UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

SISTEMA DE TRANSACCIONES COOPERATIVAS PARA UN AMBIENTE DE *CASE*

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A:

DANTE ORTIZ ANCONA

DIRECTORA DE TESIS: DRA. AMPARO LÓPEZ GAONA

MÉXICO, D.F. **DICIEMBRE DE 2003.**

Dedicatoria

A mi esposa Claudia y a mis hijas
Olga Elisa y Claudia Gloria.

Agradecimientos

Agradezco el apoyo y atención de todas las personas que me ayudaron en la culminación de este trabajo, en especial a mi profesor y amigo Cristóbal Juárez Castellanos.

Muchas gracias Dra. Amparo López Gaona, Dra. María Garza, Dr. Federico O´Reilly y Dr. Manuel Romero Salcedo por brindarme su apoyo en los momentos que mas lo necesité.

Agradezco a mis profesores su dedicación y el conocimiento que depositaron en mi

Dr. Horacio Carvajal Sánchez
Dr. David Arturo Rosenblueth Laguette
Ing. Mario Rodríguez Manzanera
M. en C. Gustavo Arturo Márquez Flores
M. en C. María Guadalupe E. Ibargüengoitia González
Dra. Hanna Oktaba
M. en C. Javier García García
M. en C. Cristóbal Juárez Castellanos
Dr. Ángel Kuri Morales
Dr. Sergio Marcellin Jacques

Agradezco la amistad y el apoyo administrativo de Lourdes, Violeta, Alfredo, Juanita, Don Mardonio y Cecilia.

Agradezco el apoyo a mis amigos y colegas Rita Carolina Rodríguez, Ricardo Villarreal, Apolinar Calderón, Mauricio Fuentes, Francisco Cárdenas y María de Jesús Madera.

Muchas gracias a mis sinodales por haber dedicado su tiempo en la revisión y mejoramiento de este trabajo.

ÍNDICE

1 INTRODUCCIÓN

1.1 Antecedentes

Con el surgimiento y maduración de la ingeniería de software, nace una necesidad nueva: el desarrollo de herramientas que faciliten la construcción y documentación de los proyectos de desarrollo de software. Es común referirse a este tipo de herramientas con el nombre CASE (Computer Aided Software Engineering) que significa ingeniería de software asistida por computadora.

Existe una lista amplia de herramientas comerciales que apoyan el proceso de desarrollo de software. Sin embargo, la mayoría de estas herramientas sólo apoyan una pequeña parte del proceso de desarrollo de software. Por ejemplo, algunas herramientas sirven únicamente para capturar los requisitos de un sistema, otras para mejorar la productividad en la programación utilizando algún lenguaje, etc. Sería conveniente poder combinar muchas de estas herramientas con la finalidad de crear un ambiente integrado de desarrollo de software.

El desarrollo de un proyecto de software grande involucra la participación de un gran número de personas que se agrupan comúnmente de acuerdo a una funcionalidad. Por ejemplo, analistas, diseñadores, programadores, un grupo de pruebas, líderes de proyecto y arquitectos de software. Todos ellos colaboran para cumplir el objetivo común de terminar el proyecto. Las herramientas CASE comerciales son monousuario, esto implica que cada participante del grupo de desarrollo de software debe tener su propia herramienta CASE y el trabajo se debe mezclar para obtener el producto final, o bien, seguir un enfoque orientado a fases (es decir, primero se termina con el análisis para poder empezar el diseño, posteriormente, hay que terminar el diseño para seguir con la programación y así sucesivamente).

El problema consiste en desarrollar una herramienta CASE que soporte todas las etapas del ciclo de vida del software y que además facilite y promueva el trabajo cooperativo.

La motivación más importante para la solución de este problema es aumentar la productividad en el desarrollo de proyectos complejos de software.

El desarrollo de un proyecto de software grande involucra inherentemente el trabajo cooperativo. Una tendencia actual, en el desarrollo de herramientas CASE, es el soporte del trabajo cooperativo.

1.2 Objetivo de la Tesis

El objetivo de esta tesis consiste en describir los principios para el desarrollo de un sistema de transacciones cooperativas que se requiere para dar el soporte a una herramienta CASE cooperativa.

1.3 Alcances de la Tesis

Describir los fundamentos para el diseño y desarrollo de un sistema de transacciones cooperativas.

Desarrollo y programación de un componente del sistema de transacciones cooperativas denominado “Especificador de Criterios de Consistencia para un Sistema de Transacciones Cooperativas”.

1.4 Organización de la Tesis

En el capítulo dos se definen algunos conceptos fundamentales de las bases de datos, los ambientes de CASE y el ambiente cooperativo. Se describen las características fundamentales de las transacciones tradicionales y las transacciones avanzadas. Se explican resumidamente algunos de los modelos de transacciones avanzadas más utilizados.

En el capítulo tres se describen los conceptos fundamentales de la programación orientada a objetos y el lenguaje utilizado para construir, visualizar, especificar y documentar el sistema de software desarrollado en esta tesis.

En el capítulo cuatro se efectúa la descripción detallada del Sistema de Transacciones Cooperativas, ilustrando su arquitectura. Se efectúa también una descripción de la arquitectura del Sistema Especificador de Criterios de Consistencia para el Sistema de Transacciones Cooperativas.

En el capítulo cinco se describe el diseño e implantación del sistema objeto de esta tesis, ilustrando y describiendo cada uno de sus módulos.

En el capítulo seis se describen las cualidades representativas, que se tomaron en cuenta, para obtener la calidad en el desarrollo del sistema. Se describen también las perspectivas del trabajo futuro. Por último, se describen las conclusiones en base a la evaluación de resultados.

2 AMBIENTE COOPERATIVO DE DESARROLLO DE SOFTWARE

En éste capítulo se definen algunos conceptos fundamentales de los ambientes de CASE, las bases de datos, y el ambiente cooperativo, así como las relaciones entre ellos. Se describen las características fundamentales de las transacciones tradicionales y las transacciones avanzadas. Se explican resumidamente algunos de los modelos de transacciones avanzadas más utilizados.

2.1 Ambiente de CASE

Un ambiente de CASE es una colección integrada de herramientas accedidas por medio de un lenguaje o de una interfaz que brinde soporte de programación durante el ciclo de vida del software. Estas herramientas deben soportar todas las tareas relativas a la administración y desarrollo de software, tales como la especificación de requerimientos, análisis, diseño, programación, prueba, mantenimiento, administración de la configuración, administración de versiones, ingeniería directa, ingeniería inversa y ejecución.

Los ambientes de CASE disponibles en la actualidad son herramientas relativamente simples, es decir, soportan el desarrollo de un tipo particular de componente de software o el enfoque orientado a fases (la salida de una herramienta es transformada a un formato que será la entrada de la siguiente herramienta). Decir que un ambiente de CASE es una colección integrada de herramientas significa que estas no deben conectarse vagamente como sucede en el enfoque orientado a fases. A continuación se muestran los requerimientos esenciales de un ambiente de CASE:

- **Modelado de información compleja:** se refiere al modelado de objetos de tamaño variable, tal como documentos, programas, gráficas dirigidas, árboles de análisis sintáctico, diagramas de flujo, gráficas de dependencias, etc.

- **Manejo de versiones:** se refiere a la administración y almacenamiento de varias versiones de documentos, programas y otros objetos

- **Restricciones de integridad semántica:** Son reglas que se utilizan para restringir los estados y/o las transiciones de estado del sistema.

- **Manejo de transacciones avanzadas:** se refiere al manejo de transacciones de duración larga, interactivas y cooperativas.

La figura 2.1 ilustra seis de los ambientes de CASE comerciales más completos que existen. La primera columna muestra el nombre del producto, la segunda columna muestra la dirección de su página de internet y la tercer columna muestra el nombre de la compañía propietaria del producto. Estos ambientes de CASE se utilizan para el desarrollo de software orientado a objetos.

CASE	URL	Empresa propietaria
ObjectiF	www.microtool.de	MicroTOOL GmbH
Enterprise Architect	www.sparxsystems.com.au	Sparx Systems
Rational Rose	www.rational.com	IBM
Cradle	www.threesl.com	3SL
Select	www.selectbs.com	Select Business Solutions Inc.
Stp	www.aonix.com	Aonix

Figura 2.1. Ambientes de CASE comerciales.

La figura 2.2 ilustra las características más importantes de estos seis ambientes de CASE. La primer columna muestra las tareas relativas a la administración y desarrollo de software así como los requerimientos esenciales que debe tener un ambiente de CASE. Las columnas restantes muestran el nombre de cada ambiente de CASE así como el cumplimiento de cada una de las características enunciadas en la primer columna.

Cabe aclarar que la figura 2.2 se construyó utilizando las características que los proveedores aseguran sobre sus productos. Sin embargo, resulta conveniente hacer las pruebas correspondientes sobre cada producto para corroborar la veracidad de sus características. Estas pruebas no se realizaron porque se tienen que adquirir los productos y cada uno de ellos es bastante caro, además de que está fuera de los alcances de este trabajo.

	ObjectiF	Enterprise Architect	Rational Rose	Cradle	Select	Stp
Especificación de requerimientos	Si	Si	Si	Si	Si	Limitada
Análisis	Si	Si	Si	Si	Si	Si
Diseño	Si	Si	Si	Si	Si	Si
Programación	Limitada	Limitada	Limitada	Limitada	Limitada	Limitada
Prueba	Si	Si	Si	Limitada	No	No
Mantenimiento	Si	Si	Si	Limitada	Si	Limitada
Administración de la configuración	Si	Limitada	Si	No	No	No
Administración de versiones	No	Limitada	Si	No	No	No
Ingeniería directa	Si	Si	Si	Si	Si	Si
Ingeniería inversa	No	Si	Si	Si	Si	Si
Ejecución Modelado de información compleja	Si Limitada	No Limitada	No Limitada	No Limitada	Limitada Limitada	No Limitada
Restricciones de integridad semántica	Limitada	Limitada	Limitada	Limitada	Limitada	Limitada
Almacenamiento	Archivo	Archivo	Archivo	Base de datos	Archivo	Base de datos
Transacciones avanzadas	No	No	No	No	No	No
Soporte al trabajo cooperativo	No	No	No	No	No	Limitada

Figura 2.2. Características de algunos de los ambientes de CASE comerciales más completos.

Ninguno de los ambientes de CASE descritos en la figura 2.1 es completamente un ambiente integrado. Para cumplir con las características descritas en la figura 2.2 se utilizan varios productos del mismo proveedor; cada uno tiene una interfaz con los demás para permitir la integración de los componentes de software generados. De estos seis ambientes de CASE el líder es Rational Rose, no necesariamente por ser el mejor, sino porque ganó mucho prestigio al salir a la venta primero. Actualmente es el más vendido y el más caro.

Se observa en la figura 2.2 que únicamente Cradle y Stp utilizan como sistema de almacenamiento bases de datos y en el caso de STP, esto es precisamente lo que promueve el trabajo cooperativo. Resulta conveniente utilizar como

sistema de almacenamiento bases de datos ya que estas brindan una gran variedad de servicios.

2.2 Bases de Datos

Una Base de Datos es una colección de datos interrelacionados y almacenados permanentemente en una computadora tal que:

a) Los datos son compartidos por diferentes usuarios y programas de aplicación, pero existe un mecanismo común para la inserción, actualización, borrado y consulta de los datos.
b) Tanto los usuarios finales como los programas de aplicación no necesitan conocer los detalles de las estructuras de almacenamiento.

Un **DBMS** (**D**ata **B**ase **M**anagement **S**ystem - Sistema Administrador de la Base de Datos) es un sistema formado por una Base de Datos y un conjunto de programas para administrarla y explotarla.

Servicios proporcionados por un DBMS:

- Almacenamiento y recuperación eficiente de los datos
- Minimización de la redundancia de los datos.
- Aseguramiento de la consistencia de los datos.
- Mantenimiento de la integridad de los datos.
- Otorgamiento de la seguridad de los datos
- Control de la concurrencia de los datos.
- Protección de los datos contra fallas del sistema.
- Administración del diccionario de datos.
- Otorgamiento de una interfaz de alto nivel con los programadores.

Los usuarios y los programas de aplicación interactúan con la base de datos a través de transacciones. Estas constituyen el mecanismo esencial para asegurar la consistencia y la integridad de los datos.

2.2.1 Modelo Tradicional de Transacciones

Una transacción tradicional es un conjunto de operaciones sobre la base de datos que la llevan de un estado consistente a otro estado también consistente y que se considera como una operación atómica, es decir, o se ejecutan todas las operaciones de la transacción o no se ejecuta ninguna.

Las transacciones tradicionales tienen la característica principal de ser serializables, es decir, las transacciones se ejecutan de manera concurrente pero el resultado final es equivalente a una ejecución en serie de las transacciones. Otra característica es que son transacciones cortas, es decir, tienen una duración en el orden de segundos.

Las transacciones cortas deben satisfacer las siguientes propiedades para mantener la consistencia de la base de datos:

- **Atomicidad**: Se refiere al hecho de que todas las operaciones de una transacción deben ser tratadas como una unidad simple, es decir, se ejecutan todas o ninguna.

- **Consistencia**: La transacción deberá llevar a la base de datos de un estado consistente a otro estado consistente. Si se ejecutan algunas transacciones concurrentemente, el DBMS deberá asegurar que la ejecución concurrente de transacciones deje a la base de datos en un estado consistente.

- **Aislamiento**: requiere que cada transacción observe una base de datos consistente, es decir, no deberá leer los resultados intermedios de otras transacciones.

- **Durabilidad**: Requiere que el resultado de una transacción que termina con la instrucción commit se haga permanente en la base de datos aun en caso de fallas. A lo largo del presente trabajo, se utilizará el término commit, para denotar la instrucción para hacer permanentes los resultados de las operaciones de una transacción sobre la base de datos.

estas propiedades se conocen como ACID (del inglés: Atomicity, Consistency, Isolation, Durability).

2.2.2 Usos de un DBMS en Ingeniería de Software

Un DBMS puede ayudar a la administración de información en ingeniería de software al ofrecer una variedad de servicios como son: mecanismos de recuperación, control de concurrencia, aseguramiento de la consistencia, etcétera. Además de que puede ser usado por el ingeniero de software para:

- Ayudar a los usuarios para asociar lógicamente documentación y código.
- Mantener un seguimiento de las anotaciones de los usuarios que contengan explicaciones y asunciones.
- Manejar diferentes versiones de software y la documentación asociada.
- Controlar diferentes vistas de un sistema bajo desarrollo y mantener interfaces estándares.
- Ayudar al manejo de partes en desarrollos en grupo.
- Mantener un histórico acerca de las decisiones de desarrollo.

2.2.3 Características de un DBMS para un CASE

El proceso de desarrollo de software, involucra numerosos tipos de información y cantidades enormes de datos que pueden representarse en diferentes formatos. Se pueden aprovechar los servicios (anteriormente mencionados) que ofrece un DBMS para reducir los esfuerzos en administrar estos datos por parte del CASE. Sin embargo, los datos son generalmente, enormes, complejos y relacionados por numerosas reglas de consistencia haciendo difícil su administración. Asimismo, el proceso de desarrollo de software, en un proyecto grande, requiere la participación de uno o más grupos de personas que colaboren de manera coordinada y coherente para la terminación del proyecto. Por lo tanto, para que un DBMS pueda dar el soporte a un CASE requerirá incorporar adicionalmente las siguientes características:

Modelado de información compleja
Manejo de versiones
Restricciones de integridad semántica
Manejo de transacciones avanzadas.
Actualización de esquemas y objetos
Facilidades de consultas Ad Hoc
Distribución y trabajo cooperativo
Extensibilidad

2.3 Ambiente Cooperativo y CASE

El trabajo cooperativo es un conjunto de tareas realizadas por un conjunto de entidades autónomas (llamadas agentes y que pueden ser entes humanos o computacionales) y que se realizan de una manera organizada, coordinada y coherente para alcanzar una meta individual o global. Permite resolver problemas que por su complejidad, su naturaleza inherentemente distribuida y/o sus requerimientos de eficiencia, no podrían ser resueltos adecuadamente de manera aislada; es evidente que para tal tipo de problemas se requiere la participación de grupos de trabajo. Algunos ejemplos de este trabajo cooperativo son: CASE, Diseño Asistido por Computadora, Edición de documentos, etc.

Un sistema cooperativo es la conformación de un conjunto de agentes autónomos con capacidad de interactuar para realizar tareas comunes. Un agente se define como una entidad que existe en un medio ambiente sobre el cual interactúa con otros agentes y con capacidad de procesamiento y conocimiento.

2.4 Ambiente Cooperativo y Bases de Datos para CASE

Un ambiente cooperativo requiere de una plataforma que permita el intercambio de mensajes entre los agentes que integran un grupo de trabajo cooperativo. Además requiere del acceso a Bases de Datos que pueden ser locales o distribuidas. Los agentes pueden competir por los mismos datos y pueden realizar cambios sobre estos, pudiendo entrar en conflicto con cambios similares efectuados por otros agentes al momento de almacenarlos en la base de datos. En un ambiente cooperativo de desarrollo de software los grupos de trabajo podrían ser los siguientes:

Grupo de análisis
Grupo de diseño
Grupo de codificación
Grupo de pruebas
Grupo de simulación
Grupo de ingeniería inversa

Cada grupo puede estar formado, a su vez, por uno o más agentes, o bien por uno o más grupos de trabajo.

Comúnmente las transacciones utilizadas en los ambientes cooperativos son de duración larga (es decir, del orden de minutos, días o meses), interactúan unas con otras y pueden estar relacionadas por muchas reglas de consistencia. Las noción clásica de serialización de transacciones no es muy adecuada ya que reduce de manera significativa la concurrencia. La atomicidad de transacciones puede ser un precio muy alto al abortar transacciones de duración larga. Por último el aislamiento de transacciones tendría lugar cuando las transacciones no interactúen. Esto da lugar al surgimiento de un tipo nuevo de transacciones conocido como transacciones avanzadas [Won95].

2.5 Transacciones Avanzadas

Las transacciones avanzadas (también conocidas como transacciones no tradicionales) están caracterizadas por:

- **Duración larga:** las actividades consisten de una secuencia de accesos a la base de datos que pueden durar de minutos a meses.
- **Control interactivo:** los usuarios seleccionan las acciones junto con sus actividades conforme van avanzando.
- **Cooperación entre usuarios:** los usuarios comparten resultados parciales de sus actividades mientras están en progreso.
- **Manejo de versiones**: Una versión representa el estado de un objeto en algún instante en la historia de su desarrollo.

Existen diversos modelos de transacciones avanzadas, de los cuales se presenta un resumen de algunos:

2.5.1 Modelo Checkout

En este modelo los usuarios copian los objetos de una base de datos a áreas privadas para su manipulación. Las dos operaciones básicas en este modelo son el checkout y el checkin. La Operación de checkout consiste en reservar un objeto de la base de datos, es decir, una vez que un usuario recupera un objeto para editarlo, este se vuelve inaccesible a los demás usuarios a menos que se genere una versión nueva del objeto. La operación de checkin consiste en depositar en la base de datos un objeto reservado previamente. Dos o más usuarios pueden modificar el mismo objeto únicamente trabajando sobre versiones paralelas.

El modelo básico checkout es ampliamente usado en actividades interactivas y de duración larga, no maneja la cooperación entre usuarios y sufre de dos problemas principales:

a) No soporta la noción de objetos compuestos o agregados, forzando a los usuarios a reservar o depositar en la base de datos cada subobjeto individualmente.

b) Los mecanismos de reserva/depósito usualmente no proveen control de concurrencia sobre los objetos reservados mas allá de los que ofrece la base de datos.

2.5.2 Modelo de Consistencia Multinivel

En este modelo las copias de los objetos son reservadas en bases de datos experimentales que representan áreas de trabajo. Las bases de datos experimentales no son necesariamente privadas para un solo usuario.

Todos los objetos son reservados desde una base de datos principal en la misma base de datos experimental de nivel superior compartida entre todos los usuarios, desde la cual los objetos pueden ser reservados en bases de datos experimentales hijas representando grupos de usuarios, los cuales pueden tener así mismo sus bases de datos experimentales hijas para subgrupos, hasta que eventualmente

alcanzan la base de datos experimental hoja donde los cambios actuales son normalmente hechos por individualidades. La verificación de consistencia se realiza desde las hojas hasta la base de datos principal.

2.5.3 Modelo de Esferas de Interfaz, Dependencia y Sincronización

Este es un modelo de transacciones anidadas que considera los siguientes tres aspectos importantes:

1) **Interfaz**: es la comunicación entre una transacción padre y una transacción hija y que puede realizarse de dos maneras:

   a) Por solicitud simple: El padre solicita una consulta o actualización al hijo y espera hasta que el hijo regrese el resultado completo atómicamente

   b) Por conversación: significando que el control alterna entre el padre que genera una serie de solicitudes y el hijo responde a estas solicitudes individualmente.

2) **Dependencia:** Es la necesidad de que una transacción padre haga commit cuando la transacción hija ya lo hizo.

3) **Sincronización** Es el mecanismo para controlar la concurrencia al acceder los objetos compartidos entre las transacciones padres e hijas, a través de candados u otro esquema.

Las transacciones se agrupan en esferas dependiendo de los tres aspectos anteriormente mencionados. Por ejemplo, una esfera de interfaz incluye todas las transacciones que están involucradas en una cadena de conversaciones. Del mismo modo, una esfera de permanencia es un conjunto de transacciones dependientes.

La interfaz por conversación requiere agrupar las transacciones padre e hijas en la misma esfera de permanencia, esto es debido a que si una transacción hija aborta (por cualquier razón) en la mitad de la conversación, no únicamente deshace los cambios hechos por la transacción hija, sino también los realizados por la transacción padre hasta el inicio de la conversación.

Dados estos tres aspectos, se definen tres atributos para cada transacción anidada cuando es creada. El primer atributo refleja el criterio de interfaz, puede ser INTERFAZ o NOINTERFAZ, con INTERFAZ indica que la transacción hija está asociada con su propia esfera de interfaz y NOINTERFAZ significa que ésta comparte una esfera de interfaz con la transacción padre. El atributo de dependencia es activar PERMANENCIA o NOPERMANENCIA, y el tercer atributo, refleja el modo de sincronización y puede ser SINCRONIZA o NOSINCRONIZA. Las ocho combinaciones de estos atributos definen niveles de coordinación entre una transacción padre y su hija.

2.5.4 Modelo de Notificación

En este modelo se utiliza la notificación como un mecanismo para implementar políticas de control de concurrencia cooperativa. Estas políticas incorporan la intervención humana como parte del algoritmo para la solución del conflicto (es decir, propician la comunicación entre las partes involucradas vía telefónica, plática por computadora o correo electrónico). El modelo usa dos tipos de notificación: **inmediata** y **retardada**. La notificación inmediata alerta a los usuarios afectados de algún intento en accesos conflictivos, así como cuando ocurre el conflicto. La notificación retardada alerta a los usuarios de todos los conflictos que han ocurrido únicamente cuando una de las transacciones del conflicto intenta hacer commit.

2.5.5 Modelo Orientado a Grupos

El modelo orientado a grupos agrupa en dos categorías las transacciones largas: en **transacciones de grupo (TG**) y **transacciones de usuario (TU)**. Cada TU es una subtransacción de alguna TG. El modelo provee primitivas para definir grupos de usuarios, con la intención de asociar cada TG con un grupo usuario. Una TG reserva objetos de una base de datos pública en la base de datos correspondiente al grupo, con lo cual los usuarios crean sus propias bases de datos e invocan TUs para reservar objetos de la base de datos del grupo en sus bases de datos.

El modelo soporta cinco modos de permiso sobre la versión de un objeto: (1) sólo-lectura, el cual deja disponible una

versión únicamente para lectura; (2) lectura-derivación el cual permite que múltiples usuarios puedan leer la misma versión o derivar una versión nueva; (3) derivación compartida, el cual permite al propietario del permiso tanto leer la versión como derivar una versión nueva, mientras permite lecturas paralelas de la misma versión y derivación de versiones nuevas por otros usuarios; (4) derivación exclusiva, el cual permite al propietario leer la versión de un objeto y derivar una nueva y permite únicamente lecturas paralelas de la versión original; y (5) permiso exclusivo, el cual permite al propietario leer, modificar y derivar una versión y no permite operaciones de otros usuarios sobre esa versión.

2.5.6 Modelo de Transacciones Cooperativas

Este modelo consiste de una estructura jerárquica formada por **grupos de transacciones (GT)** y **transacciones cooperativas (TC).** Cada nodo interno es un grupo de transacciones y cada hoja es una transacción cooperativa. Siempre existe el grupo que está en el tope de la jerarquía y se le denomina grupo Raíz.[Elm95].

Cada grupo de transacciones contiene un conjunto de miembros que cooperan para hacer una tarea determinada. Un miembro puede ser una TC o un GT. Cada grupo controla activamente la interacción entre sus miembros. Debido a la naturaleza cooperativa de los miembros, la secuencia de operaciones de un solo miembro no necesariamente deja a la base de datos en un estado consistente. En lugar de esto, cada GT tiene su propia especificación de consistencia. Para especificación de consistencia, se utiliza la noción de patrones y conflictos [Ska89, Ska9l]. Los patrones en el GT indican las maneras en que las operaciones de los miembros se deben ordenar para terminar la tarea. Los conflictos especifican cómo las operaciones de los miembros no pueden ser ordenadas, para prevenir efectos secundarios indeseados. El GT se asegura que sus miembros interactúen solamente de las maneras permisibles por su especificación de consistencia y de esta manera garantiza que las operaciones de sus miembros dejen a la base de datos en un estado consistente.

Cada GT tiene un conjunto local de versiones del objeto. Así, puede haber muchas versiones de un objeto dispersado a través

de la jerarquía de la transacción. Para un objeto específico, la versión en la raíz es la versión más vieja y las que están abajo en la jerarquía son más recientes. La visibilidad se define naturalmente según la jerarquía. Es decir, la versión de un objeto en un GT es accesible a cualesquiera de sus descendientes.

Cada GT proporciona sus propias garantías sobre como proteger sus copias ante diversas clases de fallas (permanencia). Así, un GT puede tener su propia Base de Datos de la cual puede recuperar las copias de sus objetos después de un fallo del sistema. El GT Raíz en el tope de la jerarquía contiene la versión más estable de cada objeto y brinda las garantías más fuertes sobre la permanencia de sus copias.

Una versión del objeto se copia automáticamente desde el GT hacia el miembro o desde el padre del GT cuando el miembro inicia una operación de lectura sobre el objeto. La versión nueva del objeto se escribe en el padre del GT cuando el miembro indica que ha acabado de modificar el objeto y la operación de escritura es aceptable. Todas las operaciones deben cumplir con la especificación de consistencia del GT.

Se permite a los miembros hacer commit o deshacer parte de su trabajo mientras está en progreso. Cuando un tarea se completa, se permite al miembro realizar un punto de verificación (checkpoint). El punto de verificación propaga los efectos de las operaciones al padre del GT transformando las secuencias de operaciones de los miembros del GT en secuencias más cortas y que tienen el mismo efecto neto sobre esos objetos. Por ejemplo, varias operaciones de escritura sobre la misma versión de un objeto del GT pueden ser colapsados como una sola operación de escritura sobre la versión del objeto en el padre del GT. Una secuencia de números iguales de operaciones de incremento y decremento no tiene ningún efecto, así que no se propaga nada al padre en este caso.

Una **Transacción Cooperativa** de un GT es un conjunto de operaciones ejecutadas por un miembro del GT y que no necesariamente pueden considerarse como una operación atómica. Las TCs representan entes humanas o aplicaciones. Asumimos que las TCs son duraderas, ampliables y que pueden interactuar con otras TCs en su GT, tanto externamente como a través de los objetos de la base de datos. Así, pueden tener una noción razonable de lo que están haciendo los otros miembros en su GT. Durante su tiempo de vida, una TC realiza

operaciones sobre versiones de los objetos en su GT. Estas operaciones son ejecutadas o rechazadas por el GT una vez que este determina que cumplen o no con sus especificaciones de consistencia.

Los miembros acceden y actualizan las versiones locales de sus objetos usando operaciones. Una operación define una acción atómica realizada por un miembro de un GT sobre un objeto. Es decir, una **operación** es una tupla δ = <***M***, **o**, ***O***>, donde ***M*** es un miembro, ***o*** es un operador y ***O*** es un identificador de objeto. Eventualmente, los efectos de una operación se propagan hacia el GT raíz o se deshacen por uno de los miembros del grupo.

Una **historia** de una GT es una secuencia de operaciones ordenadas parcialmente y ejecutadas por los miembros del GT, sobre las versiones de los objetos pertenecientes al GT. La historia del GT se refleja en la historia de su padre y se mantiene en el registro del GT.

2.6 Discusión

Los ambientes de CASE comerciales más completos que existen, utilizan en su mayoría el archivo como medio persistente de almacenamiento de datos. Para compartir los datos y promover el trabajo cooperativo, resulta más conveniente utilizar una base de datos; además de que pueden aprovecharse los diferentes servicios que provee un DBMS. Mas aún, uno de los requerimientos esenciales de un DBMS para una ambiente de CASE, es el manejo de transacciones avanzadas.

En la actualidad, existen diversos modelos de transacciones avanzadas, de los cuales se presentaron, en este capítulo, seis de los mas utilizados. La figura 2.3 muestra una comparación entre los modelos de transacciones avanzadas descritos previamente. Se toman como características de comparación el mantenimiento de la consistencia, la duración, la interacción y la cooperación. El modelo que cumple mejor con estas características, de acuerdo a la figura 2.3, es el modelo de transacciones cooperativas, por tal motivo, se decidió usar éste modelo, como base para el desarrollo del sistema del presente trabajo.

Modelo	Consistencia	Larga	Interactiva	Cooperativa
Checkout	No	Si	Si	No
Consistencia Multinivel	Si	Si	Si	Limitada
Esferas de Interfaz, Dependencia y Sincronización	Posible	Si	Si	No
Notificación	No	Si	Si	Limitada
Orientado a Grupos	Mínima	Si	Si	Limitada
Transacciones Cooperativas	Mínima	Si	Si	Si

Figura 2.3 Comparación entre modelos de transacciones avanzadas

Por ser un estándar, se decidió utilizar el Lenguaje Unificado de Modelado (UML) para la descripción, análisis, diseño e implantación del sistema desarrollado en esta tesis. También se decidió utilizar las tecnologías orientadas a objetos. Por tal motivo, el siguiente capítulo describe los fundamentos del modelado y desarrollo orientado a objetos, así como UML. En el capítulo cuatro se muestra con más detalle la descripción de modelo de transacciones cooperativas.

3 MODELADO Y DESARROLLO ORIENTADO A OBJETOS

En éste capítulo se definen algunos conceptos fundamentales de la programación orientada a objetos y el Lenguaje Unificado de Modelado utilizado para construir, visualizar, especificar y documentar el sistema de software desarrollado en ésta tesis.

3.1 Clases y Objetos.

Un **objeto** es la representación de una entidad, ya sea conceptual o del mundo real. Un objeto puede representar una cosa concreta como un camión, una computadora o una concepto tal como un proceso químico, una transacción bancaria, una orden de compra, una historia de crédito o una tasa de interés [Quat01].

Un objeto es un concepto, abstracción o cosa con significado y fronteras bien definidas para una aplicación. Cada objeto en un sistema tiene tres características: estado, comportamiento e identidad.

Figura 3.1. Objeto

La abstracción de objeto se caracteriza por tener una identidad única que lo distingue de otros objetos. También tiene un estado, que permite informar lo que éste representa y su comportamiento, es decir lo que él sabe hacer (Figura 3.1). Hablando en términos computacionales, la identidad del objeto se puede interpretar como la referencia. El estado del objeto es una lista de variables conocidas como sus atributos, cuyos valores representan el estado que

caracteriza al objeto. El comportamiento es una lista de métodos, procedimientos, funciones u operaciones que un objeto puede ejecutar a solicitud de otros objetos.

Una clase es una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones con otros objetos y semántica [BRJ99][Quat01]. Es decir, una clase es una plantilla, molde, esquema o un patrón para crear objetos. Cada objeto es una instancia de una clase. Los objetos no pueden ser instancias de más de una clase.

3.2 Herencia y Polimorfismo.

Definición de Herencia

La herencia es una relación entre clases en donde una clase puede heredar sus atributos y métodos a varias subclases (la clase que hereda es llamada superclase). Esto significa que una subclase, aparte de los atributos y métodos propios, tiene incorporados los atributos y métodos heredados de la superclase. Una subclase puede a su vez comportarse como una superclase y heredar a otras clases, creando de esta manera la jerarquía de herencia. El hecho de que una clase hereda de más de una superclase se conoce como herencia múltiple.

Definición de Polimorfismo

En un esquema general, el polimorfismo representa la capacidad de una entidad para tener múltiples formas. El polimorfismo es un concepto de la teoría de tipos. De acuerdo a [CW86], el polimorfismo se divide en dos categorías (universal y Ad hoc) y cuatro variedades: coerción, sobrecarga, paramétrico e inclusión. (ver Figura 3.2).

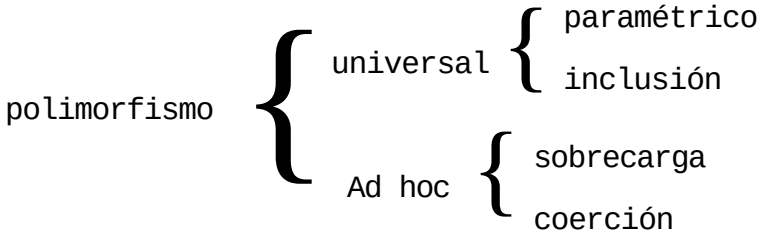


Figura 3.2. Tipos de polimorfismo.

En el polimorfismo universal un método actúa sobre un número infinito de tipos y todos los tipos tienen una estructura común. En el polimorfismo Ad hoc un método actúa sobre un número finito de tipos que no necesariamente tienen una estructura común.

La **coerción** es una operación semántica para convertir el tipo de un argumento al esperado por una método. Por ejemplo, en el siguiente fragmento de código en java, se definen las variables any_operator y dbobj que pertenecen a las clases CTAnyOperator y CTOperationArgument respectivamente. CTAnyOperator es una subclase de CTOperationArgument. Con la instrucción de asignación, el objeto que contiene la variable any_operator y que pertenece a la clase CTAnyOperator es convertido implícitamente a un objeto que pertenece a la clase CTOperationArgument.

```
protected static CTAnyOperator any_operator = new CTAnyOperator ();
protected static CTOperationArgument dbobj = null;

  public void Execute()
  {
    dbobj = any_operator;
  }
```

la siguiente instrucción de java, convierte explícitamente el objeto que se encuentra en el tope de la pila a un objeto que pertenece a la clase Symbol.

```
stack_symbol= (Symbol) parser_stack.pop();
```

En la **sobrecarga**, el mismo nombre es utilizado para denotar diferentes métodos y sus argumentos deciden que método se debe ejecutar. Por ejemplo, el siguiente fragmento de código en java, ilustra la duplicidad en la definición del método Grammar. Se observa, sin embargo que tanto la lista de argumentos como su implantación son diferentes en cada caso.

```
public Grammar()
  {
  }

public Grammar(ScannerBuffer termbuf, ScannerBuffer nontermbuf, ScannerBuffer
                    rulesbuf) throws Exception
  {
        LoadGrammar(termbuf, nontermbuf, rulesbuf);
  }
```

El **polimorfismo paramétrico** permite el uso de una abstracción simple con diferentes tipos. Por ejemplo, la abstracción *Lista* representa una lista homogénea de objetos que deberá proporcionarse a un módulo genérico. La abstracción puede usarse infinidad de veces al especificar el tipo de objetos contenidos en *Lista*. Puesto que el tipo parametrizado, puede ser cualquier tipo definido por el usuario, existe un número infinito de usos para la abstracción genérica haciendo que este sea el tipo de polimorfismo más poderoso. No es posible implantar este tipo de polimorfismo en el lenguaje Java y por consiguiente no se aplicó en el desarrollo de éste trabajo. Sin embargo, para fines didácticos y de claridad se proporciona el siguiente fragmento de código hecho en c++ que muestra una plantilla para construir la clase Lista con una infinidad de tipos. Se sustituye la variable T con el tipo de dato correspondiente.

```
Template class Lista for T {
  Attributes
  .....
  methods:
        append (T element)
        T getFirst()
        T getNext()
}
```

para construir una lista cuyos elementos sean los números reales 30.5 y 20.2 se ejecutan las siguientes sentencias.

```
Lista for float Numeros
Numeros.append(30.5)
Numeros.append(20.2)
```

El **polimorfismo por inclusión** consigue un comportamiento polimórfico con la relación de inclusión entre tipos o conjuntos de valores. En algunos lenguajes de programación orientados a objetos la relación de inclusión es una relación de subtipos. Por ejemplo, un nombre (tal como una declaración de variable) puede denotar objetos de diferentes clases que están relacionadas por una superclase común o por que implantan la misma interfaz. Así, cualquier objeto denotado por este nombre es capaz de responder a un conjunto común de operaciones de manera diferente [Boo94].

El siguiente fragmento de código ejemplifica el polimorfismo por inclusión.

```
Symbol stack_symbol;
stack_symbol = new NonTerminator("NT");
System.out.println(stack_symbol.IsTerminator());
stack_symbol = new Terminator("ID");
System.out.println(stack_symbol.IsTerminator());
```

En éste ejemplo, las clases NonTerminator y Terminator son subclases de la clase Symbol. Se observa que el tipo del objeto que contiene la variable stack_symbol se resuelve a tiempo de ejecución, la primera instrucción de asignación hace que la variable contenga un objeto de la clase *NonTerminator* mientras que la segunda instrucción de asignación hace que la variable contenga un objeto de la clase *Terminator*. La invocación del método *IsTerminator*, en la tercera instrucción, retorna el valor **FALSE** mientras que en la quinta intrucción retorna el valor de **TRUE**.

3.3 UML

El Lenguaje Unificado de Modelado (UML) es un lenguaje estándar para visualizar, especificar, construir y documentar los componentes de un sistema que involucra una gran cantidad de software.

UML es sólo un lenguaje y por tanto es tan sólo una parte de un método de desarrollo de software. UML es independiente del proceso de desarrollo de software.

UML no es lenguaje de programación visual, pero sus modelos pueden conectarse directamente a una gran variedad de lenguajes de programación. Esto significa que es posible transformar un modelo de UML a un lenguaje de programación tal como Java, C++, Visual Basic, o aún más, a relaciones en una base de datos relacional o al almacenamiento persistente en una base de datos orientada a objetos.

Esta transformación permite ingeniería directa: la generación de código a partir de un modelo UML en un lenguaje de programación. La ingeniería inversa también es posible: se puede reconstruir un modelo en UML a partir de una implementación. La ingeniería inversa requiere de herramientas que la soporten e intervención humana. La combinación de la ingeniería directa y la ingeniería inversa producen una ingeniería de "ida y vuelta", entendiendo por

esto la posibilidad de trabajar en una vista gráfica o textual, mientras las herramientas mantienen la consistencia entre las dos vistas.

Bloques de construcción de UML

El vocabulario de UML incluye tres clases de bloques de construcción [BRJ99]:

1. Elementos
2. Relaciones
3. Diagramas.

Los elementos son abstracciones que son ciudadanas de primera clase en un modelo, las relaciones ligan estos elementos entre sí; los diagramas agrupan colecciones interesantes de elementos.

3.3.1 Elementos de UML

Hay cuatro tipos de Elementos en UML:

1. Estructurales
2. De comportamiento
3. De Agrupamiento
4. De Anotación

Estos elementos son los bloques básicos de construcción orientados a objetos de UML. Se utilizan para escribir modelos bien formados. Es decir, modelos precisos, no ambiguos y completos.

Elementos Estructurales: Los Elementos estructurales, son los nombres de los modelos de UML. En su mayoría son las partes estáticas de un modelo, y representan cosas que son materiales o conceptuales. En total, hay siete tipos de elementos estructurales.

Una **clase,** tal y como se definió en la sección 3.1, se representa gráficamente con un rectángulo, que usualmente incluye su nombre, atributos y operaciones, como en la Figura 3.3. Este ejemplo muestra la clase *Symbol* que contiene un atributo y cuatro métodos. Existen reglas de convención que indican que la primer letra del nombre de una clase se escribe con mayúscula, los nombres de los atributos se

escriben con minúsculas y la primer letra del nombre de un método se escribe con minúscula, posteriormente, la primer letra de cada palabra del nombre del método se escribe con mayúscula. Para el desarrollo de este trabajo se utilizará una convención diferente para el nombre de los métodos con la intención de poder distinguir entre un método que pertenezca a una clase del lenguaje Java de un método que pertenezca a una clase desarrollada en este trabajo. La primer letra de cada palabra del nombre de un método se escribirá con mayúscula.

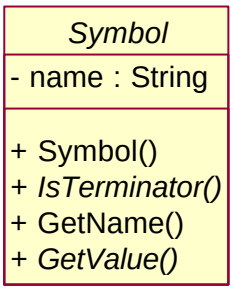


Figura 3.3. Notación en UML de una clase

Una **interfaz** es una colección de operaciones que especifican un servicio de una clase o componente. Por lo tanto, una interfaz describe el comportamiento visible externamente de ese elemento. Una interfaz puede representar el comportamiento completo de una clase o componente o sólo una parte de ese comportamiento. Una interfaz define un conjunto de especificaciones de operaciones (o sea, su signatura) pero nunca un conjunto de implementaciones de operaciones. Gráficamente, una interfaz se representa con un círculo con su nombre como se muestra en la Figura 3.4. Opcionalmente puede contener (debajo del nombre y entre paréntesis), el nombre del paquete al que pertenece.

SECommand
(from sebasic)

Figura 3.4. Notación en UML de una interfaz

Una **colaboración** define una interacción y es una colección de clases y otros elementos que colaboran para proporcionar un comportamiento cooperativo mayor que la suma de los comportamientos de sus elementos. Por lo tanto, las

colaboraciones tienen dimensión tanto estructural como de comportamiento. Una clase dada puede participar en diversas colaboraciones. Estas colaboraciones representan la implementación de patrones que integran un sistema. Gráficamente, una colaboración se representa como una elipse de borde discontinuo, incluyendo sólo su nombre, como se muestra en la figura 3.5. Este ejemplo representa la implantación del sistema para la especificación de los criterios de consistencia.

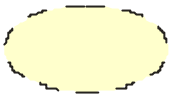


Figura 3.5. Notación en UML de una colaboración

Un **caso de uso** es una descripción de un conjunto de secuencias de acciones que un sistema ejecuta y que produce un resultado observable de interés para un actor particular. Un caso de uso se utiliza para estructurar los aspectos de comportamiento en un modelo. Un caso de uso es realizado por una colaboración. Gráficamente, un caso de uso se representa con una elipse de borde continuo, incluyendo normalmente su nombre, como se muestra en la Figura. 3.6. Un caso de uso representa una funcionalidad provista por un sistema y en este caso, la figura ilustra la funcionalidad del sistema para especificar los criterios de consistencia.

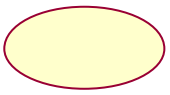


Figura 3.6. Notación en UML de un caso de uso

Los tres elementos restantes (clases activas, componentes y nodos) son todos semejantes a las clases, en cuanto que también describen un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica. Sin embargo estos tres son suficientemente diferentes y son necesarios para modelar ciertos aspectos de un sistema orientado a objetos.

Una **clase activa** es una clase cuyos objetos tienen uno o más procesos o hilos de ejecución y por lo tanto pueden dar origen a actividades de control. Una clase activa es igual que una clase excepto que sus objetos representan elementos cuyo comportamiento es concurrente con otros elementos. Gráficamente una clase activa se representa igual que una clase pero con líneas más anchas, incluyendo su nombre, atributos y operaciones, como en la Figura 3.7.

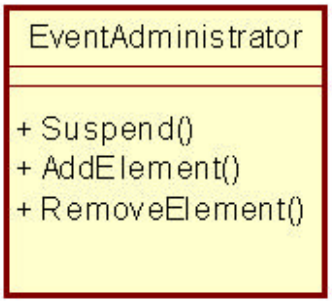


Figura 3.7. Notación en UML de una clase activa

Los dos elementos restantes (componente y nodos) también son diferentes. Representan elementos físicos, mientras que los cinco elementos anteriores representan elementos lógicos o conceptuales.

Un **componente** es un una parte física y reemplazable de un sistema que conforma y proporciona la realización de un conjunto de interfaces. En un sistema se pueden encontrar diferentes tipos de componentes, tal como componentes COM+ o Java Beans, además de componentes que son elementos de procesos de desarrollo, tal como los archivos de código fuente. Un componente típicamente representa el empaquetado físico de diferentes elementos lógicos tales como clases, interfaces, y colaboraciones. Gráficamente, un componente es representado por un rectángulo con pestañas, incluyendo sólo su nombre como en la Figura 3.8. Este ejemplo ilustra la representación del programa ejecutable de un analizador sintáctico.

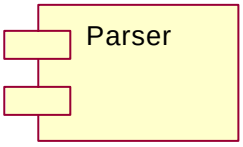


Figura 3.8. Notación en UML de un componente

Un **nodo** es un elemento físico que existe en tiempo de ejecución y representa un recurso computacional, generalmente tiene algo de memoria y frecuentemente capacidad de procesamiento. Un conjunto de componentes puede residir en un nodo y puede también emigrar de un nodo a otro. Gráficamente un nodo es representado por un cubo incluyendo sólo su nombre como en la figura 3.9 que representa una computadora con la funcionalidad de un servidor.

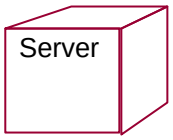


Figura 3.9. Notación en UML de un nodo

Estos siete elementos (clases, interfaces, colaboraciones, casos de uso, clases activas, componentes y nodos) son los elementos estructurales básicos que se pueden incluir en un modelo de UML. Hay también variaciones de estos siete, tales como actores, señales, y utilidades (tipos de clase), procesos e hilos (tipos de clases activas), aplicaciones, documentos, archivos, bibliotecas, páginas y tablas (tipos de componentes).

Elementos de comportamiento.

Los elementos de comportamiento son la parte dinámica de los modelos UML. Estos son los verbos de un modelo, y representan comportamiento en el tiempo y el espacio. De hecho, hay dos tipos principales de elementos de comportamiento:

Una **interacción** es un comportamiento que comprende un conjunto de mensajes intercambiados entre un conjunto de objetos, dentro de un contexto particular, para alcanzar un propósito específico. El comportamiento de una sociedad de objetos o de una operación individual puede especificarse con una interacción. Una interacción involucra muchos otros elementos incluyendo mensajes, secuencias de acción (el comportamiento invocado por un mensaje), ligas (la conexión entre objetos). Gráficamente un mensaje se representa con una línea dirigida incluyendo sólo el nombre de su operación. tal como en la Figura 3.10.

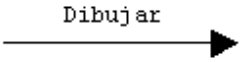


Figura 3.10. Notación en UML de una interacción

Una **máquina de estados** es un comportamiento que especifica la secuencia de estados por los que pasa un objeto o una interacción durante su tiempo de vida en respuesta a eventos, junto con sus reacciones a esos eventos. El comportamiento de una clase individual o una colaboración de clases puede especificarse con una máquina de estados. Una máquina de estados involucra a otros elementos incluyendo estados, transiciones (el flujo de un estado a otro), eventos y actividades. Gráficamente, un estado se representa con un rectángulo redondeado, incluyendo su nombre y sus subestados si los tiene, como se muestra en la Figura 3.11.

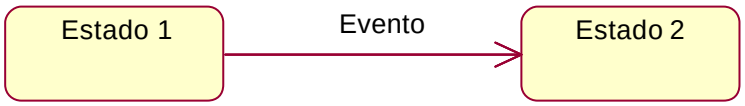


Figura 3.11. Notación en UML para un estado y para una máquina de estados

Estos dos elementos (interacciones y máquinas de estado) son los elementos básicos de comportamiento que se pueden incluir en un modelo UML. Semánticamente, estos elementos están conectados normalmente a varios elementos estructurales, principalmente clases, colaboraciones y objetos.

Elementos de agrupamiento:

Los *elementos de agrupamiento* son las partes organizativas de los modelos UML. Estos son los elementos en que puede descomponerse un modelo. En total, hay un elemento de agrupamiento principal, el paquete.

Un paquete es un mecanismo de propósito general para organizar elementos en grupos. Los elementos estructurales, los de comportamiento, e incluso otros elementos pueden incluirse en un paquete. A diferencia de los componentes (los cuales existen en tiempo de ejecución) un paquete es puramente conceptual (significa que existe durante el tiempo de desarrollo). Gráficamente un paquete se representa como una carpeta incluyendo normalmente sólo su nombre y, en

ocasiones, su contenido como se muestra en la Figura 3.12. Este ejemplo representa el paquete que contiene a todas las clases que forman la gramática del Parser.

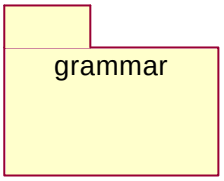


Figura 3.12. Notación en UML para un paquete

Los paquetes son los elementos de agrupamiento básicos con los cuales se puede organizar un modelo de UML. Hay variaciones, tal como los Frameworks, los modelos y los subsistemas (tipos de paquetes).

Elementos de anotación:

Los elementos de anotación son las partes explicativas de los modelos de UML. Son comentarios que se pueden aplicar para describir, clarificar y hacer observaciones sobre cualquier elemento de un modelo. Existe únicamente un elemento de anotación llamado nota. Una nota es simplemente un símbolo para representar las limitaciones y comentarios asociados a un elemento o una colección de elementos. Gráficamente una nota se representa con un rectángulo con una esquina doblada, junto con un comentario textual o gráfico, tal y como se muestra en la Figura 3.13. Este ejemplo representa una nota que contiene dos elementos gráficos con dos comentarios textuales asociados respectivamente.

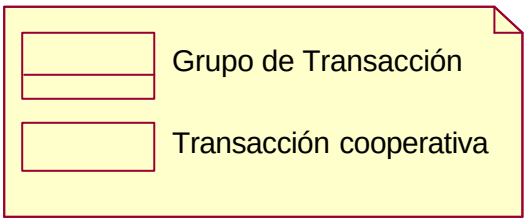


Figura 3.13. Representación en UML de una nota

3.3.2 Relaciones

Hay cuatro tipos de relaciones en UML.

1. Dependencia
2. Asociación
3. Generalización
4. Realización

Una **dependencia** es una relación semántica entre dos elementos, en la cual, un cambio a un elemento (el independiente) puede afectar a la semántica del otro elemento (el dependiente). Gráficamente una dependencia se representa con una línea discontinua, posiblemente dirigida, que incluye a veces una etiqueta, como se muestra en la Figura 3.14.

Figura 3.14. Notación en UML
para una relación de dependencia

Una **asociación** es una relación estructural que describe un conjunto de enlaces. Los cuales son una conexión entre objetos. La agregación es un tipo especial de asociación que representa una relación estructural entre un todo y sus partes. Gráficamente, una asociación es representada con una línea continua, posiblemente dirigida, que ocasionalmente incluye una etiqueta y frecuentemente contiene otros elementos tales como la multiplicidad y los nombres de rol, como en la Figura 3.15.

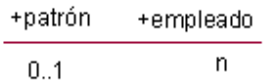


Figura 3.15. Notación en UML
para una relación de asociación

Una **generalización** es una relación especialización-generalización en la cual los objetos del elemento especializado (el hijo) son sustituidos por elementos del elemento generalizado (el padre). De esta forma, el hijo comparte la estructura y función del padre. Una generalización es equivalente a una relación de herencia. Una relación de generalización se representa gráficamente con una

línea sólida con una flecha vacía hacia el padre, como en la Figura 3.16.

Figura 3.16. Notación en UML
para una relación de generalización

Una **realización** es una relación semántica entre clasificadores, en donde un clasificador especifica un contrato que otro clasificador garantiza que cumplirá. Un clasificador es un mecanismo que describe características estructurales y de comportamiento. Los clasificadores pueden ser clases, interfaces, tipos de datos, señales, componentes, nodos, casos de uso y subsistemas. Las relaciones de realización se encuentran en dos sitios: entre interfaces y las clases o componentes que las realizan y entre casos de uso y las colaboraciones que las realizan. Gráficamente una relación de realización se representa por un híbrido entre una relación de generalización y una de dependencia, como en la Figura 3.17.

Figura 3.17. Notación en UML
para una relación de realización

3.3.3 Diagramas

Un diagrama es la representación gráfica de un conjunto de elementos, visualizado la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones). Los diagramas se utilizan para visualizar un sistema desde diferentes perspectivas. Así, un diagrama es una proyección de un sistema. Un diagrama representa un panorama de los elementos que integran un sistema. Los mismos elementos pueden aparecer en todos los diagramas, sólo en una parte de los diagramas o en ninguno (un caso muy raro). En teoría un diagrama puede contener alguna combinación de objetos y relaciones. UML incluye nueve tipos de diagramas:

1. Diagrama de clases
2. Diagrama de objetos
3. Diagrama de casos de uso
4. Diagrama de secuencia
5. Diagrama de colaboración
6. Diagrama de estados
7. Diagrama de actividades
8. Diagrama de componentes
9. Diagrama de despliegue

Un *diagrama de clases* muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Estos diagramas son los más comunes en el modelado de sistemas orientados a objetos. Cubren la vista de diseño estática del sistema.

Un *diagrama de objeto*s muestra un conjunto de objetos y sus relaciones. Representan instancias de los objetos encontrados dentro de los diagramas de clases. Estos diagramas cubren la vista de diseño estática o vista de proceso estática de un sistema tal como los diagramas de clase pero desde la perspectiva de casos reales o prototípica.

Un *diagrama de casos de uso* muestra un conjunto de casos de uso y actores (un tipo especial de clases) y sus relaciones. Un diagrama de casos de uso cubre la vista de casos de uso estática de un sistema. Estos diagramas son especialmente importantes en el modelado y organización del comportamiento del sistema.

Tanto los diagramas de secuencia como los de colaboración son un tipo de diagramas de interacción. Un *diagrama de interacción* muestra una interacción, que consta de un conjunto de objetos y sus relaciones, incluyendo los mensajes que pueden ser enviados entre ellos. Los diagramas de interacción cubren la vista dinámica de un sistema. *Un diagrama de secuencia* es un diagrama de interacción que resalta un ordenamiento en el tiempo de los mensajes; *un diagrama de colaboración* es un diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben mensajes. Los diagramas de secuencia y colaboración son isomorfos, es decir, se puede tomar uno y transformarlo en el otro.

Un *diagrama de estados* muestra una máquina de estados que consiste de estados, transiciones, eventos y actividades. Estos cubren la vista dinámica del sistema. Son importantes

para modelar el comportamiento de una interfaz, una clase o una colaboración y resaltan el comportamiento dirigido por eventos de un objeto, lo cual es especialmente útil en el modelado de sistemas reactivos.

Un *diagrama de actividad* es un tipo especial de diagrama de estados que muestra el flujo de actividades dentro de un sistema. Cubren la vista dinámica de un sistema. Son importantes para modelar la función de un sistema y enfatizan el flujo de control de los objetos.

Un *diagrama de componentes* muestra la organización y las dependencias entre un conjunto de componentes. Estos diagramas cubren la vista de implementación estática de un sistema. Se relacionan con los diagramas de clases en que un componente se corresponde, por lo común, con una o más clases, interfaces y colaboraciones.

Un *diagrama de despliegue* muestra la configuración de nodos de procesamiento a tiempo de ejecución y los componentes que residen en ellos. Cubren la vista de despliegue estática de una arquitectura. Se relacionan con los diagramas de componentes, en que un nodo incluye, por lo común, uno o más componentes.

Estos diagramas constituyen el medio principal para modelar y documentar los diferentes componentes de un sistema de software orientado a objetos. Por ejemplo, para modelar y documentar los patrones de diseño se utilizan los diagramas de clases y los diagramas de objetos.

3.4 Patrones de Diseño

El diseño de software orientado a objetos es una tarea difícil y el diseño de software orientado a objetos reutilizable es aún más difícil. Se tienen que encontrar los objetos adecuados con las clases correspondientes y con la granularidad correcta, posteriormente se tiene que determinar la interfaz de cada clase y las jerarquías de herencia, por último se tienen que especificar las relaciones existentes entre las clases. Un diseño debe ser acorde al problema que se debe resolver y lo suficientemente general para resolver problemas y requerimientos futuros.

Gran parte del éxito de los diseñadores expertos, en los sistemas orientados a objetos, radica en la aplicación de sus experiencias pasadas en el diseño de sistemas nuevos. Es muy común encontrar patrones de clases e interacciones de objetos recurrentemente en muchos sistemas orientados a objetos. Estos patrones resuelven problemas de diseño específicos y hacen que los diseños orientados a objetos sean más flexibles, elegantes y reutilizables. Un diseñador que esté familiarizado con estos patrones puede aplicarlos inmediatamente para resolver problemas nuevos evitando perder tiempo en encontrar una solución nueva.

Los patrones de diseño se aplican análogamente en muchas áreas del conocimiento (Cine, Música, Literatura, etc.) de hecho, éstos se originan de la teoría arquitectónica de Christopher Alexander [AIS77]. Aunque esta teoría se refiere a la arquitectura, se aplica igual al diseño orientado a objetos. Cada patrón describe la solución a un problema que ocurre una y otra vez, por lo que dicha solución puede ser usada muchas veces.

Cada patrón de diseño sistemáticamente nombra, explica y evalúa un diseño importante y recurrente en muchos sistemas orientados a objetos. Resulta de mucha utilidad, documentar las experiencias de diseño de tal forma que los diseñadores de sistemas puedan utilizarlas efectivamente. En [GHJV94] aparece un catálogo con 23 patrones de diseño utilizados en una gran cantidad de sistemas de software orientados a objetos. Cada uno de estos patrones de diseño se encuentra perfectamente documentado.

En general un patrón tiene 4 elementos:

1. **El Nombre del patrón**. Este nombre pasa a formar parte del vocabulario del diseño, lo que permite diseñar a un nivel mayor de abstracción.

2. **El problema** describe cuando aplicar el patrón. A veces incluye una lista de condiciones que deben cumplirse para que tenga sentido aplicar el patrón.

3. **La solución** describe los elementos que dan forma al diseño, sus relaciones, responsabilidades y colaboraciones.

4. **Las consecuencias** describen el resultado y los riesgos de aplicar el patrón. Son importantes para evaluar las

alternativas de diseño y para entender los costos y beneficios de aplicar el patrón.

Los patrones de diseño promueven y facilitan el reuso de arquitecturas y diseños exitosos. Mejoran la documentación y el mantenimiento de los sistemas existentes al incorporar una especificación explícita de las interacciones entre las clases y entre los objetos con su respectiva intención. Ayudan a obtener un diseño correcto más rápidamente. Identifican las clases e instancias, sus roles y colaboraciones y la distribución de responsabilidades para la solución de un problema específico de diseño orientado a objetos.

3.5 Discusión

UML es un lenguaje estándar para visualizar, especificar, construir y documentar los componentes de un sistema de software orientado a objetos. Esto se logra, en su mayoría, mediante la utilización de los diferentes diagramas incorporados en este lenguaje. Por tal motivo, los capítulos subsecuentes incorporan algunos de estos diagramas.

Los modelos de UML pueden conectarse directamente a una gran variedad de lenguajes de programación. Esto significa que dichos modelos se pueden transformar a algunos lenguajes de programación.

La herencia, el polimorfismo y los patrones de diseño se modelan mediante los diagramas de clases.

Todos los ambientes de CASE descritos en el capítulo 2, utilizan como lenguaje de modelado UML. Algunos de ellos generan código y realizan ingeniería inversa en varios lenguajes de programación.

4 DESCRIPCIÓN DEL SISTEMA DE TRANSACCIONES COOPERATIVAS

Este trabajo se basa en el modelo de transacciones cooperativas explicado resumidamente en la sección 2.5.6. En éste capítulo se ilustrará cómo aplicar los fundamentos teóricos del modelo, mediante un ejemplo orientado a un proyecto de desarrollo de software.

4.1 Introducción

Suponer que una empresa desea desarrollar un sistema de software. El proyecto se divide en cuatro partes importantes: Especificar los requerimientos del sistema, Análisis, Diseño y Desarrollo (por simplicidad no se muestran todas las partes que involucra el Proceso de Desarrollo de Software). Las divisiones naturales también ocurren con el Análisis y el Diseño. El desarrollo se divide a su vez en tres bloques: dos bloques de programación realizados por los programadores Oscar y Roberto, un bloque de Pruebas sobre los módulos programados por Oscar y Roberto. Lo que debe observarse aquí es que las divisiones naturales no separan el problema en problemas independientes. Persistirán numerosas restricciones a través de las divisiones. La Figura 4.1. muestra un ejemplo de la división del problema mencionado.

Se pueden hacer ahora algunas observaciones sobre las propiedades de las transacciones del modelo y del impacto que tienen sobre el sistema de control de transacciones de la base de datos que utilizan. En la figura 4.2. se muestra un cuadro comparativo de las transacciones tradicionales y las transacciones cooperativas.

En primer lugar, las transacciones cooperativas manipulan los datos que son a menudo complejos e intrínsicamente conectados por numerosas restricciones de consistencia. Debido a la complejidad mencionada, las bases de datos orientadas a objetos se eligen usualmente para estas aplicaciones.

En segundo lugar, las transacciones cooperativas tienden a ser muy largas. Un proyecto de desarrollo de software puede tomar una cantidad de tiempo significativa. En el modelo de transacciones tradicionales, la falla de transacciones se maneja con la atomicidad. La duración larga de las

transacciones cooperativas implica que se pagará un precio muy alto para hacer cumplir esta propiedad. Una cantidad significativa de trabajo tiene que ser hecha de nuevo, lo cual, no puede ser siempre posible. Esto implica que el mecanismo de recuperación tiene que ser de gran alcance y bastante versátil para deshacer parcialmente transacciones mientras que preserva la consistencia de la base de datos.

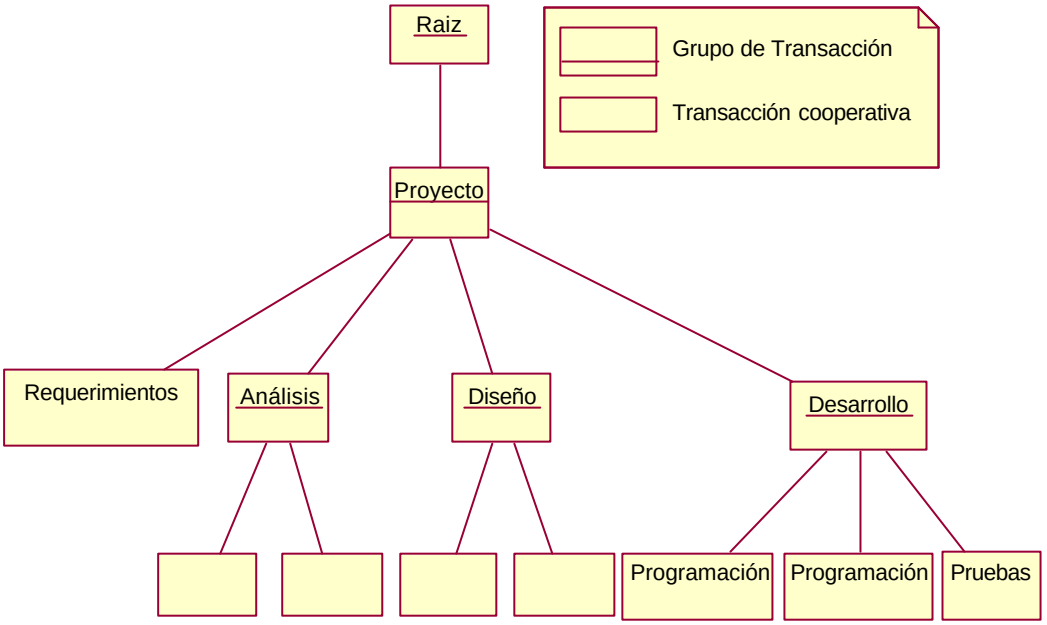


Figura 4.1. Jerarquía de Transacciones

	Tradicional	Cooperativa
Datos con que trabaja	simples	complejos
Duración	corta	larga
Atomicidad	atómica	puede no ser atómica
Anidamiento	atómico	cooperativo
Recuperación	recupera deshaciendo	deshace parcialmente con capacidad de ayuda
Interacción	nula	frecuente

Figura 4.2. Transacciones tradicionales y cooperativas

En tercer lugar, las tareas del proceso de desarrollo de software se dividen a menudo en jerarquías, con la característica de que los hermanos en un nivel dado pueden interactuar y restringirse uno a otro su comportamiento.

Las transacciones en este ambiente necesitan cooperar algunas veces. Esto significa que pueden compartir los datos asociados a su tarea. En estos casos, la serializabilidad no es un criterio apropiado de consistencia. La consistencia tiende a ser intrínsecamente dependiente de la aplicación. Por lo tanto, se debe proveer un mecanismo para controlar la interacción entre las transacciones. La serialización puede ser útil en los casos donde la cooperación no es necesaria. Este mecanismo del control debe ser bastante flexible para permitir que los participantes del proceso de desarrollo de software especifiquen fácilmente sus criterios de consistencia.

4.2 Características del Modelo de Transacciones Cooperativas

4.2.1 Organización Jerárquica de Transacciones

Existen buenas razones para organizar jerárquicamente las transacciones de un ambiente CASE. En primer lugar, esta jerarquía reflejará una división natural del trabajo común. Los detalles de una subtarea se pueden aislar de los padres y hermanos. Una subtransacción puede ser la abstracción de una subtarea, facilitando al manejo global. En segundo lugar, la estructura jerárquica se puede explotar para permitir que las transacciones que pertenecen a la misma transacción padre se ejecuten concurrentemente, acortando el tiempo de ejecución percibida de la transacción padre. En tercer lugar, las transacciones jerarquizadas ayudan a tener bajo control la recuperación. Cuando una transacción falla, tenemos que deshacer solamente sus efectos. Esto protege con eficacia los hermanos. Una organización plana afectaría todo el trabajo, relacionado o no relacionado.

En general, no es posible determinar de antemano el anidamiento máximo que un problema puede tener ni las ramificaciones de una transacción. Por lo tanto, se debe permitir que una transacción se pueda dividir en dos o más transacciones al presentarse la necesidad. Aún cuando algunas partes de la transacción hayan hecho ya un cierto trabajo. Por lo tanto, la jerarquía utilizada debe ser dinámica.

La Figura 4.1 representa una división inicial posible en un proyecto de desarrollo de software. Se observa que la

jerarquía facilita la abstracción y el control de las tareas. Es deseable una jerarquía dinámica. Por ejemplo, el director del proyecto podría sugerir la división de la transacción "Análisis de requerimientos" por un grupo de transacciones con diferentes miembros y en donde cada miembro realiza una tarea. Se necesitaría entonces la capacidad de abortar la transacción cooperativa "Análisis de requerimientos" y en su lugar crear un grupo de transacciones.

4.2.2 Criterios de Consistencia

En el dominio de aplicaciones cooperativas, no existe un criterio de consistencia simple, como la consistencia global y la atomicidad, que soporte la cooperación. En lugar de esto, la noción de lo que es correcto puede variar de una aplicación a otra y de una tarea a otra. Por lo tanto, se tienen que permitir criterios de consistencia especificados por el usuario.

Los criterios de consistencia de un grupo de transacciones (en adelante GT) describen sus historias válidas. La historia del GT en turno se refleja en la historia de su padre.

Se desea que los criterios de consistencia que pertenecen a los miembros de un GT se apliquen de acuerdo a su localización. Por ejemplo, un GT no decide la consistencia de sus nietos. Sin embargo, debe decidir la consistencia de sus hijos, los cuales deciden a su vez esta para sus hijos y así sucesivamente. Con las especificaciones por localización, podemos decir que la historia de un GT es correcta si satisface sus propios criterios y las historias son respectivamente correctas para cada uno de sus hijos.

El control de concurrencia entre los miembros de la jerarquía de la transacción requiere que cada GT haga cumplir sus criterios de consistencia especificados por el usuario. Los criterios de consistencia se definen en términos de un **lenguaje**. El alfabeto del lenguaje consiste de invocaciones de operaciones realizadas por los miembros del GT. Se utilizan **gramáticas** para definir cómo ordenar las operaciones para formar oraciones o historias correctas.

Se debe realizar en línea la verificación del cumplimiento de los criterios de consistencia. Es decir, se informa al miembro (GT o TC) inmediatamente cuando se somete una operación inválida. Esto significa que el algoritmo utilizado

para garantizar el cumplimiento de los criterios de consistencia no solamente debe reconocer historias correctas, también debe ser capaz de reconocer prefijos válidos de historias correctas. Una especificación tiene la *propiedad de prefijo viable* si se pueden reconocer todos los prefijos válidos de sus miembros en línea.

4.2.3 Sistema Multicopia

Con este enfoque, se tienen que manejar versiones, es decir; se crean objetos nuevos cuando se crean las transacciones, los objetos se combinan cuando hay commit y se borran cuando hay abortos.

Cada GT mantiene copias privadas de los objetos que son utilizados por sus miembros. El GT adquiere la copia de un objeto la primera vez que uno de sus miembros lo solicita. Así, el GT proyecta un ambiente completo, una base de datos virtual a cada uno de sus hijos. Las copias de los objetos que pertenecen al GT son accedidas y modificadas por las operaciones de sus miembros, y se presentan al padre del GT cuando el GT desea hacer commit. Si la transacción principal hace commit, los cambios a la base de datos principal son realizados por el GT raíz que está en el tope de la jerarquía.

4.2.4 Recuperación Basada en Operación

El proceso de desarrollo de software implica inherentemente iteración y errores. Después de que un participante del proyecto de desarrollo de software haya terminado una parte de una tarea, puede decidir qué parte de lo que ha hecho no es correcto. Es esencial que cualquier transacción prevea deshacer los cambios realizados a la base de datos.

Deshacer los cambios en la base de datos de un ambiente CASE está acompañado de complicaciones. Las transacciones mencionadas son largas, ampliables, e interactúan entre sí. Deshacer una transacción entera implica deshacer mucho trabajo, algo del cual se desearía preservar. Además, el aborto puede conectar en cascada a otras transacciones mientras se permita interacción. Así, es más apropiado manejar **deshace** (undo) y restaura a nivel de la operación; quitando los efectos de la operación inválida así como las operaciones que dependen de ella. Esto puede afectar muchas

transacciones, pero cada transacción se afecta lo menos posible.

Este esquema de recuperación implica que necesitamos tener la capacidad para determinar correctamente qué operaciones son afectadas por las fallas de una operación (dependencias). Se necesitan mantener estas dependencias en el registro para su uso posterior durante la recuperación.

A veces, una transacción no desea que los efectos de una operación inválida sean quitados totalmente de la base de datos. Puesto que ciertas partes de esa operación pueden todavía ser válidas. Por ejemplo, una edición grande puede estar parcialmente correcta, aunque un pedazo de ésta depende de una operación que no sea válida. Por lo tanto, en vez de borrar totalmente los efectos de las operaciones inválidas, es mejor permitir que los miembros del GT puedan clasificar los cambios, suprimiendo los que deban ser suprimidos definitivamente y modificando los archivos que han sido afectados por la anulación. Las operaciones asociadas al trabajo que debe ser preservado se pueden someter otra vez de una manera correcta.

4.3 Consistencia

Los criterios de consistencia correspondientes a un GT se representan por medio de patrones y conflictos. Estos constituyen el mecanismo esencial para garantizar la consistencia de la base de datos correspondiente al GT.

4.3.1 Patrones y Conflictos

Los patrones y conflictos reflejan de manera natural los criterios de consistencia de las transacciones cooperativas. El usuario usa el conocimiento semántico que tiene sobre las tareas realizadas por el GT, para definir los criterios de consistencia apropiados a la tarea del grupo. Estos criterios de consistencia restringen el orden en que se someten las operaciones al GT para asegurarse de que mantienen alguna forma de consistencia en la base de datos.

Los patrones especifican la intercalación de operaciones permitidas en un GT. Por ejemplo, un patrón para el GT

Desarrollo puede indicar, "Si uno de los programadores cambia la Interfaz Gráfica de Usuario (*IGU)*, la transacción de pruebas debe verificar que la *IGU* no presente errores de ejecución".

Los conflictos especifican la intercalación de operaciones prohibidas. Un conflicto puede indicar que, si uno de los programadores no ha visto la última versión de la *IGU*, entonces no la puede modificar porque puede sobrescribir los cambios realizados por alguien más. Con transacciones atómicas, los candados (locks) son el mecanismo principal del conflicto. Por ejemplo, un candado en escritura garantiza que ninguna otra transacción puede tener acceso a un objeto en el tiempo que esté habilitado el candado.

Cada GT tiene un conjunto activo de patrones y conflictos. Estos restringen en su totalidad el orden permisible de operaciones en la historia del GT y se representan por gramáticas libres de contexto o autómatas de pila.

4.3.2 Gramáticas Libres de Contexto y Autómatas de Pila

Una gramática libre de contexto (**GLC**) es un cuarteto (N,T,S,R), donde N es un conjunto finito de símbolos no terminales, T es un conjunto finito de símbolos terminales o alfabeto, S es el símbolo de inicio que pertenece a N y R es un conjunto finito de reglas de producción en donde cada regla consta de un símbolo no terminal de lado izquierdo y una combinación de símbolos terminales y no terminales del lado derecho. Una producción ε, especifica que no existen símbolos a la derecha de la regla.

Una cadena en la gramática es generada por una derivación. Una derivación comienza con el símbolo de inicio. Utiliza las reglas de producción para sustituir los símbolos no terminales por sus combinaciones asociadas, y continua hasta que la combinación sólo contiene símbolos terminales. Una derivación por la derecha es una derivación en la cual, el símbolo no terminal de la derecha se substituye en cada paso.

Los puntos se utilizan para no perder de vista la posición actual en una regla durante una derivación. Un elemento para una GLC dada es una producción con un punto en cualquier lugar de la parte derecha, incluyendo el principio o el fin. En el caso de una producción ε, B -> ε, B->. es un elemento.

Un elemento está completo si el punto es su símbolo más a la derecha.

Una forma de oración es una combinación de símbolos terminales y no terminales que son derivables del símbolo de inicio. Una forma de oración derecha es derivable con una derivación de derecha.

Las gramáticas LL(1) y LR(1) son un tipo especial de gramáticas libres de contexto que tienen la propiedad de prefijo viable. Intuitivamente, lo que esto significa es que en cualquier punto en el reconocimiento de la cadena de entrada, se puede determinar la acción correcta que se realizará mirando únicamente un símbolo en la entrada. La primer “L” representa (por left, en inglés, izquierda) el examen de la entrada de izquierda a derecha. En el caso de la Gramática LL(1) la segunda “L” representa una derivación por la izquierda, y el “1” es por utilizar anticipadamente un símbolo en la entrada y en cada paso, para tomar las decisiones de la acción en el análisis sintáctico. En la Gramática LR(1) la “R” representa (por right, en inglés, derecha) una derivación por la derecha.

Un autómata de pila es una máquina de estado finito aumentada con una pila. Además de que cambia de estado después de leer un símbolo de la entrada de información, el autómata de pila tiene la opción para meter un símbolo en la pila, sacar un símbolo de ella o no hacer nada. Un autómata de pila acepta una cadena cuando su control finito está en un estado final y la pila esta vacía.

Formalmente, un autómata de pila es un sexteto de la forma (E,Σ,Γ,T,I,F) donde:

- E es una colección finita de estados.
- Σ es el alfabeto de la máquina.
- Γ es la colección finita de símbolos de pila.
- T es una colección finita de transiciones.
- I (un elemento de E) es el estado inicial.
- F (un subconjunto de S) es la colección de estados de aceptación.

Un autómata de pila determinista (**APD**) es un autómata de pila en el cual T es una función de transición.

Es un resultado elemental de la teoría del lenguaje que cada autómata de pila tiene una gramática libre de contexto equivalente y viceversa. Mas detalles ver [ASU88][Bro93].

4.3.3 Historias Correctas en un Grupo de Transacciones

La especificación de consistencia en un GT es un conjunto activo de gramáticas de patrones y conflictos. Este conjunto varía con respecto al tiempo conforme los miembros se agregan o eliminan del GT, o conforme interactúen con diversos objetos.

Intuitivamente, la historia del GT está correcta si se conforma con todos los patrones y ninguno de los conflictos de la especificación de consistencia del GT. Es decir, una historia correcta se puede ver como una intercalación de patrones. Las operaciones en la historia que participan en un patrón son llamadas relevante para ese patrón.

Formalmente: Una **historia es correcta** si cada patrón que estuvo activo durante la historia satisface el criterio del patrón, y cada conflicto que estuvo activo durante la historia satisface el criterio del conflicto.

Criterio del Patrón.
Para la gramática del patrón P_i, sea H_i la porción de la historia que fue generada mientras que la gramática estuvo activa en el grupo de transacción. Sea S_i la secuencia de operaciones que constituyen la proyección de Hi de las operaciones relevantes para la gramática P_i. Si la secuencia S_i es una forma de oración en la gramática P_i, entonces P_i satisface el criterio del patrón.

Criterio del conflicto.
Para la gramática del conflicto C_j, sea H_j la porción de la historia que fue generada mientras que la gramática estuvo activa en el grupo de transacción. Sea S_j la secuencia de operaciones que constituyen la proyección de H_j de las operaciones relevantes para la gramática C_j. Si la secuencia S_j no es una forma de oración en la gramática C_j, entonces C_j satisface el criterio del conflicto

4.3.4 Ejemplo

Este ejemplo ilustra las especificaciones de consistencia para el GT *Desarrollo* y una historia que es correcta para ese GT (ver Figura 4.1). Como se mostró previamente, *IGU* es el programa de la interfaz gráfica de usuario y *Reporte* es un informe que contiene el detalle de los errores de ejecución encontrados por la transacción cooperativa *Pruebas*; *cont_pru* es un contador de los problemas reportados en la ejecución de los programas.

La Figura 4.3 muestra las especificaciones de consistencia para el GT *Desarrollo* de dos maneras. La primera está como una GLC, los símbolos no terminales están representados con las letras A, B y S, el símbolo de inicio es S. Los símbolos terminales son operaciones de la forma <M, o, O>, donde ***M*** es un miembro, ***o*** es un operador y ***O*** es un identificador de objeto. La segunda representación está como autómata de pila y se usa la notación de UML para modelar las máquinas de estados.

La primera especificación de consistencia está representada por las gramáticas del patrón P_1 y el Conflicto C_1. La semántica está dada por la frase "si alguien modifica la interfaz gráfica de usuario se deben realizar las pruebas necesarias para encontrar errores de ejecución y se debe generar un reporte con los resultados de las pruebas realizadas" dicho de otro modo, si alguien modifica el programa *IGU*, la Transacción Cooperativa *Pruebas* deberá leer y actualizar el *Reporte* para dejar un registro de las pruebas que se realizaron.

La segunda especificación de consistencia está representada por la gramática del patrón P_2. Esta se utiliza para garantizar que todos los problemas reportados, referentes a programas diferentes, deben estar solucionados antes de que el sistema de software sea liberado. De acuerdo a lo mostrado en la gramática de patrón P_2 de la Figura 4.3., se observa del autómata de pila, que solamente las historias que tienen un número igual de incrementos y decrementos para el contador *cont_pru* están completas para esta gramática.

La última especificación de consistencia está representada por las gramáticas del patrón P_s y el Conflicto C_s. La semántica está dada por la frase "si varias transacciones editan el mismo objeto entonces, la primer transacción que escriba el objeto obliga a las demás transacciones a leer

nuevamente el objeto". Las gramáticas P_s y C_s son plantillas que se utilizan para construir dinámicamente los patrones y conflictos requeridos para manejar la sincronización. La siguiente sección ilustra con mayor detalle la sincronización y la funcionalidad de estas gramáticas.

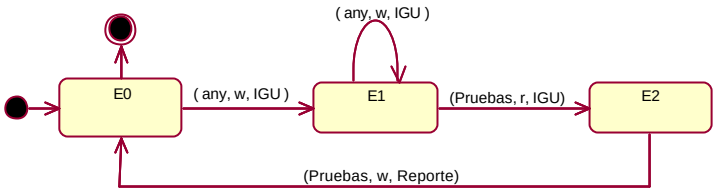
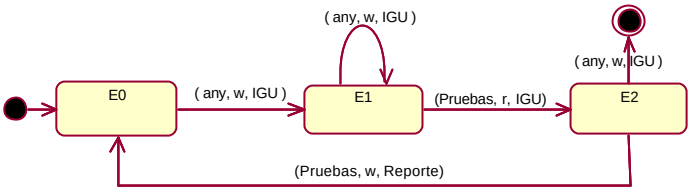
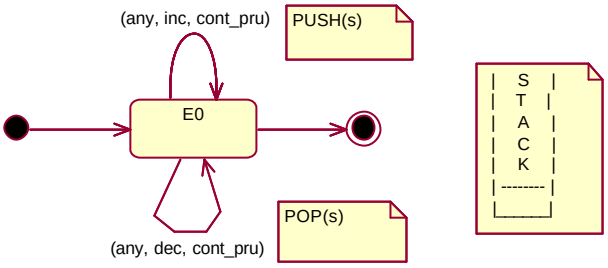
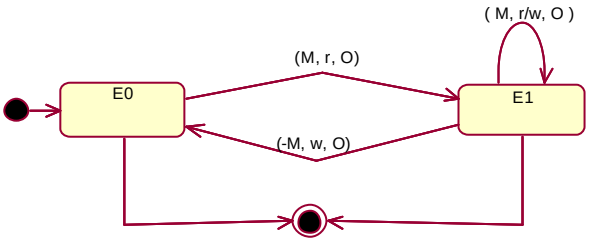
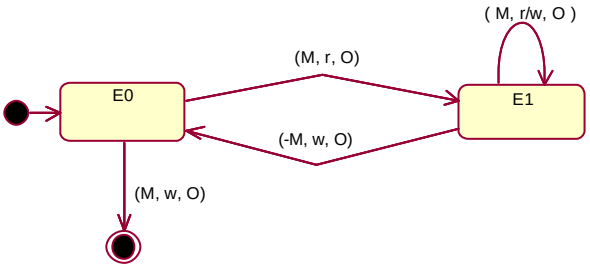
P_1: S::=(*any*,*w*,*IGU*)A|∈ A::=(*any*,*w*,*IGU*)A|(*Pruebas*,*r*,*IGU*)B B::=(*Pruebas*,w,*Reporte*)S	
C_1: S::=(*any*,*w*,*IGU*)A A::=(*any*,*w*,*IGU*)A|(*Pruebas*,*r*,*IGU*)B B::=(*any*,*w*,*IGU*)|(*Pruebas*,w,*Reporte*)S	
P_2: S::=B B::=AB|A A::=(*any*,*inc*,*cont_pru*)B(*any*,*dec*,*cont_pru*)| (*any*,*inc*,*cont_pru*)(*any*,*dec*,*cont_pru*)	
P_s: S::=(M, *r*, O)A|∈ A::=(-M, *w*, O)S|(M, *r*/*w*, O)A|∈	
C_s: S::=(M, *w*, O)|(M, *r*, O)A A::=(-M, *w*, O)S|(M, *r*/*w*, O)A	

Figura 4.3. Ejemplos de Patrones y Conflictos como gramáticas y como máquinas de estados.

4.4 Sincronización

El GT puede especificar un protocolo que coordine las interacciones de sus miembros con los objetos para asegurar que estas interacciones dejen a los datos en un estado consistente. Este protocolo puede ser serializabilidad u otra cosa. Se puede usar un conjunto especial de gramáticas denominadas ***gramáticas de sincronización***. Cada gramática de sincronización asegura que únicamente un miembro interactúe con un objeto, de una manera que preserve la consistencia del objeto dentro del GT. Una de tales gramáticas tomará lugar por cada par *<Miembro, Objeto>*. Todas estas gramáticas tienen una estructura idéntica, debido a que el GT mantiene una noción uniforme de consistencia para todos sus objetos.Esto se puede implementar definiendo, para cada GT, una plantilla que se utilice para construir dinámicamente las gramáticas de sincronización, manteniendo a **M** y **O** como variables que se sustituirán con los identificadores del miembro y del objeto, respectivamente. Cuando un miembro lee por primera vez un objeto, del conjunto de su grupo, se crea una instancia de las gramáticas del patrón y del conflicto para el protocolo de la sincronización del grupo. Es decir, se construye una copia de cada plantilla de la gramática, substituyendo **M** por el identificador del miembro y **O** con el identificador del objeto, y agregando las dos nuevas gramáticas al conjunto de gramáticas utilizadas para definir la consistencia del grupo.

En este caso, es deseable que los miembros del grupo *Desarrollo* no sobrescriban los cambios realizados por otros. La Figura 4.3. muestra las plantillas de la gramática de sincronización (patrón P_S y conflicto C_S) para el grupo *Desarrollo*. La Figura 4.4 muestra P_3 y C_3, las gramáticas de sincronización para las interacciones entre la TC *Oscar* y el objeto *IGU*. Examinando minuciosamente la gramática P_3, se observa que una vez que la TC *Oscar* lea el objeto *IGU*, puede hacer cualquier cosa sobre éste hasta que alguna otra TC escriba una nueva versión. En ése momento, la TC *Oscar* debe leer la nueva versión antes de que pueda continuar trabajando con el objeto.

Se construyen también las instancias P_4 y C_4 para que sean gramáticas de patrón y de conflicto en la sincronización con **M** = *Pruebas* y **O** = *IGU*, y con las plantillas P_S y C_S de la gramática de sincronización. Semejantemente, P_5 y C_5 son las gramáticas de sincronización del patrón y del conflicto con **M** = *Pruebas* y **O** = *Reporte*.

P_3: S::=(*Oscar, r, IGU)*A|∈ A::=(-*Oscar, w, IGU*)S|(*Oscar*, *r/w, IGU*)A|∈
C_3: S::=(*Oscar, w, IGU)*|(*Oscar, r, IGU*)A A::=(-*Oscar, w, IGU*)S|(*Oscar*, *r/w, IGU*)A

Figura 4.4 Gramáticas de sincronización de Patrón y Conflicto con **M**=*Oscar* y **O**=*IGU*

La Figura 4.5 muestra una historia correcta para el GT *Desarrollo*, de acuerdo a la especificación de consistencia del ejemplo. En esta figura, se indica qué operaciones en la historia son relevantes a las gramáticas del patrón usando círculos. Los círculos rellenos indican que la operación relevante también completa una oración en la gramática. Notar que, mientras que la primera operación <*Pruebas, w, Reporte*> completa una oración en la gramática de patrón P_1, la tarea no está completa porque la TC *Pruebas* detectó un problema e incrementó *cont_pru*. Los errores detectados en *IGU* necesitan corregirse y posteriormente debe probarse nuevamente antes de que la tarea se termine.

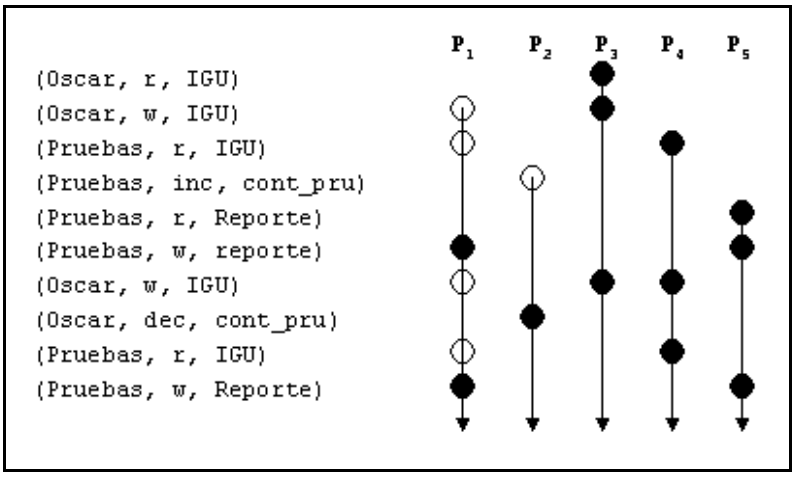


Figura 4.5 Ejemplo de una historia correcta.

4.4.1 Algoritmo

En un punto dado en la ejecución de un GT, el mecanismo de sincronización ha permitido la ejecución de algunas operaciones de su entrada y ha rechazado otras. Se denomina prefijo de la historia a la secuencia de operaciones que se han ejecutado. Cada gramática, definida en las especificaciones de consistencia, puede estar en una parte intermedia de una cierta derivación basada en las operaciones del prefijo de la historia. También, el mecanismo de la sincronización puede estar examinando una operación sometida para determinar si debe aceptarse o rechazarse.

Considerar la acción que se debe tomar ahora. Este proceso es más fácil de visualizar en términos de APDs. Por lo tanto, considerar la acción que se debe tomar cuando una operación se somete a un APD.

Si el APD corresponde a una gramática de un patrón, y la operación causa la transición a un estado muerto, la operación se rechaza. Por otra parte, si la operación causa una transición a un estado desde el cual es accesible un estado final, la operación se acepta. Si el estado actual en el APD no tiene ningún arco saliente etiquetado con la operación, la operación no afecta el APD y se ignora ésta. Probablemente, la operación afecte a otro APD.

Cuando el APD corresponde a una gramática de un conflicto y la operación causa una transición a un estado final, la operación se rechaza. Si la operación causa una transición a un estado por el cual es accesible un estado no final, la operación se acepta. Si no hay arco saliente etiquetado con la operación, se ignora ésta.

Una operación es relevante cuando provoca que el APD la acepte o la rechace. Una historia está completa con respecto a un conjunto de APDs si se cumple el siguiente criterio: después de que las operaciones en la historia se sometan en secuencia a los APDs, todos los APDs de patrón están en un estado final con la pila vacía, y todos los APDs de conflicto están en configuraciones no aceptadas.

El análisis con gramáticas corresponde al análisis con APDs. Las acciones realizadas son iguales. Es decir, una operación se puede rechazar, aceptar o ignorar. La aceptación o el rechazo de una operación, que causa una transición en el APD,

causa un cambio en el estado de análisis de la gramática correspondiente.

El algoritmo de la sincronización se puede resumir de la siguiente manera: una operación se acepta cuando ninguna de las gramáticas la rechazan. Si una operación se acepta, se debe actualizar la información del análisis para cada gramática.

El Apéndice A ilustra cómo trabaja la sincronización. El ejemplo muestra una secuencia de operaciones sometidas al GT *Desarrollo*. En este ejemplo, se utiliza la especificación de consistencia de la Figura 4.3. La Figura 4.5 muestra la historia aceptada por el mecanismo de sincronización.

4.4.2 Punto de Verificación

Cuando un miembro pone un punto de verificación y termina, es porque cree que ha terminado con éxito su tarea. Un miembro puede poner un punto de verificación (y terminar) solamente cuando todas las gramáticas relevantes a las operaciones que el miembro ha sometido están completas. Esto indica que las tareas en las que está implicado el miembro, dejan los objetos en la base de datos del GT en un estado consistente de acuerdo a la especificación de consistencia del GT.

P_1: S::=(*any,w,IGU*) A . A::=(*Pruebas,r,IGU*) B . B::=(*Pruebas,*w,*Reporte*) S . S::=(*any,w,IGU*) A . A::=(*Pruebas,r,IGU*) B . B::=(*Pruebas,*w,*Reporte*) S . S::=∈ . P_2: S::= B . B::= A . A::=(*any,inc,cont_pru*) (*any,dec,cont_pru*) .	P_3: S::=(*Oscar, r, IGU)* A . A::=(*Oscar, r/w, IGU*) A . A::=(*Oscar, r/w, IGU*) A . A::=∈ . P_4: S::=(*Pruebas, r, IGU)* A . A::=(-*Pruebas, w, IGU*) S . S::=(*Pruebas, r, IGU*) A . A::=∈ . P_5: S::=(*Pruebas, r, Reporte)* A . A::=(*Pruebas, r/w, Reporte*) A . A::=(*Pruebas, r/w, Reporte*) A . A::=∈ .

Figura 4.6 Gramáticas de patrón cuando las TCs *Oscar* y *Pruebas* colocan un punto de verificación.

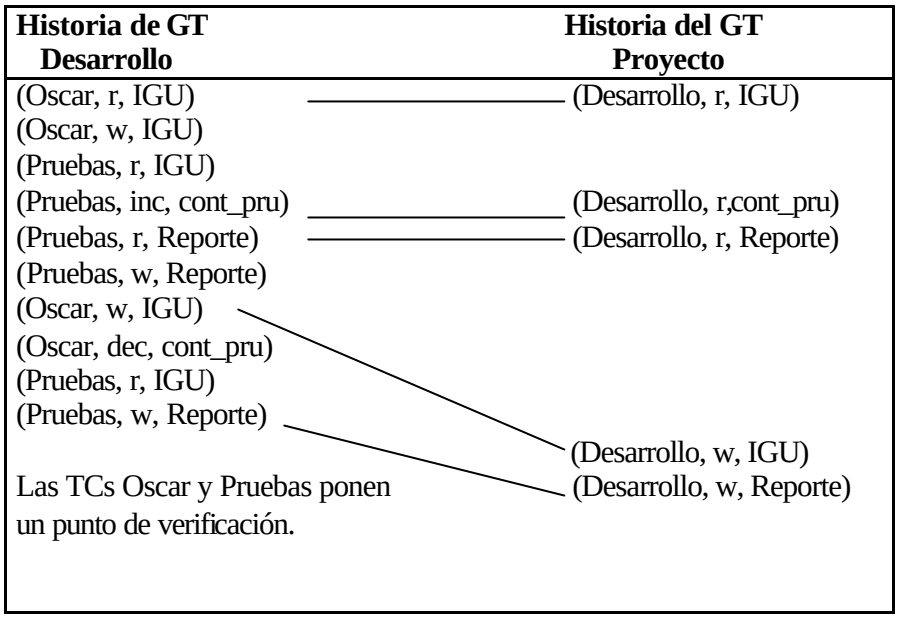


Figura 4.7 Relaciones entre historias.

La Figura 4.6 muestra las gramáticas de los patrones cuando las TCs *Oscar* y *Pruebas* ponen el punto de verificación después de la secuencia de operaciones del ejemplo. El punto de verificación hace que los efectos de esta secuencia de operaciones se propaguen al padre del GT; en este caso, el GT *Proyecto*. La Figura 4.7 muestra como se propagan las operaciones hacia el padre como resultado del punto de verificación, y cómo se relacionan con la historia del GT *Desarrollo*.

4.5 Recuperación

Como se explicó en la sección 4.2.4. se desea preservar tanto trabajo como sea posible cuando se abortan las operaciones realizadas sobre la base de datos. Por lo tanto, en vez de abortar transacciones enteras, se purgan de la base de datos únicamente los efectos de todas las operaciones que son directa o transitivamente afectadas por el aborto de una operación. Para hacer esto, se debe registrar tanto la historia de un grupo de transacciones como las dependencias entre las operaciones de la historia.

4.5.1 Mantenimiento de Dependencias y Registro

Para cada patrón definido en un GT, las *dependencias de patrón* se forman con las operaciones que participan en ese patrón. Puesto que los patrones definen ordenamientos de operaciones, cada operación en la secuencia relevante para algún patrón depende de la validez de la operación anterior en esa secuencia. Sin embargo, puesto que la secuencia de operaciones asociadas con el patrón está completamente ordenada en la historia, la definición de dependencia de patrón puede ser simplificada como sigue: Cada operación Op es dependiente de patrón, con respecto a la operación anterior, en cada patrón en donde participa Op.

Las *dependencias de lectura* ocurren por lo siguiente: Una operación de lectura realizada por el miembro **M,** sobre la versión de un objeto, está correcta si la operación que escribió la versión está también correcta. Si la operación de escritura es invalidada posteriormente, entonces la operación de lectura por parte de **M** se vuelve inválida.

Las *dependencias Padre-Hijo* ocurren entre las operaciones de los niveles adyacentes de la jerarquía de la transacción. Por ejemplo, cuando un miembro **M** de un GT lee un objeto, el GT debe tener una copia del objeto que le proporcionará a **M**. La lectura de **M** está correcta solamente si el GT leyó una versión correcta. Si esa versión se invalida más adelante como resultado de algún aborto, entonces la lectura de **M** se debe invalidar. Ocurre una situación similar con la escritura; cuando el GT escribe una versión sobre su padre, la validez de esa operación se basa en la validez de la última operación de escritura realizada por uno de sus miembros.

Las dependencias de cada operación quedan registradas en su correspondiente archivo de registro. De este modo se pueden calcular fácilmente las dependencias de patrón y de lectura/escritura. Las dependencias de patrón se procesan para cada gramática de patrón relevante a la operación que se ejecuta. Las dependencias de lectura/escritura pueden ser calculadas manteniendo en cada versión del objeto un indicador en su archivo de registro de la operación que escribió la versión, e incluir la información en el archivo de registro en cualquier operación que lea la versión. Las dependencias Padre-hijo señalan a otros archivos de registro. Por ejemplo, una operación de lectura señala hacia el archivo de registro del padre.

La Figura 4.8 muestra el archivo de registro para el ejemplo del GT *Desarrollo*. El archivo de registro incluye el registro de las secuencias de operaciones, las dependencias de patrones y las dependencias de lectura para cada operación.

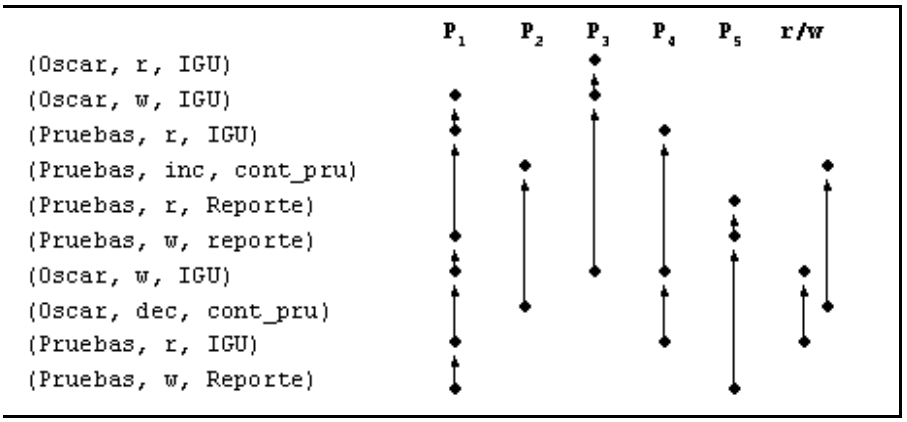


Figura 4.8 Contenido del archivo de registro.

La primer columna contiene las operaciones ejecutadas ordenadas cronológicamente. Las siguientes cinco columnas ilustran las dependencias asociadas a los patrones P_1 a P_5. Para un patrón, un círculo negro en el renglón indica que la operación es relevante para ese patrón. Una flecha entre dos puntos negros indica que la operación en la cola de la flecha depende de la operación en la cabeza de la flecha. La última columna indica las diferentes dependencias de lectura.

4.5.2 Algoritmo

Cuando se aborta una operación, la base de datos entra en una fase de anulación donde purga todos los efectos de esa operación. Esta fase procesa el archivo de registro cronológicamente, comenzando en la operación abortada, e invalidando cualquier operación que cumpla alguno de los criterios siguientes:

1. Fue abortada directamente por el miembro.
2. Es dependiente de alguna operación inválida, debido a una dependencia de patrón, una dependencia de lectura, o una dependencia padre-hijo.
3. Esta en conflicto. Es decir, debido a las diferencias entre la nueva historia y la vieja historia, ahora completa alguna gramática del conflicto.

Para el ejemplo de la Figura 4.8, suponer que el miembro *Oscar* decide abortar las operaciones (*Oscar,w,IGU*) y (*Oscar,dec,cont_pru*). Entonces la operación (*Pruebas,r,IGU*) deberá también ser abortada, debido a la dependencia del patrón P_4 y a la dependencia de lectura de la operación (*Oscar,w,IGU*). También, la operación (*Pruebas,w,Reporte*) debe ser abortada debido a la dependencia del patrón P_1.

Una vez que se hayan detectado las operaciones inválidas, todas las versiones de un objeto escritas por operaciones inválidas se purgan de la base de datos. También, el estado de análisis de cada gramática afectada se ajusta para reflejar el estado de análisis en la historia nueva. Así, en el ejemplo, P_1 se ajusta al estado de análisis en que se encontraba antes de reconocer la operación (*Oscar,w,IGU*). P_2 se ajusta al estado de análisis antes de reconocer la operación (*Oscar,dec,cont_pru*). El proceso es similar para las otras gramáticas de Patrón y de Conflicto pertenecientes a la especificación de consistencia.
Al final de la fase de anulación, cada miembro que ha tenido una operación inválida envía un mensaje de anulación, indicando cuáles de sus operaciones se han invalidado. En la fase de la recuperación, estos miembros utilizan los mensajes de anulación como punto de partida para recuperar su trabajo.

Durante la fase de recuperación, los miembros que han tenido operaciones inválidas tiene permitido realizar cualquiera de las cosas siguientes para recuperar su trabajo:

1. Leer nuevamente las versiones inválidas para determinar exactamente qué trabajo fue invalidado y decidir que partes del trabajo se deben preservar. Esto ayuda al miembro a determinar qué operaciones de compensación necesitan someterse.

2. Ejecutar operaciones de compensación. Estas se deben validar por el mecanismo de sincronización, de acuerdo al estado de análisis actual de las gramáticas de patrón y conflicto. Las operaciones de recuperación permiten que el miembro mantenga los cambios que pudieron haber sido válidos, mientras que quita cambios inválidos.

3. Abortar cualquier operación que no tenga un punto de verificación y emprender acciones de recuperación. Esta opción es útil en la emulación de muchos procedimientos de aborto tradicionales.

4.6 Arquitectura General del Sistema de Transacciones Cooperativas

La Figura 4.9 muestra la arquitectura general del Sistema de Transacciones Cooperativas. Este sistema se compone de tres módulos: el especificador de criterios de consistencia, el verificador de consistencia y el monitor de transacciones.

El especificador de criterios de consistencia es un sistema utilizado por un GT para especificar las reglas de consistencia que deberán satisfacer las operaciones que sometan los miembros del GT sobre la base de datos correspondiente al GT. Este sistema generará una representación interna de los criterios de consistencia para que puedan ser utilizados por cualquier otro sistema.

El verificador de consistencia es un sistema para validar cada una de las operaciones sobre la base de datos correspondiente a un GT que deseen realizar los miembros de dicho GT. Este sistema utiliza la representación interna de los criterios de consistencia generada previamente por el especificador de criterios de consistencia. Además, el verificador de consistencia coordina las interacciones de los miembros del GT utilizando las gramáticas de sincronización.

El monitor de transacciones es un sistema que se encarga de coordinar la ejecución concurrente de las transacciones cooperativas pertenecientes a un GT. Este sistema le transfiere el control al verificador de consistencia cuando una transacción cooperativa desea realizar una operación sobre la base de datos. También realiza las actividades de recuperación manejando un archivo de registro y manteniendo las dependencias entre las operaciones ejecutadas sobre la base de datos.

Como puede observarse en la figura 4.9, deberá existir una interfaz que permite la comunicación entre el sistema de transacciones cooperativas (vía el monitor de transacciones) y una herramienta CASE que soporte el trabajo cooperativo.

La Figura 4.10 muestra la arquitectura del sistema objeto de esta tesis denominado “Especificador de Criterios de Consistencia para un Sistema de Transacciones Cooperativas”. Los principales componentes del sistema son: Lenguaje orientado a especificar criterios de consistencia e Intérprete.

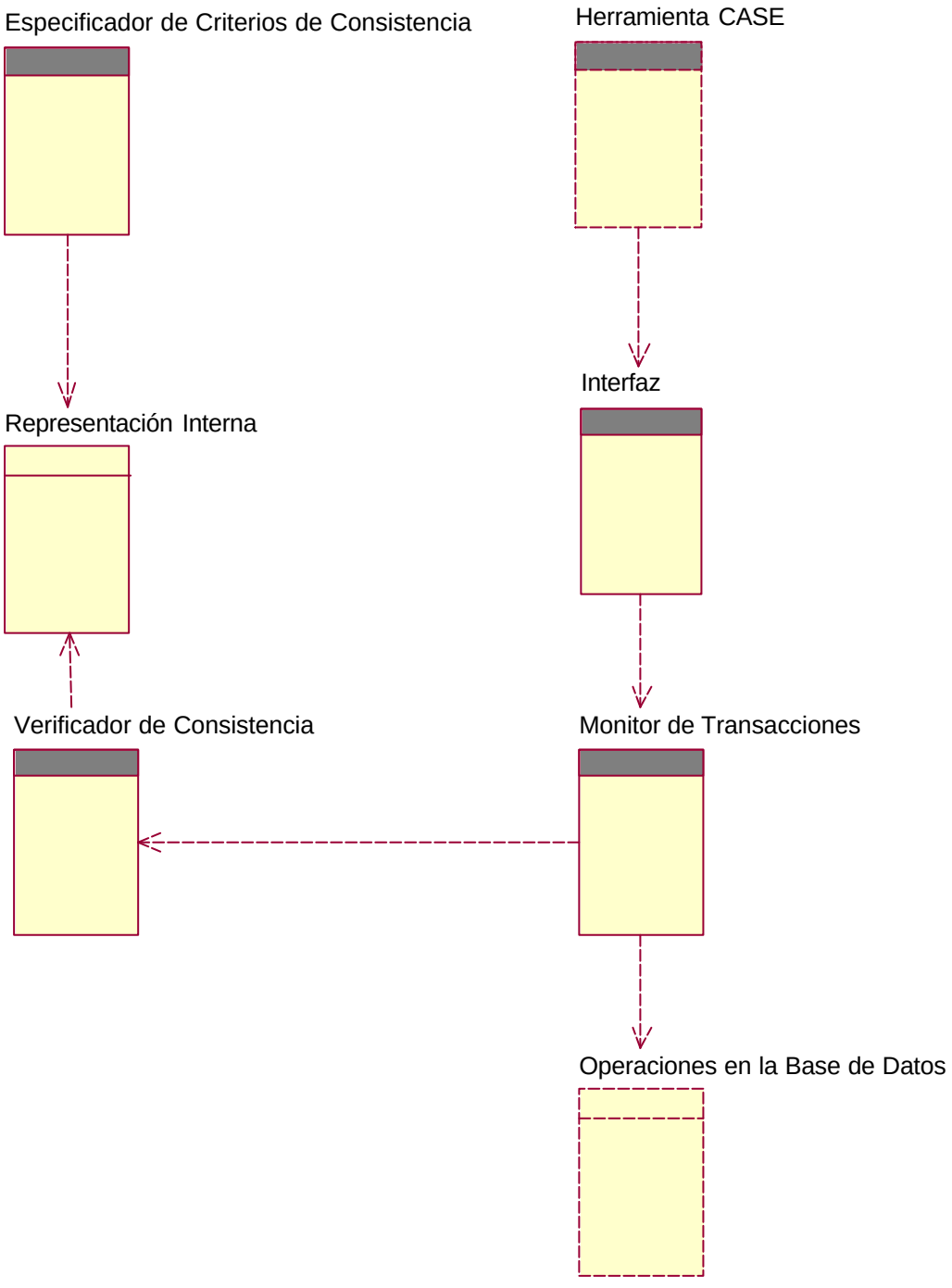


Figura 4.9. Arquitectura General del Sistema de Transacciones cooperativas

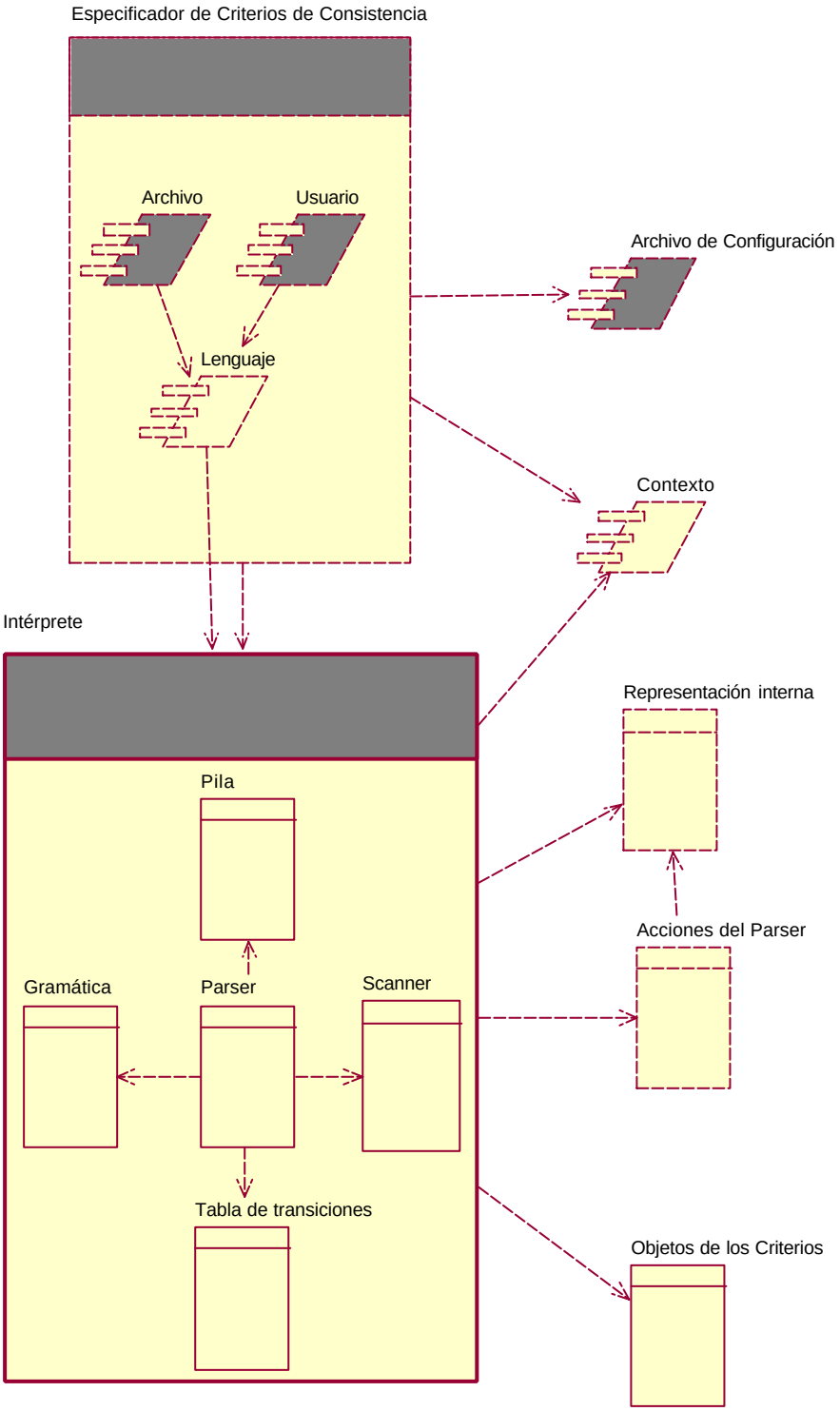


Figura 4.10. Especificador de Criterios de Consistencia.

El lenguaje orientado a especificar criterios de consistencia consta de dos objetos: Patrones y Conflictos. La Figura 4.10 ilustra cómo el Especificador de Criterios de Consistencia lee un archivo de configuración, a partir del cual, crea un contexto que contiene información referente a la fuente de donde el intérprete leerá las oraciones del lenguaje, los nombres de los archivos para construir la gramática y la tabla de transiciones del intérprete, por último el nombre de la clase utilizada para cargar las acciones asociadas a la tabla de transición. (Ver archivos de datos en el Apéndice **E**).

El intérprete se compone de una pila, una gramática, una tabla de transiciones, un parser (analizador sintáctico y traductor) y un scanner (analizador léxico). El parser constituye la columna vertebral del intérprete.

4.6.1 Descripción del Lenguaje Orientado a Especificar los Criterios de Consistencia

Debido a que un criterio de consistencia puede ser un patrón o un conflicto, el lenguaje está organizado en dos bloques: patrones y conflictos, no existe un orden en la aparición de estos bloques. La Gramática del lenguaje se muestra en el Apéndice B y se usa la notación BNF para representarla de manera precisa. Para identificar el inicio de cada bloque se emplea una palabra clave: **PATTERN** para un patrón y **CONFLICT** para un conflicto. Por ejemplo las oraciones de la Figura 4.11 representan los patrones y conflictos ilustrados en la Figura 4.3. Puede observarse que en estos ejemplos los símbolos no terminales se componen de una letra mayúscula, sin embargo, la gramática permite que un símbolo no terminal sea una identificador (es decir, una letra seguida de una combinación de caracteres alfanuméricos), por esta razón los símbolos no terminales deben separarse con un espacio en blanco ya que para el caso de AB habría una ambigüedad y no se podría determinar si se trata del símbolo AB o del símbolo A seguido del símbolo B. Puede observarse del ejemplo de la sección 4.3.4 que la **operación** (definida como <M,o,O>) es el único símbolo terminal.

La palabra reservada **INIT** se utiliza para indicar que el símbolo no terminal que sigue a continuación es el símbolo de inicio; si se omite esta palabra reservada, el símbolo de

inicio de cada criterio de consistencia será el consecuente de su primera regla.

```
PATTERN P1:

INIT S<-(ANY,W,IGU)A|@;
A<-(ANY,W,IGU)A|(Pruebas,R,IGU)B;
B<-(Pruebas,W,Reporte)S;


CONFLICT C1:

INIT S<-(ANY,W,IGU)A;
A<-(ANY,W,IGU)A|(Pruebas,R,IGU)B;
B<-(ANY,W,IGU)|(Pruebas,W,Reporte)S;


PATTERN P2:

INIT S<-B;
B<-A B|A;
A<-(ANY,INC,cont_pru)B(ANY,DEC,cont_pru)|
   (ANY,INC,cont_pru)(ANY,DEC,cont_pru);


PATTERN Ps:

INIT S<-(VAR M,R,VAR O)A|@;
A<-(VAR -M,W,VAR O)S|(VAR M,R/W,VAR O)|@;


CONFLICT Cs:

INIT S<-(VAR M,W,VAR O)|(VAR M,R,VAR O)A;
A<-(VAR -M,W,VAR O)S|(VAR M,R/W,VAR O)A;
```

Figura 4.11. Ejemplo de oraciones del lenguaje.

Para eliminar el no determinismo en la gramática del Apéndice B se realiza una factorización por la izquierda y se obtiene una gramática equivalente, la cual se muestra en el Apéndice C. La gramática factorizada se utiliza para construir la tabla de transiciones del intérprete.

Las fuentes de información para proporcionarle al Intérprete, las oraciones del lenguaje, pueden ser: un archivo de texto, una interfaz gráfica de usuario o cualquier otro medio. Para este trabajo se utiliza un archivo de texto por razones de productividad. El especificador de criterios de consistencia es demasiado robusto, flexible y extensible ya que permite añadir fuentes de información nuevas. Lo que tiene que hacerse es construir el conjunto de clases para manejar la fuente nueva y hacer la especificación en el archivo de configuración sin tener que recompilar el sistema completo.

4.6.2 Descripción del Intérprete

El núcleo del intérprete lo constituye el parser, este último toma del contexto los nombres y rutas de los archivos de texto donde se encuentran las definiciones de la gramática y de la tabla de transiciones, posteriormente procede a construir los objetos que representan tanto a la gramática como a la tabla de transiciones. Una vez cargada la gramática y la tabla de transiciones, el parser obtiene del contexto el nombre de la clase que se utilizará para cargar las acciones utilizadas en el proceso de traducción, estas acciones tienen una fuerte dependencia con la representación interna de los criterios de consistencia porque a partir de esta, generarán una gráfica cíclica dirigida con los objetos que representan internamente a los criterios de consistencia. Por último, el parser inicializa la pila y construye el analizador léxico (scanner) dándole a conocer a este la fuente de datos donde se encuentran las oraciones del leguaje. Una vez terminado este proceso de inicialización, se comienza con el proceso de traducción.

4.6.2.1 Descripción de la Tabla de Transiciones

La tabla de transiciones se basa en las tablas de análisis sintáctico LL(1). Una Tabla de análisis sintáctico para un analizador sintáctico (parser) LL(1) es una matriz bidimensional. Las filas se etiquetan con los símbolos no terminales de la gramática sobre la cual se basa el parser. Las columnas se etiquetan con los símbolos terminales de la gramática más una columna adicional FDC (o el carácter “$”, que representa la marca de fin de cadena). El elemento (M,N) de la tabla indica la acción que debe seguirse cuando el no

terminal M aparece en la cima de la pila y el símbolo de preanálisis es N. Cada celda de la matriz contiene un par (I,J) en donde I indica la regla que debe aplicarse, es decir, la regla cuyo antecedente debe remplazar el símbolo M del tope de la pila (Ver el archivo Rules.txt del Apéndice E) y j indica la acción semántica o la traducción que debe ejecutarse (ver la Figura 4.12 que muestra el arreglo que contiene los nombres de las clases utilizadas para representar las acciones semánticas). Una celda vacía indica una situación de error (color oscuro o rojo) o un situación que jamás se puede presentar (color claro o amarillo). Una celda con la palabra ***sync*** indica un carácter especial para recuperación de errores en modo de oración o modo de pánico. Esto quiere decir que cuando ocurre un error, se ignoran los símbolos de la entrada hasta encontrar el carácter de sincronización. El Apéndice D ilustra la tabla de transiciones del parser.

No. Acción	Acción
0	ComplementAction
1	CompositeOperatorAction
2	ConflictAction
3	GrammarAction
4	NonTerminatorAction
5	Nothing
6	NTInitialAction
7	NTListAction
8	ObjectAction
9	ObjectActionAny
10	OperationAction
11	OperatorAction
12	OperatorActionAny
13	PatternAction
14	ProductionAction
15	RuleAction
16	SimpleOperatorAction
17	TransactionAction
18	TransactionActionAny
19	VariableAction

Figura 4.12. Acciones semánticas del Parser.

4.6.2.2 Algoritmo del Parser

Una vez que se ha construido la tabla de transiciones, la tarea de escribir el segmento del programa que efectúe el análisis sintáctico del lenguaje es demasiado sencilla. Lo único que tiene que hacer el programa es insertar en la pila el símbolo de inicio de la gramática y posteriormente, mientras no se vacíe la pila, igualar los símbolos de la cima de la pila con los de la entrada o remplazar el no terminal de la cima de la pila siguiendo las directrices de la tabla de transición. Ver el algoritmo de la Figura 4.13.

```
Insertar (Símbolo de inicio)
Leer (Símbolo de entrada)
Mientras pila no esta vacía haz
Comienza
  En caso de que cima de la pila sea
    Terminal:
      Si cima de la pila = Símbolo de entrada entonces
        Extraer símbolo de la pila y Leer(Símbolo de la entrada)
      En otro caso
        Salir a la rutina de error
    No Terminal:
      Si tabla[cima de la pila, Símbolo de entrada] ¹ Vacío entonces
        Remplazar cima de la pila por
        tabla[cima de la pila, Símbolo de entrada].Antecedente
      En otro caso
        Salir a la rutina de error
Termina
Si Símbolo de entrada ¹ marca de fin de cadena entonces
  Salir a la rutina de error.
```

Figura 4.13. Algoritmo de análisis sintáctico.

4.7 Discusión

A continuación se resumen las características del modelo de transacciones cooperativas.

- Las transacciones deben estar organizadas jerárquicamente. Cada nodo interno en la jerarquía, es un grupo de transacciones cooperativas y cada hoja es una transacción cooperativa.

- Cada grupo de transacciones cooperativas debe tener su propia base de datos en donde se mantienen y editan copias de los objetos pertenecientes al grupo raiz. Esto da lugar a que existan diferentes versiones de un objeto a lo largo de la jerarquía.

- Cada grupo de transacciones cooperativas debe incorporar un conjunto de criterios de consistencia que deben satisfacer las operaciones, sometidas por cada uno de sus miembros, sobre su base de datos.

- La recuperación de una transacción cooperativa deberá realizarse en base a la última operación abortada. Es decir, se deshace esta última operación y todas las operaciones dependientes.

Algunas de la figuras del presenta capítulo, se construyeron utilizando diagramas de UML. La figura 4.1, que representa una jerarquía de transacciones cooperativas, se construyó con un diagrama de colaboración. Las máquinas de estados de la figura 4.3 se construyeron con los diagramas de estados. Por último, la arquitectura del sistema de transacciones cooperativas y la arquitectura del especificador de criterios de consistencia, representadas por las figuras 4.9 y 4.10 respectivamente, se construyeron con los diagramas de componentes.

El sistema de transacciones cooperativas se divide en tres subsistemas: El Especificador de Criterios de Consistencia, El Verificador de Consistencia y el Monitor de Transacciones. El alcance de este trabajo, contempla únicamente el diseño e implantación del Especificador de Criterios de Consistencia.

5 DISEÑO E IMPLANTACIÓN DEL SISTEMA

Para que el Sistema Especificador de Criterios de Consistencia sea portable a diferentes plataformas se decidió usar el lenguaje de programación Java. Puesto que en el desarrollo de este sistema se utilizan muchas bibliotecas de clases del lenguaje java y sus nombres están en inglés, las clases pertenecientes al sistema también se escribieron en inglés para tener una homologación. La Figura 5.1. ilustra la jerarquía de paquetes del sistema. El Apéndice E.2. ilustra el diccionario de clases del sistema, es decir, describe globalmente cada paquete y cada una de las clases que integran el paquete.

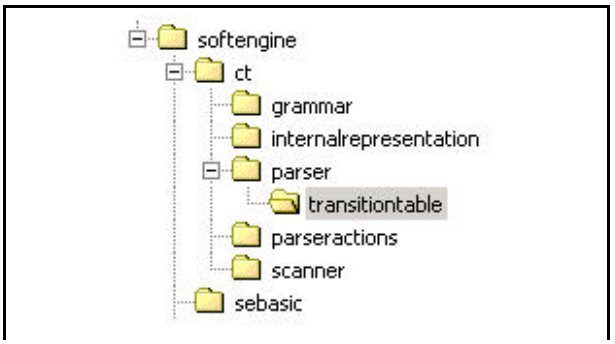


Figura 5.1. Jerarquía de paquetes del sistema.

5.1 El paquete sebasic

Este paquete desarrollado por el M.C. Cristóbal Juárez Castellanos contiene un conjunto de clases básicas que representan algunas estructuras de datos bastante usadas. Las clases básicas de este paquete son la secuencia (representado por la clase ***SESequence***) y el arreglo dinámico (representado por la clase ***SEDynamicArray***), ambas son contenedoras de objetos. El siguiente grupo de clases: ***SECommand***, ***SECommandHistory***, ***SECommandSequence*** y ***SECommandSeqIterator*** forma parte del patrón de diseño **Comando**. Este patrón de diseño encapsula un comando como un objeto, permitiendo parametrizar clientes con diferentes comandos y soportando el deshacer y reahacer de un comando. Por último la clase ***SESequenceIterator*** forma parte del patrón de diseño **Iterador**. Este patrón de diseño brinda un medio para acceder

secuencialmente a los elementos de un contenedor sin exponer su estructura interna. Ver paquete sebasic en el Apéndice E.2 y [GHJV94]. El Apéndice F muestra el diagrama de clases del paquete sebasic.

5.2 Diseño e Implantación del Scanner

El objeto correspondiente al scanner (Analizador léxico) lo construye el parser. La Figura 5.2. muestra el diagrama de clases correspondiente al diseño del Scanner. Las clases que pertenecen al lenguaje java y son utilizadas por el Scanner son: ***FileReader***, ***BufferedReader*** y ***StringTokenizer.*** La clase ***FileReader*** se utiliza para construir un objeto apuntador que haga referencia a un archivo. La clase ***BufferedReader*** se utiliza para construir un objeto apuntador que haga referencia a un Buffer. Por último la clase ***StringTokenizer*** se utiliza para construir un objeto que descompone una línea de texto en una lista de tokens (palabras) en base a un conjunto de delimitadores.

El Scanner se compone de la interfaz ***ScannerBuffer*** y de las clases ***Scanner*** y ***ScannerFileBuffer***. La interfaz ***ScannerBuffer*** define únicamente al método ***GetInputLine().*** La clase ***ScannerFileBuffer*** implementa el método de esta interfaz usando un archivo de texto como la fuente de entrada para las oraciones del lenguaje de especificación de criterios de consistencia. Si deseamos agregar una fuente de entrada nueva, como por ejemplo una ventana de usuario, se puede implantar la clase ***ScannerGUIBuffer*** y definirla para que implemente el método ***GetInputline()*** de la interfaz.

La clase ***Scanner*** tiene definidos tres atributos. El primero de ellos, denominado “white_char”, contiene los delimitadores que se consideran como espacios en blanco (el tabulador “\t”, el espacio en blanco “ ”, el retorno de carro “\r” y el salto de línea “\n”). El segundo atributo “delims” contiene los delimitadores usados para diferenciar un token de otro. El tercer atributo “return_delim” contiene un valor booleano (true para que un delimitador se considere como un token y false para el caso contrario). El parser le envía el mensaje de solicitud de token al Scanner mediante el método GetToken() y a través de este mismo método el Scanner le proporciona al Parser el token.

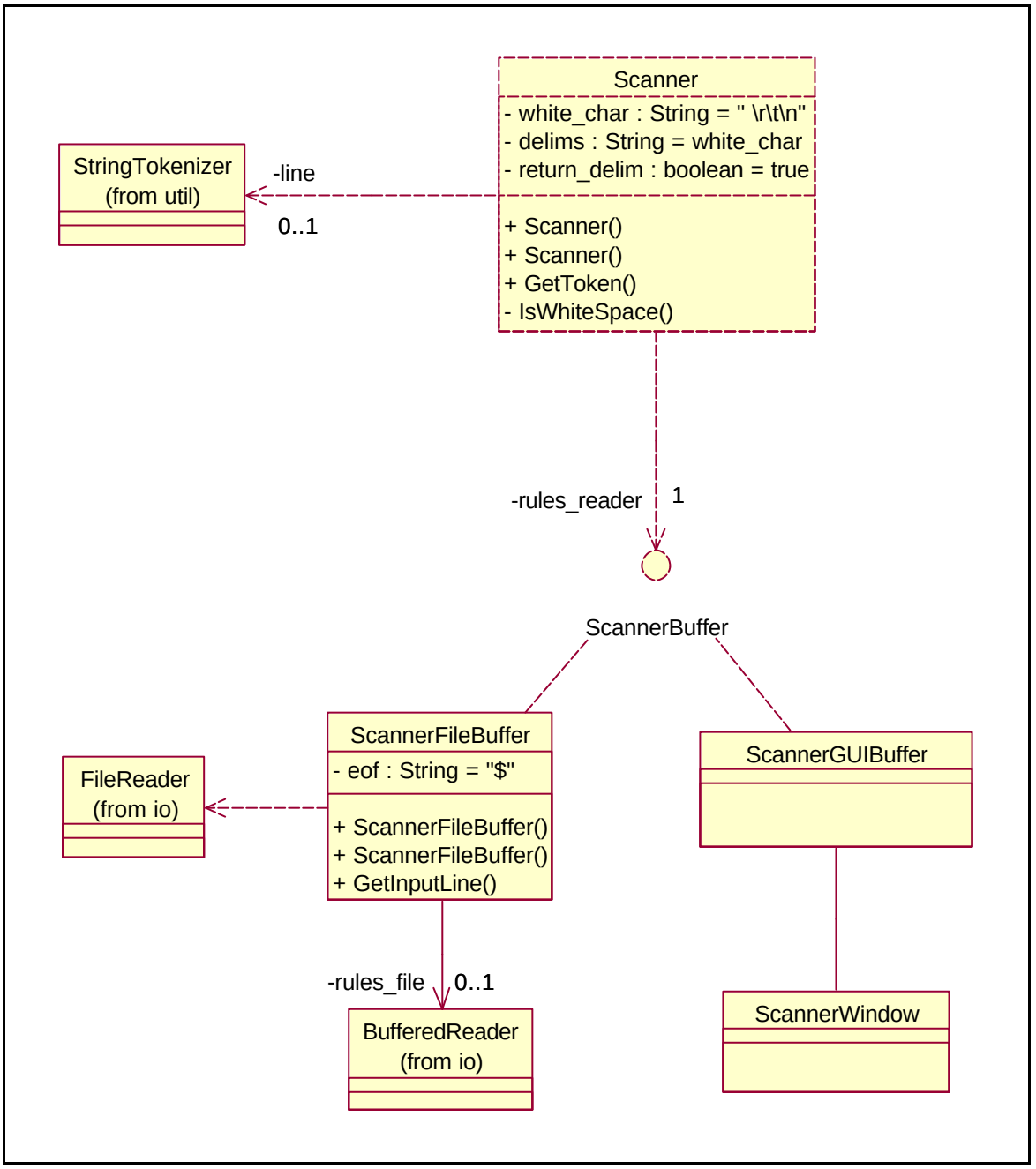


Figura 5.2. Diagrama de clases del Scanner.

La clase Scanner tiene definidos dos métodos constructores:

public Scanner(ScannerBuffer rules_buffer) throws Exception
public Scanner(ScannerBuffer rules_buffer, String delimiters) throws Exception

El primer constructor se utiliza para construir un scanner usando como delimitadores a los caracteres contenidos en el atributo white_chars. El segundo constructor utiliza como delimitadores los caracteres especificados en el atributo white_char y la variable delimiters.

Cuando el scanner termina de leer la fuente de datos le manda al parser el token especial de fin de cadena (por defecto este carácter es "$").

5.3 Diseño e Implantación de la Gramática del Parser

La Figura 5.3 ilustra el diagrama de clases correspondiente a la gramática del parser, este diagrama se apega a la definición dada en la sección 4.3.2. Los símbolos no terminales de la gramática están representados por la clase ***NTSequence*** (del inglés non terminator sequence), los símbolos terminales de la gramática están representados por la clase ***TSequence*** (del inglés terminator sequence), las reglas de producción de la gramática están representadas por la clase ***RulesArray*** y el símbolo de inicio (que es un símbolo no terminal) queda representado por la clase ***NonTerminator.*** Una regla (representada por la clase ***Rule***) está formada por un símbolo no terminal denominado consecuente (clase ***NonTerminator***) y una secuencia de símbolos terminales (clase ***Terminator***) y no terminales (clase ***NonTerminator***) denominada antecedente y representada por la clase ***SymbolSequence.*** La clase principal de la gramática del parser es la clase **Grammar,** esta clase contiene un constructor con tres argumentos:

```
public Grammar(ScannerBuffer termbuf, ScannerBuffer nontermbuf, ScannerBuffer rulesbuf)
   throws Exception
  {
    LoadGrammar(termbuf, nontermbuf, rulesbuf);
  }
```

El parser toma del contexto los nombres y rutas de tres archivos de configuración para poder construir la gramática (ver archivos de datos en el apéndice E), el primer archivo contiene una lista con los símbolos terminales, el segundo archivo contiene una lista con los símbolos no terminales y el tercer archivo contiene la gramática de lenguaje en un formato tipo texto y con notación BNF.

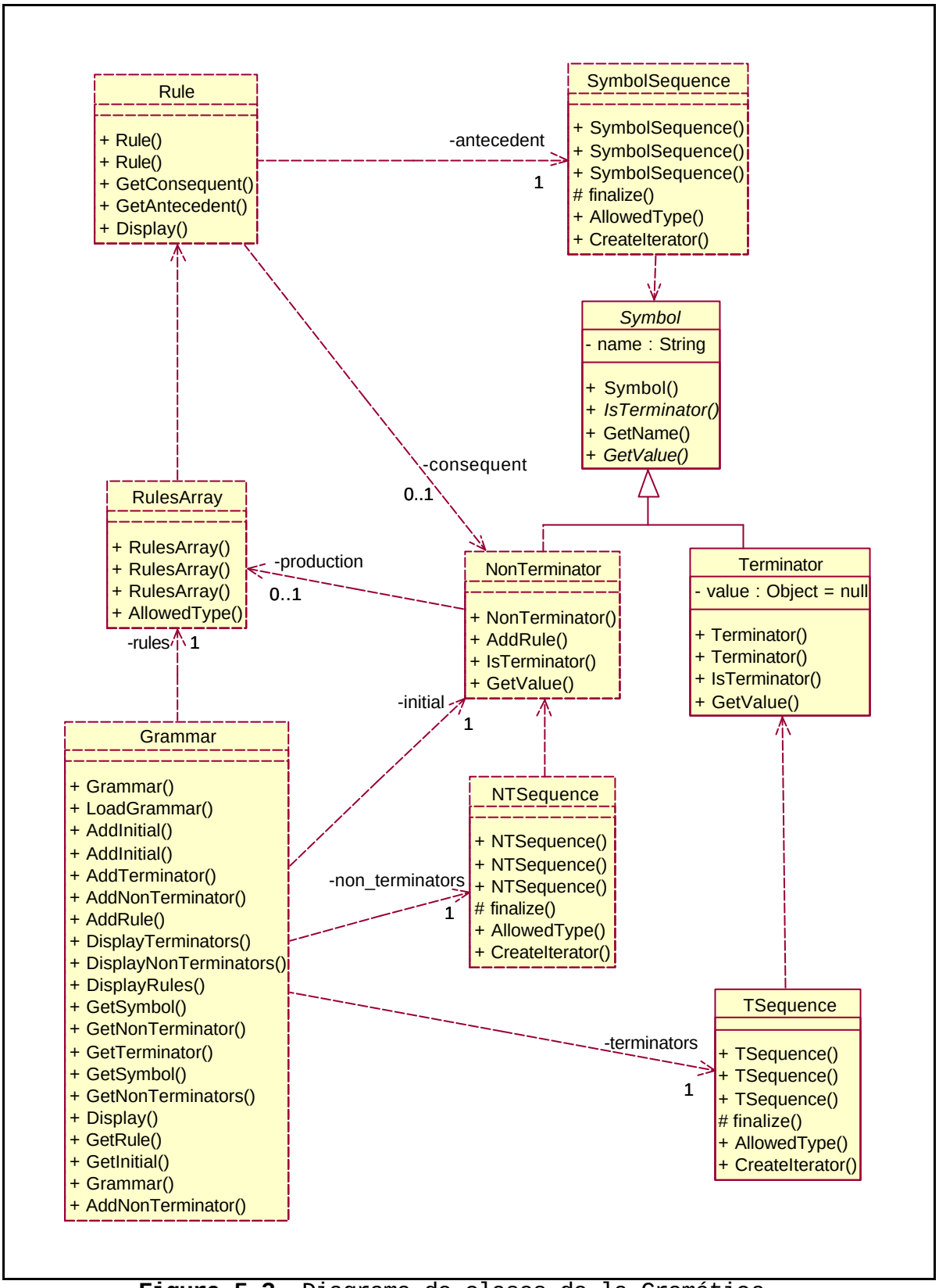


Figura 5.3. Diagrama de clases de la Gramática.

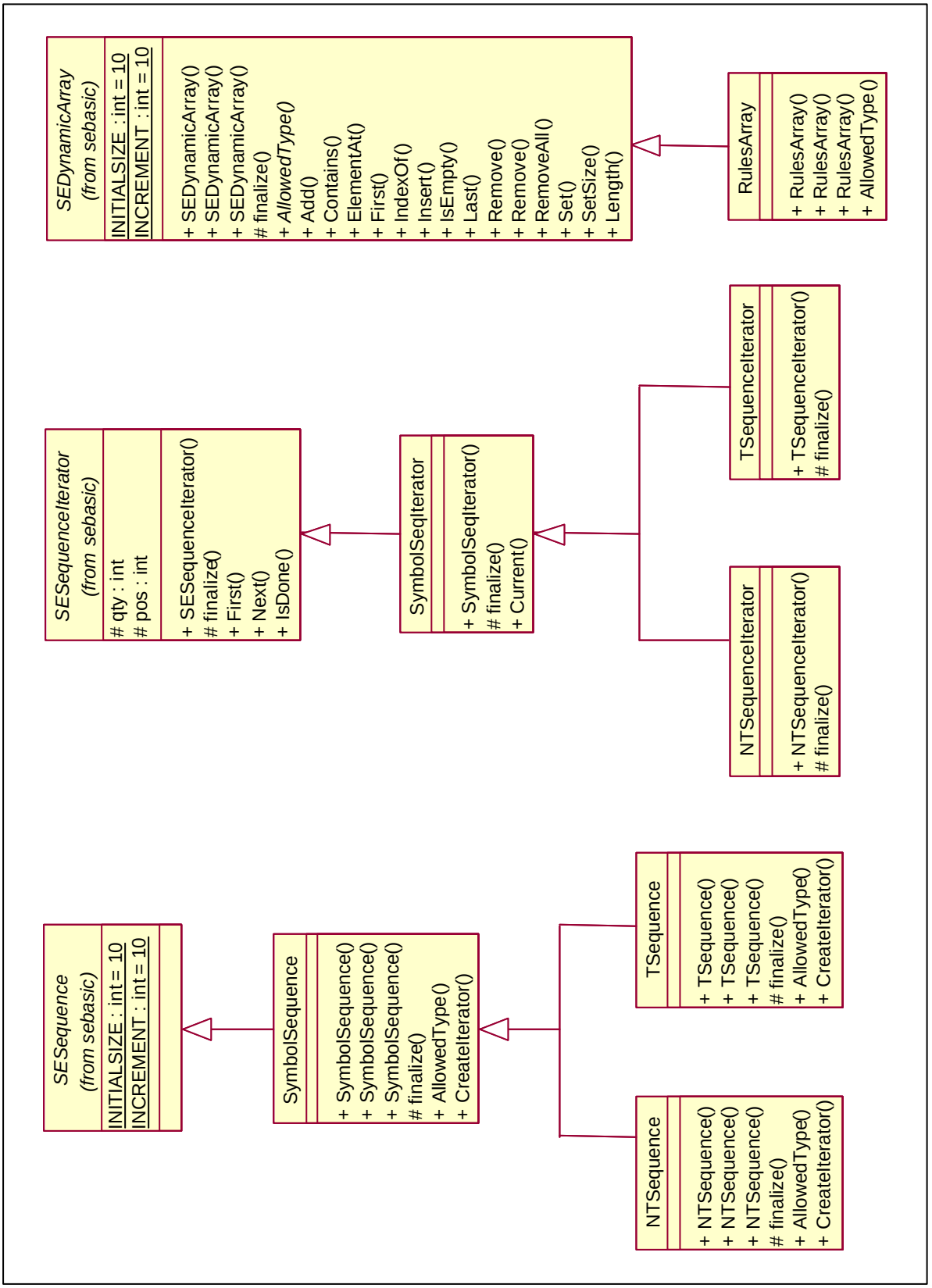


Figura 5.4. Parte complementaria del diagrama de clases de la Gramática.

Una vez que el parser toma del contexto los nombres de los archivos, construye un buffer de datos para cada archivo y genera el objeto correspondiente a la gramática mandándole al constructor estos tres buffers como argumentos (termbuf, nontermbuf, rulesbuf).

La gramática tiene adicionalmente el constructor Grammar(), utilizado para crear una gramática vacía y poder adicionarle nuevos elementos conforme progrese una tarea. Este constructor es de utilidad para poder construir los objetos con la representación interna de los criterios de consistencia.

Los métodos utilizados en el proceso de contrucción de la gramática son: **Grammar(), LoadGrammar(), AddInitial(), AddInitial(), AddTerminator(), AddNonTerminator(), AddNonTerminator()** y **AddRule().**

Los métodos utilizados para obtener o acceder los elementos de la gramática son: **GetSymbol(), GetNonTerminator(), GetTerminator(), GetSymbol(), GetNonTerminators(), GetRule()** y **GetInitial()**

Los métodos utilizados para visualizar el contenido de la gramática son: **DisplayTerminators(), DisplayNonTerminators(), DisplayRules()** y **Display().**

Para el diseño de la gramática se utilizó una variante del patrón de diseño **Compositor**. Este patrón de diseño compone objetos en estructuras de árbol para representas jerarquías parte-todo. Permite tratar de manera uniforme objetos individuales y objetos compuestos. Las clases de la figura 5.3 que integran este patrón de diseño son: **RulesArray, Rule, SymbolSequence, Symbol, NonTerminator y Terminator**.

La Figura 5.4 muestra la parte complementaria correspondiente al diagrama de clases de la gramática del parser. Se ilustra la jerarquía de herencia y las clases ***SymbolSeqIterator***, ***TsequenceIterator*** y ***NTSequenceIterator*** utilizadas para crear los objetos iteradores que recorrerán los elementos de las secuencias **SymbolSequence, TSequence** y **NTSequence** respectivamente.

5.4 Representación Interna de los Criterios de Consistencia

Tal y como se definieron los criterios de consistencia en la sección 4.3, la Figura 5.5 ilustra el diagrama de clases correspondiente a la representación interna de dichos criterios de consistencia. Un criterio de consistencia queda representado por la clase ***Criterium***. En la jerarquía de herencia se observa que un Patrón (representado por la clase ***Pattern***) y un conflicto (representado por la clase ***Conflict***) son criterios de consistencia (tal como se definió previamente). Un criterio de consistencia tiene un nombre (dado por el atributo name) y una gramática (dada por la relación de asociación entre las clases ***Criterium*** y ***Grammar***). El conjunto de todos los criterios de consistencia queda representado por la clase ***CorrectnessCriteria*** que contiene dos secuencias de criterios de consistencia (representado por las asociaciones entre las clases ***CorrectnessCriteria*** y ***CTCriteriaSequence***) y un iterador por cada secuencia(representado por las asociaciones entre las clases ***CorrectnessCriteria*** y ***CTCriteriaSeqIterator***). La asociación con el nombre ***patternseqit*** representa a la secuencia de patrones y tiene asignado el iterador ***patternseqit***. La asociación con el nombre ***conflictseqit*** representa a la secuencia de conflictos y tiene asignado el iterador ***conflictseqit***.

Como se mencionó en la sección 4.6.1, la operación es el único símbolo terminal. Puede observarse de la Figura 5.3 que un terminador (clase *Terminator*) tiene un nombre (heredado de la clase *Symbol*) y un valor. El nombre del terminador, en este caso, es ***CTDBOperation*** (Cooperative Transaction Data Base Operation) y valor es una tupla con tres elementos cuya representación interna esta dada por la clase ***CTDBOperationValue*** (Cooperative Transaction Data Base Operation Value) en el diagrama de clases de la Figura 5.6. El primer elemento de la tupla (es decir, la transacción) queda representada por la asociación entre las clases **CTDBOperationValue** y ***CTOperationArgument*** con el nombre ***dbtransaction***. Por la relación de herencia se infiere que la transacción puede ser el operador **any** (representado por la clase ***CTAnyOperator***), un identificador (representado por la clase ***CTIdentifier***) o una variable (representada por la clase ***CTVariable***). Lo mismo ocurre para el tercer elemento (es decir el objeto), este queda representada por la asociación entre las clases **CTDBOperationValue** y ***CTOperationArgument*** con el nombre ***dbobject***.

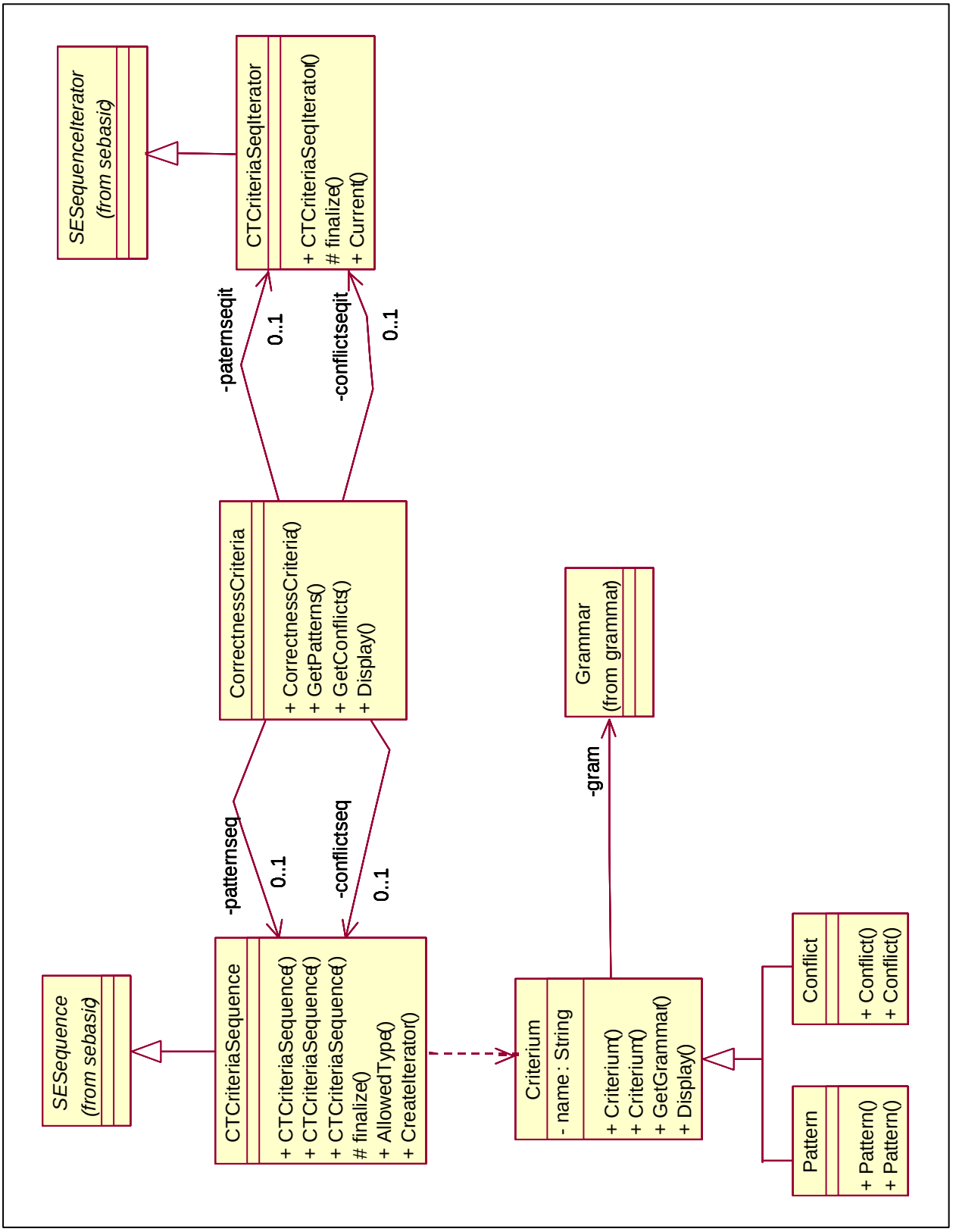


Figura 5.5. Representación interna de los criterios de consistencia.

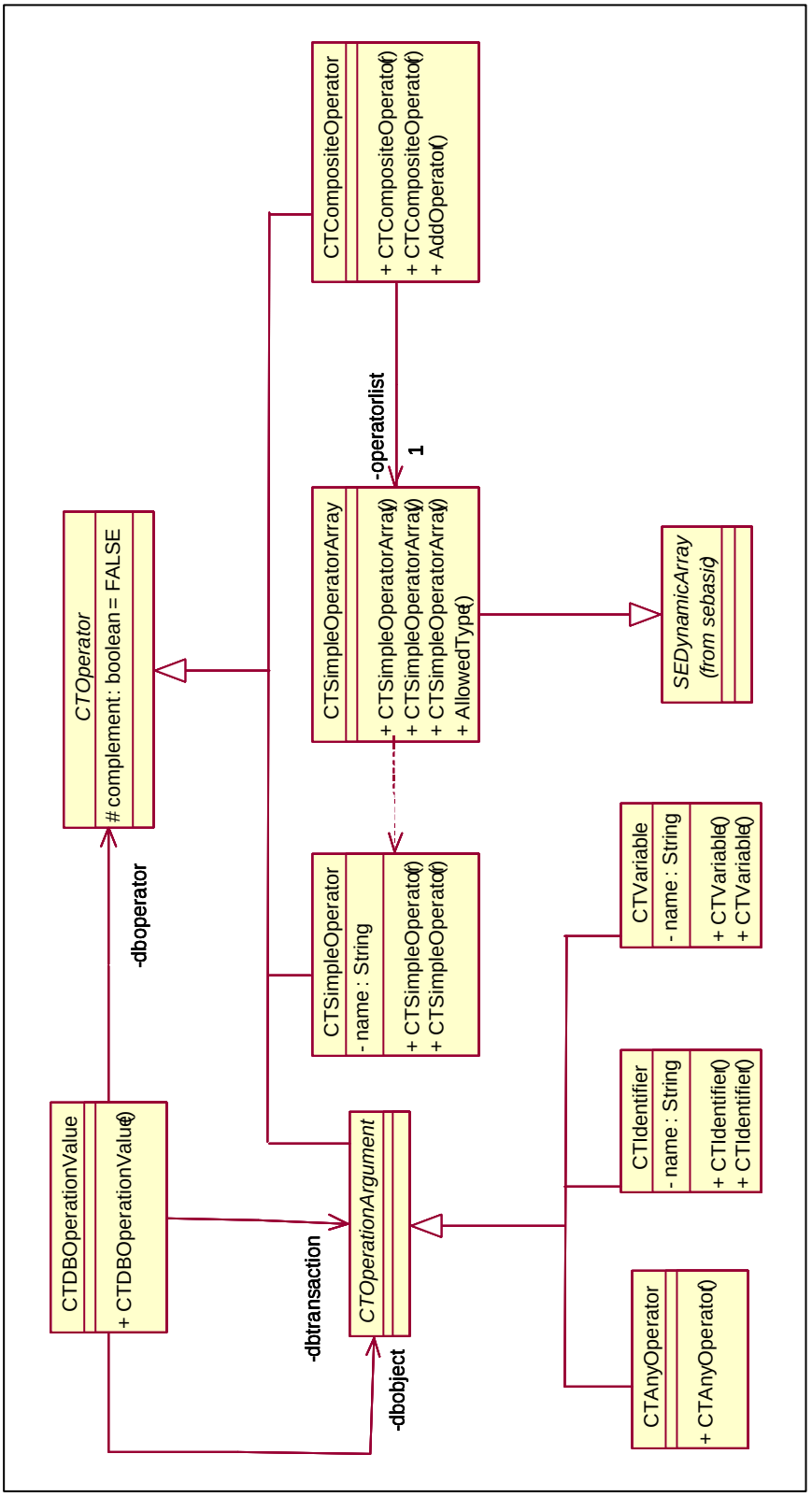
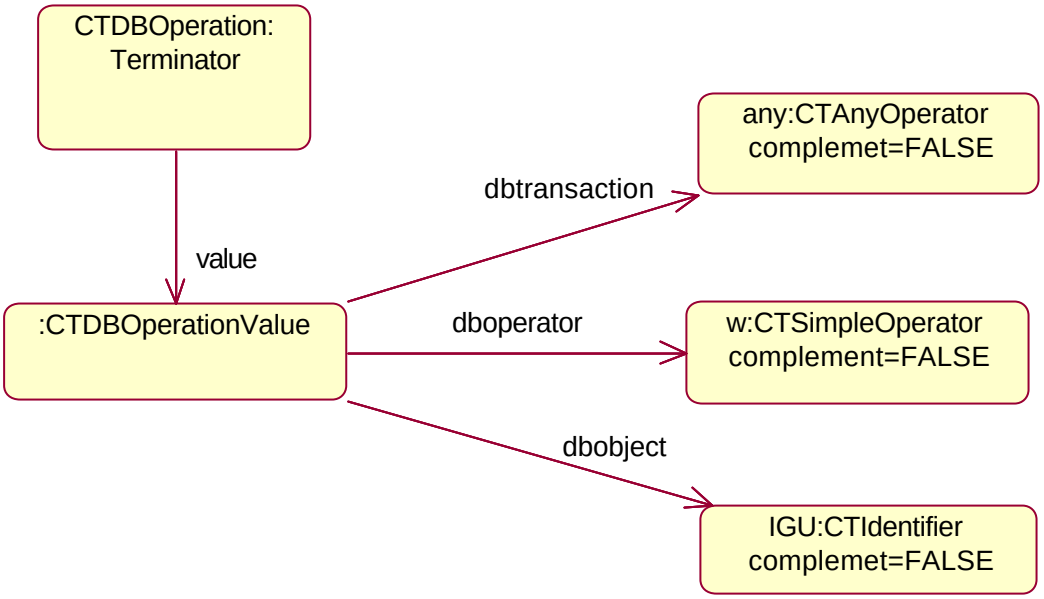
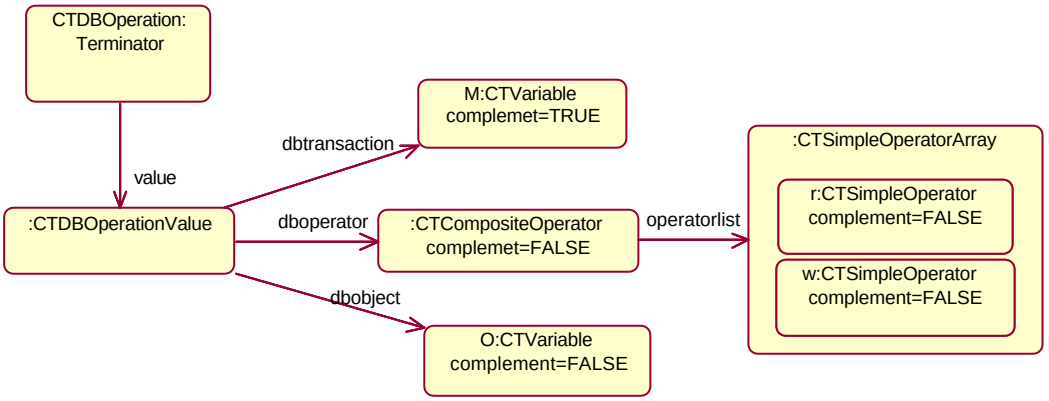


Figura 5.6. Representación interna de una operación ***(Transacción, Operador, Objeto)***.



a. Diagrama de objetos para la operación (any, w IGU).



b. Diagrama de objetos para la plantilla de operación (-W, r/w O).

Figura 5.7. Diagrama de objetos de dos operaciones.

El segundo elemento (es decir, el operador) queda representada por la asociación entre las clases ***CTOpertor*** y **CTDBOperationValue** con el nombre ***dboperator***. El operador puede ser un operador simple (representado por la clase ***CTSimpeOperator***)

un operador compuesto (representado por la clase ***CTCompositeOperator***), el operador **any**, un identificador o una variable. Un operador compuesto es un arreglo de operadores simples. Si cualquier elemento de una operación es una variable, entonces el patrón o el conflicto a que pertenece será una plantilla. La Figura 5.7a muestra un ejemplo del diagrama de objetos para la operación (any, w IGU) y la Figura 5.7b muestra un ejemplo del diagrama de objetos para la plantilla de operación (-M, r/w, O). Los objetos y relaciones entre ellos, en ambos diagramas, se construyen en base al diagrama de clases de la Figura 5.6.

5.5 Diseño e Implantación de la Tabla de Transiciones

Observando la tabla de transiciones del Apéndice D y contando las celdas vacías, se obtiene que estas constituyen aproximadamente el 84% del total de celdas. Esta es una característica muy común de las tablas de transiciones. Por lo tanto la tabla de transición es una tipo de arreglo bidimensional conocido como matriz esparcida. Una matriz esparcida es aquella cuyo contenido esta formado en su mayoría por celdas vacías. Dicho de otro modo, la cantidad real de datos diferentes del vacío es muy pequeña comparada con el almacenamiento asignado para la matriz.

Una matriz esparcida puede representarse mejor usando memoria ligada. Se mantiene en la memoria una estructura que contiene únicamente a los elementos que son diferentes del vacío y encadenándolos uno con otro [TA1983]. La Figura 5.8 ilustra el diagrama de clases para representar la tabla de transiciones y La Figura 5.9 muestra el diagrama de objetos correspondiente, ilustrando solo una parte de la tabla.

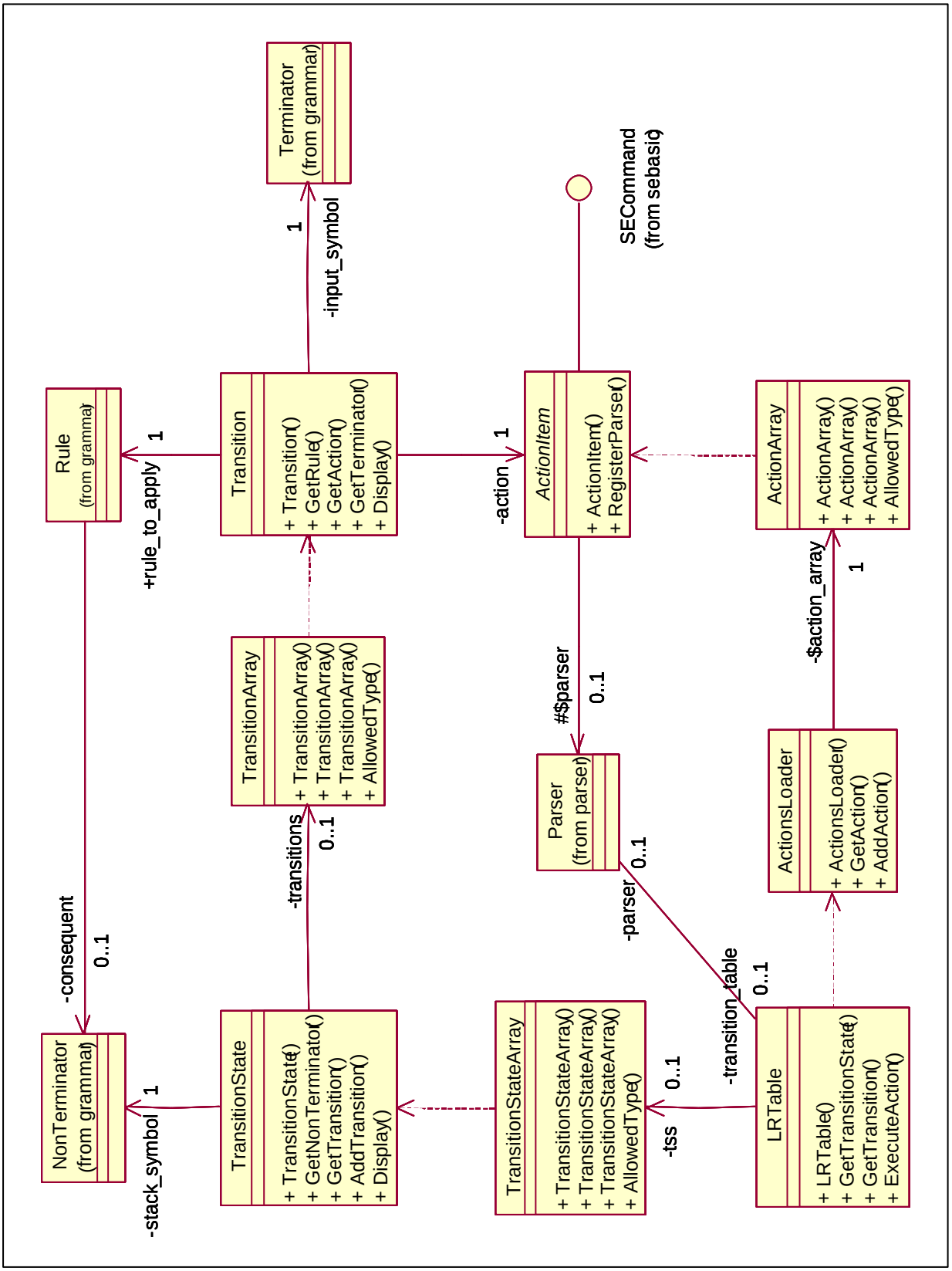


Figura 5.8. Diagrama de clases de la tabla de transiciones.

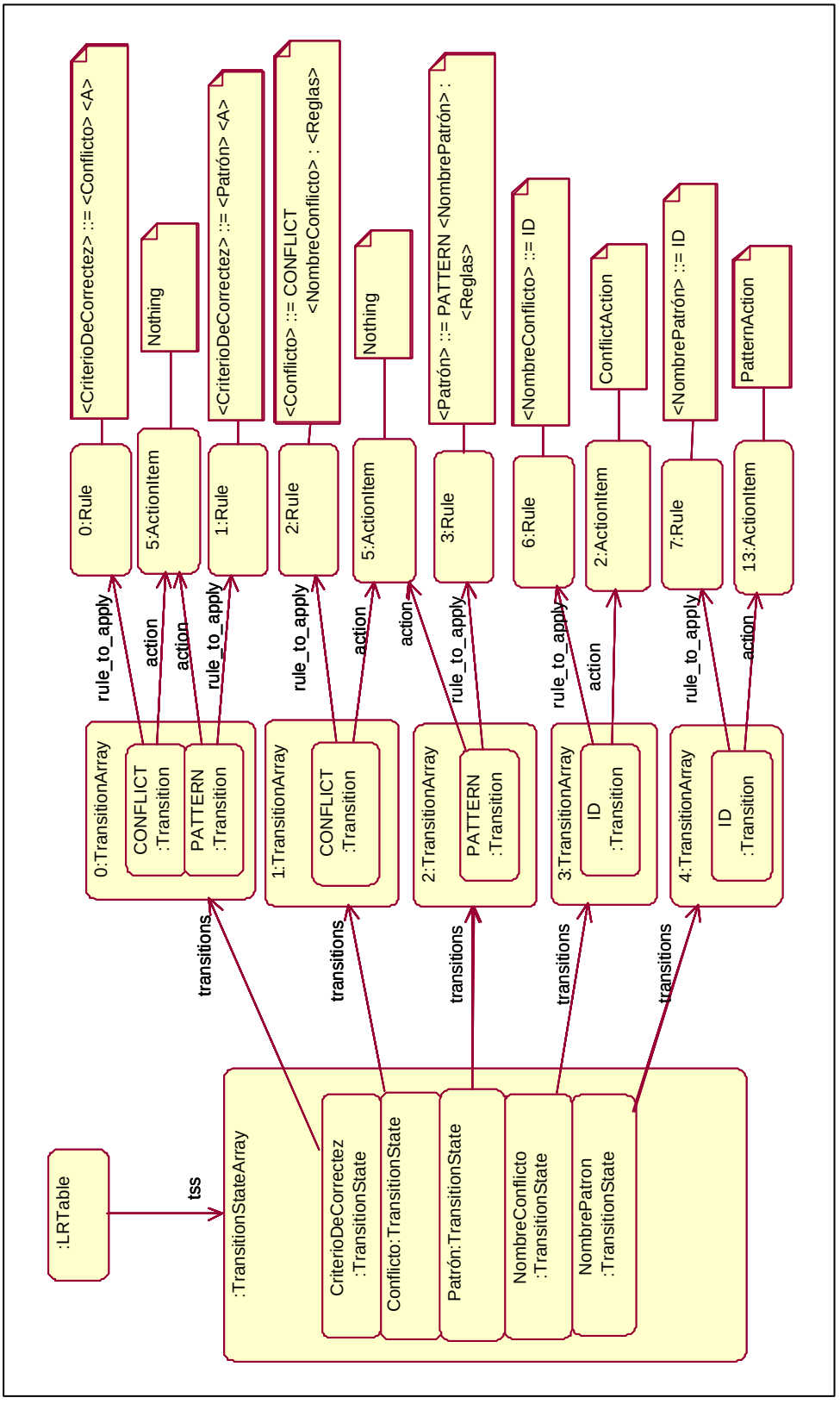


Figura 5.9. Diagrama de objetos de la tabla de transiciones. (Un subconjunto de los objetos que integran la tabla)

5.6 *Diseño e Implantación del Parser*

El parser es un analizador sintáctico predictivo no recursivo y se basa en las gramáticas del tipo LL(1). Las cuales, tienen la característica importante de que su tabla de transición no tiene entradas con definiciones múltiples. La Figura 5.10 muestra el diagrama de clases correspondiente al parser. La pila del parser queda representada por la clase ***Stack***. Esta clase forma parte de las bibliotecas de clases del lenguaje Java.

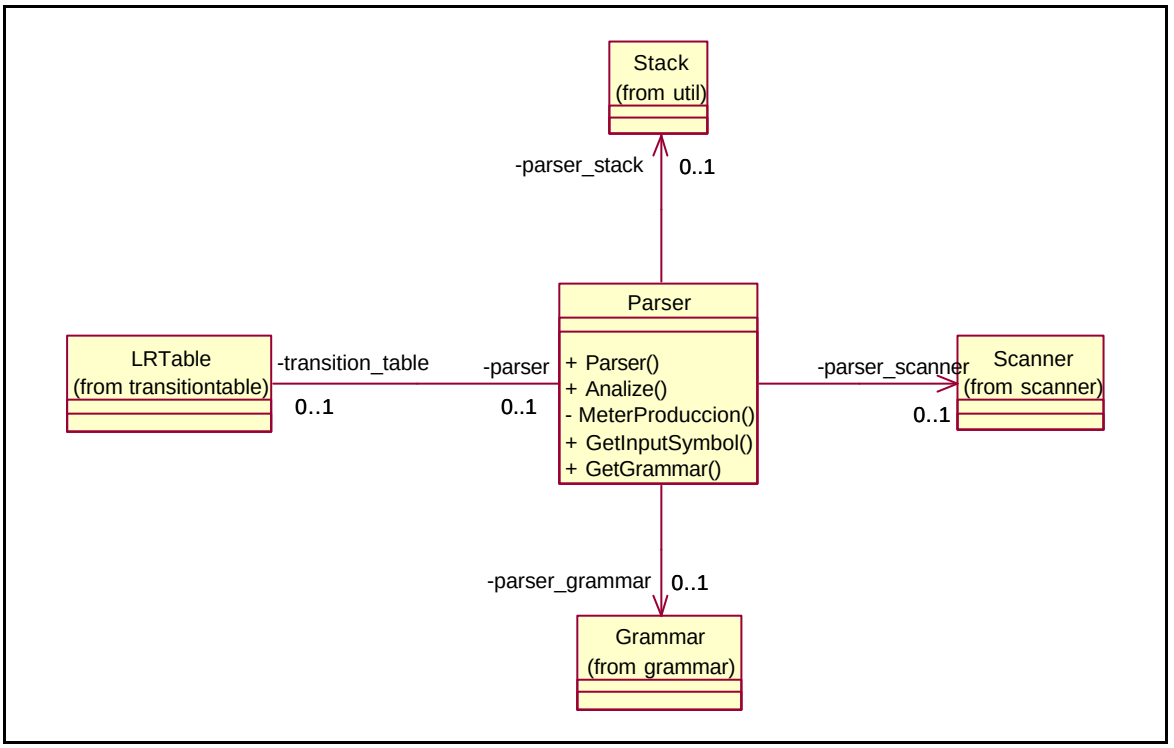


Figura 5.10. Diagrama de clases del Parser.

El especificador de criterios de consistencia construye el parser invocando el método ***Parser()*** y enviándole como parámetro el contexto. El método ***Analize()*** implementa el algoritmo descrito en la sección 4.6.2.2. y es quien realiza el análisis sintáctico y la traducción, realiza la invocación del método ***MeterProducción()*** para introducir el antecedente de una regla en la pila. A continuación se ilustra la codificación en java del método ***Analize():***

```
  public void Analize() throws Exception
  {
    String str;
    Symbol stack_symbol = parser_grammar.GetInitial();
    Vector Action;
    parser_stack.push(parser_grammar.GetSymbol(eof));
//  Apuntar al primer Simbolo de la entrada
    str = parser_scanner.GetToken();
    input_symbol = parser_grammar.GetSymbol(str);
    if (input_symbol == null)
      input_symbol = new Terminator("ID",str);
    do {
//    System.out.println(stack_symbol.GetName()+" "+input_symbol.GetName());
      if (stack_symbol.IsTerminator()) {
        if (stack_symbol.GetName().compareTo(input_symbol.GetName())==0)
          stack_symbol= (Symbol) parser_stack.pop();
        else
          System.out.println("Error: No se espera el simbolo "+str);
        str = parser_scanner.GetToken();
        input_symbol = parser_grammar.GetSymbol(str);
        if (input_symbol == null)
        input_symbol = new Terminator("ID",str);
      }
      else {
        Action=transition_table.ExecuteAction((NonTerminator) stack_symbol,
                                              (Terminator) input_symbol);
        MeterProduccion(Action);
        stack_symbol= (Symbol) parser_stack.pop();
        //Si la produccion es nula NonTerminator -> $
        if (stack_symbol.GetName().compareTo(eof)==0)
          stack_symbol= (Symbol) parser_stack.pop();
      }

    } while (!parser_stack.empty());
  }
```

5.7 Las Acciones Semánticas del Parser

Como se ilustra en la Figura 4.10., tanto la representación interna de los criterios de consistencia como las acciones semánticas del parser son programas externos al interprete.

Esto le da robustez al interprete ya que le permite cargar a tiempo de ejecución, las acciones semánticas y la representación interna de las reglas de consistencia. El intérprete funciona con cualquier tabla de transición basada en una gramática LL(1) y cada tabla de transición tiene asociado su respectivo conjunto de acciones semánticas y su respectiva gramática. A su vez, cada conjunto de acciones semánticas esta asociado con las clases de los objetos que se construirán como resultado del proceso de la interpretación.

La Figura 5.11. ilustra el diagrama de clases principal de las acciones semánticas del parser. En esta figura se puede observar la relación de las acciones semánticas con la representación interna de los criterios de consistencia, la tabla de transiciones y la gramática. Las Figuras 5.12. y 5.13. ilustran los diagramas de clases de las acciones semánticas del parser descritas previamente en la Figura 4.12.

Las acciones semánticas del parser son cargadas a tiempo de ejecución por la tabla de transiciones, con el método de clase **forName** tal y como se muestra a continuación:

Class.forName(action_manager);

La clase **Class** pertenece a las bibliotecas de clases del lenguaje Java. La variable **action_manager** contiene el nombre de la clase utilizada para cargar las acciones semánticas, para este caso su valor es **"CTActionsLoader"**. La Figura 5.14. muestra el código de la clase ***CTActionsLoader***; en el que se puede observar el bloque siguiente:

```
static {
   new CTActionsLoader();
}
```

este bloque invoca al constructor de la clase ***CTActionsLoader*** en el instante en que se carga la definición de la clase con el método **forName**. Se observa además en el código del constructor, que crea los objetos correspondientes a las acciones semánticas y los agrega a un arreglo de acciones perteneciente a la clase ***ActionsLoader***. El método ***AddAction*** es un método estático (o método de clase).

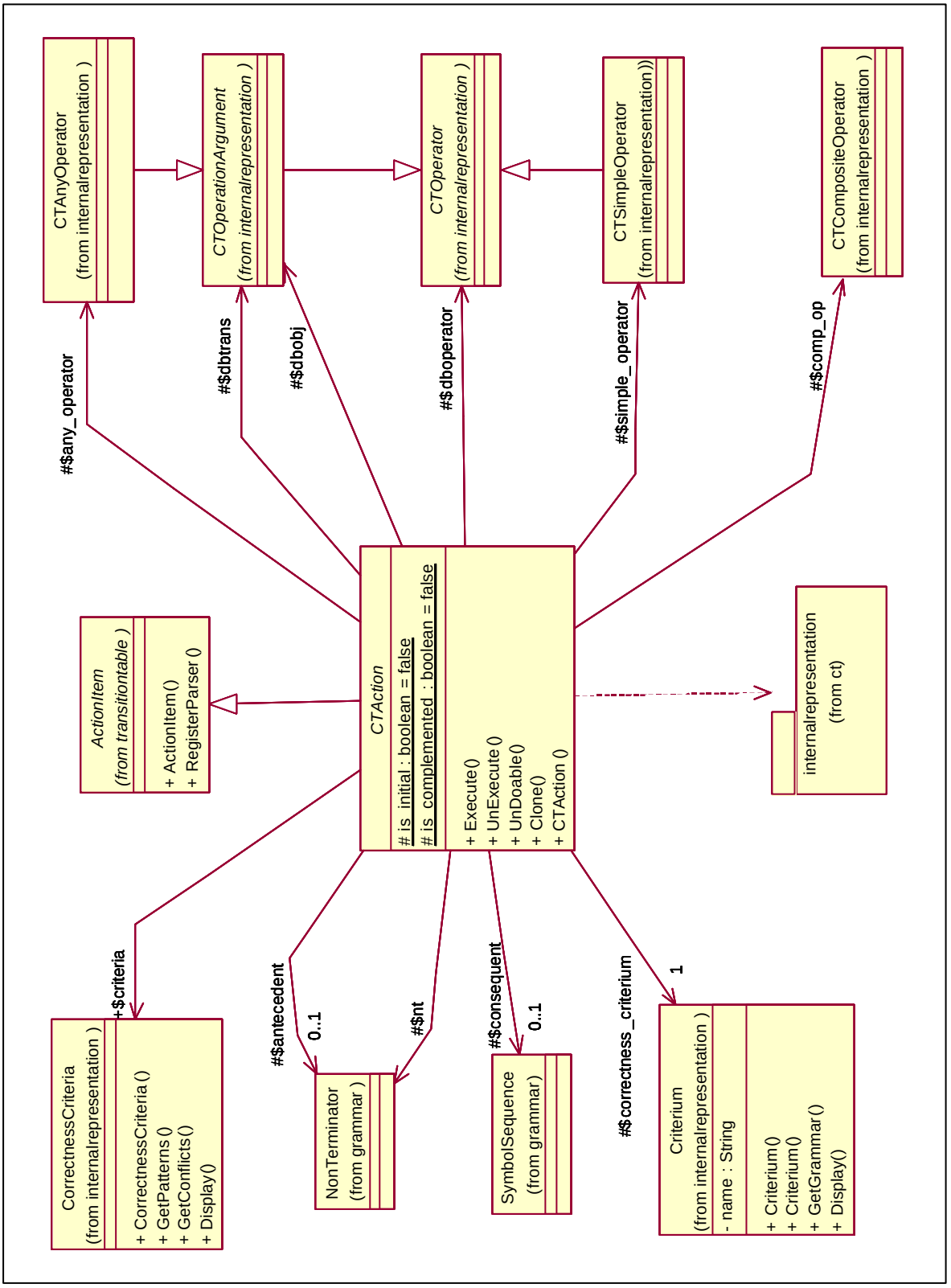


Figura 5.11. Diagrama de clases principal de las acciones semánticas del parser.

```
//Source file: D:\\users\\Tesis\\softengine\\ct\\parseractions\\CTActionsLoader.java

package softengine.ct.parseractions;

import softengine.ct.parser.transitiontable.ActionsLoader;

/**
CLASE: CTActionsLoader                AUTOR: Dante Ortiz Ancona
Fecha de Creacion: 02.07.2002         Fecha Ultima Modificacion: 03.07.2002
Descripcion: Carga a memoria las acciones semánticas del parser
*/
public class CTActionsLoader
{

  public CTActionsLoader()
  {
    ActionsLoader.AddAction(new ComplementAction());
    ActionsLoader.AddAction(new CompositeOperatorAction());
    ActionsLoader.AddAction(new ConflictAction());
    ActionsLoader.AddAction(new GrammarAction());
    ActionsLoader.AddAction(new NonTerminatorAction());
    ActionsLoader.AddAction(new Nothing());
    ActionsLoader.AddAction(new NTInitialAction());
    ActionsLoader.AddAction(new NTListAction());
    ActionsLoader.AddAction(new ObjectAction());
    ActionsLoader.AddAction(new ObjectActionAny());
    ActionsLoader.AddAction(new OperationAction());
    ActionsLoader.AddAction(new OperatorAction());
    ActionsLoader.AddAction(new OperatorActionAny());
    ActionsLoader.AddAction(new PatternAction());
    ActionsLoader.AddAction(new ProductionAction());
    ActionsLoader.AddAction(new RuleAction());
    ActionsLoader.AddAction(new SimpleOperatorAction());
    ActionsLoader.AddAction(new TransactionAction());
    ActionsLoader.AddAction(new TransactionActionAny());
    ActionsLoader.AddAction(new VariableAction());
  }

  static {
    new CTActionsLoader();
  }
}
```

Figura 5.14. Código de la clase CTActionsLoader.

5.8 Manejo de Excepciones

Un analizador sintáctico puede utilizar muchas estrategias para la detección y recuperación de errores. La estrategia empleada en el desarrollo de este sistema es conocida como "Recuperación en modo de pánico". Esta estrategia consiste en lo siguiente; Cuando el parser descubre un error, desecha los símbolos de entrada, de uno en uno, hasta que encuentra uno perteneciente a un conjunto designado de componentes léxicos de sincronización.

El conjunto de componentes léxicos de sincronización esta formado por las palabras reservadas **CONFLICT** y **PATTERN** y por los símbolos ";" y **"$"** (o fin de cadena).

Existen dos tipos de excepciones; la primera de ellas se presenta cuando ocurren errores graves que le impiden al parser construir los objetos necesarios para el proceso de análisis, por ejemplo, si no existe el archivo que contiene la gramática del parser o dicho archivo se encuentra dañado. Lo mismo sería para la tabla de transiciones o para las acciones semánticas. Cuando ocurre una excepción de este tipo, esta se lanza inmediatamente y se aborta la ejecución del programa.

El segundo tipo de excepción se presenta cuando se realiza el análisis del archivo que contiene las oraciones del lenguaje de especificación de criterios de consistencia y se presenta un error léxico o sintáctico. Cuando ocurre una excepción de este tipo; se construye una excepción simple que contiene un código de error, el número de la línea donde ocurrió el error y la línea que contiene al error. Posteriormente, la excepción se agrega a un arreglo de excepciones y se continúa con el proceso del análisis. Finalmente, cuando el parser termina el proceso de análisis, se lanzan todas las excepciones contenidas en el arreglo de excepciones.

El Apéndice G muestra un listado con los códigos de error y una descripción de cada error. La Figura 5.15. ilustra el diagrama de clases correspondiente al manejo de excepciones. El primer tipo de excepción queda representado con la clase ***CTException*** (cooperative transaction exception) y el segundo tipo de excepción queda representado con las clases ***ParserException***, ***MultipleException*** y ***ParserExceptionArray***. La clase ***Exception*** es una clase perteneciente a las bibliotecas de clases de lenguaje Java.

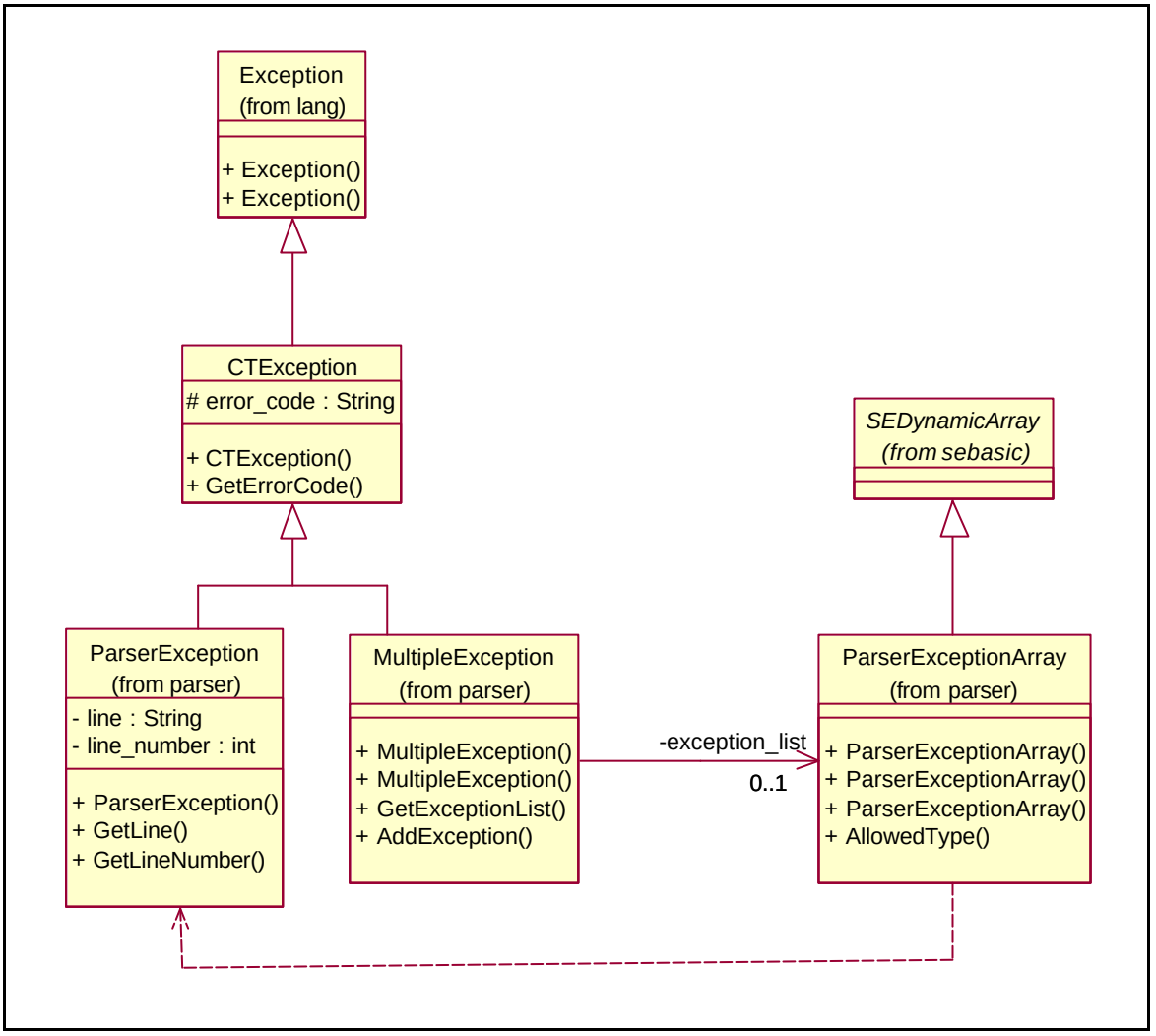


Figura 5.15. Diagrama de clases para manejo de excepciones.

5.9 Discusión

La figuras del presente capítulo se construyeron utilizando dos tipos de diagramas de UML: Los diagramas de clases y los diagramas de objetos. Este tipo de diagramas son los más comunes para el diseño orientado a objetos.

El paquete ***softengine,*** que se encuentra en el nivel mas alto de la jerarquía de paquetes, debe contener al conjunto de clases

y paquetes para poder implantar un ambiente de CASE cooperativo. El paquete ***ct*** (cooperative transactions) es el paquete que se integrará con las clases y paquetes para la implantación del sistema de transacciones cooperativas, es decir, contiene las clases y paquetes del Especificador de Criterios de Consistencia y contendrá las clases y paquetes del Verificador de Consistencia y del Monitor de Transacciones.

El sistema de especificación de criterios de consistencia se integra de los siguientes paquetes: ***grammar***, ***internalrepresentation***, ***transitiontable***, ***scanner***, ***parser*** y ***parseractions***.

La parte más interesante del proceso de diseño e implantación del sistema del presente trabajo, es el proceso de carga dinámica de las acciones semánticas del parser, ya que esto implica la aplicación de conceptos muy avanzados del lenguaje de programación Java y no vienen documentados en la literatura. Esto significa que tiempo de ejecución se carga la definición de una clase y se construyen los objetos correspondientes a la clase. Para este fin se realizó ingeniería inversa en algunos programas que realizan carga dinámica de administradores de dispositivos de cómputo y se estudió cada programa fuente obtenido en este proceso para tomar algunas ideas importantes y poderlas aplicar.

6 PERSPECTIVAS Y CONCLUSIONES

En éste capítulo se describen las cualidades representativas, que se tomaron en cuenta, para obtener la calidad en el desarrollo del sistema. También se describen las perspectivas del trabajo futuro. Por último se describen las conclusiones en base a la evaluación de resultados.

6.1 Calidad del Software Desarrollado.

Para el diseño e implantación del sistema se tomaron en cuenta cualidades representativas especificadas por la ingeniería de software que indican que un sistema de software, de buena calidad, debe ser: correcto, modular, portable, reusable, robusto, mantenible, eficiente y amigable. Estas cualidades representativas mantienen una analogía con las cualidades definidas en el ISO/IEC 9126-1 e incluso resultan ser una ampliación, ya que la modularidad y la reusabilidad no están contempladas en dicho estándar.

Correcto

Un sistema es correcto si se comporta de acuerdo a las especificaciones del análisis de requerimientos. El sistema se modeló apegándose a las especificaciones dadas en el análisis de requerimientos y a las definiciones planteadas para cada uno de sus componentes. En ocasiones esto no se ve tan claro porque se realizaron algunas variantes con el fin de usar de manera óptima los recursos computacionales. Se realizaron bastantes pruebas y siempre se obtuvieron los resultados esperados.

Modular

Se refiere al hecho de organizar el sistema en módulos lógicos o paquetes. Cada paquete agrupa un conjunto de clases que se encuentran fuertemente relacionadas ya sea conceptualmente, o por alguna relación de dependencia, herencia, agregación o asociación. Tomando en consideración estos aspectos se construyó para el sistema la jerarquía de paquetes ilustrada en la sección E.2 del Apéndice E.

Portable

Se refiere al hecho de hacer independiente al sistema de la plataforma o el equipo de cómputo en el que fue implantado, siendo posible implantarlo y ejecutarlo en otras plataformas o equipos de cómputo. En este trabajo se escogió el lenguaje Java por su portabilidad y se evitó el empleo de instrucciones no estándares del lenguaje de programación. El sistema se ejecutó con éxito en los siguientes equipos: Pc Pentium IV, Sun Sparc, Mac G4 y para las plataformas Solaris, Linux, Windows y MacOs.

Reusable

Se refiere al hecho de poder utilizar partes o todo el sistema para la construcción de un sistema nuevo. El sistema desarrollado en esta tesis puede utilizarse por cualquier sistema de traducción basado en gramáticas LL(1). Cabe aclarar que al momento de terminar este trabajo se encontró una forma de aplicarlo en la traducción de especificaciones hechas en el lenguaje XML a sentencias de SQL. El sistema también se puede aplicar en el desarrollo del verificador de consistencia del sistema de transacciones cooperativas.

Robusto

Se refiere al hecho de que el sistema se comporte razonablemente aún en circunstancias no anticipadas en la especificación de requerimientos. Para hacer cumplir esta propiedad se incorporó el módulo de manejo de excepciones.

Mantenible

Se refiere al hecho de poder hacerle modificaciones al sistema con poco trabajo, con la idea de ampliarlo para obtener una nueva versión o modificar su comportamiento. Para cumplir con esta cualidad se documentó muy bien el sistema con el apoyo de un CASE y se incorporó el polimorfismo por inclusión para poder agregar dinámicamente módulos al sistema sin tener que compilarlo nuevamente.

Eficiente

Se refiere al hecho de que el sistema use los recursos de cómputo adecuadamente y que los tiempos de respuesta sean razonables. Para hacer cumplir esta propiedad se usaron estructuras de datos dinámicas como por ejemplo: los arreglos

dinámicos y las secuencias, los cuales, reservan únicamente el espacio de memoria que necesitan. Se construyó el intérprete usando una analizador sintáctico predictivo, cuyo tiempo de respuesta es bastante reducido comparado con otros analizadores sintácticos.

Amigable

Se refiere al hecho de que el sistema sea fácil de usar por el usuario final. Para introducir los datos al sistema se usa la notación BNF por ser precisa, es decir, libre de ambigüedades. El sistema es muy simple de usar, sin embargo, un trabajo que tiene realizar el usuario final y que resulta complicado es la construcción de la gramática del parser y la definición de las acciones semánticas correspondientes.

6.2 Perspectivas

Por razones de productividad y simplicidad en las pruebas del sistema, las oraciones del lenguaje, para especificar los criterios de consistencia, se introdujeron por medio de un archivo de texto, quedaría pendiente ampliar el sistema desarrollando los módulos para introducir las oraciones a través de una interfaz gráfica de usuario o a través de una base de datos.

Los alcances definidos para este trabajo fue desarrollar el sistema especificador de criterios de consistencia para el sistema de transacciones cooperativas. Quedaría pendiente desarrollar en otros trabajos el verificador de consistencia y el monitor de transacciones.

Por razones de eficiencia, el parser del sistema de especificación de criterios de consistencia se construyó basándose en una gramática de tipo LL(1). Esto quiere decir que realiza un análisis sintáctico predictivo utilizando un solo símbolo en la entrada. Esto limita su reusabilidad. Sin embargo, este tipo de gramática modela perfectamente el sistema en cuestión. Puede desarrollarse un parser basado en una gramática de tipo LL(k), pudiendo reutilizarse en una mayor gama de aplicaciones, o bien, puede desarrollarse un parser basado en una gramática de tipo LR(k). Sin embargo, se estaría sacrificando la eficiencia. Se prefirió la eficiencia en lugar de la reusabilidad. Como sugerencia, puede

implantarse un parser para cada tipo de gramática y usar el polimorfismo por inclusión para determinar, a tiempo de ejecución, el parser que se utilizará.

6.3 Conclusiones

El intérprete puede reutilizarse en cualquier sistema de traducción basado en una gramática de tipo LL(1). Para construir el intérprete, se estudió la posibilidad de aplicar el patrón de diseño “Intérprete” pero el código generado hubiera sido bastante rígido y el sistema resultante no se podría reutilizar en otros sistemas de traducción. Aunado a esto, se tendrían que construir una gran cantidad de clases haciendo el sistema inmanejable. Valdría la pena revisar muy a fondo este sistema y estudiar la posibilidad de construir algunos patrones de diseño y de código nuevos.

El sistema es portable debido a que se utilizó Java como lenguaje de programación y existe una máquina virtual para interpretar este lenguaje en cualquier plataforma, además se evitó el empleo de instrucciones no estándares. Debido a que los programas en Java son interpretados por una máquina virtual tardan más en ejecutarse que un programa compilado en lenguaje C++. Pudo desarrollarse el sistema usando el lenguaje C++ ya que este lenguaje existe en todas las plataformas, sin embargo, resulta más difícil construir código portable afectando con esto la productividad. Se prefirió la productividad en lugar de la eficiencia.

La parte más complicada de implantar fue el proceso de carga dinámica de las acciones semánticas del parser, ya que esto implica la aplicación de conceptos muy avanzados del lenguaje de programación Java y no vienen documentados en la literatura. Para este fin se tuvo que realizar ingeniería inversa en algunos programas que realizan carga dinámica de administradores de dispositivos de cómputo y estudiar los programas fuente obtenidos en este proceso para tomar algunas ideas importantes y poderlas aplicar al presente trabajo.

La realización de este trabajo implicó el estudio a profundidad de varias áreas del conocimiento en ciencias de la computación como son: Bases de datos, Ingeniería de Software, Tecnologías orientadas a objetos, Teoría matemática de la computación, Compiladores y Lenguajes de programación.

APÉNDICE A. Ejemplo de una secuencia de operaciones para el Grupo de Transacción Desarrollo.

La primer columna de la tabla muestra las operaciones que se someten al GT, ordenadas cronológicamente de arriba hacia abajo. La segunda columna especifica la aceptación o rechazo de la operación. La tercer columna muestra el estado de análisis de cada una de las gramáticas relevantes a la operación. Para cada gramática relevante, se especifican los elementos que participan en el proceso del análisis. Un punto entre dos símbolos indica la posición del análisis. Si una operación se rechaza, se indica en esta columna solamente el análisis que causó el rechazo. Si la operación es aceptada, se indica el progreso del análisis en las gramáticas relevantes.

El subconjunto del prefijo de la historia, relevante a cada gramática, se puede deducir del análisis, leyendo los elementos de arriba hacia abajo, y considerando únicamente las partes del elemento que preceden al punto.

OPERACIÓN	**DECISIÓN**	**ANÁLISIS DE LAS GRAMÁTICAS RELEVANTES**
(*Oscar*, w, *IGU*)	Rechazar	C_3: S::=(*Oscar, w, IGU*) .
(*Oscar*, r, *IGU*)	Aceptar	C_3: S::=(*Oscar, r, IGU*) **.** A P_3: S::=(*Oscar, r, IGU)* . A
(*Oscar*, w, *IGU*)	Aceptar	C_1: S::=(*any,w,IGU*) **.** A C_3: S::=(*Oscar, r, IGU*) **.** A A::=(*Oscar, r/w, IGU*) **.** A P_1: S::=(*any,w,IGU*) **.** A P_3: S::=(*Oscar, r, IGU)* . A A::=(*Oscar, r/w, IGU*) **.** A
(*Pruebas*, r, *IGU*)	Aceptar	C_1: S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B C_4: S::=(*Pruebas, r, IGU*) **.** A P_1: S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B P_4: S::=(*Pruebas, r, IGU)* . A
(*Pruebas*, inc, *cont_pru*)	Aceptar	P_2: S::= **.** B B::= **.** A **...** A::=(*any,inc,cont_pru*) **....**
(*Pruebas*, r, *Reporte*)	Aceptar	C_5: S::=(*Pruebas, r, Reporte*) **.** A P_5: S::=(*Pruebas, r, Reporte)* . A
(*Pruebas*, w, *Reporte*)	Aceptar	C_1: S::=(*any,w,IGU*) **.** A

		A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S C_5: S::=(*Pruebas, r, Reporte*) **.** A A::=(*Pruebas, r/w, Reporte*) **.** A P_1: S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S P_5: S::=(*Pruebas, r, Reporte)* . A A::=(*Pruebas, r/w, Reporte*) **.** A
(*Oscar*, w, *IGU*)	Aceptar	C_1: S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S S::=(*any,w,IGU*) **.** A C_3: S::=(*Oscar, r, IGU*) **.** A A::=(*Oscar, r/w, IGU*) **.** A A::=(*Oscar, r/w, IGU*) **.** A C_4: S::=(*Pruebas, r, IGU*) **.** A A::=(*-Pruebas, w, IGU*) **.** S P_1: S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S S::=(*any,w,IGU*) **.** A P_3: S::=(*Oscar, r, IGU)* . A A::=(*Oscar, r/w, IGU*) **.** A A::=(*Oscar, r/w, IGU*) **.** A P_4: S::=(*Pruebas, r, IGU)* . A A::=(*-Pruebas, w, IGU*) **.** S
(*Oscar*, dec, *cont_pru*)	Aceptar	P_2: S::= **.** B B::= **.** A **...** A::=(*any,inc,cont_pru*) (*any,dec,cont_pru*) **.**
(*Pruebas*, r, *IGU*)	Aceptar	C_1: S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B C_4: S::=(*Pruebas, r, IGU*) **.** A A::=(*-Pruebas, w, IGU*) **.** S S::=(*Pruebas, r, IGU*) **.** A P_1: S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B P_4: S::=(*Pruebas, r, IGU)* . A A::=(*-Pruebas, w, IGU*) **.** S

		S::=(*Pruebas, r, IGU*) **.** A
(*Pruebas*, w, *Reporte*)	Aceptar	C_1: S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S C_5: S::=(*Pruebas, r, Reporte*) **.** A A::=(*Pruebas, r/w, Reporte*) **.** A A::=(*Pruebas, r/w, Reporte*) **.** A P_1: S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S S::=(*any,w,IGU*) **.** A A::=(*Pruebas,r,IGU*) **.** B B::=(*Pruebas*,w,*Reporte*) **.** S P_5: S::=(*Pruebas, r, Reporte)* . A A::=(*Pruebas, r/w, Reporte*) **.** A A::=(*Pruebas, r/w, Reporte*) **.** A

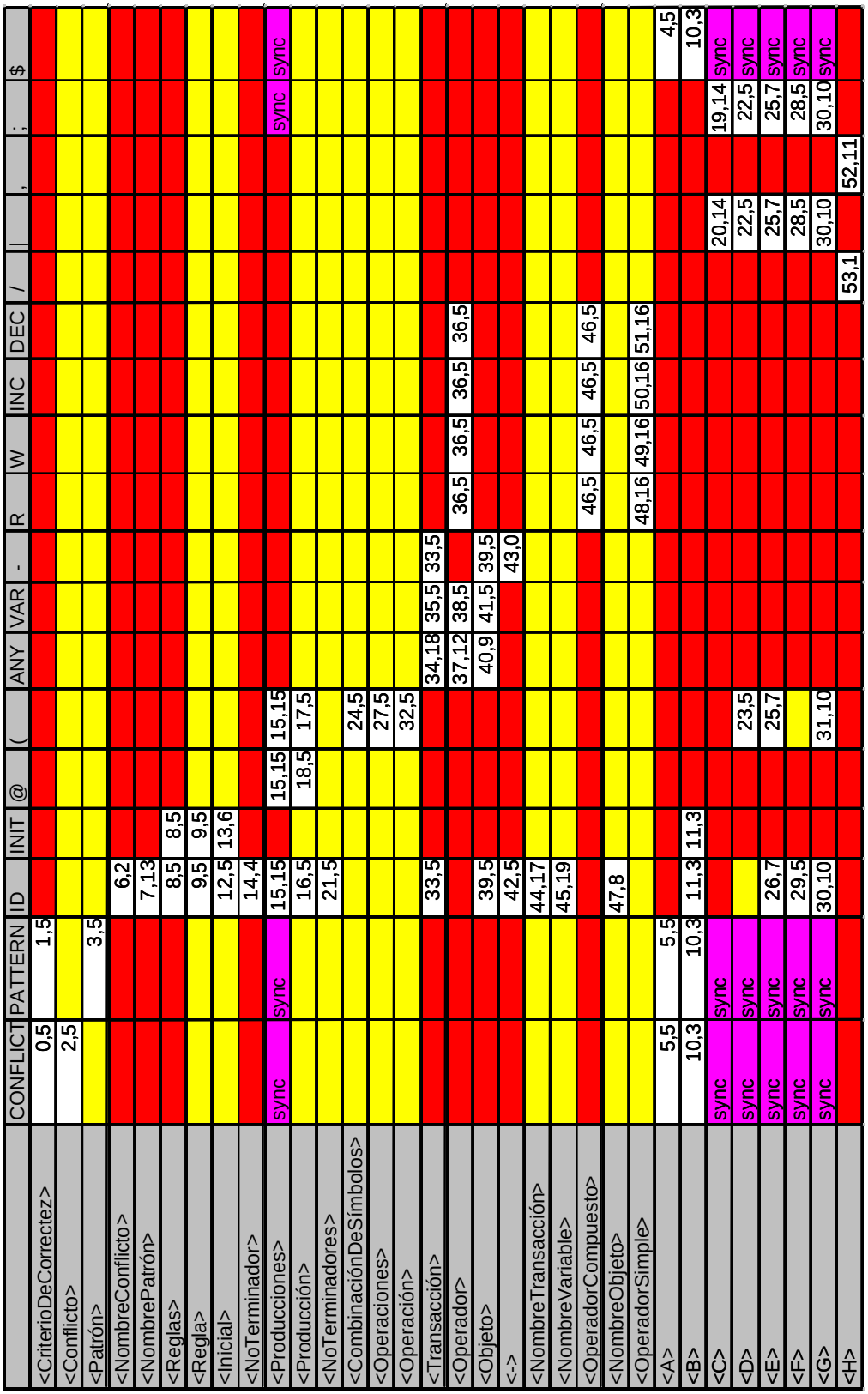
APÉNDICE B. Descripción en BNF de la gramática para la especificación de los criterios de consistencia.

```
<CriterioDeCorrectez>::=<Conflicto>|<Patrón>|
          <Conflicto><CriterioDeCorrectez>|
          <Patrón><CriterioDeCorrectez>
<Conflicto>::=CONFLICT <NombreConflicto>:<Reglas>
<Patrón>::=PATTERN <NombrePatrón>:<Reglas>
<Reglas>::=<Regla>|<Regla><Reglas>
<Regla>::=[INIT]<NoTerminador> <- <Producciones>;
<Producciones>::=    <Producción>
                     |<Producción><Vertical><Producciones>
<Producción>::=<NoTerminadores>
               |<CombinaciónDeSímbolos>
               |<NoTerminadores><CombinaciónDeSímbolos>
               |<Nada>
<CombinaciónDeSímbolos>::=
     <Operaciones>
     |<Operaciones><NoTerminadores>
     |<Operaciones><NoTerminadores><CombinaciónDeSímbolos>
<Operaciones>::=<Operación>|<Operación><Operaciones>
<NoTerminadores>::=  <NoTerminador>
                     |<NoTerminador><NoTerminadores>
<Operación>::=(<Transacción>,<Operador>,<Objeto>)
<NoTerminador>::=    <Identificador>
<NombrePatrón>::=    <Identificador>
<NombreConflicto>::=<Identificador>
<Transacción>::=[-]<NombreTransacción>|ANY|
                VAR[-]<NombreVariable>
<Operador>::=<OperadorCompuesto>|ANY|VAR[-]<NombreVariable>
<Objeto>::=[-]<NombreObjeto>|ANY|VAR[-]<NombreVariable>
<Vertical>::="|"
<Nada>::="e"
<NombreTransacción>::=<Identificador>
<NombreVariable>::=<Identificador>
<OperadorCompuesto>::= <OperadorSimple>|
     <OperadorSimple>/<OperadorCompuesto>
<OperadorSimple>::= R|W|INC|DEC
<Identificador>::=   <Letra>|<Letra><Alfanumerico>
<Alfanumerico>::=    <Letra>
                     |<Dígito>
                     |<Letra><Alfanumerico>
                     |<Dígito><Alfanumerico>
<Letra>::=<Mayúscula>|<Minúscula>
<Mayúscula>::=
     A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<Minúscula>::=
     a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<Dígito>::=0|1|2|3|4|5|6|7|8|9
```

APÉNDICE C. Descripción en BNF de la gramática factorizada y resumida para la especificación de los criterios de consistencia.

```
<CriterioDeCorrectez>::=<Conflicto> <A>|<Patrón><A>
<Conflicto>::=CONFLICT <NombreConflicto>:<Reglas>
<Patrón>::=PATTERN <NombrePatrón>:<Reglas>
<A>::= e |<CriterioDeCorrectez>
<NombreConflicto>::=ID
<NombrePatrón>::=ID
<Reglas>::=<Regla><B>
<Regla>::=<Inicial><NoTerminador> <- <Producciones>;
<B>::= e |<Reglas>
<Inicial>::= e | INIT
<NoTerminador>::= ID
<Producciones>::=<Producción><C>
<Producción>::=<NoTerminadores><D>
                |<CombinaciónDeSímbolos>
                |NADA
<C>::=e |VERTICAL<Producciones>
<NoTerminadores>::=<NoTerminador><E>
<D>::=e |<CombinaciónDeSímbolos>
<CombinaciónDeSímbolos>::= <Operaciones><F>
<E>::=e |<NoTerminadores>
<Operaciones>::=<Operación><G>
<F>::=e |<NoTerminadores><D>
<G>::= e |<Operaciones>
<Operación>::=(<Transacción>,<Operador>,<Objeto>)
<Transacción>::=<->><NombreTransacción>|ANY|
                VAR<->><NombreVariable>
<Operador>::=  <OperadorCompuesto>|ANY|
                VAR<->><NombreVariable>
<Objeto>::=<->><NombreObjeto>|ANY|VAR<->><NombreVariable>
<->::=e |-
<NombreTransacción>::=ID
<NombreVariable>::=ID
<OperadorCompuesto>::= <OperadorSimple><H>
<NombreObjeto>::=ID
<OperadorSimple>::= R|W|INC|DEC
<H>::= e | / <OperadorCompuesto>
```

APÉNDICE D. Tabla de transiciones del Parser.



	CONFLICT	PATTERN	ID	INIT	@	(	ANY	VAR	-	R	W	INC	DEC	/	|	,	;	$
<CriterioDeCorrectez>	0,5	1,5																
<Conflicto>	2,5																	
<Patrón>		3,5																
<NombreConflicto>			6,2															
<NombrePatrón>			7,13															
<Reglas>			8,5	8,5														
<Regla>			9,5	9,5														
<Inicial>			12,5	13,6														
<NoTerminador>			14,4															
<Producciones>		sync	15,15		15,15	15,15											sync	sync
<Producción>			16,5		18,5	17,5												
<NoTerminadores>			21,5															
<CombinaciónDeSímbolos>						24,5												
<Operaciones>						27,5												
<Operación>						32,5												
<Transacción>			33,5				34,18	35,5	33,5									
<Operador>							37,12	38,5		36,5	36,5	36,5	36,5					
<Objeto>			39,5				40,9	41,5	39,5									
<->			42,5						43,0									
<NombreTransacción>			44,17															
<NombreVariable>			45,19															
<OperadorCompuesto>										46,5	46,5	46,5	46,5					
<NombreObjeto>			47,8															
<OperadorSimple>										48,16	49,16	50,16	51,16					
<A>	5,5	5,5																4,5
<B>	10,3	10,3	11,3	11,3														10,3
<C>	sync	sync													20,14		19,14	sync
<D>	sync	sync				23,5									22,5		22,5	sync
<E>	sync	sync	26,7			25,7									25,7		25,7	sync
<F>	sync	sync	29,5												28,5		28,5	sync
<G>	sync	sync	30,10			31,10									30,10		30,10	sync
<H>														53,1		52,11		

APÉNDICE E. Archivos del sistema.

E.1. Archivos de datos

ActionTable.txt Contiene la tabla de transiciones en un formato tipo texto de 4 columnas. La Primer columna es un número entero asociado a la posición de un símbolo no terminal dentro del archivo NonTerminators.txt. La Segunda columna es el nombre de un símbolo terminal especificado en el archivo Terminators.txt. La Tercera columna es un número entero asociado a la posición de una regla dentro del archivo Rules.txt. La Cuarta columna es un número entero asociado a la acción asociada a la regla que se aplicará como resultado de la transición de estado. Las acciones se cargarán dinámicamente utilizando el nombre de archivo proporcionado por el sistema**.**

Contenido del archivo			
NT	**Terminador**	**Regla**	**Acción**
0	CONFLICT	0	5
0	PATTERN	1	5
1	CONFLICT	2	5
2	PATTERN	3	5
3	ID	6	2
4	ID	7	13
5	ID	8	5
5	INIT	8	5
6	ID	9	5
6	INIT	9	5
7	ID	12	5
7	INIT	13	6
8	ID	14	4
9	ID	15	15
9	@	15	15
9	(	15	15
10	ID	16	5
10	@	18	5
10	(	17	5
11	ID	21	5
12	(	24	5
13	(	27	5
14	(	32	5
15	ID	33	5
15	ANY	34	18
15	VAR	35	5
15	-	33	5
16	ANY	37	12
16	VAR	38	5
16	R	36	5
16	W	36	5
16	INC	36	5
16	DEC	36	5
17	ID	39	5
17	ANY	40	9
17	VAR	41	5
17	-	39	5
18	ID	42	5
18	-	43	0
19	ID	44	17
20	ID	45	19
21	R	46	5
21	W	46	5
21	INC	46	5
21	DEC	46	5
22	ID	47	8
23	R	48	16
23	W	49	16
23	INC	50	16
23	DEC	51	16
24	CONFLICT	5	5
24	PATTERN	5	5
24	$	4	5
25	CONFLICT	10	3
25	PATTERN	10	3
25	ID	11	3
25	INIT	11	3
25	$	10	3
26	|	20	14
26	;	19	14
27	(	23	5
27	|	22	5
27	;	22	5
28	ID	26	7
28	(	25	7
28	|	25	7
28	;	25	7
29	ID	29	5
29	|	28	5
29	;	28	5
30	ID	30	10
30	(	31	10
30	|	30	10
30	;	30	10
31	/	53	1
31	,	52	11

NonTerminators.txt Contiene una lista con los símbolos no terminales de la gramática utilizada por el Parser y la tabla de transiciones. El símbolo de inicio deberá estar en la primera línea.

No. Línea	Contenido del Archivo
0	<CriterioDeCorrectez>
1	<Conflicto>
2	<Patrón>
3	<NombreConflicto>
4	<NombrePatrón>
5	<Reglas>
6	<Regla>
7	<Inicial>
8	<NoTerminador>
9	<Producciones>
10	<Producción>
11	<NoTerminadores>
12	<CombinaciónDeSímbolos>
13	<Operaciones>
14	<Operación>
15	<Transacción>
16	<Operador>
17	<Objeto>
18	<->
19	<NombreTransacción>
20	<NombreVariable>
21	<OperadorCompuesto>
22	<NombreObjeto>
23	<OperadorSimple>
24	<A>
25	<B>
26	<C>
27	<D>
28	<E>
29	<F>
30	<G>
31	<H>

reglas.txt Contiene frases del lenguaje utilizado para la especificación de los criterios de consistencia, es decir, contiene una lista con los criterios de consistencia.

Contenido del archivo

```
PATTERN P1:
INIT S<-(ANY,W,IGU)A|@;
A<-(ANY,W, IGU)A|(Pruebas,R, IGU)B;
B<-(Pruebas,W,Reporte)S;
CONFLICT C1:
INIT S<-(ANY,W, IGU)A;
A<-(ANY,W, IGU)A|(Pruebas,R, IGU)B;
B<-(ANY,W, IGU)|(Pruebas,W, Reporte)S;
PATTERN P2:
INIT S<-B;
```

```
B<-A B|A;
A<-(ANY,INC,cont_pru)B(ANY,DEC,cont_pru)|
  (ANY,INC, cont_pru)(ANY,DEC,cont_pru);
PATTERN Ps:
INIT S<-(VAR M,R,VAR O)A|@;
A<-(VAR -M,W,VAR O)S|(VAR M,R/W,VAR O)|@;
CONFLICT Cs:
INIT S<-(VAR M,W,VAR O)|(VAR M,R,VAR O)A;
A<-(VAR -M,W,VAR O)S|(VAR M,R/W,VAR O)A;
```

Rules.txt Contiene una lista con las reglas de la gramática utilizada por el Parser y la tabla de transiciones. Las reglas tienen que estar en BNF.

No. Línea	Contenido del Archivo
0	<CriterioDeCorrectez> ::= <Conflicto> <A>
1	<CriterioDeCorrectez> ::= <Patrón> <A>
2	<Conflicto> ::= CONFLICT <NombreConflicto> : <Reglas>
3	<Patrón> ::= PATTERN <NombrePatrón> : <Reglas>
4	<A> ::= $
5	<A> ::= <CriterioDeCorrectez>
6	<NombreConflicto> ::= ID
7	<NombrePatrón> ::= ID
8	<Reglas> ::= <Regla> <B>
9	<Regla> ::= <Inicial> <NoTerminador> < - <Producciones> ;
10	<B> ::= $
11	<B> ::= <Reglas>
12	<Inicial> ::= $
13	<Inicial> ::= INIT
14	<NoTerminador> ::= ID
15	<Producciones> ::= <Producción> <C>
16	<Producción> ::= <NoTerminadores> <D>
17	<Producción> ::= <CombinaciónDeSímbolos>
18	<Producción> ::= @
19	<C> ::= $
20	<C> ::= | <Producciones>
21	<NoTerminadores> ::= <NoTerminador> <E>
22	<D> ::= $
23	<D> ::= <CombinaciónDeSímbolos>
24	<CombinaciónDeSímbolos> ::= <Operaciones> <F>
25	<E> ::= $
26	<E> ::= <NoTerminadores>
27	<Operaciones> ::= <Operación> <G>
28	<F> ::= $
29	<F> ::= <NoTerminadores> <D>
30	<G> ::= $
31	<G> ::= <Operaciones>
32	<Operación> ::= (<Transacción> , <Operador> , <Objeto>)
33	<Transacción> ::= <-> <NombreTransacción>
34	<Transacción> ::= ANY
35	<Transacción> ::= VAR <-> <NombreVariable>
36	<Operador> ::= <OperadorCompuesto>
37	<Operador> ::= ANY
38	<Operador> ::= VAR <-> <NombreVariable>
39	<Objeto> ::= <-> <NombreObjeto>
40	<Objeto> ::= ANY
41	<Objeto> ::= VAR <-> <NombreVariable>
42	<-> ::= $
43	<-> ::= -
44	<NombreTransacción> ::= ID
45	<NombreVariable> ::= ID
46	<OperadorCompuesto> ::= <OperadorSimple> <H>
47	<NombreObjeto> ::= ID
48	<OperadorSimple> ::= R
49	<OperadorSimple> ::= W
50	<OperadorSimple> ::= INC
51	<OperadorSimple> ::= DEC
52	<H> ::= $
53	<H> ::= / <OperadorCompuesto>

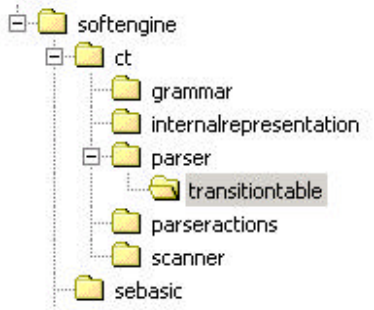
Terminators.txt Contiene una lista con los símbolos terminales de la gramática utilizada por el Parser y la tabla de transiciones. Para los identificadores se utilizara la palabra ID.

Contenido del Archivo
CONFLICT
PATTERN
ID
INIT
@
(
ANY
VAR
-
R

W
INC
DEC
/
|
,
;
:
<
)

E.2. Programas fuentes (diccionario de clases del sistema)

Jerarquía de los paquetes



Paquete **ct:** (cooperative transactions) Conjunto de clases y paquetes que participan en el sistema de transacciones cooperativas.

CTException.java	Cooperative Transaction Exception. Superclase principal para el sistema de transacciones cooperativas. Extiende a la clase Exception agregando un código de error.

Paquete **grammar**: Conjunto de clases que integran una gramática.

Grammar.java	Clase para el manejo de Gramaticas.
NonTerminator.java	Representa a un símbolo no terminal.
NTSequence.java	Clase para el manejo de secuencias de símbolos no terminales.
NTSequenceIterator.java	Iterador que recorre los elementos de una secuencia de símbolos no terminales.
Rule.java	Representa a una regla de la gramática.
RulesArray.java	Arreglo dinámico de reglas.
Symbol.java	Representa a un símbolo que puede ser terminal o no terminal.
SymbolSeqIterator.java	Iterador que recorre los elementos de una secuencia de símbolos.
SymbolSequence.java	Clase para el manejo de secuencias de simbolos.
Terminator.java	Representa a un símbolo terminal.
TSequence.java	Clase para el manejo de secuencias de símbolos terminales.
TSequenceIterator.java	Iterador que recorre los elementos de una secuencia de símbolos terminales.

Paquete **internalrepresentation**: Conjunto de clases utilizadas para la representación interna de los criterios de consistencia.

Conflict.java	Representa un criterio de consistencia para un conflicto.
CorrectnessCriteria.java	Representa una lista con los criterios de consistencia.
Criterium.java	Representa un criterio de consistencia que puede ser un patrón o un conflicto.
CTAnyOperator.java	Representa el operador comodín any. Este operador representa cualquie cosa. La comparación de cualquier operador con el operador any es verdadera.
CTCompositeOperator.java	Cooperative Transaction Composite Operator. Representa un operador compuesto, es decir, es una lista de operadores. Si un operador simple se compara con un operador compuesto, el resultado será verdadero si el operador simple es un elemento del operador compuesto.
CTCriteriaSeqIterator.java	Iterador para recorrer los elementos de una secuencia de criterios de consistencia.
CTCriteriaSequence.java	Clase para el manejo de secuencias de criterios de consistencia.
CTDBOperationValue.java	Cooperative Transaction Data Base Operation Value. Representa una operación de una transacción cooperativa.
CTIdentifier.java	Representa a un identificador que forma parte de una operación para una transacción cooperativa.
CTOperationArgument.java	Representa a un argumento de una operación para una transacción cooperativa.
CTOperator.java	Representa a un operador (simple o compuesto) de una operación para una transacción cooperativa.
CTSimpleOperator.java	Representa un operador simple de una operación para una transacción cooperativa.
CTSimpleOperatorArray.java	Arreglo dinámico de operadores simples.
CTVariable.java	Representa a una variable que forma parte de una operación para una transacción cooperativa.
Pattern.java	Representa un criterio de consistencia para un patrón.

Paquete **parser**: Contiene el parser y el conjunto de excepciones que pueden generarse durante el análisis sintáctico.

MultipleException.java	Denota al conjunto de excepciones que se generaron durante el proceso del análisis léxico.
Parser.java	Realiza el análisis léxico, sintáctico y la traducción.
ParserException.java	Extiende la clase CTException agregando un número de línea y la línea en donde ocurre un error en el análisis sintáctico.
ParserExceptionArray.java	Arreglo dinámico de excepciones.

Paquete **parseractions**: Contiene al conjunto de clases que representan las acciones semáticas del parser.

ComplementAction.java	Acción que se ejecuta cuando reconoce al operador de complemento denotado por el símbolo "-".
CompositeOperatorAction.java	Acción que se ejecuta cuando reconoce a un operador compuesto. Un operador compuesto es una lista de operadores simples separados por el símbolo "/".
ConflictAction.java	Acción que se ejecuta cuando se reconoce un conflicto.
CTActionsLoader.java	Carga a memoria las acciones semánticas del parser.
GrammarAction.java	Acción que se ejecuta cuando se reconoce una gramática de un patrón o de un conflicto.
NonTerminatorAction.java	Acción que se ejecuta cuando se reconoce un símbolo no terminal.
Nothing.java	Acción que representa no hacer nada.
NTInitialAction.java	Acción que se ejecuta cuando se reconoce el símbolo INIT.
NTListAction.java	Acción que se ejecuta cuando se reconoce una secuencia de símbolos no terminales.
ObjectAction.java	Acción que se ejecuta cuando se reconoce al argumento Object como parte de una operación.

ObjectActionAny.java	Acción que se ejecuta cuando se reconoce al comodín any, en el lugar del argumento Object de una operación.
OperationAction.java	Acción que se ejecuta cuando se reconoce una operación.
OperatorAction.java	Acción que se ejecuta cuando se reconoce un operador.
OperatorActionAny.java	Acción que se ejecuta cuando se reconoce al comodín any, en el lugar del argumento Operator de una operación.
PatternAction.java	Acción que se ejecuta cuando se reconoce un patrón.
ProductionAction.java	Acción que se ejecuta cuando se reconoce el antecedente de una regla.
RuleAction.java	Acción que se ejecuta cuando se reconoce una regla.
SimpleOperatorAction.java	Acción que se ejecuta cuando se reconoce a un operador simple.
TransactionAction.java	Acción que se ejecuta cuando se reconoce al argumento Transaction como parte de una operación.
TransactionActionAny.java	Acción que se ejecuta cuando se reconoce al comodín any, en el lugar del argumento Transaction de una operación.
VariableAction.java	Acción que se ejecuta cuando se reconoce una variable como un argumento de una operación.

Paquete **transitiontable**: Conjunto de clases que componen la tabla de transiciones.

ActionArray.java	Arreglo dinámico que contiene las acciones semánticas del parser.
ActionItem.java	Representa una acción semántica.
ActionsLoader.java	Contiene un arreglo que almacena las acciones semánticas del parser y es de utilidad para cargar dinámicamente dichas acciones.
LRTable.java	Left Right Table. Tabla de transiciones.
Transition.java	Representa una transición de estado en la tabla de transiciones.
TransitionArray.java	Arreglo dinámico de transiciones de estado.
TransitionState.java	Representa un conjunto de transiciones de estado.
TransitionStateArray.java	Arreglo dinámico que contiene un conjunto de arreglos de transiciones de estado.

Paquete **sebasic**: Paquete perteneciente a la compañía softengine, desarrollado por Cristóbal Juárez Castellanos. El uso de estas clases fue con la autorización del autor. Contiene un conjunto de clases basicas.

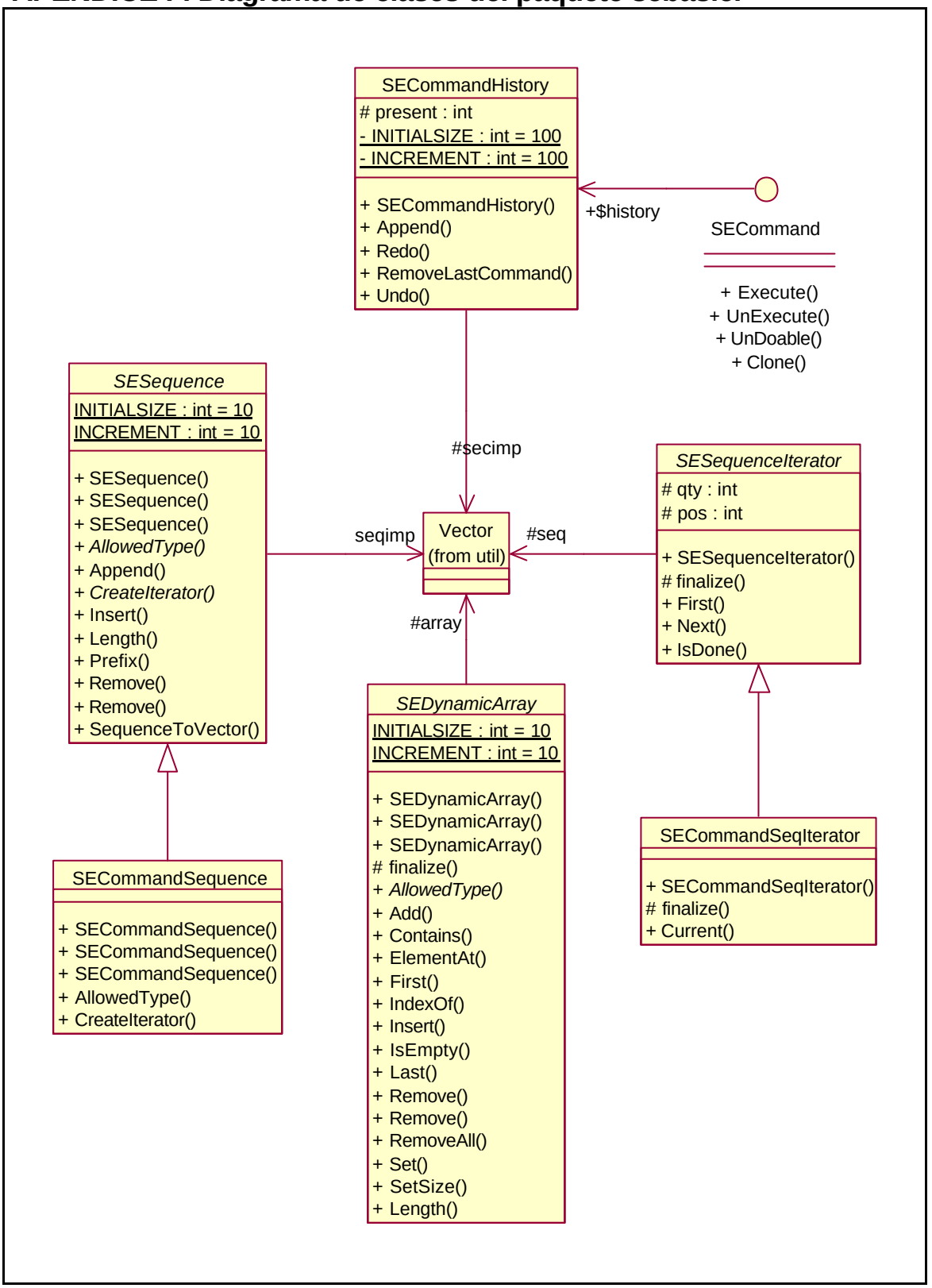
SECommand.java	Interfaz para el manejo de comandos o acciones.
SECommandHistory.java	Clase para el manejo de historias de comandos, esta clase almacena los comandos ejecutados e incorpora las funciones para deshacer o reconstruir un comando.
SECommandSeqIterator.java	Iterador para recorrer los elementos de una secuencia de comandos.
SECommandSequence.java	Clase contenedora de una secuencia de comandos.
SEDynamicArray.java	Clase contenedora para manejo de arreglos dinámicos.
SESequence.java	Clase contenedora para manejo de secuencias.
SESequenceIterator.java	Iterador para recorrer los elementos de una secuencia.

Paquete **scanner:** Paquete que contiene las clases que colaboran en el análisis léxico.

Scanner.java	Clase principal del analizador léxico.
ScannerBuffer.java	Interfaz para el manejo del Buffer de entrada del Analizador Léxico.
ScannerFileBuffer.java	Clase que implementa el método GetInputLine() de la interfaz ScannerBuffer para que la fuente de entrada del analizador léxico provenga de un archivo.

Paquete **softengine**: Paquete perteneciente a la compañía del mismo nombre y es la raíz en la jerarquía de paquetes del sistema objeto de esta tesis. Este paquete se originó con la idea principal del desarrollo de un CASE cooperativo.

APÉNDICE F. Diagrama de clases del paquete sebasic.



APÉNDICE G. Mensajes de error.

CÓDIGO	MENSAJE
CT-00001	El valor del parser es nulo. No existe la gramática para generar la tabla de transiciones. La excepción es lanzada en el constructor del la tabla de transiciones. LRTable es el nombre de la clase para la tabla de transiciones.
CT-00002	El valor de la gramática es nulo. No se puede generar la tabla de transiciones. La excepción es lanzada en el constructor del la tabla de transiciones. LRTable es el nombre de la clase para la tabla de transiciones.
CT-00003	La gramática carece de símbolos no terminales. La tabla de transiciones no contiene trancisiones de estado. La excepción es lanzada en el constructor del la tabla de transiciones. LRTable es el nombre de la clase para la tabla de transiciones.
CT-00004	La fuente de datos para construir la tabla de transición debe tener el formato: "No Terminal" "Terminal" "Regla" "Acción" por ejemplo: 0 CONFLICT 0 5 "No Terminal" debe ser un número entero asociado al orden secuencial de un símbolo no terminal dentro de la gramática. "Terminal" debe ser el nombre de un símbolo terminal de la gramática. "Regla" debe ser un número entero asociado al orden secuencial de una regla dentro de la gramática. "Acción" debe ser un número entero asociado al orden secuencial de la acción semántica que se debe de ejecutar como resultado de la transición de estado.
CT-00005	El índice del símbolo terminal X, de la fuente de datos para construir la tabla de transición, esta fuera del rango permitido de [0..N].
CT-00006	El símbolo terminal, de la fuente de datos para construir la tabla de transición, no se encuentra en la lista de los símbolos terminales de la gramática.
CT-00007	El índice de la regla gramatical X, de la fuente de datos para construir la tabla de transición, esta fuera del rango permitido de [0..N].
CT-00008	El índice de la acción semántica X", de la fuente de datos para construir la tabla de "transición, esta fuera del rango permitido de [0 .. N]
CT-00009	La fuente de datos para construir la tabla de transición presenta múltiples errores.

REFERENCIAS

[AIS77] Christopher Alexander, Sara Ishikawa, Murray Silverstein. A Pattern Language. Oxford University Press, 1977.

[ASU98] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compiladores principios, técnicas y herramientas. Addison Wesley, 1998

[Boo94] Grady Booch, Object-Oriented Analysis and Design with Applications, Second Edition, Benjamin/Cummings, 1994.

[BRJ99] Grady Booch, James Rumbaugh, Ivar Jacobson. The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[Bro93] J. Glenn Brookshear. Teoría de la Computación Lenguajes Formales, Autómatas y Complejidad. Addison Wesley, 1993

[CW86] Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, Vol 17, No.4, 471-522, 1986.

[Elm95] Ahmed K. Elmagarmid. Database Transaction Models for Advanced Aplications, Morgan Kaufman, 1995

[GHJV94]Erich Gamma, Richard Helm, Ralph Johnson, Jhon Vlissides. Design Patterns. Professional Computing. Addison-Wesley,1994.

[Quat01] Terry Quatrani. Visual Modeling with Rational Rose 2000 and UML. Addison Wesley, 2001

[Ska89] Andrea H. Skarra. Concurrency control for cooperating transactions in an object oriented database. *SIGPLAN Notices*,24(4) April 1989

[Ska9l] Andrea H. Skarra. Localized correctness specifica-tions for cooperating transactions in an object oriented database. *Office Knowledge Engineering*, (1):79-106,1991.

[TA83] Aaron M. Tenenbaum, Moshe J. Augenstein. Estructura de Datos en Pascal. Prentice Hall, 248-258, 1983.

[Won95] Modern database systems: the object model, interoperability, and beyond, 409 - 433, Won Kim Editor, 1995.

BIBLIOGRAFÍA

[BRJ$_1$99] Booch G., Rumbaugh J., Jacobson I.,, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.

[BRJ$_2$99] Booch G., Rumbaugh J., Jacobson I., The Unified Software Development Process, Addison-Wesley, 1999.

[Coa95] Coad P. at. al., Object Models, Strategies, Patterns and Applications, Yourdon Press Computing Series, Prentice Hall, 1995

[CR98] Patrick Chan, Rossana Lee, The Java Class Libraries Second Edition, Volume 2, Addison-Wesley, 1998.

[CRK98] Patrick Chan, Rossana Lee, Douglas Kramer, The Java Class Libraries Second Edition, Volume 1, Addison-Wesley, 1998.

[CRK99] Patrick Chan, Rossana Lee, Douglas Kramer, The Java Class Libraries Second Edition, Volume 1, Supplement for the Java 2 Platform Standard Edition, V. 1.2, Addison-Wesley, 1999.

[Lar98] Larman C., Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design, Prentice Hall 1998.

[www03] www.cs.queensu.ca/Software-Engineering/tools.html Index of CASE tools, 2003.