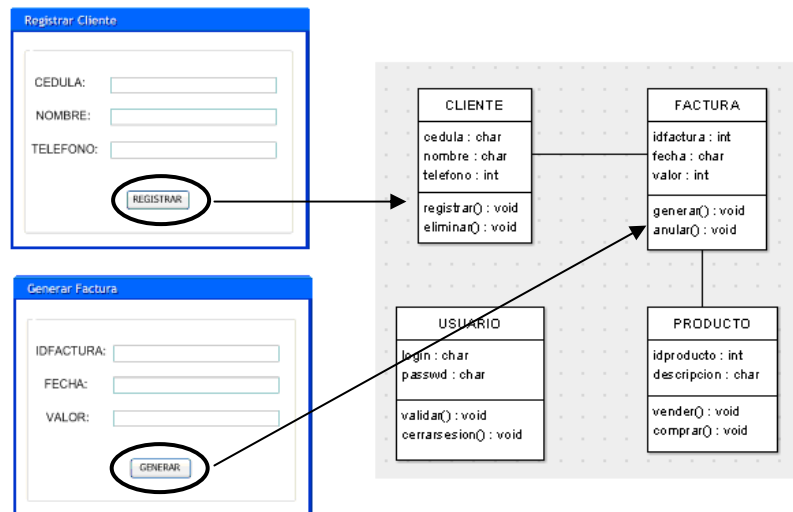


Figura 5.5. Descripción gráfica de la Regla 5.

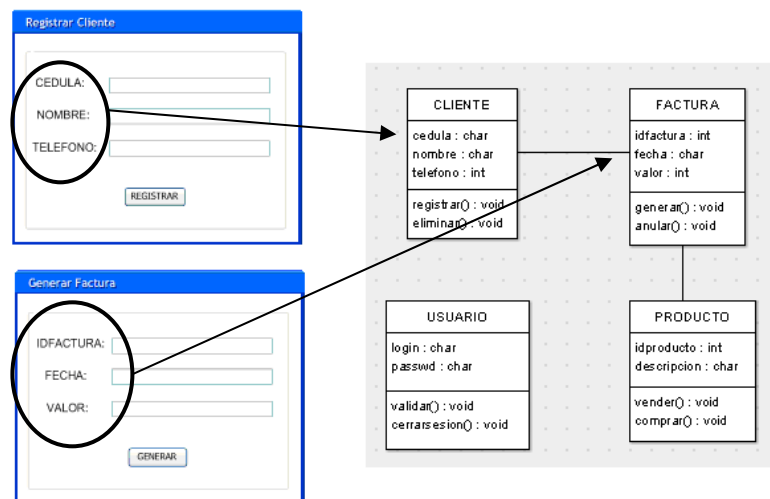
## 5.1.6 Regla 6

Si una interfaz gráfica de usuario posee campos de texto para que el usuario ingrese información, estos campos deben ir precedidos por sus respectivas etiquetas, las cuales informan acerca de lo que se debe digitar en los campos. Dichas etiquetas deben tener sus atributos correspondientes en una clase. Dicho de otra forma, para toda interfaz de usuario  $I$  en la que existan etiquetas (labels)  $E_1, E_2, E_3, \dots, E_q$  con sus respectivos campos de texto  $T_1, T_2, T_3, \dots, T_o$  y/o campos de selección  $S_1, S_2, S_3, \dots, S_p$  debe existir una clase  $C$  con atributos  $A_1, A_2, A_3, \dots, A_n$ , que contengan a las etiquetas  $E_x$ . La expresión gráfica de esta regla se puede apreciar en la Figura 5.6.

La expresión en OCL que representa esta regla es la siguiente:

### Classifier

*self.DiagramElement->forall(de: DiagramElement, c: Class, x: Integer, y: Integer | de->exists(de.key=3) and (de.key=4 or de.key=5) implies c.attribute->includes(de.value))*



**Figura 5.6. Descripción gráfica de la Regla 6.**

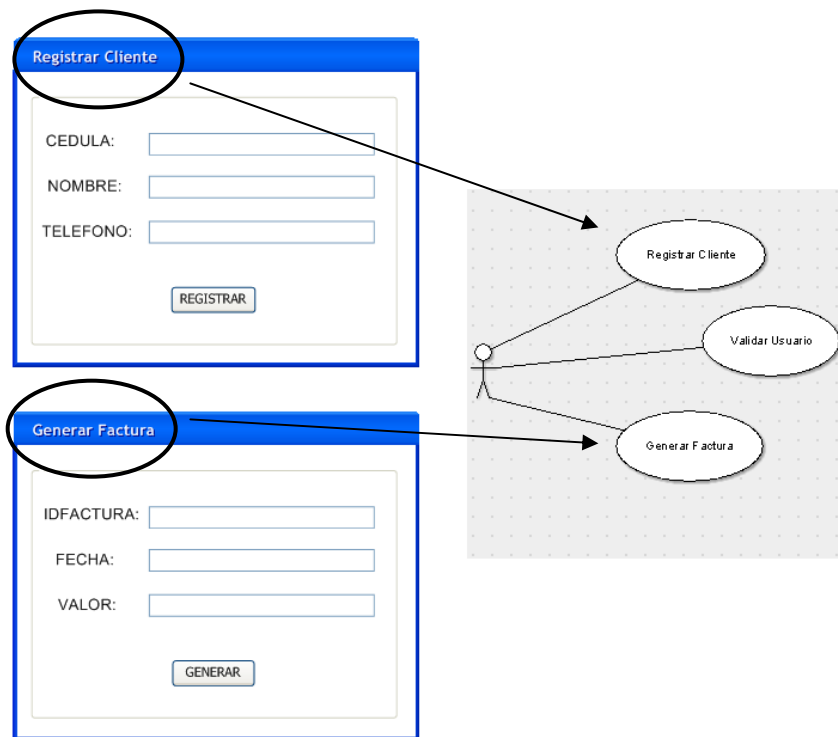
### 5.1.7 Regla 7

Como se describió en la sección 5.1.3, en el título de cada interfaz gráfica de usuario debe ir un verbo y un sustantivo. Dicho verbo y sustantivo debe corresponder con el nombre de un caso de uso, los cuales también están compuestos de un nombre y un sustantivo. En otras palabras, para toda interfaz de usuario I debe existir un Caso de Uso U en el que  $I.key=1(\text{título})$  y que  $I.value = U.name$ . La expresión gráfica de esta regla se puede apreciar en la Figura 5.7.

La expresión en OCL que representa esta regla es la siguiente:

Classifier

*self.DiagramElement->exists(de: DiagramElement, uc: UseCase | de.key=1 and de.value.toUpper=uc.name.toUpper)*



**Figura 5.7. Descripción gráfica de la Regla 7.**

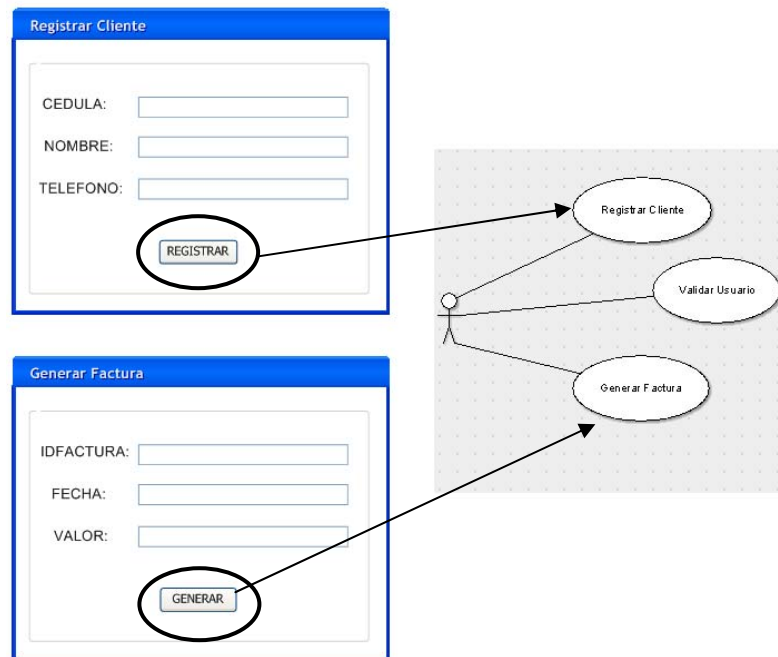
### 5.1.8 Regla 8

Como se especificó en la sección 5.1.5, en una interfaz gráfica de usuario debe existir un formulario con un botón para aplicar cambios o enviar información a otro formulario. Dicho botón tiene en su etiqueta un verbo. Dicho verbo debe corresponder a un verbo de un nombre de un caso de uso. Dicho de otra forma, para toda interfaz de usuario I en la que exista un botón de enviar (submit) B (I.key=3) debe existir un Caso de Uso U tal que U.name contenga a I.value. La expresión gráfica de esta regla se puede apreciar en la Figura 5.8.

La expresión en OCL que representa esta regla es la siguiente:

Classifier

*self.DiagramElement->exists(de: DiagramElement, uc: UseCase, x: Integer, y: Integer | y > x and de.key=3 and uc.name.toUpper.substring(x,y)=de.value.toUpper)*



**Figura 5.8. Descripción gráfica de la Regla 8.**

## 5.2 Validación de Reglas de Consistencia

Para la validación de las reglas mencionadas anteriormente, se hizo uso de dos herramientas que poseen la utilidad de exportar los contenidos de sus modelos en formato XMI (OMG, 2007): ArgoUML® y Visio®. Por estar ArgoUML® enfocado al manejo de diagramas UML, se trabajaron en esta herramienta los diagramas de casos de uso y de clases; en Visio® se trabajaron las Interfaces Gráficas de Usuario (GUI).

Después de hacer los respectivos modelos e interfaces, estos se exportan en formato XMI y luego se analizan con una herramienta que se desarrolló para efectos de este trabajo en el lenguaje de programación Java®. Esta herramienta hace uso del Lenguaje Xquery por medio de una aplicación especializada denominada *Saxonb* para la validación de las reglas. En este programa se evalúan las diferentes reglas con los tres modelos y para cada regla sale un informe que indica si la regla se cumple (estado correcto), si no se cumple (estado de error) o si posiblemente hay un error de sinónimos por lo que se presenta una advertencia (warning).

### 5.2.1 Integración de sinónimos

Con el fin de tener la posibilidad de mostrarle al usuario que posiblemente algunas de las inconsistencias encontradas se debieron al uso de sinónimos por parte del analista y que no son errores como el caso de que faltara una clase o un caso de uso, se da la posibilidad de mostrar advertencias donde se indique que el posible error puede ser debido a uso de palabras similares. Se implementó una lista de los sustantivos y verbos más utilizados en el ámbito del área de software, teniendo como base los entregables de varios semestres presentados por los estudiantes en la asignatura Ingeniería de software de la Universidad Nacional.

A continuación se presenta un fragmento del archivo XML generado:

```

<?xml version="1.0" encoding="UTF-8"?>
<equivalencias>
  <empleado>
    <sinonimo>trabajador</sinonimo>
    <sinonimo>operario</sinonimo>
  </empleado>
  <cedula>
    <sinonimo>identificacion</sinonimo>
  </cedula>
  <cargo>
    <sinonimo>funcion</sinonimo>
  </cargo>
  <salario>
    <sinonimo>sueldo</sinonimo>
  </salario>
  <registrar>
    <sinonimo>inscribir</sinonimo>
    <sinonimo>ingresar</sinonimo>
  </registrar>
  <validar>
    <sinonimo>autenticar</sinonimo>
    <sinonimo>ingresar</sinonimo>
    <sinonimo>autorizar</sinonimo>
    <sinonimo>aprobar</sinonimo>
    <sinonimo>admitir</sinonimo>
    <sinonimo>aceptar</sinonimo>
  </validar>
</equivalencias>

```

En la sección 5.2.2 se presenta un fragmento de la consulta Xquery que utiliza las reglas de consistencia definidas en Xpath para determinar si los modelos presentados en formato XMI son consistentes entre sí.

## 5.2.2 Consulta Xquery

Para las versiones ArgoUML 0.22 y Visio 2003 se tienen los *Namespaces* respectivos `org.omg.xmi.namespace.UML` y <http://schemas.microsoft.com/visio/2003/core> con los que se

conforma y valida el archivo XML resultante. Posteriormente, a este archivo se le aplica la consulta XQuery, la cual genera un archivo con estructura HTML para ser visualizado en un Browser convencional. En la aplicación se omiten los botones que tengan el texto ACEPTAR, CANCELAR, SALIR, CERRAR, APLICAR, CONTINUAR Y ENVIAR, ya que son botones comunes a todas las aplicaciones y no tiene sentido analizarlos. Un fragmento de la consulta realizada se presenta a continuación:

NOTA: La consulta completa, donde se muestra el código para cada regla, se puede consultar en los anexos.

```
<resultado xmlns:UML = 'org.omg.xmi.namespace.UML'> (: Es la raiz del documento :)
<br/> (: Los brs se utilizan para que haya cambio de linea una vez generado el archivo XML :)
<grafico xmlns='http://schemas.microsoft.com/visio/2003/core'>
{
for $itemgrafico in doc('visio.xml')/VisioDocument/Pages/Page/Shapes/Shape (: Se recorren los elementos gráficos del archive
de visio :)
return
<br>
<br>
{if ($itemgrafico/@NameU="Blank form") then (: Cuando hay un formulario :)
<formulario>
$itemgrafico/Text/text() (: Se muestra el texto del formulario (titulo) :)
else
"no es formulario"
</formulario>
}
</br>
<textos>
{$itemgrafico/text()}
</textos>
</br>
}<br/>
</grafico>
{for $clases in doc('argo.xml')/XMI/XML.content/UML:Model/UML:Namespace.ownedElement/UML:Class
let $atributos:=$clases/UML:Classifier.feature/UML:Attribute/@name (: Se crea una secuencia con los atributos de las clases :)
let $operaciones:=$clases/UML:Classifier.feature/UML:Operation/@name (: Se crea una secuencia con las operaciones de las
clases :)
return
<br>
```

```

<clase><br/>
<nombre>
{upper-case($clases/@name)} (: Se escribe el nombre de la clase en mayúscula :)

</nombre>
    {for $i in 1 to count($atributos)
    return
        <br><atributo>{upper-case($atributos[position()=$i])}</atributo><br> (: Se imprimen los atributos :)
        }
    {for $i in 1 to count($operaciones)
    return
        <br><operacion>{upper-case($operaciones[position()=$i])}</operacion><br> (: Se imprimen las operaciones :)
        }
    <br/>
</clase><br/>
<br/>
{for $casos in doc('argo.xml')/XML/XML.content/UML:Model/UML:Namespace.ownedElement/UML:UseCase
let $operacion:= $casos/@name
return
<br>
<casouso>
    {for $i in 1 to count($operacion) (: Se imprimen los casos de uso :)
    return
        <br><nombre>{upper-case($operacion[position()=$i])}</nombre><br>
        }
    <br/>
</casouso>
<br/>
}
<br/>
<br/>
{
for $grafico in doc('graph.xml')/graph
for $diagrama in doc('argo.xml')/XML/XML.content/UML:Model/UML:Namespace.ownedElement

let $cuadrotexto:= $grafico/textbox (: Se crea una secuencia con los cuadros de texto :)
let $combo:= $grafico/combobox (: Se crea una secuencia con los cuadros combinados :)
let $label:= $grafico/label (: Se crea una secuencia con las etiquetas :)
let $button:= $grafico/button (: Se crea una secuencia con los botones :)

let $clase:= $diagrama/UML:Class
let $casouso:= $diagrama/UML:UseCase
let $operacion:= $diagrama/UML:Class/UML:Classifier.feature/UML:Operation

```

```

return
<consistencia>
<br/>
<existeclaseencasodeuso>{
for $i in 1 to count($clase)
for $j in 1 to count($casouso)
return
(: Se revisa que el nombre de la clase exista en el caso de uso :)
if (contains(upper-case($casouso[position()=$j]/@name), upper-case($clase[position()=$i]/@name))) then
("<br/>La clase <b> ", upper-case($clase[position()=$i]/@name), "</b> existe en el caso de uso <b>", upper-
case($casouso[position()=$j]/@name), "</b>")
else
("<br/>La clase <b> ", upper-case($clase[position()=$i]/@name), "</b> NO existe en el caso de uso <b>", upper-
case($casouso[position()=$j]/@name), "</b>")
}<br/></existeclaseencasodeuso><br/><br/>
<existeoperacionencasodeuso>
{
for $i in 1 to count($operacion)
for $j in 1 to count($casouso)
return
(: Se revisa que la operacion de la clase exista en el caso de uso :)
if (contains(upper-case($casouso[position()=$j]/@name), upper-case($operacion[position()=$i]/@name))) then
("<br/>La operacion <b>",upper-case($operacion[position()=$i]/@name), "</b> existe en el caso de uso <b>".upper-
case($casouso[position()=$j]/@name), "</b>")
else
("<br/>La operacion <b>".upper-case($operacion[position()=$i]/@name), "</b> NO existe en el caso de uso <b>".upper-
case($casouso[position()=$j]/@name), "</b>")
}<br/></existeoperacionencasodeuso><br/><br/>
{
<existeoperacionenboton>
{
for $i in 1 to count($button)
for $j in 1 to count($operacion)
return
(: Se revisa que la operacion de la clase exista en el texto del boton :)
if(contains(upper-case($button[position()=$i]/label),upper-case($operacion[position()=$j]/@name))) then
("<br/>La operacion <b> " ,upper-case($operacion[position()=$j]/@name), " </b> existe en el boton <b> " ,upper-
case($button[position()=$i]/label), " </b>")
else
("<br/>La operacion <b> " ,upper-case($operacion[position()=$j]/@name), " </b> NO existe en el boton <b> " ,upper-
case($button[position()=$i]/label), " </b>")
}<br/>
</existeoperacionenboton>

```

```

}
<br/>
<br/>
</consistencia>
}
<br/>
</resultado>

```

### 5.3 Comparación con otros trabajos

En las tablas 5.1 y 5.2 se muestra un cuadro comparativo de las características del método propuesto y de los proyectos y trabajos encontrados.

Característica	PROYECTO				
	Xlinkit	Sunetnanta	Chiorean, <i>et al.</i>	Kösters, <i>et al.</i>	Liu, <i>et al.</i>
Utiliza Procesamiento de Lenguaje Natural?	NO	SI	NO	SI	SI
Se realiza de manera formal?	NO	NO	SI	NO	NO
Se define un conjunto de reglas de consistencia intermodelos?	NO	SI	NO	NO	NO
Chequea consistencia inter-modelos?	Chequea consistencia entre diagramas y el código resultante	SI	NO	SI	NO
Chequea consistencia entre el diagrama de clases y de casos de uso de UML?	NO	NO	NO	SI	NO
Utiliza Representación intermedia?	SI	SI	SI	SI	SI
Tipo de Representación intermedia	Conjunto de Documentos y Reglas en XML	Grafos Conceptuales de Sowa	XMI, OCLE	Especificación gráfica de casos de uso	Descripción en un lenguaje controlado

Tabla 5.1 Comparación de trabajos afines al método propuesto (parte 1 de 2)

	<b>PROYECTO</b>				
<b>Característica</b>	<b>Shiskov <i>et al.</i></b>	<b>Glinz</b>	<b>Buhr</b>	<b>Grundy <i>et al.</i></b>	<b>González <i>et al.</i></b>
<b>Utiliza Procesamiento de Lenguaje Natural?</b>	SI	SI	NO	NO	NO
<b>Se realiza de manera formal?</b>	NO	NO	NO	NO	SI
<b>Se define un conjunto de reglas de consistencia intermodelos?</b>	NO	NO	NO	NO	SI
<b>Chequea consistencia intermodelos?</b>	SI	SI	SI	SI	SI
<b>Chequea consistencia entre el diagrama de clases y de casos de uso de UML?</b>	SI	SI	SI	NO	SI
<b>Utiliza Representación intermedia?</b>	NO	SI	SI	SI	SI
<b>Tipo de Representación intermedia</b>	-	Especificación textual de los casos de uso	Modelo Objetual	Herramientas de entorno de desarrollo de software	XMI

**Tabla 5.2 Comparación de trabajos afines al método propuesto (parte 2 de 2)**

Como se puede observar en las tablas 5.1 y 5.2, el método propuesto complementa a los otros principalmente en aspectos de normalización, definición de un conjunto de reglas de consistencia intermodelos, independencia de la plataforma y la integración con las Interfaces gráficas de usuario, permitiendo así una mayor cobertura de manera formal y ampliando la posibilidad de extenderse a otros modelos.

## Capítulo 6

### Resultados

Todas las pruebas fueron realizadas utilizando un computador con procesador PENTIUM 4 de 3.0 Ghz., 1.0 Gb. de memoria RAM, corriendo bajo Sistema Operativo Windows XP® Service Pack 2. Las implementaciones de los modelos fueron realizadas en Java® y SaxonB.

Se utilizó como caso de estudio un Sistema de Reservaciones de Vuelos con acceso vía Internet (Weitzenfeld, 2005). El enunciado que propone Weitzenfeld es el siguiente:

Se trata de un sistema que permite al usuario hacer consultas y reservas de vuelos, además de poder comprar los boletos aéreos de forma remota, sin la necesidad de recurrir a un agente de viajes humano. Se desea que el sistema de reservaciones sea accesible a través del Internet (World Wide Web). Como base para estos sistemas, existen en la actualidad múltiples bases de datos de reservaciones de vuelos que utilizan las agencias de viajes para dar servicio a los clientes, por ejemplo, Sabre, Apollo, Worldspan, Amadeus, Sahara, Panorama, Gemini, Galileo, Axess, etc. Muchos de estas bases de datos y sistemas correspondientes son la base para los sistemas de reservaciones de vuelos de acceso por Internet, como por ejemplo, Travelocity, Expedia, Viajando, DeViaje, etc.

El sistema presenta en su pantalla principal un mensaje de bienvenida describiendo los servicios ofrecidos junto con la opción para registrarse por primera vez, o si ya se está registrado, poder utilizar el sistema de reservaciones de vuelos. Este acceso se da por medio de la inserción de un login previamente especificado y un password previamente escogido y que debe validarse.

Una vez registrado el usuario, y después de haberse validado el registro y contraseña del usuario, se pueden seleccionar las siguientes actividades: Consulta de vuelos, Reserva de vuelos, Pago de boletos.

La consulta de vuelos se puede hacer de tres maneras diferentes: Horarios de Vuelos, Tarifas de Vuelos, Estado de Vuelo.

La consulta según horario muestra los horarios de las diferentes aerolíneas dando servicio entre dos ciudades. La consulta según tarifas muestra los diferentes vuelos entre dos ciudades dando prioridad a su costo. El estado de vuelo se utiliza principalmente para consultar el estado de algún vuelo, incluyendo información de disponibilidad de asientos y, en el caso de un vuelo para el mismo día, si está en hora. Se pueden incluir preferencias en las búsquedas, como fecha y horario deseado, categoría de asiento, aerolínea deseada y si se desea sólo vuelos directos.

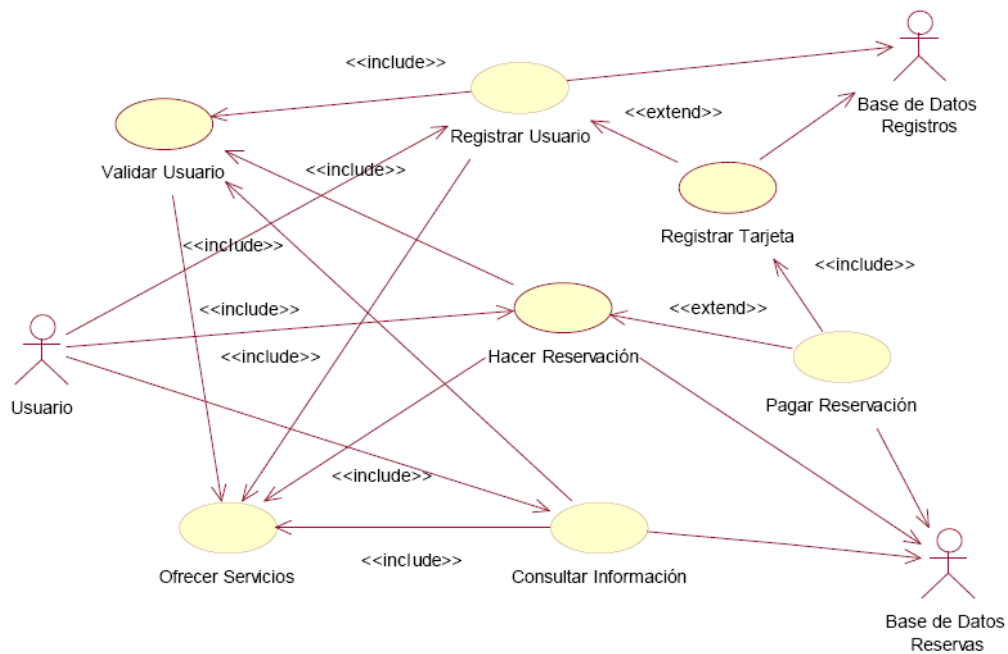
La reservación de vuelo permite al cliente hacer una reserva para un vuelo particular, especificando la fecha y horario, bajo una tarifa establecida. Es posible reservar un itinerario compuesto de múltiples vuelos, para uno o más pasajeros, además de poder reservar asientos. El pago permite al cliente, dada una reserva de vuelo previa y una tarjeta de crédito válida, adquirir los boletos aéreos. Los boletos serán posteriormente enviados al cliente, o estarán listos para ser recogidos en el mostrador del aeropuerto previo a la salida del primer vuelo.

Es necesario estar previamente registrados con un número de tarjeta de crédito válida para poder hacer compras de boletos, o de lo contrario proveerla en el momento de la compra. Además de los servicios de vuelo, el usuario podrá en cualquier momento acceder, modificar o cancelar su propio registro, todo esto después de haber sido el usuario validado en el sistema.

En las secciones 6.1 y 6.2 se presenta para cada diagrama su representación UML así como una fracción de su representación en formato XMI, el cual es la base para aplicar las reglas de

consistencia definidas en *Xquery* por medio de *Xpath*. En la sección de anexos se presenta el código completo.

## 6.1 Casos de uso para el Sistema de Reservas de Vuelo



**Figura 6.1. Casos de uso para el sistema de reservas de vuelo** (Weitzenfeld, pág 208)

Al exportar la sección de casos de uso del proyecto de ArgoUML en formato XMI, se obtuvo lo siguiente (sólo se expone una parte significativa del código, la versión completa se puede obtener en los anexos):

```

<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Thu Feb 08 02:00:29 GMT 2007'>
  <XMI.header> <XMI.documentation>
    <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
    <XMI.exporterVersion>0.20.x</XMI.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
  <XMI.content>

```

```

<UML:Model xmi.id = '-84-21-8--50--56153916:110e9c919d3:-8000:00000000000077B'
  name = 'untitledModel' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
  isAbstract = 'false'>
<UML:Namespace.ownedElement>
  <UML:UseCase xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000086B'
    name = 'Validar Usuario' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
    isAbstract = 'false'/>
  <UML:UseCase xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000086C'
    name = 'Registrar Usuario' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
    isAbstract = 'false'>
    <UML:UseCase.include>
      <UML:Include xmi.idref = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000089F'/>
    </UML:UseCase.include>
    <UML:UseCase.extensionPoint>
      <UML:ExtensionPoint xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000089D'
        name = 'newEP' isSpecification = 'false' location = 'loc'/>
    </UML:UseCase.extensionPoint>
  </UML:UseCase>
  <UML:UseCase xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000086D'
    name = 'Registrar Tarjeta' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
    isAbstract = 'false'>
    <UML:UseCase.extend>
      <UML:Extend xmi.idref = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000089E'/>
    </UML:UseCase.extend>
  </UML:UseCase>
  <UML:UseCase xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000086E'
    name = 'Consultar Información' isSpecification = 'false' isRoot = 'false'
    isLeaf = 'false' isAbstract = 'false'>
    <UML:UseCase.include>
      <UML:Include xmi.idref = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:000000000000898'/>
    </UML:UseCase.include>
  </UML:UseCase>
  <UML:UseCase xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000086F'
    name = 'Pagar Reservación' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
    isAbstract = 'false'>
    <UML:UseCase.extend>
      <UML:Extend xmi.idref = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000089C'/>
    </UML:UseCase.extend>
    <UML:UseCase.include>
      <UML:Include xmi.idref = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000089A'/>
    </UML:UseCase.include>
  </UML:UseCase>
  <UML:UseCase xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:000000000000870'

```

```

name = 'Ofrecer Servicios' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'/>
<UML:UseCase xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:0000000000000871'
name = 'Hacer Reservación' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'>
<UML:UseCase.include>
  <UML:Include xmi.idref = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:0000000000000899'/>
  <UML:Include xmi.idref = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:00000000000008A0'/>
</UML:UseCase.include>
<UML:UseCase.extensionPoint>
  <UML:ExtensionPoint xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:000000000000089B'
name = 'newEP' isSpecification = 'false' location = 'loc'/>
</UML:UseCase.extensionPoint>
</UML:UseCase>
<UML:Actor xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:0000000000000872'
name = 'Usuario' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'/>
<UML:Actor xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:0000000000000873'
name = 'Base de Datos Reservas' isSpecification = 'false' isRoot = 'false'
isLeaf = 'false' isAbstract = 'false'/>
<UML:Actor xmi.id = '-64--88-105--122--66a1d0b2:1112f3ac0dd:-8000:0000000000000874'
name = 'Base de Datos Registros' isSpecification = 'false' isRoot = 'false'
isLeaf = 'false' isAbstract = 'false'/>

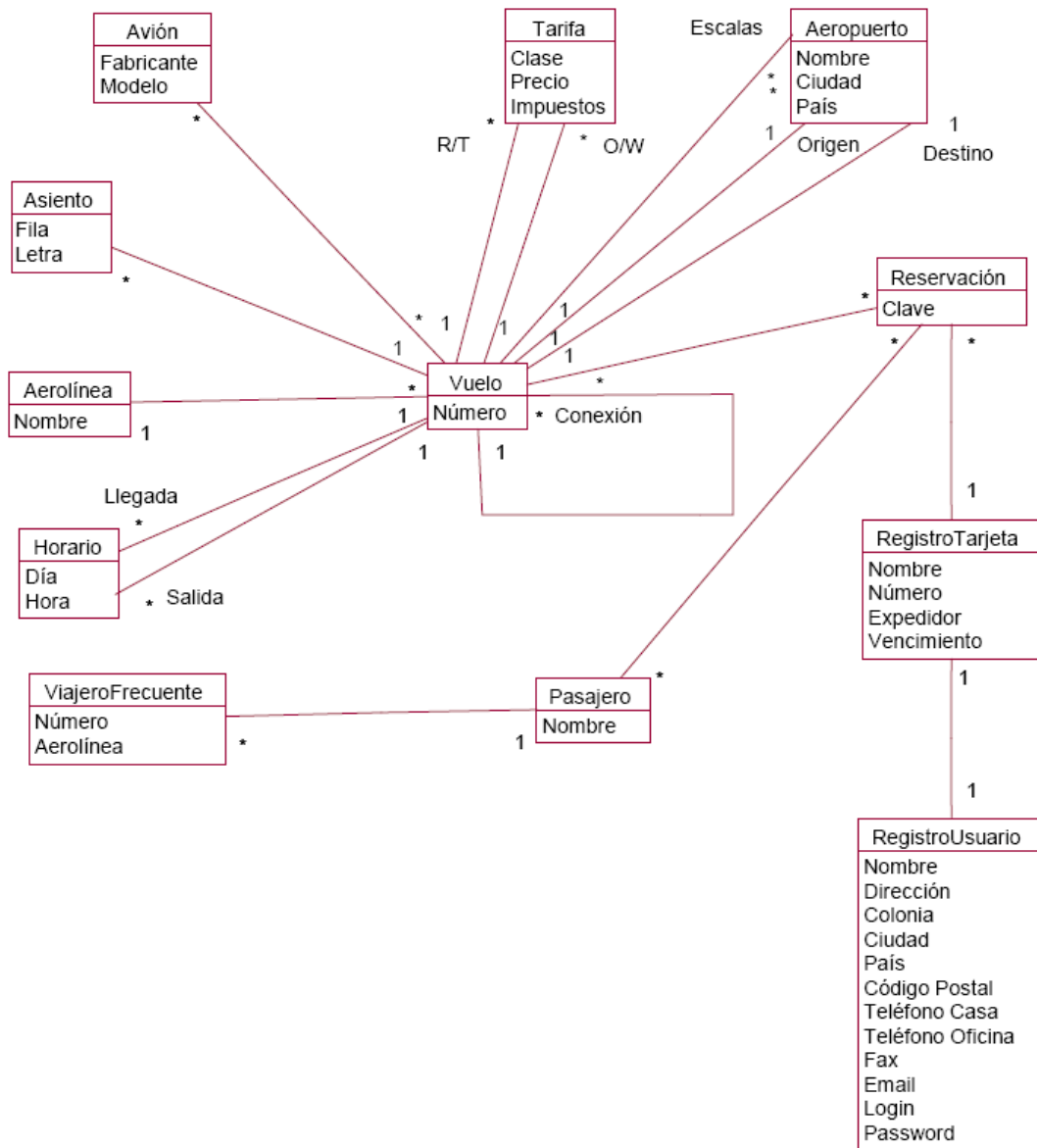
```

## 6.2 Diagrama de Clases para el Sistema de Reservas de Vuelo

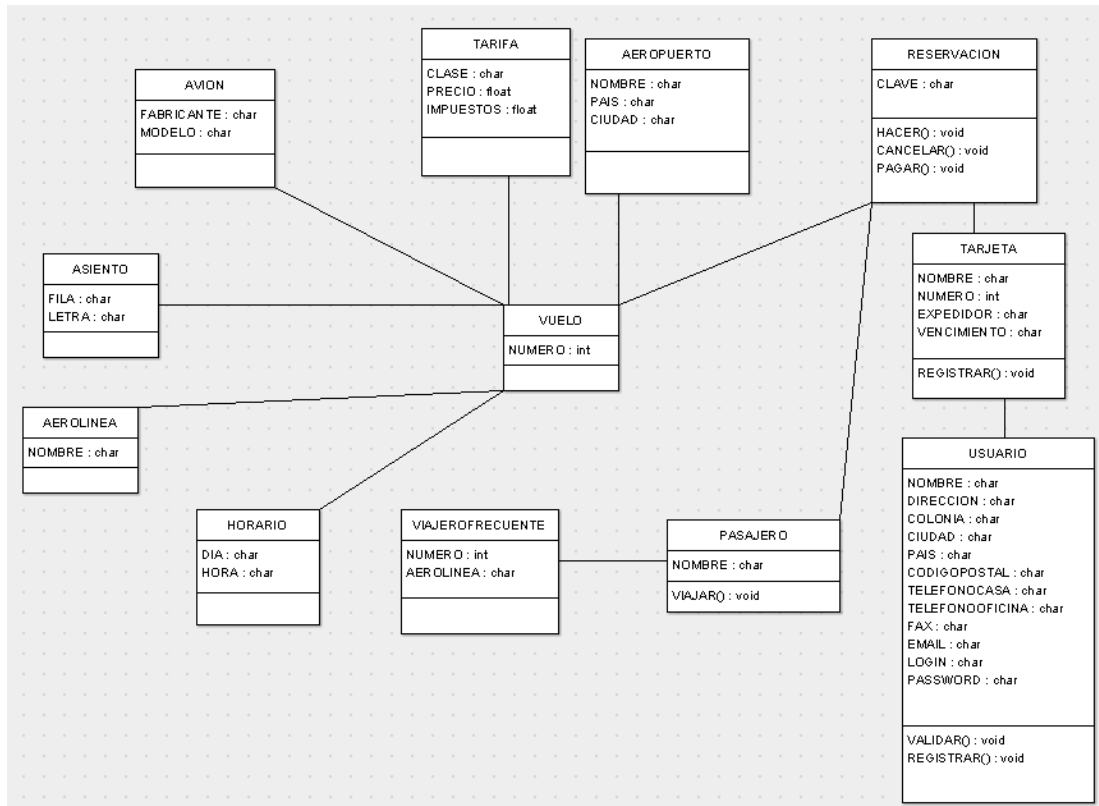
Debido a que Weitzenfeld (2005) propone un diagrama de clases sin operaciones (parecido más bien a un modelo del dominio), se propuso unos cambios en los nombres de las clases así como cambios en los nombres de las operaciones en algunas clases para validar las reglas de consistencia:

En la clase Reservación se agregaron las operaciones Hacer(), Pagar() y Cancelar(). La clase RegistroTarjeta se renombró a Tarjeta y se le agregó la operación Registrar(). La clase RegistroUsuario se renombró a Usuario y se le adicionaron las operaciones Validar() y

Registrar(). En la Figura 6.2 Se muestra el modelo concebido por Weitzenfeld y en la Figura 6.3 se muestra el modelo de Clases propuesto, con los cambios respectivos.



**Figura 6.2 Diagrama de Clases para el sistema de reservaciones de vuelo propuesto por Weitzenfeld**



**Figura 6.3 Diagrama de Clases para el sistema de reservas de vuelo (modificado)**

Al exportar la sección del diagrama de clases del proyecto de ArgoUML en formato XMI, se obtuvo lo siguiente (sólo se expone la primera parte del código, la versión completa se puede obtener en los anexos):

```

<?xml version = '1.0' encoding = 'UTF-8' ¿>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Thu Feb 08 02:20:42 GMT 2007'>
<XMI.header> <XMI.documentation>
  <XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
  <XMI.exporterVersion>0.20.x</XMI.exporterVersion>
</XMI.documentation>
<XMI.metamodel xmi.name="UML" xmi.version="1.4"/></XMI.header>
<XMI.content>
<UML:Model xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:000000000000077B'
  name = 'untitledModel' isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
  isAbstract = 'false'>

```

```

<UML:Namespace.ownedElement>
<UML:Class xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000077E'
  name = 'AVION' visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
<UML:Classifier.feature>
<UML:Attribute xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:000000000000797'
  name = 'FABRICANTE' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
  changeability = 'changeable' targetScope = 'instance'>
<UML:StructuralFeature.multiplicity>
<UML:Multiplicity xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:000000000000798'>
  <UML:Multiplicity.range>
    <UML:MultiplicityRange xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:000000000000799'
      lower = '1' upper = '1'>
    </UML:Multiplicity.range>
  </UML:Multiplicity>
</UML:StructuralFeature.multiplicity>
<UML:StructuralFeature.type>
  <UML:DataType xmi.idref = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000079A'>
</UML:StructuralFeature.type>
</UML:Attribute>
<UML:Attribute xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000079B'
  name = 'MODELO' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
  changeability = 'changeable' targetScope = 'instance'>
<UML:StructuralFeature.multiplicity>
<UML:Multiplicity xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000079C'>
  <UML:Multiplicity.range>
    <UML:MultiplicityRange xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000079D'
      lower = '1' upper = '1'>
    </UML:Multiplicity.range>
  </UML:Multiplicity>
</UML:StructuralFeature.multiplicity>
<UML:StructuralFeature.type>
  <UML:DataType xmi.idref = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000079A'>
</UML:StructuralFeature.type>
</UML:Attribute>
</UML:Classifier.feature>
</UML:Class>
<UML:Class xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:000000000000780'
  name = 'TARIFA' visibility = 'public' isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
<UML:Classifier.feature>
<UML:Attribute xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000079E'
  name = 'CLASE' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'

```

```

changeability = 'changeable' targetScope = 'instance'>
<UML:StructuralFeature.multiplicity>
  <UML:Multiplicity xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000079F'>
    <UML:Multiplicity.range>
      <UML:MultiplicityRange xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000070'
        lower = '1' upper = '1' />
      </UML:Multiplicity.range>
    </UML:Multiplicity>
  </UML:StructuralFeature.multiplicity>
<UML:StructuralFeature.type>
  <UML:DataType xmi.idref = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000079F' />
</UML:StructuralFeature.type>
</UML:Attribute>
<UML:Attribute xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000071'
  name = 'PRECIO' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
  changeability = 'changeable' targetScope = 'instance'>
<UML:StructuralFeature.multiplicity>
  <UML:Multiplicity xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000072'>
    <UML:Multiplicity.range>
      <UML:MultiplicityRange xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000073'
        lower = '1' upper = '1' />
      </UML:Multiplicity.range>
    </UML:Multiplicity>
  </UML:StructuralFeature.multiplicity>
<UML:StructuralFeature.type>
  <UML:DataType xmi.idref = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000077' />
</UML:StructuralFeature.type>
</UML:Attribute>
<UML:Attribute xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000074'
  name = 'IMPUESTOS' visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
  changeability = 'changeable' targetScope = 'instance'>
<UML:StructuralFeature.multiplicity>
  <UML:Multiplicity xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000075'>
    <UML:Multiplicity.range>
      <UML:MultiplicityRange xmi.id = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000076'
        lower = '1' upper = '1' />
      </UML:Multiplicity.range>
    </UML:Multiplicity>
  </UML:StructuralFeature.multiplicity>
<UML:StructuralFeature.type>
  <UML:DataType xmi.idref = '-84-21-8—50—56153916:110e9c919d3:-8000:00000000000077' />
</UML:StructuralFeature.type>
</UML:Attribute>

```

</UML:Classifier.feature>  
</UML:Class>

### 6.3 Interfaces para el Sistema de Reservaciones de Vuelo

De las 18 interfaces gráficas de usuario propuestas por Weitzenfeld sólo se tomaron en cuenta 3 de las más importantes para mostrar el proceso. Dichas interfaces fueron implementadas en Java por Weitzenfeld, por lo que se replantearon en Visio para tener el soporte de XMI, y así poder integrarlas en el análisis de las reglas de consistencia por medio de Xquery. A continuación se presentan las interfaces con sus respectivos fragmentos de código en formato XMI.

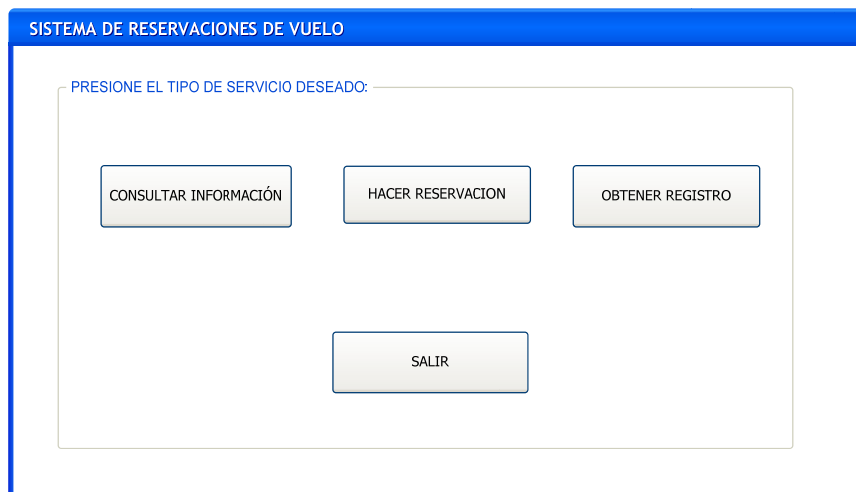


Figura 6.4. Pantalla de Menú de Servicios

SISTEMA DE RESERVACIONES DE VUELO

PANTALLA CREAR REGISTRO USUARIO

NOMBRE:  APELLIDO:

CALLE:  COLONIA:

CIUDAD:  PAIS:

CODIGO POSTAL:  TEL CASA:

TEL OFICINA:  FAX:

LOGIN:  E-MAIL:

PASSWORD:  REPETIR  
PASSWORD:

**Figura 6.5. Pantalla de Registro de Usuario por Primera Vez**

SISTEMA DE RESERVACIONES DE VUELO

PANTALLA CREAR REGISTRO TARJETA

NOMBRE:

NÚMERO DE TARJETA  
(REQUERIDO PARA PAGOS)

TIPO:

FECHA VENCIMIENTO:

**Figura 6.6. Pantalla de Registro de Tarjeta por Primera Vez**

A continuación se muestra una fracción del código XML que se obtuvo al exportar el proyecto de Visio en formato XMI.

```

<Pages><Page ID='0' NameU='Page-1' ViewScale='0.5' ViewCenterX='3.9370078740157'
ViewCenterY='5.1968503937008'><PageSheet LineStyle='0' FillStyle='0' TextStyle='0'><PageProps><PageWidth
Unit='MM'>8.26771653543307</PageWidth><PageHeight Unit='MM'>11.69291338582677</PageHeight><ShdwOffsetX
Unit='MM'>0.125</ShdwOffsetX><ShdwOffsetY Unit='MM'>0.125</ShdwOffsetY><PageScale
Unit='MM'>0.03937007874015748</PageScale><DrawingScale
Unit='MM'>0.03937007874015748</DrawingScale><DrawingSizeType>0</DrawingSizeType><DrawingScaleType>0</Drawi
ngScaleType><InhibitSnap>0</InhibitSnap><UIVisibility>0</UIVisibility><ShdwType>0</ShdwType><ShdwObliqueAngle>
0</ShdwObliqueAngle><ShdwScaleFactor>1</ShdwScaleFactor></PageProps><XForm><PinX>0</PinX><PinY>0</PinY><
Width>1</Width><Height>1</Height><LocPinX F='Width*0'>0</LocPinX><LocPinY
F='Height*0'>0</LocPinY><Angle>0</Angle><FlipX>0</FlipX><FlipY>0</FlipY><ResizeMode>0</ResizeMode></XForm
><PageLayout><ResizePage F='Inh'>0</ResizePage><EnableGrid F='Inh'>0</EnableGrid><DynamicsOff
F='Inh'>0</DynamicsOff><CtrlAsInput F='Inh'>0</CtrlAsInput><PlaceStyle F='Inh'>0</PlaceStyle><RouteStyle
F='Inh'>0</RouteStyle><PlaceDepth F='Inh'>0</PlaceDepth><PlowCode F='Inh'>0</PlowCode><LineJumpCode
F='Inh'>1</LineJumpCode><LineJumpStyle F='Inh'>0</LineJumpStyle><PageLineJumpDirX
F='Inh'>0</PageLineJumpDirX><PageLineJumpDirY F='Inh'>0</PageLineJumpDirY><LineToNodeX
Unit='MM'>0.125</LineToNodeX><LineToNodeY Unit='MM'>0.125</LineToNodeY><BlockSizeX
Unit='MM'>0.25</BlockSizeX><BlockSizeY Unit='MM'>0.25</BlockSizeY><AvenueSizeX
Unit='MM'>0.375</AvenueSizeX><AvenueSizeY Unit='MM'>0.375</AvenueSizeY><LineToLineX
Unit='MM'>0.125</LineToLineX><LineToLineY Unit='MM'>0.125</LineToLineY><LineJumpFactorX
F='Inh'>0.666666666666667</LineJumpFactorX><LineJumpFactorY
F='Inh'>0.666666666666667</LineJumpFactorY><LineAdjustFrom F='Inh'>0</LineAdjustFrom><LineAdjustTo
F='Inh'>0</LineAdjustTo><PlaceFlip F='Inh'>0</PlaceFlip><LineRouteExt
F='Inh'>0</LineRouteExt><PageShapeSplit>1</PageShapeSplit></PageLayout></PageSheet><Shapes><Shape ID='1'
NameU='Blank form' Type='Group'
Master='0'><XForm><PinX>4.232283464566929</PinX><PinY>5.610236220472441</PinY><Width>7.677165354330708</
Width><Height>7.677165354330708</Height><LocPinX F='Inh'>3.838582677165354</LocPinX><LocPinY
F='Inh'>3.838582677165354</LocPinY><Angle F='Inh'>0</Angle><FlipX F='Inh'>0</FlipX><FlipY
F='Inh'>0</FlipY><ResizeMode F='Inh'>0</ResizeMode></XForm><Event><TheData F='No
Formula'>0</TheData><TheText F='No Formula'>0</TheText><EventDbfClick F='Inh'>0</EventDbfClick><EventXFMod
F='No Formula'>0</EventXFMod><EventDrop F='No Formula'>0</EventDrop></Event><TextBlock><LeftMargin Unit='PT'
F='Inh'>0</LeftMargin><RightMargin Unit='PT' F='Inh'>0</RightMargin><TopMargin Unit='MM'
F='Inh'>0.08858267716535433</TopMargin><BottomMargin Unit='PT' F='Inh'>0</BottomMargin><VerticalAlign
F='Inh'>0</VerticalAlign><TextBkgnd F='Inh'>0</TextBkgnd><DefaultTabStop
F='Inh'>0.5905511811023622</DefaultTabStop><TextDirection F='Inh'>0</TextDirection><TextBkgndTrans
F='Inh'>0</TextBkgndTrans></TextBlock><TextXForm><TxtPinX Unit='MM'
F='Inh'>0.1476377952755905</TxtPinX><TxtPinY F='Inh'>7.677165354330708</TxtPinY><TxtWidth
F='Inh'>2.5631603525625</TxtWidth><TxtHeight Unit='MM' F='Inh'>0.2460629921259843</TxtHeight><TxtLocPinX
F='Inh'>0</TxtLocPinX><TxtLocPinY Unit='MM' F='Inh'>0.2460629921259843</TxtLocPinY><TxtAngle
F='Inh'>0</TxtAngle></TextXForm></User NameU='Pixel' ID='1'><Value Unit='MM'

```

F='Inh'>0.00984251968503937</Value><Prompt F='No Formula'/></User><User NameU='MinWidth' ID='4'><Value F='Inh'>1</Value><Prompt F='No Formula'/></User><User NameU='MinHeight' ID='5'><Value F='Inh'>1</Value><Prompt F='No Formula'/></User><Char IX='0'><Font F='Inh'>147</Font><Color F='Inh'>1</Color><Style F='Inh'>1</Style><Case F='Inh'>0</Case><Pos F='Inh'>0</Pos><FontScale F='Inh'>1</FontScale><Size Unit='PT' F='Inh'>0.138888888888889</Size><DbUnderline F='Inh'>0</DbUnderline><Overline F='Inh'>0</Overline><Strikethru F='Inh'>0</Strikethru><Highlight F='Inh'>0</Highlight><DoubleStrikethrough F='Inh'>0</DoubleStrikethrough><RTLText F='Inh'>0</RTLText><UseVertical F='Inh'>0</UseVertical><Letterspace F='Inh'>0</Letterspace><ColorTrans F='Inh'>0</ColorTrans><AsianFont F='Inh'>0</AsianFont><ComplexScriptFont F='Inh'>0</ComplexScriptFont><LocalizeFont F='Inh'>0</LocalizeFont><ComplexScriptSize F='Inh'>1</ComplexScriptSize><LangID>1034</LangID></Char><Act ID='1'><Menu F='Inh'>Fondo & blanco</Menu><Action F='Inh'>0</Action><Checked F='Inh'>1</Checked><Disabled F='Inh'>0</Disabled><ReadOnly F='No Formula'>0</ReadOnly><Invisible F='No Formula'>0</Invisible><BeginGroup F='No Formula'>0</BeginGroup><TagName F='No Formula'><ButtonFace F='No Formula'><SortKey F='No Formula'></Act><Act ID='2'><Menu F='Inh'>Fo&ndo gris</Menu><Action F='Inh'>0</Action><Checked F='Inh'>0</Checked><Disabled F='Inh'>0</Disabled><ReadOnly F='No Formula'>0</ReadOnly><Invisible F='No Formula'>0</Invisible><BeginGroup F='No Formula'>0</BeginGroup><TagName F='No Formula'><ButtonFace F='No Formula'><SortKey F='No Formula'></Act><Act ID='3'><Menu F='Inh'>Fon&do personalizado</Menu><Action F='Inh'>0</Action><Checked F='Inh'>0</Checked><Disabled F='Inh'>0</Disabled><ReadOnly F='No Formula'>0</ReadOnly><Invisible F='No Formula'>0</Invisible><BeginGroup F='No Formula'>0</BeginGroup><TagName F='No Formula'><ButtonFace F='No Formula'><SortKey F='No Formula'></Act><Connection IX='0'><X Unit='MM' F='Inh'>0.03937007874015748</X><Y Unit='MM' F='Inh'>7.381889763779526</Y><DirX F='Inh'>0</DirX><DirY F='Inh'>0</DirY><Type F='Inh'>0</Type><AutoGen F='Inh'>0</AutoGen><Prompt F='No Formula'></Connection><Connection IX='1'><X Unit='MM' F='Inh'>7.63779527559055</X><Y Unit='MM' F='Inh'>7.381889763779526</Y><DirX F='Inh'>0</DirX><DirY F='Inh'>0</DirY><Type F='Inh'>0</Type><AutoGen F='Inh'>0</AutoGen><Prompt F='No Formula'></Connection><Connection IX='2'><X Unit='MM' F='Inh'>0.03937007874015748</X><Y Unit='MM' F='Inh'>0.03937007874015748</Y><DirX F='Inh'>0</DirX><DirY F='Inh'>0</DirY><Type F='Inh'>0</Type><AutoGen F='Inh'>0</AutoGen><Prompt F='No Formula'></Connection><Connection IX='3'><X Unit='MM' F='Inh'>7.63779527559055</X><Y Unit='MM' F='Inh'>0.03937007874015748</Y><DirX F='Inh'>0</DirX><DirY F='Inh'>0</DirY><Type F='Inh'>0</Type><AutoGen F='Inh'>0</AutoGen><Prompt F='No Formula'></Connection><Connection IX='4'><X Unit='MM' F='Inh'>0.05905511811023622</X><Y Unit='MM' F='Inh'>7.519685039370078</Y><DirX F='Inh'>0</DirX><DirY F='Inh'>0</DirY><Type F='Inh'>0</Type><AutoGen F='Inh'>0</AutoGen><Prompt F='No Formula'></Connection><Connection IX='5'><X Unit='MM' F='Inh'>7.62795275590551</X><Y Unit='MM' F='Inh'>7.514763779527558</Y><DirX F='Inh'>0</DirX><DirY F='Inh'>0</DirY><Type F='Inh'>0</Type><AutoGen F='Inh'>0</AutoGen><Prompt F='No Formula'></Connection><Text><cp IX='0'>SISTEMA DE RESERVACIONES DE VUELO</Text><Shapes><Shape ID='2' Type='Shape' MasterShape='6'><XForm><PinX F='Inh'>3.838582677165354</PinX><PinY F='Inh'>0</PinY><Width F='Inh'>7.677165354330708</Width>

## 6.4 Validación de Reglas de Consistencia

Para la validación de las reglas de consistencia entre los diagramas expuestos en las secciones 6.1 y 6.2 primero se exportaron los diagramas a formato XMI, luego se ejecutó la aplicación realizada, en la cual se aplicaron las diferentes reglas propuestas en la sección 5.1, dicha aplicación generó como resultado un archivo en formato XML (resultado.xml), el cual fue luego procesado por la aplicación realizada en java, la cual eliminó información no relevante y le dio formato para poder ser mostrado vía web, creando un archivo (resultado.html) para ser interpretado por un browser (explorador) y así poder ser mostrado cómo página web.

A continuación se muestra un fragmento del resultado obtenido, donde se puede observar la información del análisis realizado. En los anexos se presenta todo el resultado obtenido.

```
<resultado >
<!-- se presentan algunas clases con sus atributos y operaciones -->
<clase>
<nombre>TARJETA</nombre>
<atributo>NOMBRE</atributo>
<atributo>NUMERO</atributo>
<atributo>EXPEDIDOR</atributo>
<atributo>VENCIMIENTO</atributo>
<operacion>REGISTRAR</operacion>
</clase>
<clase>
<nombre>RESERVACION</nombre>
<atributo>CLAVE</atributo>
<operacion>HACER</operacion>
<operacion>CANCELAR</operacion>
<operacion>PAGAR</operacion>
</clase>
<clase>
<nombre>USUARIO</nombre>
<atributo>NOMBRE</atributo>
<atributo>DIRECCION</atributo>
<atributo>COLONIA</atributo>
<atributo>CIUDAD</atributo>
<atributo>PAIS</atributo>
```

```

<atributo>CODIGOPOSTAL</atributo>
<atributo>TELEFONOCASA</atributo>
<atributo>TELEFONOOFICINA</atributo>
<atributo>FAX</atributo>
<atributo>EMAIL</atributo>
<atributo>LOGIN</atributo>
<atributo>PASSWORD</atributo>
<operacion>VALIDAR</operacion>
<operacion>REGISTRAR</operacion>
</clase>
<!-- se presentan algunos casos de uso -->
<casouso>
<nombre>HACER RESERVACION</nombre>
</casouso>
<casouso>
<nombre>REGISTRAR USUARIO</nombre>
</casouso>
<casouso>
<nombre>VALIDAR USUARIO</nombre>
</casouso>
<casouso>
<nombre>CONSULTAR INFORMACION</nombre>
</casouso>
<casouso>
<nombre>REGISTRAR TARJETA</nombre>
</casouso>

<!-- se presentan algunas interfaces con sus etiquetas y botones -->

<interfaz>
<titulo>PRESIONE EL TIPO DE SERVICIO DESEADO</titulo>
<boton>CONSULTAR INFORMACION</boton>
<boton>HACER RESERVACION</boton>
<boton>OBTENER REGISTRO</boton>
<boton>SALIR</boton>
</interfaz>
<interfaz>
<titulo>PANTALLA CREAR REGISTRO USUARIO</titulo>
<etiqueta>NOMBRE</etiqueta>
<etiqueta>APELLIDO</etiqueta>
<etiqueta>CALLE</etiqueta>
<etiqueta>COLONIA</etiqueta>
<etiqueta>CIUDAD</etiqueta>

```

<etiqueta>PAIS</etiqueta>  
<etiqueta>CODIGO POSTAL</etiqueta>  
<etiqueta>TEL CASA</etiqueta>  
<etiqueta>TEL OFICINA</etiqueta>  
<etiqueta>FAX</etiqueta>  
<etiqueta>LOGIN</etiqueta>  
<etiqueta>E-MAIL</etiqueta>  
<etiqueta>PASSWORD</etiqueta>  
<etiqueta>REPETIR PASSWORD</etiqueta>  
<boton>REGISTRAR</boton>  
<boton>SALIR</boton>  
</interfaz>

<!-- se presentan algunas reglas de consistencia -->

<consistencia>

<existecasodeusoenclase>

El caso de uso CONSULTAR INFORMACION no tiene relación con ninguna clase

</existecasodeusoenclase>

<existeoperacionencasodeuso>

La operacion PAGAR de la clase RESERVACION no existe en ningun caso de uso

</existeoperacionencasodeuso>

<existebotonecasodeuso>

El botón OBTENER REGISTRO de la interfaz PRESIONE EL TIPO DE SERVICIO DESEADO no existe en ningun caso de uso

</existebotonecasodeuso>

</consistencia>

</resultado>

## **6.4.1 Análisis de Resultados**

### **6.4.1.1 Regla 1**

En el contexto propuesto por Weitzenfeld sólo se cumple para los casos de uso Hacer Reservación y Pagar Reservación. Los casos de uso Registrar Tarjeta y Registrar Usuario salen con advertencias ya que existen clases cuyos nombres son sinónimos de Tarjeta y usuario (RegistroTarjeta y RegistroUsuario respectivamente)

### **6.4.1.2 Regla 2**

Sólo se cumple para los casos de uso Hacer Reservación y Pagar Reservación. Los casos de uso Registrar Tarjeta y Registrar Usuario también salen con advertencias ya que existen clases cuyos nombres son sinónimos de Tarjeta y usuario (RegistroTarjeta y RegistroUsuario respectivamente) y ahí las operaciones corresponden (Registrar)

### **6.4.1.3 Regla 3**

Las interfaces de las figuras 6.5 y 6.6 (crear registro usuario y crear registro tarjeta) salieron con advertencias ya que son sinónimos de los casos de uso registrar usuario y registrar tarjeta respectivamente.

### **6.4.1.4 Regla 4**

Las interfaces de las figuras 6.5 y 6.6 (crear registro usuario y crear registro tarjeta) salieron con advertencias ya que contienen en sus títulos sinónimos de las operaciones que se encuentran en las clases (registrar).

### **6.4.1.5 Regla 5**

La interfaz de la Figura 6.4 contiene tres botones con los textos Consultar Información, Hacer reservación y Obtener Registro, mostrando cierta correspondencia con dos de las operaciones de las clases correspondientes.

#### **6.4.1.6 Regla 6**

En la interfaz de usuario de la Figura 6.5 se muestra los campos requeridos para registrar un usuario. Nótese que el campo apellido no tiene correspondencia con ningún atributo de la clase correspondiente por lo que sale un error de consistencia.

En la Figura 6.6 se muestra el Registro de Tarjeta y se observa que hay un campo llamado tipo que no tiene correspondencia con ningún atributo de su clase correspondiente por lo que también generará un error de consistencia.

#### **6.4.1.7 Regla 7**

Se puede observar en las Figuras 6.5 y 6.6 que los títulos de las interfaces no corresponden exactamente a los casos de uso, por lo que hubo una serie de advertencias debido a que corresponden a sinónimos de los casos de uso.

#### **6.4.1.8 Regla 8**

La interfaz de la Figura 6.4 contiene tres botones con los textos Consultar Información, Hacer reservación y Obtener Registro, mostrando correspondencia exacta con dos casos de uso.

A continuación se presenta un cuadro de resumen donde se muestran algunos aspectos de los resultados obtenidos por medio de la aplicación, tras aplicar la validación de las reglas de consistencia propuestas.

REGLA	CUMPLE	NO CUMPLE	ADVERTENCIA
1.	Hacer Reservación (Caso de Uso) Pagar Reservación (Caso de Uso)	Validar Usuario (Caso de Uso) Ofrecer Servicios (Caso de Uso) Consultar Información (Caso de Uso)	Registrar Tarjeta (Caso de Uso) Registrar Usuario (Caso de Uso)
2.	Hacer Reservación (Caso de Uso) Pagar Reservación (Caso de Uso)	Validar Usuario (Caso de Uso) Ofrecer Servicios (Caso de Uso) Consultar Información (Caso de Uso)	Registrar Tarjeta (Caso de Uso) Registrar Usuario (Caso de Uso)
3.			crear registro usuario (Interfaz) crear registro tarjeta (Interfaz)
4.			crear registro usuario (Interfaz) crear registro tarjeta (Interfaz)
5.	Consultar Información Hacer reservación Obtener Registro		
6.		Registro de Tarjeta Registro de Usuario	
7.		Crear registro tarjeta Crear registro usuario	
8.	Consultar Información Hacer reservación Obtener Registro		

**Figura 6.7 Reglas de Consistencia entre el diagrama de clases y el diagrama de casos de uso de UML**

# Capítulo 7

## Conclusiones

### 7.1 Contribuciones Principales

Los resultados obtenidos en las diferentes áreas de trabajo de la investigación muestran un número considerable de inconsistencias que pueden ser encontradas con el método propuesto, debido que al incluir las interfaces gráficas de usuario se tiene un apoyo extra en la verificación de correspondencia de sus elementos con los atributos de las clases, garantizando así una mayor consistencia entre los modelos de casos de uso y de diagramas de clases de UML.

Por otro lado, al presentar una especificación formal de reglas de consistencia entre el diagrama de casos de uso y diagrama de clases en OCL, se formaliza la validación de los aspectos necesarios para garantizar que no exista ambigüedad entre estos modelos y que no se tenga diagramas con objetos aislados. Al integrar las interfaces de usuario con estos dos modelos utilizando archivos en formato XMI, se hace al método independiente de la plataforma y que sea regido por el estándar, asegurando así interoperabilidad y modularización para facilitar el intercambio de información.

#### 7.1.1 Definición de reglas Intermodelos

A diferencia de otros trabajos, en esta Tesis se planteó un conjunto de reglas de consistencia intermodelos, específicamente entre el diagrama de casos de uso y el diagrama de clases. Además de definir dicho conjunto de reglas, se implementó un método que permite validar

dichas reglas y que puede ser utilizado para revisar otras reglas, brindando la posibilidad de extender el método a otros modelos.

### **7.1.2 Especificación Formal**

La integración del Lenguaje de Especificación de objetos, OCL, en el método propuesto es un aspecto a tener en cuenta debido a que no se encontraron evidencias de que algún autor haya presentado reglas de consistencia entre el modelo de clases y el modelo de casos de uso de UML de manera formal, aportando así un método de validación de reglas de consistencia sin ambigüedades y bien formulado. Esto implica una posible integración con las reglas de buena formación en la especificación de UML, definiendo así una posibilidad de integrarse en el estándar.

Al formalizar las reglas se elimina la posibilidad de darle varias interpretaciones y se descarta el manejo de lenguaje natural, estableciendo así un mecanismo consistente y concreto para la verificación de consistencia entre dichos modelos.

### **7.1.3 Integración con Interfaces Gráficas de Usuario**

Ninguna Investigación que trabaja la consistencia entre el diagrama de clases y el modelo de casos de uso de UML ha integrado las interfaces de usuario en sus reglas de consistencia, dejando así un gran vacío en los prototipos de usuario para los casos de uso. Al integrar las interfaces de usuario en el método propuesto, se logró validar la correspondencia de los atributos de las clases con los casos de uso, haciéndolo formalmente y pudiendo así integrar de una manera completa las reglas para la validación de la consistencia entre los modelos mencionados previamente. También se compararon los elementos de las interfaces gráficas de usuario con los casos de uso para garantizar una correspondencia en el funcionamiento del sistema tanto en los prototipos como en los casos de uso iniciales, dejando la posibilidad de encontrar situaciones específicas sin modelar o sin su correspondiente prototipo.

#### **7.1.4 Integración con Xquery**

Al implementar el método con búsquedas en documentos XMI cuya estructura está basada en XML, se garantiza la interoperabilidad y la independencia de plataforma, ya que al usar expresiones relativas de XPath y Xquery se garantiza que se encontrará la información deseada sin importar la profundidad del árbol en la que se está buscando, que se puedan utilizar varios documentos para integrarlos en la búsqueda y que se pueda ejecutar las consultas desde cualquier API que soporte el estándar de Xquery.

Al usar estos tipos de archivos, se puede extender el método para otras plataformas que soporten el formato XMI, sin obligar al usuario a trabajar en una plataforma específica, dándole la posibilidad incluso de trabajar en otro sistema operativo o con software libre.

#### **7.1.4 Transformación de OCL a XQuery.**

Se definió la forma de transformar reglas definidas en OCL a sentencias XQuery, en las que se hizo uso de los operadores FLOWR y las sentencias XPath, permitiendo así una forma estándar de implementar dichas reglas en un entorno estándar con formato XML para luego ser llevadas a la WEB.

#### **7.1.4 Utilización de diccionario de sinónimos**

Al utilizar un diccionario de sinónimos (en formato XML) se brindó la posibilidad de generar advertencias a la hora de validar las reglas, ya que el usuario pudo haber utilizado palabras similares en los modelos o interfaces. Con esto se busca que no se genere un error sino que se presente la información al usuario como una alternativa para los sustantivos o verbos que utilizó, para que realice los cambios respectivos y pueda obtener modelos consistentes.

### **7.1.5 Resultados Vía Web**

Otro aspecto positivo es la forma como se muestran los resultados, ya que se obtiene un archivo de formato HTML con una estructura tipo XML donde se define para cada regla si se cumplió o no, o se presenta la advertencia correspondiente. Al presentar los resultados vía Web, se garantiza accesibilidad y disponibilidad de la información, ya que por ejemplo la aplicación se puede montar en un servidor Apache Tomcat o en cualquier servidor que soporte Java, específicamente JSP y Servlets para acceder remotamente a él, obteniendo los resultados por medio de un Browser. Esto es posible debido a que la aplicación fue desarrollada con Java, un lenguaje independiente de la plataforma que puede ser ejecutado prácticamente en cualquier equipo.

## **7.2 Limitaciones del Método Propuesto**

### **7.2.1 Include y Extend**

La parte de inclusión y extensión en los diagramas de casos de uso no fue considerada en el método actual, ya que esto requiere particularización y consideraciones especiales al relacionar los casos de uso con las clases y las interfaces.

### **7.2.2 Herencia**

No se trabajó la herencia ni en casos de uso ni en el diagrama de clases ya que no existe correspondencia directa entre las clases que heredan y los casos de uso que se especializan.

## **7.3 Trabajo Futuro**

Al observar las limitaciones del método propuesto, se busca integrar en futuros métodos la parte de inclusión y extensión de los casos de uso relacionándolos con las clases y las interfaces de usuario.

Se busca además investigar en la relación existente de la especialización de casos de uso y el manejo de herencia en el diagrama de clases, considerando también su relación con las interfaces de usuario. Por otro lado también se tiene pensado como trabajo futuro lo siguiente:

### **7.3.1 Extender el método a otros Modelos**

Se busca extender el método a otras parejas de modelos, con el fin de que exista más consistencia en los diagramas UML que genere el analista, garantizando así un buen producto de software. Se piensa inicialmente integrar el método con el diagrama de actividades y el diagrama de secuencias, debido a su gran relación y afinidad con los diagramas presentados.

### **7.3.2 Integración de OCL en UN-Método**

En la Escuela de Sistemas de la Universidad Nacional se creó el UN-Método, que incluye diagramas de UML y de otros sitios. Se busca verificar las reglas de consistencia entre modelos de requisitos de Un-Método (Zapata, *et al.*, 2006) por medio del método propuesto y además formalizarlas en OCL, para brindar a la comunidad universitaria un método con especificaciones formales.

### **7.3.3 Corpus**

Se busca enlazar el diccionario de datos con un corpus de mayor tamaño, incluso buscar la forma de utilizar un lexicón como el de la Real Academia de la Lengua (RAE).

## Bibliografía

1. Blankenhorn, Kai. A UML Profile for GUI Layout. 2004
2. Boehm, B.W. R & D Trends and defense needs. En: P. Wegner, editor, Research Directions in Software Technology, pages 44–86. MIT Press, Cambridge, MA, 1979.
3. Booch G., Jacobson I., and Rumbaugh J.,“Object Constraint Language Specification”, UML Documentation Set Version 1.1, Rational Software Cooperation, September, 1997.
4. Buhr, R.J.A. Use Case Maps as Architectural Entities for Complex Systems. IEEE Transactions on Software Engineering 24, 12 (Dec. 1998). 1131-1155.
5. Bustos, Guillermo. Integración Informal De Modelos En Uml. Publicado en la Revista Ingeniare N° 14, Facultad de Ingeniería, Pontificia Universidad Católica de Valparaíso, Valparaíso (Chile), Diciembre 2002.
6. Clark, J. and DeRose, S. XML Path Language (XPath) Version 1.0. Recommendation <http://www.w3.org/TR/1999/REC-xpath-19991116>, World Wide Web Consortium, November 1999.
7. Dan Chiorean, Mihai Pasca, Adrian Carcu, Cristian Botiza, Sorin Moldovan. Ensuring UML models consistency using the OCL Environment. Sixth International Conference on the Unified Modelling Language - the Language and its applications, San Francisco, 2003.
8. DeRose, S.; Maler, E. and Orchard, D. XML Linking Language (XLink) Version 1.0. W3C Recommendation <http://www.w3.org/TR/xlink/>, World Wide Web Consortium, June 2001.
9. Document Object Model (DOM) Level 1 Specification. W3C Recommendation <http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001>, World Wide Web Consortium, 1998.
10. Egyed, A. Scalable Consistency Checking between Diagrams – The ViewIntegra Approach. En: 16th IEEE International Conference on Automated Software Engineering. 2001.

11. Fowler, M. UML Distilled: A brief guide to the Standard Object Modeling Language. Addison-Wesley. 2004.
12. Fuentes L., Vallecillo A., Using UML Profiles: A case study. Workshop on Automating Object-Oriented Software Development Methods. 2002
13. Glinz, M. A lightweight approach to consistency of Scenarios and Class Models. En: Fourth International Conference on Requirements Engineering, Illinois (USA), June 10-23, 2000.
14. Grundy, J., Hosking, J., Mugridge, W.B. Inconsistency Management for Multiple-View Software Development Environments. IEEE Transactions on Software Engineering 24, 11 (Nov. 1998). 960-981.
15. Gryce, Clare; Finkelstein, Anthony and Nentwich, Christian. Lightweight Checking for UML Based Software Development. En: Workshop on Consistency Problems in UML-based Software Development., Dresden, Germany, 2002.
16. Herramienta Objecteering, Disponible en Web en: <http://www.objecteering.com/>
17. Herramienta OCLE. Disponible vía Web en <http://lci.cs.ubbcluj.ro/OCLE>  
Disponible vía Web en: <http://www.therationaledge.com/rosearchitect/mag/index.html>
18. Herramienta Together. Disponible en Web en: <http://www.borland.com/together/controlcenter/index.html>.
19. Jackson M., "Software Requirements & Specifications: a lexicon of practice, principles and prejudices", Addison Wesley, Great Britain, 1995.
20. Jacobson, Ivar; I. Booch and G. Rumbaugh J. The Unified Software Development Process, Addison Wesley, 1999.
21. Jacobson, Ivar. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.
22. Jacobson, Ivar. Object-Oriented Development in an Industrial Environment. En: Proceedings OOPSLA' 87, special issue of SIGPLAN Notices. Vol 22, N°12, pp.183-191, 1987.
23. Java Technology. Disponible vía web en: <http://java.sun.com/>
24. Kösters, G., Pagel, B.-U., Winter, M. Coupling Use Cases and Class Models. En: BCS FACS/EROS Workshop on Making Object-oriented Methods more Rigorous. London, 1997.

25. Liu, D. Subramaniam, K. Far, B.H. Eberlein, A. Automating transition from use-cases to class model. Canadian Conference on Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Pp. 831-834 vol.2
26. Mandel, T. The elements of user interface design. New York: Wiley, 1997.
27. Marcus, A. "Principles of effective visual communication for graphical user interface design". En: Baecker, R.M.; Rudin, J.; Buxton, W.A.S. Readings in human computer interaction: toward the year 2000. San Francisco: Morgan Kaufman, 1995, pp. 425-441.
28. MDA – Model Driven Architecture. Disponible vía Web en <http://www.omg.org/mda>
29. Mellor J., Scott K., Uhl Aa., y Weise D. "MDA Distilled: Principles of Model-Driven Architecture". Addison Wesley. 2002
30. Meta Object Facility (MOF) Specification. OMG Document: formal/2002-04-03, 2003.
31. Minoli, Mariano. Presentando XQuery en sociedad. En: <http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art183.asp>, 2004
32. MOF – Meta Object Facility. Disponible vía web en <http://www.omg.org/mof>
33. Moors, Michael. Consistency Checking, Rose Architect, Spring Issue, April 2000.
34. OCL. Object Constraint Language Specification. Disponible en Web en: <http://www.omg.org/technology/documents/formal/ocl.htm>
35. OMG – Object Management Group. Disponible vía Web en <http://www.omg.org>
36. Shishkov B, Xie Z, Lui K, Dietz J. Using norm analysis to derive use case from business processes. 5th Workshop on Organizations semiotics. June 14-15. Delft the Netherlands. 2002
37. Sunetnanta, T. y Finkelstein, A. Automated Consistency Checking for Multiperspective Software Specifications. Proceedings of the 26th Australasian computer science conference, Volume 16, 2003, Pp. 291-300.
38. UML - Unified Modeling Language. Versión 1.1. Septiembre de 1997. Disponible vía Web en <http://www.omg.org/uml>
39. Unified Modeling Language: Superstructure versión 2.0. OMG Final Adopted Specification, 2002.
40. W3C - World Wide Web Consortium. Disponible via web en: <http://www.w3.org/>

41. Warmer J, Kleppe A. The Object Constraint Language: Getting your models ready for MDA, Addison Wesley, 2003.
42. Weitzenfeld, Alfredo. Ingeniería de Software orientada a objetos con Uml, Java e Internet. Thomson editores. 2005.
43. XMI – XML Metadata Interchanger. Disponible vía Web en <http://www.omg.org/technology/xml/>
44. XML - Extensible Markup Language. Disponible vía web en <http://www.w3.org/XML/>
45. Zowghi, D. and Gervasi, V. The Three Cs of requirements: consistency, completeness, and correctness. En: International Workshop on Requirements Engineering: Foundations for Software Quality, Essen, Germany: Essener Informatik Beitiage, 2002, pp. 155-164.
46. Zapata Carlos Mario, Villegas Sandra Milena, Arango Fernando. Reglas de consistencia entre modelos de requisitos de Un-Método. Revista Universidad EAFIT. Vol 42, No 141, 2006.