



UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA

HERRAMIENTA CASE PARA LA GENERACIÓN DE CÓDIGO C++ A PARTIR DE DIAGRAMAS DE CLASE UML

TESIS PARA OBTENER EL TÍTULO DE:
INGENIERO EN COMPUTACIÓN

PRESENTA:

IRVING ALBERTO CRUZ MATÍAS

DIRECTOR DE TESIS:

M.C. CARLOS ALBERTO FERNÁNDEZ Y FERNÁNDEZ

Huajuapán de León, Oaxaca. Julio de 2003

Dedicatoria

A Dios

... por darme la oportunidad de existir.

A mis padres

... por darme la oportunidad de ser alguien en la vida.

A mi hermano Julio Cesar

...por su fraternidad.

A Sayde

...con quien comparto este momento de alegría y orgullo.

Agradecimientos

Agradezco a todos mis profesores de la Universidad Tecnológica de la Mixteca por sus conocimientos aportados y ejemplos de profesionalidad que me servirán para desempeñar orgullosamente mi carrera.

A mi asesor M.C. Carlos Alberto Fernández y Fernández por su confianza al aceptarme para llevar a cabo este proyecto, y más que nada por su guía y enorme paciencia.

A las familias Cruz y Matías quienes directa o indirectamente contribuyeron al desarrollo de esta tesis.

Y a todos aquellos que hicieron posible la elaboración de este trabajo.

Muchas gracias...

Resumen

Las herramientas CASE (*Computer Aided Software Engineering* – Ingeniería de Software Asistida por Computadora) nacen para auxiliar a los desarrolladores de software, lo que permite el apoyo computarizado en todo o en parte del ciclo de vida del desarrollo de un sistema de software, tal es el caso de las herramientas CASE con soporte a UML (*Unified Modeling Language* – Lenguaje Unificado de Modelado).

Hoy en día existen una gran cantidad de estas herramientas, por mencionar algunas: Rose de Rational Software Corporation, Rhapsody de I-logix, y Visual UML de Object Modelers. Todas ellas pretenden reducir significativamente el trabajo de un desarrollador de software, sin embargo, el adquirir una de estas herramientas tiene como inconveniente para muchas personas, un precio elevado, aunado a la dificultad de aprendizaje, debido a la cantidad de opciones y configuraciones que presentan, sobre todo si se quiere obtener el máximo beneficio a las funciones importantes, como la generación de código fuente o de estructura de datos.

En una institución educativa como la Universidad Tecnológica de la Mixteca, se tendrían que adquirir varias licencias para el uso de una herramienta CASE, lo que implicaría un costo significativo de dinero; por tal razón, el presente proyecto de tesis plantea el desarrollo de una herramienta CASE para generación de código C++ a partir de diagramas de clase UML, para apoyo de aquellos que estén iniciando con el aprendizaje del Análisis y Diseño Orientado a Objetos.

La herramienta CASE propuesta, a la que se llamó UMLGEC++ (acrónimo obtenido de la conjugación de las palabras UML, generador, código y C++) tiene como objetivo principal, proveer una herramienta CASE que sirva para material de apoyo a la materia de Programación Orientada Objetos II, impartida en la Universidad Tecnológica de la Mixteca.

Con UMLGEC++ se pretende:

- Ayudar en el proceso de análisis y diseño.
- Mejorar la calidad de los desarrollos.
- Mejorar la productividad.
- Impulsar al programador principiante a hacer uso de buenas prácticas de programación como la declaración de funciones *inline* y uso de la STL de C++.
- Mostrar el uso del UML y el UP (*Unified Process* - Proceso Unificado) para el desarrollo de sistemas.

El capítulo 1 da una introducción del proyecto, el problema a resolver y la justificación de la tesis. El capítulo 2 habla de las características de las herramientas CASE, ventajas y desventajas de usarlas, y sus principales componentes. El capítulo 3 es una introducción a UML, se explican las partes utilizadas para el modelado de la tesis, haciendo hincapié en los diagramas de clase, que son los que la herramienta soporta. El capítulo 4 es una introducción al UP, se explican sus características y cómo está formado, ya que este proceso es el que se sigue para el desarrollo de la tesis. El capítulo 5 describe paso a paso los resultados obtenidos siguiendo el UP con UML, desde la etapa de requerimientos de la herramienta hasta los resultados obtenidos una vez terminado el proyecto. Finalmente se exponen los resultados obtenidos y los trabajos a futuro.

Contenido

Resumen	iv
Contenido	v
Lista de figuras	viii
Lista de tablas	x
Capítulo 1: Descripción del problema	11
1.1 Introducción.....	11
1.1.1 ¿Qué son las herramientas CASE?	11
1.1.2 ¿Qué es el UML?.....	11
1.2 El problema	12
1.3 Solución propuesta	13
1.4 Objetivos.....	16
Capítulo 2: Herramientas CASE	17
2.1 Introducción.....	17
2.1.1 Herramientas CASE y los modelos de desarrollo	17
2.1.2 Historia de las herramientas CASE	18
2.2 Ventajas de usar herramientas CASE.....	19
2.2.1 Facilidad para la revisión de aplicaciones	19
2.2.2 Ambiente interactivo en el proceso de desarrollo	19
2.2.3 Generación de código	20
2.2.4 Reducción de costos	21
2.2.5 Incremento en la productividad.....	21
2.3 Desventajas de usar herramientas CASE.....	21
2.3.1 El costo	21
2.3.2 Preparación del personal.....	22
2.3.3 Métodos estructurados.....	22
2.3.4 Falta de estandarización para el intercambio de información	22
2.3.5 Función limitada.....	22
2.4 Componentes de una herramienta CASE	23
2.4.1 Depósito de datos	23
2.4.2 Módulo para creación de diagramas y modelado	24
2.4.3 Analizador de sintaxis	24
2.4.4 Generador de código.....	24
Capítulo 3: El lenguaje de modelado unificado	26
3.1 Introducción.....	26
3.1.1 ¿Qué es el UML?.....	26
3.1.2 Antecedentes del UML.....	27
3.1.3 Conceptos básicos	27
3.2 Las vistas de UML.....	28
3.2.1 Diagrama de casos de uso.....	28
3.2.2 Diagrama de clases	31
3.2.3 Diagrama de paquetes.....	34
3.2.4 Diagrama de interacción.....	36
3.2.4.1 Diagrama de secuencia	36
3.2.4.2 Diagrama de colaboración.....	38

Capítulo 4: El Proceso Unificado de Desarrollo de Software	41
4.1 Introducción.....	41
4.1.1 ¿Qué es el Proceso Unificado?	41
4.1.2 Antecedentes del Proceso Unificado	42
4.2 La vida del Proceso Unificado	43
4.2.1 Fase de inicio.....	44
4.2.2 Fase de elaboración	44
4.2.3 Fase de construcción.....	44
4.2.4 Fase de transición	44
4.3. Elementos del proceso unificado.....	45
4.4 Características del Proceso Unificado	45
4.4.1 Un proceso dirigido por casos de uso.....	46
4.4.2 Un proceso centrado en la arquitectura	47
4.4.2.1 Necesidad de la arquitectura.....	47
4.4.2.2 Casos de uso y arquitectura	47
4.4.3 Un proceso iterativo e incremental.....	48
4.5 Los flujos de trabajo fundamentales.....	48
Capítulo 5: Desarrollo de la herramienta UMLGEC++	50
5.1 Introducción.....	50
5.2 Requerimientos.....	50
5.2.1 Evaluación de herramientas CASE.....	50
5.2.2 Encontrar actores y casos de uso	54
5.2.3 Priorizar casos de uso	55
5.2.4 Detallar casos de uso	56
5.2.5 Estructurar el modelo de casos de uso.....	56
5.3 Análisis.....	60
5.3.1 Análisis de la arquitectura	60
5.3.1.1 Modelo de capas típico.....	61
5.3.2 Análisis de casos de uso	61
5.3.2.1 Identificación inicial de clases.....	61
5.3.2.2 Descripción de las interacciones entre objetos del análisis	63
5.3.3 Análisis de clases.....	65
5.3.3.1 Identificación de responsabilidades.....	65
5.3.3.2 Identificación de atributos	65
5.3.3.3 Identificación de asociaciones y agregaciones	66
5.3.3.3.1 Visibilidad	66
5.3.3.3.2 Asociaciones y agregaciones	67
5.3.3.4 Identificación de generalizaciones.....	68
5.4 Diseño.....	68
5.4.1 Diseño de la arquitectura	69
5.4.1.1 Identificación de subsistemas	69
5.4.1.2 Identificación de interfaces.....	70
5.4.2 Diseño de casos de uso	71
5.4.2.1 Identificación de clases de diseño participantes.....	71
5.4.2.2 Descripción de las interacciones entre objetos del diseño.....	71
5.4.3 Diseño de una clase	73
5.4.3.1 Identificación de operaciones	73
5.4.3.2 Identificación de atributos	73
5.4.3.3 Identificación de asociaciones y agregaciones	73

5.4.3.4 Identificación de generalizaciones.....	73
5.4.4 Diseño de un subsistema	74
5.5 Implementación	74
5.5.1 Implementación de la arquitectura	75
5.5.2 Implementación de un subsistema.....	75
5.5.3 Implementación de una clase.....	77
5.5.3.1 Generación de código a partir de una clase de diseño	77
5.5.3.2 Implementación de operaciones	79
5.5.3.3 Métodos fundamentales de la herramienta	79
5.5.3.3.1 El método “guardar()”	79
5.5.3.3.2 El método “generarCódigo()”	81
5.5.3.3.3 Análisis de sintaxis	84
5.6 Pruebas	86
5.6.1 Diseñar pruebas	86
5.6.2 Realizar pruebas	87
5.7 Resultados obtenidos	88
5.8 Comparativa de UMLGEC++ frente a otras herramientas.....	91
Conclusiones y trabajos a futuro.....	93
Bibliografía.....	95
Apéndice A. Glosario.....	97
Introducción.....	97
Términos.....	97
Apéndice B. Diagrama de clases global de UMLGEC++.....	100
Apéndice C. Clases contenidas dentro de los subsistemas.....	101
Apéndice D. Estructura para el almacenamiento de archivo.....	103
Apéndice E. Código para el almacenamiento de archivo.....	106
Símbolos.....	109
Apéndice F. Código fuente completo del sistema <i>Home Heating System</i>.....	110

Lista de figuras

Fig. 1.1: Diagrama de clases del ejemplo.....	14
Fig. 1.2: Código fuente en C++ del archivo de cabecera de la clase “Libro”	15
Fig. 2.1: Historia de las herramientas CASE.....	20
Fig. 2.2: Componentes principales de una herramienta CASE	23
Fig. 3.1: Clasificación de vistas y diagramas de UML	28
Fig. 3.2: Diagrama de casos de uso del sistema de préstamo de libros de una biblioteca.....	31
Fig. 3.3: Un fragmento del diagrama de clases del sistema de préstamo de libros de una biblioteca	34
Fig. 3.4: Diagrama de paquetes del sistema de préstamo de libros de una biblioteca.....	36
Fig. 3.5: Diagrama de secuencia para el caso de uso de reservar un libro en línea.....	38
Fig. 3.6: Diagrama de colaboración para el caso de uso de reservar un libro en línea.....	40
Fig. 4.1: Relación entre los cinco flujos del trabajo fundamentales y las cuatro fases del Proceso Unificado	43
Fig. 4.2: Los flujos de trabajo fundamentales del UP y los modelos que desarrollan.	46
Fig. 5.1: Gráfica comparativa de herramientas CASE con soporte a UML	53
Fig. 5.2: Formato expandido del caso de uso “Crear componente de clase”	57
Fig. 5.3: Diagrama de casos de uso de UMLGEC++	58
Fig. 5.4: Formato real del caso de uso “Crear componente de clase”	59
Fig. 5.5: Diagrama inicial de clases del caso de uso “Crear componente de clase”.....	62
Fig. 5.6: Iconos de UML para representar los estereotipos de clases.....	63
Fig. 5.7: Diagrama de secuencia del flujo normal del caso de uso “Crear componente de clase”	64
Fig. 5.8: Diagrama de colaboración del flujo normal del caso de uso “Crear componente de clase”	64
Fig. 5.9: Clases con atributos y responsabilidades	66
Fig. 5.10: Representación gráfica de la visibilidad entre objetos en Rational Rose.....	67
Fig. 5.11: Diagrama de clases del caso de uso “Crear componente de clase”.....	68
Fig. 5.12: Subsistemas de UMLGEC++	70
Fig. 5.13: Diagrama de secuencia de diseño del flujo normal del caso de uso “Crear componente de clase”.....	72
Fig. 5.14: Diagrama de colaboración de diseño del flujo normal del caso de uso “Crear componente de clase”	72
Fig. 5.15: Ejemplo de reuso con la clase TForm de C++ Builder 5.0	74
Fig. 5.16: Evolución de las clases de diseño en componente de implementación	76
Fig. 5.17: Implementación de clase “TClase” en C++ Builder	78
Fig. 5.18: Propuesta de código para almacenamiento en archivo	80
Fig. 5.19: Código final para almacenamiento en archivo.....	80
Fig. 5.20: Estructura del cuerpo generado para un archivo .h.....	83
Fig. 5.21: Estructura del cuerpo generado para un archivo .cpp	84
Fig. 5.22: Ventana de configuración de una asociación en UMLGEC++.....	85
Fig. 5.23: Diagrama de secuencia del flujo normal del caso de uso “Crear componente de relación”	85
Fig. 5.24: Diagrama de secuencia del flujo alterno “Nombre existente” del caso de uso “Crear componente de clase”	86
Fig. 5.25: Diagrama de clases del sistema “Home Heating System”	87
Fig. 5.26: Proceso de generación de código fuente del sistema “Home Heating System”	88

Fig. 5.27: Archivos creados a partir del diagrama del sistema “Home Heating System”	88
Fig. 5.28: Sección de implementación de asociaciones de la clase “HomeHeatingSystem”	89
Fig. 5.29: Fragmento de código que muestra el uso de la STL.	90
Fig. 5.30: Ejecución del sistema “Home Heating System”	90
Fig. 5.31: Gráfica comparativa de UMLGEC++ frente a otras herramientas CASE con soporte a UML	92
Fig. B.1: Diagrama de clases global de UMLGEC++	100
Fig. C.1: Clases contenidas dentro del subsistema “Clases Boundary”	101
Fig. C.2: Clases contenidas dentro del subsistema “ClasesControl”	101
Fig. C.3: Clases contenidas dentro del subsistema “Componentes”	102
Fig. C.4: Clases contenidas dentro del subsistema “ClasesEntity”	102

Lista de tablas

Tabla 1.1: Precios de algunas herramientas CASE en Enero de 2003	13
Tabla 3.1: Elementos estructurales de un caso de uso.....	29
Tabla 3.2: Relaciones dentro de un caso de uso.....	30
Tabla 3.3: Elementos principales del diagrama de clases.	32
Tabla 3.4: Elementos principales del diagrama de clases (continuación)	33
Tabla 3.5: Elementos principales del diagrama de paquetes.....	35
Tabla 3.6: Elementos principales del diagrama de secuencia	37
Tabla 3.7: Elementos principales del diagrama de colaboración.	39
Tabla 5.1: Tabla comparativa de herramientas CASE con soporte a UML	52
Tabla 5.2: Requisitos de UMLGEC++	55
Tabla 5.3: Evaluación de UMLGEC++.....	91

Capítulo 1: Descripción del problema

1.1 Introducción

1.1.1 ¿Qué son las herramientas CASE?

Día a día la tecnología avanza, surgen nuevas y mejores formas de hacer las cosas, siempre buscando métodos más efectivos, confiables, con mayor calidad y menos riesgos. Las herramienta CASE nacen para auxiliar a los desarrolladores de software, lo que permite el apoyo computarizado en todo o en parte del ciclo de vida del desarrollo de un sistema de software.

Las herramientas CASE han surgido para dar solución a varios problemas inherentes al diseño del software, pero como se cita en [INEI 97] principalmente nacen para solucionar el problema de la mejora de la calidad del desarrollo de sistemas de mediano y gran tamaño, y en segundo término, por el aumento de la productividad.

Para que los negocios sean competitivos deben llevar una buena calidad de los productos o servicios que ofrece. De acuerdo con las investigaciones realizadas por el INEI (Instituto Nacional de Estadística e Informática del Perú) la mejora de la calidad se logra al reducir sustancialmente muchos de los problemas de análisis y diseño relacionados con los proyectos, como la lógica en el diseño y la coherencia de módulos, entre otros. Y la mejora de la productividad se consigue a través de la automatización de tareas como la generación y reutilización de código, que son puntos importantes a considerar en una herramienta CASE. [INEI 97]

1.1.2 ¿Qué es el UML?

Con el paso del tiempo las computadoras han ido creciendo y mejorando, a diferencia del software, que no lleva el mismo nivel de crecimiento, esto debido a diversos problemas como son: productividad, costos, confiabilidad, entre otros, que son causantes de muchos de los errores del desarrollo de software. Esta complejidad es atacada por la Ingeniería del Software, la cual con el fin de facilitar el proceso de desarrollo, realiza diagramas que describen la forma de cómo hacer las cosas, lo que ha ocasionado que surjan metodologías de Análisis y Diseño Orientada a Objetos (OO – *Object Oriented*) (Orientado a Objetos) y herramientas CASE para facilitar el uso de tales métodos. Según Fowler [Fowler 98] la existencia de muchos de estos métodos OO ocasionó un problema de estandarización.

En Octubre de 1994 surge oficialmente el UML de Rational Corporation con el propósito de unificar principalmente los métodos de Grady Booch (Booch), James Rumbaugh (OMT) y Jacobson (Objectory). El UML fue aceptado por el OMG (*Object Management Group* – Grupo Administrador de Objetos) como un estándar en Noviembre de 1997. [Booch 99]

La finalidad del UML es ser un lenguaje de modelado independiente de cualquier método. Es un lenguaje gráfico utilizado para el desarrollo de componentes de software, el cual puede ser aplicado en diversas áreas como el comercio electrónico, actividades bancarias, arquitecturas de codificación, entre otras.

El UML define una cantidad de diagramas y el significado de estos. El uso de los diagramas UML como se expone en [Fowler 98], es para la mejora de la comunicación, ya que ayuda a los desarrolladores de software que utilizan diferentes métodos, a que se comuniquen al usar una misma notación. Además, al hacer uso de diagramas UML se mejora considerablemente la tarea de actualizar los diagramas ya creados, ya que si se hace un cambio en algún componente no se tendría que modificar todo el diagrama implicado.

Gracias al lenguaje de modelado estándar que provee el UML es que es considerado por muchos autores incluyendo a sus creadores James Rumbaugh, Ivar Jacobson y Grady Booch como el sucesor de las notaciones de modelado [Fowler 99]. La ventaja principal del UML que marca Jim Rumbaugh [Coleman 97] sobre otras notaciones OO es que elimina las diferencias entre semánticas y notaciones.

1.2 El problema

Hoy en día existe una gran cantidad de herramientas que soportan UML, entre las conocidas: Rose de IBM (anteriormente de Rational Software Corporation), Rhapsody de I-logix, y Visual UML de Visual Object Modelers. Muchas de ellas tienen características que reducen significativamente el trabajo de un desarrollador, lo que las hace ser consideradas herramientas CASE. El adquirir una de estas herramientas tiene como inconveniente principal el alto precio, ya que aunque existen versiones gratuitas de ellas, son limitadas de alguna manera, lo cual hace a un programador novato, pensar en no usar una herramienta CASE – al menos no en un principio.

Otros inconvenientes de las herramientas CASE tienen que ver con la dificultad de aprendizaje, debido a la cantidad de opciones y configuraciones que presentan, sobre todo si se quiere sacar el máximo provecho a las funciones importantes como la generación de código o de estructura de datos.

En el análisis de [Hetherington 99] se menciona que la mayoría de las herramientas CASE pueden tener dos versiones, las que sirven sólo para el trazo de diagramas y otras que pretenden soportar la mayoría de los tipos de diagramas UML y generar código en los lenguajes de programación más comunes. Las herramientas que sirven únicamente para dibujar son fáciles de usar y no tienen mayor problema, ya que no generan código alguno. Pero la gran mayoría de las herramientas CASE que soportan UML tienen la gran desventaja de no contar con un comprobador de sintaxis de UML. Contar con tal comprobador es importante sobre todo si el usuario comienza con el estudio del UML y si la herramienta le ayuda informándole de las inconsistencias de los diagramas en el momento de estarlos diseñando, se le puede proporcionar más interactividad y apoyo.

Al realizar una investigación sobre las principales compañías desarrolladoras de herramientas CASE con soporte a UML, se encontró una gran cantidad de ellas; la mayoría tienen disponibles una versión *trial* (prueba) o *Demo*, que aunque son gratis no permiten crear diseños completos,

ya sea por el tiempo de vida o por la falta de muchas de las opciones. Las versiones completas conocidas como *Developer* (Desarrollo) o *Enterprise* (Empresa) pueden ser adquiridas por medio de comercio electrónico. El precio de compra de una licencia de una versión completa de estas herramientas, van de los \$500.00 hasta más de los \$4,000.00 dólares americanos, esto sin contar los gastos de mantenimiento y soporte técnico. La tabla 1.1 muestra algunos de estos productos con sus respectivos precios.

Tabla 1.1: Precios de algunas herramientas CASE en Enero de 2003

Producto	Precio en dólares americanos
IBM Rational Rose 2002 Professional Edition for C++	\$ 2394.00
IBM Rational Rose 2002 Enterprise Edition	\$4194.00
Visual UML 3.1 Standard Edition	\$ 495.00
Visual UML 3.1 Plus - Developer Edition	\$ 995.00
Rhapsody in C++ / Solo Edition V 4.1	\$ 895.00

Como se puede observar, el adquirir una herramienta de éstas resultaría demasiado costoso. Tratándose de una institución educativa como la Universidad Tecnológica de la Mixteca, se tendría que adquirir más de una licencia de estos productos para adoptarla como material de apoyo para la enseñanza de programación Orientada a Objetos.

1.3 Solución propuesta

La presente tesis no pretende dar solución a los problemas que los desarrolladores de software demandan de las actuales herramientas CASE de UML, además que el desarrollo de una herramienta CASE que soporte toda o casi toda la metodología UML llevaría demasiado tiempo y se necesitaría de varios desarrolladores para acelerar el proceso.

La tesis propuesta se limita a proporcionar un herramienta CASE que soporte la notación UML para diagramas de clase – suficiente para representar la estructura estática de los sistemas –, contando además con un comprobador de sintaxis para tal notación y con la implementación de uno de los principales componentes de una herramienta CASE : el generador de código, el cual una vez diseñado un diagrama de clases, generará un código en lenguaje C++ 100% compilable que posteriormente pueda ser aprovechado por el usuario, ya que se obtiene la estructura de las clases y objetos del sistema que se hayan modelado, con lo que el trabajo de codificación será reducido en gran medida, quedando como actividad complementaria el codificar el comportamiento del sistema y realizar pruebas para tener un sistema funcional. Logrando al final, una reducción del tiempo utilizado para el desarrollo del sistema.

Asimismo, el desarrollo de esta herramienta está orientada a proporcionar un material de apoyo para la materia de Programación Orientada a Objetos II impartida en la Universidad Tecnológica de la Mixteca, asignatura en la que los alumnos aprenden el uso del UML y desarrollan un proyecto final tomando como base éste lenguaje de modelado.

Contar con una herramienta de este tipo hace que los alumnos comprendan mejor el uso y aplicación de los diagramas de clase, y al mismo tiempo, les ayuda a analizar y diseñar de una manera más rápida y eficiente el sistema a desarrollar.

A manera de ejemplo, se presenta el siguiente diagrama de clases sencillo modelado en UMLGEC++. El ejemplo muestra cuál deberá ser la funcionalidad de la herramienta propuesta:

Considere que un libro contiene atributos como el folio, el título entre otros, y está formado por algunos datos adicionales como el número de volúmenes y el número de páginas, también un libro está compuesto por uno o mas o más autores.

Un ejemplar es un tipo específico de un libro que hereda las características de un libro específico, anexado sus propios atributos, como la fecha de obtención de cada ejemplar.

Por otro lado existe la posibilidad de reservar uno o más libros para su préstamo posterior.

La siguiente figura representa el diagrama de clases en UML del sistema:

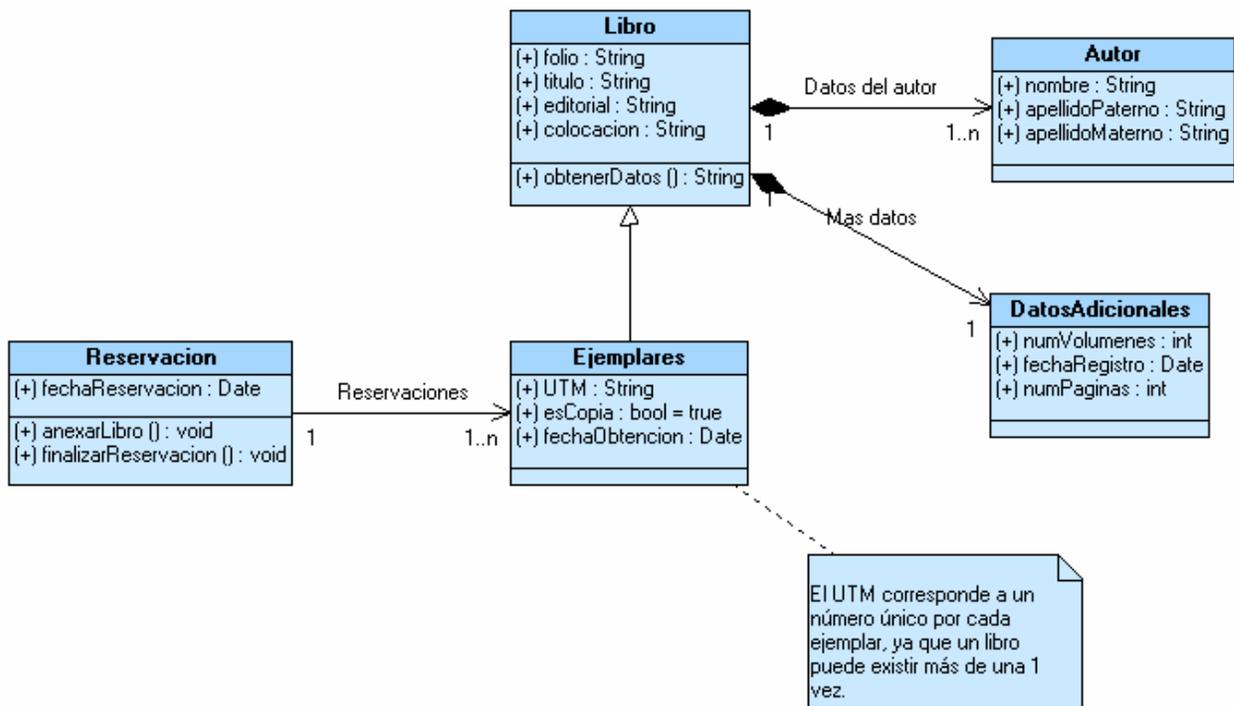


Fig. 1.1: Diagrama de clases del ejemplo

Como se observa, el problema se ve de otra forma, más claro para los desarrolladores de software y se obtiene una mejor perspectiva de lo que se desea desarrollar, se ven las estructuras de las clases, sus atributos, sus operaciones, entre otros detalles.

A continuación se presenta parte del código fuente que UMLGEC++ generó a partir de este diagrama. El código que se lista a continuación corresponde al archivo de cabecera (archivo .h en C++) de la clase *Libro* de la Fig. 1.1.

```

//Clase: Libro
#ifndef Libro_h
#define Libro_h 1
#include "Autor.h"
#include "DatosAdicionales.h"
#include <list.h>
class Libro
{
    //Declaraciones públicas
public:
    //----- Constructores generados por omisión
    Libro();
    Libro(const Libro &right);
    //----- Destructor generado por omisión
    ~Libro();
    //----- Operación de asignación generado
    Libro & operator=(const Libro &right);
    //----- Operaciones de igualdad generadas
    int operator==(const Libro &right) const;
    int operator!=(const Libro &right) const;
    //----- Operaciones definidas por el usuario
    String obtenerDatos();
    //----- Operaciones Get y Set para atributos
    //Atributo: folio
    const String get_folio() const;
    void set_folio(String valor);
    //Atributo: titulo
    const String get_titulo() const;
    void set_titulo(String valor);
    //Atributo: editorial
    const String get_editorial() const;
    void set_editorial(String valor);
    //Atributo: colocacion
    const String get_colocacion() const;
    void set_colocacion(String valor);
    //----- Operaciones Get y Set para asociaciones
    //Asociación: Datos del autor
    //Rol: Autor::obj_Autor
    const list<Autor> get_obj_Autor() const;
    void set_obj_Autor(list<Autor> valor);
    //Asociación: Mas datos
    //Rol: DatosAdicionales::obj_DatosAdicionales
    const DatosAdicionales get_obj_DatosAdicionales() const;
    void set_obj_DatosAdicionales(DatosAdicionales valor);
    //Declaraciones protegidas
protected:
    //----- Operaciones definidas por el usuario
    //----- Operaciones Get y Set para atributos
    //----- Operaciones Get y Set para asociaciones
    //Declaraciones privadas
private:
    //----- Operaciones definidas por el usuario
    //----- Operaciones Get y Set para atributos
    //----- Operaciones Get y Set para asociaciones
private: //Implementación
    //----- Atributos establecidos como públicos
    String folio;
    String titulo;
    String editorial;
    String colocacion;
    //----- Asociaciones establecidas como públicas
    //Asociación: Datos del autor
    //Rol: Autor::obj_Autor -> Multiplicidad:1..n (Composición)
    list<Autor> obj_Autor;
    //Asociación: Mas datos
    //Rol: DatosAdicionales::obj_DatosAdicionales -> Multiplicidad:1 (Composición)
    DatosAdicionales obj_DatosAdicionales;
};

```

Fig. 1.2: Código fuente en C++ del archivo de cabecera de la clase “Libro”

1.4 Objetivos

Una vez expuesta la solución propuesta, se determinó que este proyecto de tesis tiene como objetivo general: el desarrollo de una herramienta CASE (UMLGEC++) que sirva de material de apoyo a la materia de Programación Orientada a Objetos II impartida en la Universidad Tecnológica de la Mixteca.

Con los siguientes objetivos específicos:

- Ayudar en el proceso de análisis y diseño, al utilizar el lenguaje de Modelado Unificado para el diseño de un sistema y esquematizándolo en UMLGEC++.
- Mejorar la calidad de los desarrollos. Al hacer uso de UMLGEC++ se facilitan los procesos de creación de los diagramas de clase UML y por supuesto, la creación de código fuente, ya que la herramienta contará con un comprobador de sintaxis que permitirá crear diagramas válidos para el UML, y así, generar un código libre de errores.
- Mejorar la productividad: UMLGEC++ automatiza las tareas de generación de código y de reutilización de los diagramas de clase ya creados para posteriores modelados de sistemas.
- Impulsar al programador principiante a hacer uso de buenas prácticas de programación como la declaración de funciones *inline* y uso de la STL (*Standard Template Library* – Biblioteca de Plantillas Estándar) de C++.
- Aplicar la nueva tecnología aceptada por el OMG para el modelado de sistemas dentro de la programación Orientada a Objetos: el UML. Y uno de los nuevos procesos para el desarrollo de sistemas: el Proceso Unificado, creado por los mismos autores del UML.

Capítulo 2: Herramientas CASE

2.1 Introducción

El mercado de los sistemas de hoy en día demanda desarrollos de software cada vez más grandes y complejos debido a la importancia y al volumen de la información que manejan las organizaciones. Como solución a esta problemática, la Ingeniería del Software ha creado métodos y herramientas que ayuden a los programadores a desarrollar aplicaciones de forma más rápida y de mejor calidad.

La Ingeniería del software es la ciencia que ayuda a elaborar sistemas con el fin de que sea económico, fiable y funcione eficientemente sobre las máquinas reales [Pressman 93]. El uso de la Ingeniería del Software trae consigo algunas ventajas como: obtención de un nivel competitivo, mejora de la uniformidad de métodos, adaptación de la automatización del analista, cambio de métodos de trabajo, entre otros.

La IEEE define en 1990 a la Ingeniería del Software como: “La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación (funcionamiento) y mantenimiento de software” [IEEE 93]

Luego entonces, la Ingeniería de Software Asistida por Computadora (CASE) tiene como objetivo proporcionar un conjunto de herramientas bien integradas y que ahorren trabajo, uniendo y automatizando todas o algunas de las fases del ciclo de vida del software, en otras palabras, CASE es una herramienta que ayuda a un ingeniero de software a desarrollar sistemas de cómputo.

Dos definiciones de CASE más formales son las siguientes:

“Herramientas individuales para ayudar al desarrollador de software o administrador de proyectos durante una o más fases del desarrollo (o mantenimiento) del software”. [Terry 90]

“Una combinación de herramientas de software y metodologías estructuradas de desarrollo”. [McClure 89]

2.1.1 Herramientas CASE y los modelos de desarrollo

La idea principal de CASE como se expresa en [Stobart 96] es apoyar las fases del ciclo de vida del software con un conjunto de herramientas que ahorren tiempo y trabajo. Algunas herramientas se centran en las primeras fases; éstas proveen ayuda para el dibujo de diagramas y creación de pantallas. Otras se enfocan en las fases de implementación. Éstas incluyen codificación automatizada y generadores de prueba.

Con la ayuda de CASE muchas de las fases pueden ser terminadas más rápido; de esta manera se convierte en fases de menos intensidad de trabajo. Pero CASE no sólo se restringe en asistir con

la documentación, también envuelve aspectos de apoyo computacional para el diseño del software como la detección de errores en el diseño de modelos y la generación de código, entre otros.

2.1.2 Historia de las herramientas CASE

La historia de las herramientas CASE comienza a principios de los 70's con el procesador de palabras que se usaba en la creación de documentos. El desarrollo se centró inicialmente en herramientas de soporte de programas como traductores, compiladores, ensambladores y procesadores de macros [SEI 99]. Dado que la tecnología avanzó y el software se volvió más complejo, el tamaño para el soporte también tuvo que crecer. Ahora se desarrollaban herramientas para diagramas (como los diagramas Entidad-Relación y diagramas de flujo), editores de programas, depuradores, analizadores de código y utilidades de impresión, como los generadores de reportes y documentación.

Los desarrollos en el área de las herramientas CASE entre 1980 y 1990 tuvieron un enfoque hacia las herramientas que buscaban dar respuestas a los problemas en los sistemas de desarrollo. Esto dió inicio a la elaboración de los siguientes productos de herramientas CASE:

- Desarrollo Orientado a Objetos: Éstas ayudan a crear código reutilizable que pueda ser utilizado en diferentes lenguajes y plataformas. Con el gran crecimiento de desarrollos actual, este tipo de herramientas continúa incrementándose.
- Herramientas de desarrollo Visual: Estas herramientas permiten al desarrollador construir rápidamente interfaces de usuario, reportes y otras características de los sistemas, lo que hace posible que puedan ver los resultados de su trabajo en un instante, un ejemplo de estas herramientas son los lenguajes de programación visuales como Borland C++ Builder, Visual C++, entre otros.

El INEI, en su serie de publicaciones de la Colección “Cultura Informática”, tiene a disposición su vigésimo segundo número titulado: “Herramientas Case”, publicación en la que se proporciona información importante sobre éstas herramientas. [INEI 99]

Las actuales líneas de evolución de las herramientas CASE de acuerdo a las investigaciones del INEI son:

- Herramientas para sistemas bajo arquitectura cliente/servidor. Versiones que faciliten la distribución de los elementos de una aplicación entre los diferentes clientes y servidores.
- CASE multiplataforma. Herramientas que soportan combinaciones de diferentes plataformas físicas, sistemas operativos, interfaces gráficas de usuario, sistemas de gestión de bases de datos, lenguajes de programación y protocolos de red.
- CASE para ingeniería inversa y directa. Ya existen algunas herramientas de este tipo como IBM Rational Rose. Su evolución serán mejoras en la obtención de los diseños a partir del código ya existente (ingeniería inversa) y la regeneración del mismo (ingeniería directa).

- CASE para trabajo en grupo (*groupware*). Herramientas que se centran en el proceso de desarrollo más que en el producto a desarrollar.
- CASE para desarrollo de sistemas Orientados a Objetos. Casi todas las herramientas existentes cubren alguna de las fases del ciclo de vida de desarrollo de aplicaciones orientadas a objetos, ya sea la interfaz del usuario, análisis, diseño, programación, etc. Ahora el objetivo es cubrir el ciclo de vida completo.
- Otras posibles líneas de evolución serán:
 - La utilización de la tecnología multimedia.
 - La incorporación de técnicas de inteligencia artificial.
 - Sistemas de realidad virtual.

La historia de las herramientas de software se resume en la Fig. 2.1, basada en las fechas en [Stobart 96] y ampliado con información obtenida de [INEI 97] e [INEI 99]:

2.2 Ventajas de usar herramientas CASE

La tecnología CASE trae consigo una gran cantidad de ventajas al desarrollo del software, a continuación se listan y describen las más importantes: [Garmire 99], [Hogan 99] y [Stobart 96]

2.2.1 Facilidad para la revisión de aplicaciones

Muchas veces, una vez que se implementa una aplicación, se emplea por mucho tiempo. Cuando se crea un sistema en grupo, el contar con un almacenamiento central de la información agiliza el proceso de revisión ya que éste proporciona consistencia y control de estándares. La capacidad que brindan algunas herramientas para la generación de código contribuye a modificar el sistema por medio de las especificaciones en los diagramas más que por cambios directamente al código fuente.

2.2.2 Ambiente interactivo en el proceso de desarrollo

El desarrollo de sistemas es un proceso interactivo. Las herramientas CASE soportan pasos interactivos al eliminar el fastidio manual de dibujar diagramas. Como resultado de esto, los analistas pueden repasar y revisar los detalles del sistema con mayor frecuencia y en forma más segura.

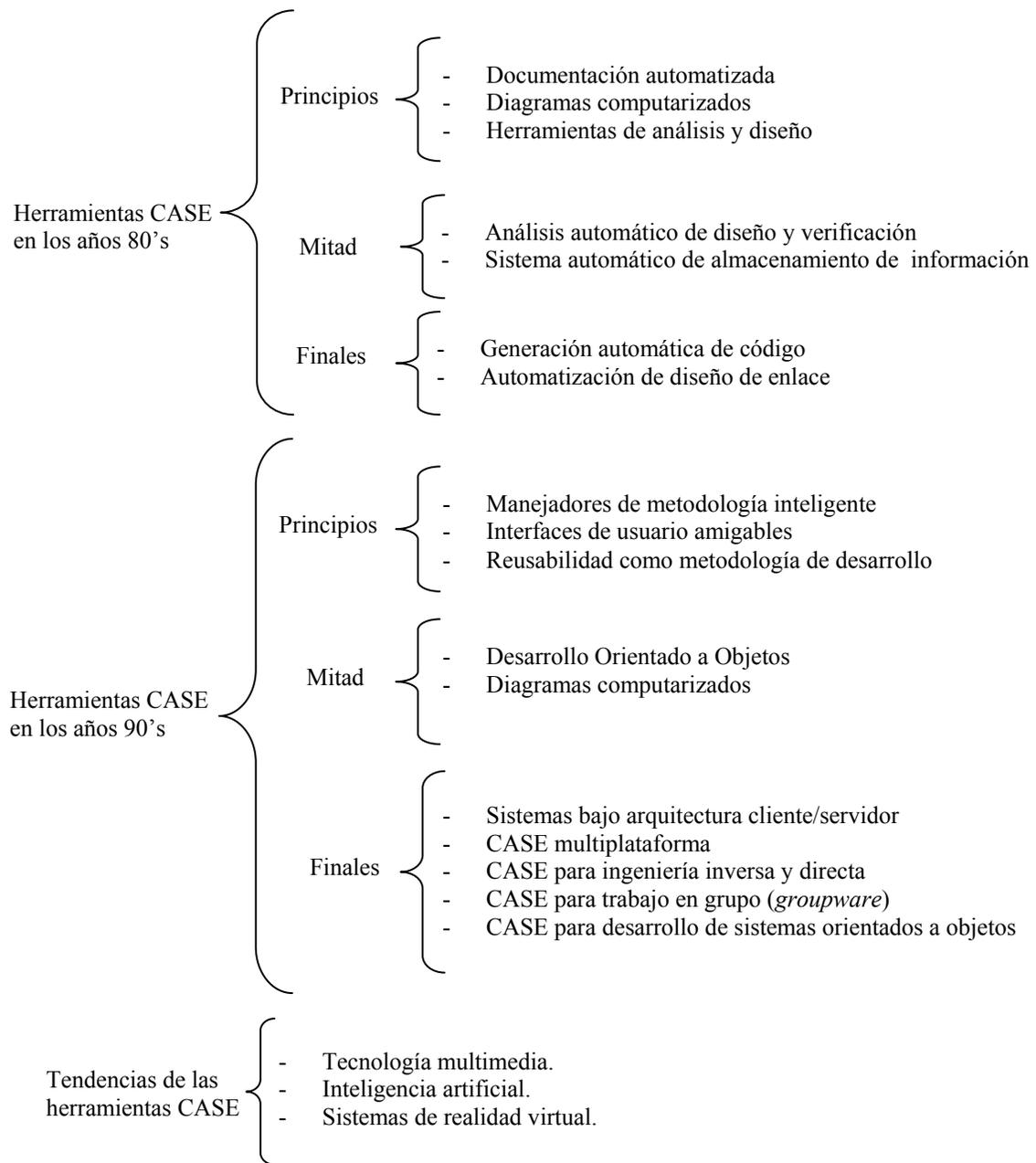


Fig. 2.1: *Historia de las herramientas CASE*

2.2.3 Generación de código

La generación de código trae consigo beneficios considerables dentro del desarrollo del software, una de ellas es que permite acelerar el proceso de desarrollo al automatizar el tedioso proceso de crear diagramas y posteriormente codificarlos. Otra ventaja es que asegura una estructura consistente (lo que ayuda en la fase de mantenimiento) y disminuye la ocurrencia de varios tipos de errores, mejorando de esta manera la calidad del software desarrollado. Además permite la reutilización, ya que se puede volver a utilizar el software y las estructuras estándares para crear

otros sistemas o modificar el actual, así como el cambio de una especificación modular: es decir, volver a generar el código y los enlaces con otros módulos.

Cabe resaltar que ninguna de las herramientas que existen actualmente en el mercado es capaz de generar un código 100% completo, en el sentido de que pueden crear una estructura estática y partes de una estructura dinámica de un sistema, pero aún no son capaces de generar todo el código que un desarrollador tiene que elaborar a lo largo del ciclo de desarrollo. Cubrir todo este ciclo es precisamente una de las tendencias de las herramientas CASE.

2.2.4 Reducción de costos

En ocasiones se desea desarrollar el diseño de pantallas y reportes con el fin de mostrar cómo quedará la interfaz del usuario, es decir, la distribución y colocación de los datos, títulos, mensajes, etc., además es muy común en las etapas de mantenimiento de un sistema, hacer correcciones al prototipo desarrollado. Todo lo anterior trae como consecuencia un costo en tiempo y dinero al tener que utilizar personal para estas tareas.

La reducción de costos se ve reflejada entonces, al utilizar una herramienta CASE para diseño de interfaces, logrando hacer cambios de interfaz con rapidez. Y usando herramientas con generación de código, el código fuente se puede volver a generar después de hacer las modificaciones requeridas rápidamente en los diagramas.

2.2.5 Incremento en la productividad

El incremento en la productividad se ve reflejado por la optimización de los recursos con que se cuentan para llevar a cabo un proceso. Al hacer uso de una herramienta CASE se administran los recursos humanos y tecnológicos, y al combinar el conocimiento de los desarrolladores con las prestaciones que brindan estas herramientas es que se obtiene un incremento en la productividad en el desarrollo del software.

2.3 Desventajas de usar herramientas CASE

Las herramientas CASE también traen consigo puntos débiles significativos, que van desde la confiabilidad en los métodos que proponen hasta su alcance limitado, a continuación se listan y describen tales debilidades: [Garmire 99] y [Hogan 99]

2.3.1 El costo

El costo es una de las mayores desventajas por la que muchas empresas y desarrolladores no utilizan herramientas CASE. De acuerdo con la Tabla 1.1 el rango de precios de éstas herramientas en sus versiones para empresas van de los \$500.00 hasta más de los \$4,000.00 dólares, esto sin contar con los costos de entrenamiento y mantenimiento. Por ello se debe hacer un buen balance entre los costos y los beneficios que traería la adopción de una herramienta CASE.

2.3.2 Preparación del personal

Al adquirir una herramienta CASE es común que se requiera invertir tiempo y dinero para el entrenamiento del personal para obtener un mejor aprovechamiento de la nueva herramienta, esto implica aprender el lenguaje de modelado que la herramienta soporta y la aplicación del proceso de desarrollo a seguir para la construcción de un sistema.

2.3.3 Métodos estructurados

Muchas herramientas CASE están construidas teniendo como base alguna metodología de análisis estructurado y un proceso de desarrollo de sistemas. Debido a esta característica se ve una limitante, ya que no todos los desarrolladores implicados en el desarrollo de un sistema en grupo están de acuerdo con el empleo de cierta metodología y/o proceso de desarrollo, por lo que antes de comenzar a diseñar el sistema tendrían que acordar la metodología a seguir para así evitar futuras complicaciones en el diseño.

2.3.4 Falta de estandarización para el intercambio de información

Aún no aparece un conjunto compatible de herramientas CASE, es decir, una herramienta puede dar soporte a los diagramas que emplea cierta metodología o bien imponer su propia metodología, reglas y procesos. Lo que trae consigo que al desarrollar un sistema de gran tamaño no exista una compatibilidad entre diferentes herramientas y sea muy poca la posibilidad de compartir fácilmente la información.

Como resultado a esta deficiencia están surgiendo herramientas que adoptan al UML como lenguaje de modelado para los diagramas. El UML tiende a ser adoptado por muchas organizaciones como lenguaje de modelado estándar. Entre las organizaciones que han contribuido para obtener como resultado la versión 1.0 del UML están Hewlett-Packard, I-Logix, IBM, Microsoft, Oracle, Rational, Texas Instruments, entre otras. De extenderse esta tendencia se reduciría en gran medida la falta de estandarización entre métodos y procesos de desarrollo.

2.3.5 Función limitada

A pesar de que las herramientas pueden apoyar muchas fases del ciclo de desarrollo o adaptarse a diferentes métodos de desarrollo, por lo general tienen un enfoque principal hacia una fase o método específico. Tal es el caso de la herramienta propuesta.

UMLGEC++ no pretende dar solución a las desventajas citadas anteriormente, simplemente se trata de una herramienta que tendrá como base el lenguaje de modelado estándar UML con lo cual ya no se dependería de una metodología, ya que como se ha mencionado el UML es independiente de ésta. Además la herramienta se enfocará únicamente a la creación de diagramas de clase, por lo que puede ser utilizada durante cualquier etapa de un proceso de desarrollo que implique el uso de este tipo de diagramas, tal es el caso de la fase de elaboración dentro del Proceso Unificado, en la que se crean prototipos de estructuras estáticas a partir de diagramas de clase bien definidos.

2.4 Componentes de una herramienta CASE

Las herramientas CASE tienen componentes que las hacen ser herramientas de ayuda para el desarrollo de software. La Fig. 2.2 muestra los componentes principales. [INEI 99] y [Lokan 99]



Fig. 2.2: *Componentes principales de una herramienta CASE*

A continuación se describen cada uno de estos componentes, sus funciones y su implementación para UMLGEC++

2.4.1 Depósito de datos

También conocido como Diccionario de Recursos de Información, es la Base de Datos de información central de la herramienta CASE, en este depósito se almacena toda la información creada y generada a través de las etapas que cubre la herramienta de determinado proceso de desarrollo, entre estos datos están: componentes de análisis y diseño, diagramas de entrada-salida, diagramas de clase, etc.

El depósito de datos coordina, integra y estandariza las diferentes componentes de información, entre sus principales características están: [INEI 99]

- **El tipo de metodología que soporta la herramienta.** Para el caso de UMLGEC++ el tipo de metodología es UML.
- **La forma de actualización de los datos, si es en tiempo real o por un proceso separado.** En UMLGEC++ la forma de actualización de los datos es mediante un proceso separado que almacena toda la información del diagrama en un archivo con un formato especial, que posteriormente puede ser abierto para ser modificado.
- **La reutilización de módulos para otros diseños.** UMLGEC++ permite modificar los diseños creados y también copiar y pegar los componentes de clases entre varios diagramas.

- **Compatibilidad con los depósitos de datos de otras herramientas para la importación o exportación de la información.** Este punto no se contempló dentro de los requisitos de UMLGEC++ dado que muchas herramientas no ponen a disposición del público el formato de su depósito de datos.

2.4.2 Módulo para creación de diagramas y modelado

Este componente consiste en dar soporte para la creación de los diagramas más utilizados para el análisis y diseño del software, entre los diagramas y modelos más usados se encuentran: los diagramas de flujo de datos, el modelo entidad-relación, entre otros. Para la notación UML se tienen los diagramas de casos de uso, diagramas de clase, diagramas de estado, etc.

Las características que hacen más fácil o más poderosa a una herramienta en cuanto al soporte de diagramas son: [INEI 99]

- **El número máximo de niveles permitido en un diagrama.** Para el caso de diagramas de clase de UML dentro de UMLGEC++ no hay profundidad en los diagramas, es decir, un diagrama de clases no contiene a otro dentro de su diseño.
- **Número máximo de objetos que permite en un diagrama.** UMLGEC++ permite crear tantos objetos como la memoria de la computadora donde se esté ejecutando lo permita, ya que los componentes se van creando y destruyendo en tiempo de ejecución.
- **Posibilidad de deshacer los cambios hechos a los diagramas.** UMLGEC++ permite deshacer los últimos cambios en el diagrama desde que se crea o se abre un archivo. Los cambios que puede deshacer son: mover una clase o una nota, borrar cualquier componente del diagrama.

2.4.3 Analizador de sintaxis

El analizador de sintaxis verifica la validez y el correcto uso de la información introducida a la herramienta de acuerdo a la metodología o lenguaje soportado por la herramienta. Este componente es muy importante para ayudar al desarrollador a no cometer errores de sintaxis al momento de crear los diagramas y generar un código más completo y preciso.

UMLGEC++ implementa este análisis basándose en las características de cada elemento del diagrama de clases de UML, ya que cada elemento determina los datos que puede contener y las conexiones con las que puede estar relacionado. En el Capítulo 5 se explica más a detalle este punto.

2.4.4 Generador de código

El generador de código fuente es uno de los componentes indispensables de las herramientas CASE. Las características más importantes de este componente son:

- **El / los lenguaje(s) de programación que genera.** UMLGEC++ genera código en lenguaje C++.
- **Alcance de la generación del cuerpo del programa.** UMLGEC++ genera la estructura estática de un sistema de software, es decir, la arquitectura de las clases participantes con sus atributos y los cuerpos vacíos de sus operaciones. En la sección 5.5.3.3.2 se explica más a detalle la generación de código.

Ya que se han revisado las características, los componente de una herramienta CASE y cómo han sido implementados para UMLGEC++, seguimos con el ciclo de vida de desarrollo del software, revisando primero en el siguiente capítulo al UML, que es el lenguaje que se utilizó para el modelado de la herramienta, donde se incluyen los diagramas de clase que son los que UMLGEC++ soporta.

Capítulo 3: El lenguaje de modelado unificado

3.1 Introducción

La Ingeniería del Software ha tratado con el paso del tiempo simplificar cada vez más las complejidades que presentan el análisis y diseño de sistemas de software. Para atacar un problema muy grande es bien sabido que se hace uso de la descomposición, para el caso de la Ingeniería del Software se puede hacer una descomposición ya sea algorítmica u Orientada Objetos, dónde esta última es la de más tendencia actualmente.

Debido a esta problemática los investigadores de la Ingeniería del Software han desarrollado diversas metodologías Orientadas a Objetos con la finalidad de proporcionar un soporte para los desarrolladores de sistemas para analizar y diseñar con más precisión los sistemas. Martin Fowler afirma que actualmente el UML es considerado por muchos autores incluyendo a sus creadores James Rumbaugh, Ivar Jacobson y Grady Booch como el lenguaje que unifica los métodos de Análisis y Diseño Orientados a Objetos. [Fowler 99]

3.1.1 ¿Qué es el UML?

El UML es un lenguaje de modelado gráfico y visual utilizado para especificar, visualizar, construir y documentar los componentes de un sistema de software. Está pensado para poder aplicarse en cualquier medio de aplicación que necesite capturar requerimientos y comportamientos del sistema que se desee construir. Ayuda a comprender y a mantener de una mejor forma un sistema basado en un área que el analista o desarrollador puede desconocer.

El UML permite captar información sobre la estructura estática y dinámica de un sistema, en donde la estructura estática proporciona información sobre los objetos que intervienen en determinado proceso y las relaciones que existen entre de ellos, y la estructura dinámica define el comportamiento de los objetos a lo largo de todo el tiempo que estos interactúan hasta llegar a su o sus objetivos. [Rumbaugh 00]

Una característica sobresaliente del UML es que no es un método, sino un lenguaje de modelado. Un método define su notación (lenguaje) y su proceso a seguir durante el ciclo de vida de desarrollo del software. El UML sólo define la notación gráfica y su significado, a partir de la cual se crearán los diseños de sistemas y no depende de un proceso, el cual sería el encargado de orientar los pasos a seguir para elaborar el diseño. [Fowler 99]. La idea de usar los diagramas creados mediante el UML es simplemente para mejorar la comunicación, porque ayuda a que los desarrolladores de software se comuniquen con un mismo lenguaje de modelado independiente de las metodologías empleadas [Fowler 98].

La ventaja principal del UML [Coleman 97] sobre otras notaciones OO es que elimina las diferencias entre semánticas y notaciones.

3.1.2 Antecedentes del UML

Antes que el UML, hubo muchos intentos por unificar métodos, el más conocido que se señala en [Rumbaugh 00] es el caso de *Fusion* por Coleman y sus colegas que incluyó conceptos de los métodos OMT y Booch, pero como los autores de estos últimos no estaban involucrados en la unificación fue tomado como otro método más.

El primer acercamiento a UML fue en 1994 cuando se da la noticia de que Jim Rumbaugh se une con Grady Booch en Rational Software Corporation con la finalidad de unificar sus métodos OMT y Booch respectivamente.

En 1995 salió a luz la primera propuesta de su método integrado que fue la versión 0.8 del entonces llamado Método Unificado (*Unified Method*). En ese mismo año Ivar Jacobson se une a Rational para trabajar con Booch y Rumbaugh en el proceso de unificación, a partir de aquí a estos tres personajes se les conoce como “Los tres amigos”.

En 1996, los tres amigos concluyen su trabajo y lo nombran UML, es entonces cuando el OMG decide convocar a otras compañías a participar con sus propuestas para mejorar el enfoque estándar que el UML pretendía.

En enero de 1997, todas las propuestas de las empresas – entre las que figuran IBM, Oracle y Rational Software – se unieron en la versión 1.0 del UML que fue presentada ante el OMG para su consideración como estándar. Y finalmente en Noviembre de 1997 el UML fue adoptado por el OMG y otras organizaciones afines como lenguaje de modelado estándar.

En diciembre de 2002 IBM adquirió las acciones de Rational Software Corporation.

3.1.3 Conceptos básicos

Como ya se ha mencionado, el UML no es un método sino un lenguaje, el cual define únicamente una notación y un metamodelo [Fowler 99]. El UML al ser un lenguaje estándar no depende de un proceso de desarrollo, y esto es precisamente lo que se quería al lograr unificar los métodos, que se tuviera un lenguaje en común entre los diferentes métodos, para que el desarrollador tuviera la libertad de escoger la metodología de su agrado. Con esto los desarrolladores implicados en un proyecto pueden tener la seguridad de que estarán creando diseños de software bajo un lenguaje que será comprendido por todos aquellos que utilicen el UML.

La notación en el UML son los componentes gráficos que se utilizan para crear los metamodelos, es decir, es la sintaxis del lenguaje de modelado. [Fowler 99]. Para el caso de los diagramas de clase, la notación es la forma en cómo se dibuja una clase, la asociación, la multiplicidad, la agregación, etc.

Un metamodelo o modelo, es la representación de algo en cierta forma, para el caso de la Ingeniería del Software un modelo es un diagrama que representa la definición de la notación. [Fowler 99]

3.2 Las vistas de UML

Las vistas de UML se refieren a la forma en que se modela una parte del sistema a desarrollar. Los autores del UML proponen una clasificación de los diagramas que proporcionan las vistas de UML, y aunque pareciera que esta clasificación es algo intuitiva, aclaran que es simplemente una propuesta y que cada desarrollador puede crear su propia clasificación. [Schmuller 00].

La Fig. 3.1 muestra una clasificación simple.

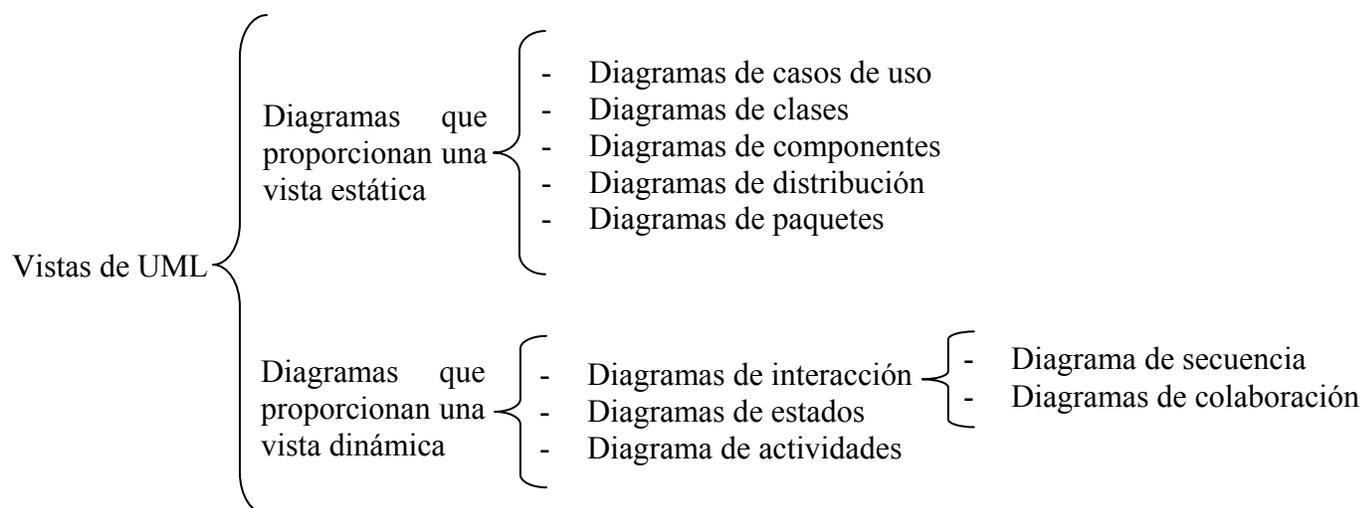


Fig. 3.1: Clasificación de vistas y diagramas de UML ¹

La vista estática es la representación de los elementos del sistema y sus relaciones, la vista dinámica muestra la especificación y la implementación del comportamiento a lo largo del tiempo, es decir muestran el cambio progresivo de los objetos [Rumbaugh 00].

Para el modelado de UMLGEC++ no se crearon todos los diagramas de UML, ya que cada sistema de software tiene características propias y requiere de diferentes vistas. A continuación se describen aquellos que se utilizaron.

3.2.1 Diagrama de casos de uso

La vista que proporcionan los diagramas de casos de uso, modela la forma en cómo un actor interactúa con el sistema, es decir, describe una interacción entre uno o más actores y el sistema o subsistemas como una secuencia de mensajes que llevan una forma, tipo y orden. El propósito de esta vista es enumerar a los actores y los casos de uso, mostrando un comportamiento y determinar qué actores participan en cada caso de uso. [Rumbaugh 00]

¹ Diagrama creado a partir de información obtenida de [Rumbaugh 00] y [Schmuller 00].

La vista de casos de uso es útil para tener una forma de comunicarse con los usuarios finales del sistema, ya que da una visión de cómo ellos esperan que el sistema se comporte.

Un diagrama de casos de uso es una descripción lógica de una parte funcional del sistema y no del sistema en su totalidad. Este diagrama consta de elementos estructurales y relaciones.

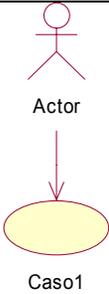
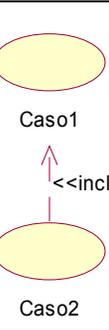
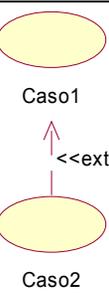
Elementos estructurales: Representan las partes físicas o conceptuales de un modelo. La tabla 3.1 muestra estos elementos. [Figueroa 97], [Rumbaugh 00] y [Schmuller 00]

Tabla 3.1: *Elementos estructurales de un caso de uso*

 <p>Actor</p>	<p>Actor</p> <p>Un actor es un rol que un usuario juega cuando interacciona con el sistema, es un comportamiento específico frente a tal sistema. Un actor no necesariamente representa a una persona en particular, sino más bien la labor que realiza ante al sistema, puede ser un humano, otro sistema de software o algún otro proceso. Se puede decir entonces que, varios usuarios pueden estar relacionados con un solo actor. Por ejemplo, para el caso de uso de un sistema de ventas, el rol de “vendedor” lo puede hacer un vendedor tal cual, el dueño de la tienda, o incluso puede ser una máquina de autoservicio. Un actor se denota como una figura en forma de persona con su nombre debajo.</p>
 <p>Caso de uso</p>	<p>Caso de uso</p> <p>Un caso de uso es una acción o tarea específica que el sistema lleva a cabo tras una petición ya sea de un actor o de otro caso de uso. Un caso de uso conduce a un estado observable de interés para un actor. Se denota como una elipse con su nombre dentro o debajo de ella.</p>

Relaciones: Las relaciones conectan a los elementos estructurales para darle sentido al diagrama de casos de uso. Estas relaciones se muestran en la tabla 3.2. [Figueroa 97], [Rumbaugh 00] y [Schmuller 00]

Tabla 3.2: Relaciones dentro de un caso de uso

 <p>Actor</p> <p>Caso1</p>	<p>Asociación</p> <p>La asociación es la relación más simple del UML, es la comunicación que se da entre un actor y un caso de uso, o entre dos casos de uso. Se denota por una línea dirigida entre ellos.</p>
 <p>Caso general</p> <p>Caso específico</p>	<p>Generalización</p> <p>Este tipo de relación se da entre un caso de uso general y un caso de uso más específico. Donde este último hereda propiedades del primero y agrega sus propias acciones. Se denota como una línea continua con una punta de flecha en forma de triángulo sin rellenar que apunta hacia el caso de uso general.</p>
 <p>Caso1</p> <p><<include>></p> <p>Caso2</p>	<p>Inclusión</p> <p>Es el tipo de relación que se da entre casos de uso cuando se tiene una parte de comportamiento que es similar en más de un caso de uso, y no se desea copiar la descripción de tal conducta para cada uno de ellos o bien cuando un caso de uso necesita utilizar a otro caso de uso. Es recomendable utilizar la inclusión cuando se repite un caso de uso más de una vez y se quiera evitar repeticiones. Se denota con una línea discontinua con una punta de flecha y con la palabra clave “include”</p>
 <p>Caso1</p> <p><<extend>></p> <p>Caso2</p>	<p>Extensión</p> <p>Esta relación se da también sólo en casos de uso cuando se tiene un caso de uso que es similar a otro, pero que hace un poco más. Es recomendable entonces, utilizar la extensión cuando se describa una variación de una conducta normal. Se denota con una línea discontinua con una punta de flecha y con la palabra clave “extend”</p>

La Fig. 3.2 muestra un ejemplo sencillo de un diagrama caso de uso para un sistema de préstamo de libros de una biblioteca:

En esta figura se observa la forma en que los actores (modelados como: el usuario de la biblioteca y el bibliotecario) interactúan con el sistema. Cada uno participa en diferentes casos de uso, por ejemplo se observa cómo la acción de “reservar libro” lo realiza únicamente el usuario de biblioteca sin necesidad que intervenga el bibliotecario, caso contrario a la acción de “prestar libro” que necesita la participación de ambos actores.

Se observa también las relaciones entre los casos de uso, como la existente entre “Prestar libro” y “Prestar libro con reservación” donde este último es una extensión del primero, ya que un préstamo de libro con reservación es una variación al proceso de un préstamo normal. En el caso

de uso “Reservar libro en línea” se aprecia una relación de inclusión con el caso de uso “Buscar libro”, ya que para que se lleve a cabo una reservación se necesita hacer una búsqueda previa de los libros.

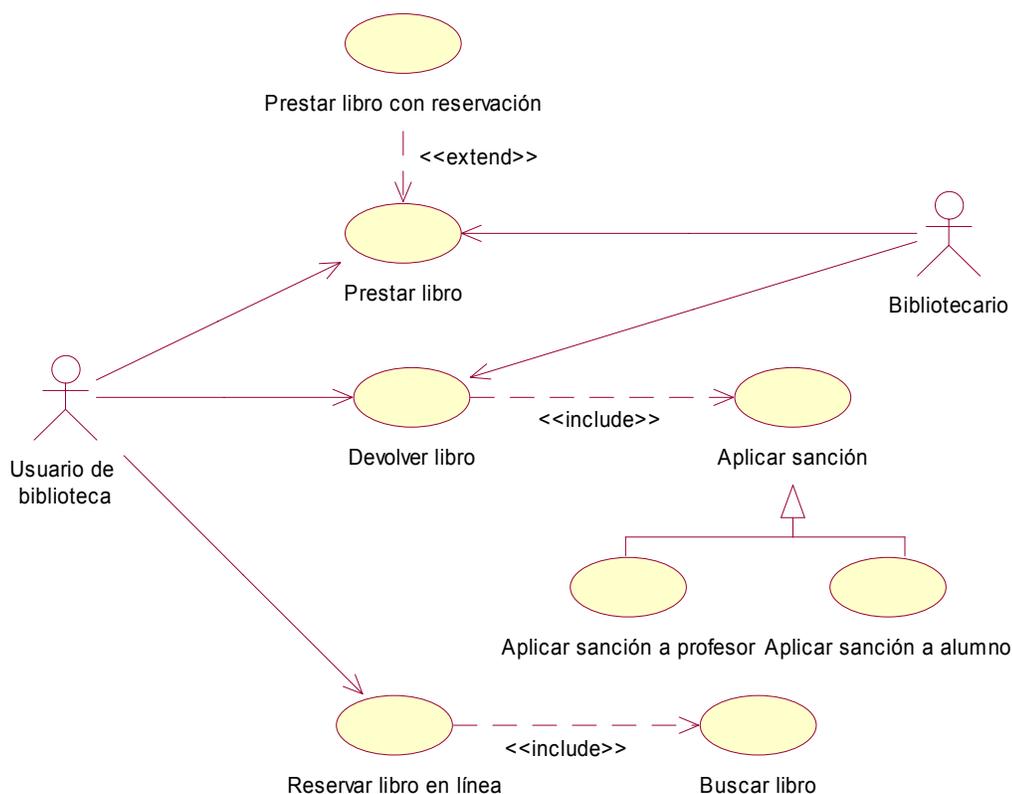


Fig. 3.2: Diagrama de casos de uso del sistema de préstamo de libros de una biblioteca²

3.2.2 Diagrama de clases

La vista de los diagramas de clase, visualiza las relaciones entre las clases que se involucran en el sistema. Un diagrama de clases muestra un conjunto de clases, interfaces y colaboraciones y las relaciones entre éstas, mostrando así, la estructura estática de un sistema. Los diagramas de clase pueden mostrar:

- Clases
- Atributos
- Operaciones
- Asociaciones
- Generalizaciones
- Agregaciones
- Composiciones

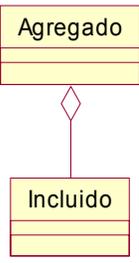
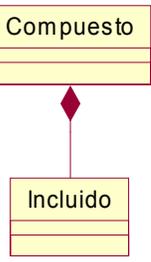
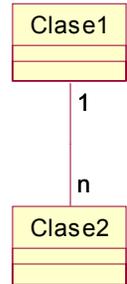
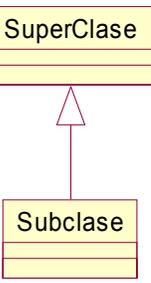
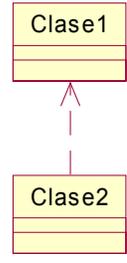
² Ejemplo basado en el sistema de biblioteca de la Universidad Tecnológica de la Mixteca.

Un diagrama de clases esta compuesto por los elementos mostrados en las tablas 3.3 y 3.4. [Figuroa 97], [Rumbaugh 00] y [Schmuller 00]

Tabla 3.3: Elementos principales del diagrama de clases.

<p>Clase</p> <p>Atributos</p> <p>Operaciones()</p>	<p>Clase</p> <p>Es la unidad básica que encapsula toda la información que comparten los objetos del mismo tipo. A través de una clase se puede modelar el entorno del sistema. En UML, una clase se representa por un rectángulo que posee tres divisiones:</p> <p><i>Superior:</i> Contiene el nombre de la Clase</p> <p><i>Intermedio:</i> Contiene los atributos que caracterizan a la Clase (pueden ser private, protected o public).</p> <p><i>Inferior:</i> Contiene las operaciones, las cuales son la forma de cómo interactúa el objeto con los demás, dependiendo de la visibilidad (pueden ser private, protected o public).</p>
<p><i>sin imagen</i></p>	<p>Atributos</p> <p>Los atributos o características de una Clase se utilizan para almacenar información, estos atributos tienen asignado un tipo de visibilidad que define el grado de comunicación con el entorno, los tipos de visibilidades son tres, de acuerdo a la Programación Orientada a Objetos: <i>public</i> (pública), <i>protected</i> (protegida), <i>private</i> (privada).</p> <p>La sintaxis en UML para un atributo es: <i>visibilidad nombre : tipo = valor_inicial</i></p>
<p><i>sin imagen</i></p>	<p>Operaciones</p> <p>Las operaciones de una clase son la forma en cómo ésta interactúa con su entorno, éstas también pueden tener uno de los tipos de visibilidad arriba mencionadas.</p> <p>La sintaxis en UML para una operación es: <i>visibilidad nombre (parámetros) : tipo_de_retorno</i></p>
<p>Clase1</p> <p>Clase2</p>	<p>Asociación</p> <p>La relación entre las instancias de las clases es conocida como Asociación, permite relacionar objetos que colaboran entre sí. La asociación puede ser unidireccional o bidireccional. Una asociación unidireccional significa que sólo existe comunicación de la clase de la que parte la flecha hacia la que apunta. En caso de que sea bidireccional significa que existe comunicación entre ambas clases. Para representar una asociación bidireccional sólo se dibuja una línea continua que una las dos clases.</p>

Tabla 3.4: Elementos principales del diagrama de clases (continuación)

	<p>Agregación</p> <p>La agregación es un tipo especial de asociación, con la cual se pueden representar entidades formadas por varios componentes. La agregación es una relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye, en la programación OO se dice que ésta es una relación por referencia. La agregación se representa con una flecha con punta de rombo en blanco en el extremo del compuesto o base.</p>						
	<p>Composición</p> <p>La composición es similar a la agregación, ya que también representa entidades formadas por componentes, sólo que esta relación es estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye, ya que el objeto base se construye a partir del objeto incluido, en programación OO se dice que esta es una relación por valor. La composición se representa con una flecha con punta de rombo relleno en el extremo del compuesto o base.</p>						
	<p>Multiplicidad</p> <p>Cada asociación tiene dos roles o papeles que se encuentran en cada extremo de la línea, cada uno de estos papeles tiene asignada una <i>multiplicidad</i>, la cual indica el número de objetos que pueden participar en la relación, es decir los límites inferior y superior de los objetos que participan. La multiplicidad puede ser:</p> <table data-bbox="479 1108 933 1224"> <tr> <td>1..* (1..n)</td> <td>Uno o más</td> </tr> <tr> <td>* (n)</td> <td>Cero o más</td> </tr> <tr> <td>m (m es un entero)</td> <td>Número fijo</td> </tr> </table> <p>Estos números se colocan sobre la línea de asociación junto a la clase correspondiente, como se ilustra.</p>	1..* (1..n)	Uno o más	* (n)	Cero o más	m (m es un entero)	Número fijo
1..* (1..n)	Uno o más						
* (n)	Cero o más						
m (m es un entero)	Número fijo						
	<p>Generalización (Herencia)</p> <p>Indica que una subclase hereda las operaciones y atributos especificados por una superclase, por ende, la subclase además de poseer sus propios métodos y atributos, poseerá las operaciones y atributos de la superclase, siempre y cuando estos tengan visibilidad pública o protegida. Se denota como una línea continua con una punta de flecha en forma de triángulo sin relleno que apunta hacia la superclase.</p>						
	<p>Dependencia</p> <p>La dependencia es un tipo de relación entre dos elementos, donde el uso más particular de este tipo de relación es para denotar una dependencia de uso que tiene una clase con otra, en otras palabras, un cambio en una de ellas causa un cambio en la otra. En la dependencia el objeto utilizado no se almacena dentro del objeto que la crea. La dependencia se denota como una línea discontinua con punta de flecha formada por dos líneas.</p>						

La Fig. 3.3 muestra parte de un diagrama clases obtenido de un sistema de préstamo de libros en una biblioteca.

En este diagrama se puede apreciar principalmente la forma en que están modelados los datos de los libros, observe los atributos de cada clase, note que un libro está “compuesto” por uno o más autores y un registro de datos adicionales, ya que sin estos, los datos de un libro estarían incompletos. También observe que los ejemplares heredan las características de un libro, anexado sus propios atributos, como la fecha de obtención de cada ejemplar.

También se observa la asociación existente entre una reservación y un libro, donde se entiende que una reservación puede ser de uno o más libros.

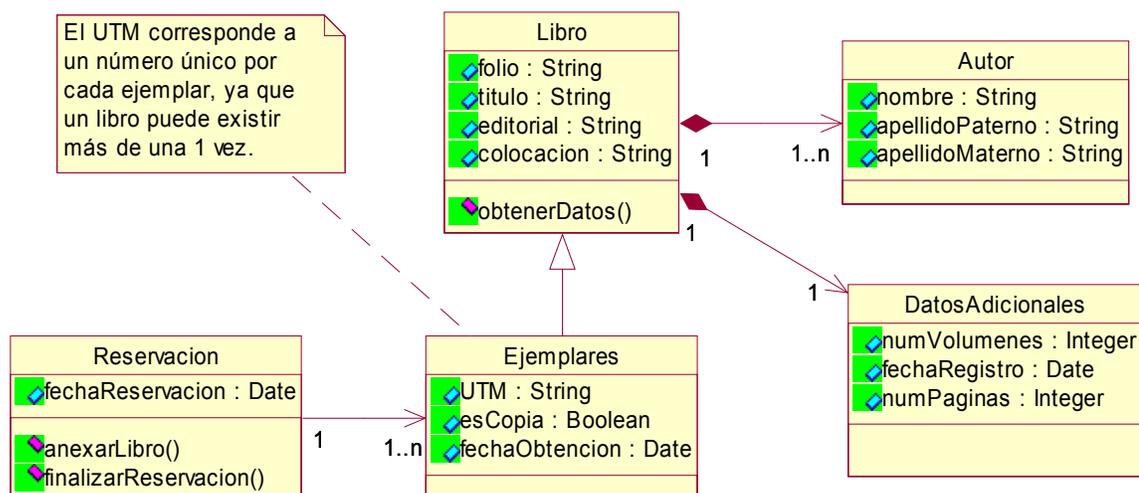


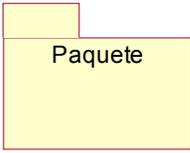
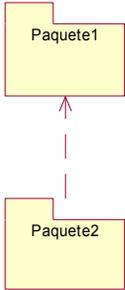
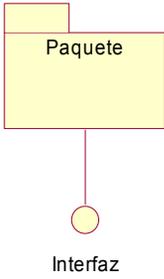
Fig. 3.3: Un fragmento del diagrama de clases del sistema de préstamo de libros de una biblioteca³

3.2.3 Diagrama de paquetes

Cuando se tiene un sistema muy grande, es conveniente separarlo en piezas más pequeñas para poder trabajar más fácil con él, y son los diagramas de paquetes los que nos muestran las relaciones y las dependencias entre los diferentes paquetes en que un sistema está dividido. El diagrama de paquetes puede mostrar los componentes mostrados en la tabla 3.5. [Figueroa 97], [Rumbaugh 00] y [Schmuller 00]

³ Ejemplo basado en el sistema de biblioteca de la Universidad Tecnológica de la Mixteca.

Tabla 3.5: Elementos principales del diagrama de paquetes

	<p>Paquete</p> <p>Un paquete es un mecanismo para organizar un conjunto de elementos. Los paquetes pueden contener otros elementos de modelo como diagramas clases, máquinas de estado, diagramas de casos de uso o diagramas de interacción, incluso puede contener otros paquetes. Comúnmente un paquete representa un subsistema.</p>
	<p>Interfaz</p> <p>Una interfaz es un elemento de modelado que define un conjunto de comportamientos a través de operaciones ofrecidas por un elemento, ya sea una clase, un subsistema u otro componente.</p>
	<p>Dependencia</p> <p>La dependencia entre paquetes se da si existiera una dependencia entre dos clases en diferentes paquetes. La dependencia se denota como una línea discontinua con punta de flecha formada por dos líneas.</p>
	<p>Realización</p> <p>La realización es una relación que se da entre un paquete y una interfaz, lo que indica que el paquete define su comportamiento a través de una o más interfaces. La realización entre un paquete y una interfaz se denota como una línea continua.</p>

La Fig. 3.4 muestra un diagrama de paquetes, donde los paquetes representan los subsistemas en que puede ser dividido el sistema de préstamos de libros de una biblioteca. En esta figura se tienen 3 subsistemas, donde el subsistema “Reservaciones” contiene aquellos elementos que tienen que ver con el proceso de reservaciones de libros, y proporciona una interfaz con operaciones que proveen información al subsistema “Préstamos” para el caso particular de prestar un libro con reservación.

El subsistema “Control de libros” contiene aquellos elementos de control de los datos de los libros, como altas, bajas y cambios de dichos datos, y proporciona una interfaz con operaciones que son utilizadas por los otros dos subsistemas, por ejemplo, una reservación o un préstamo necesitan utilizar la operación “buscar_libro”.

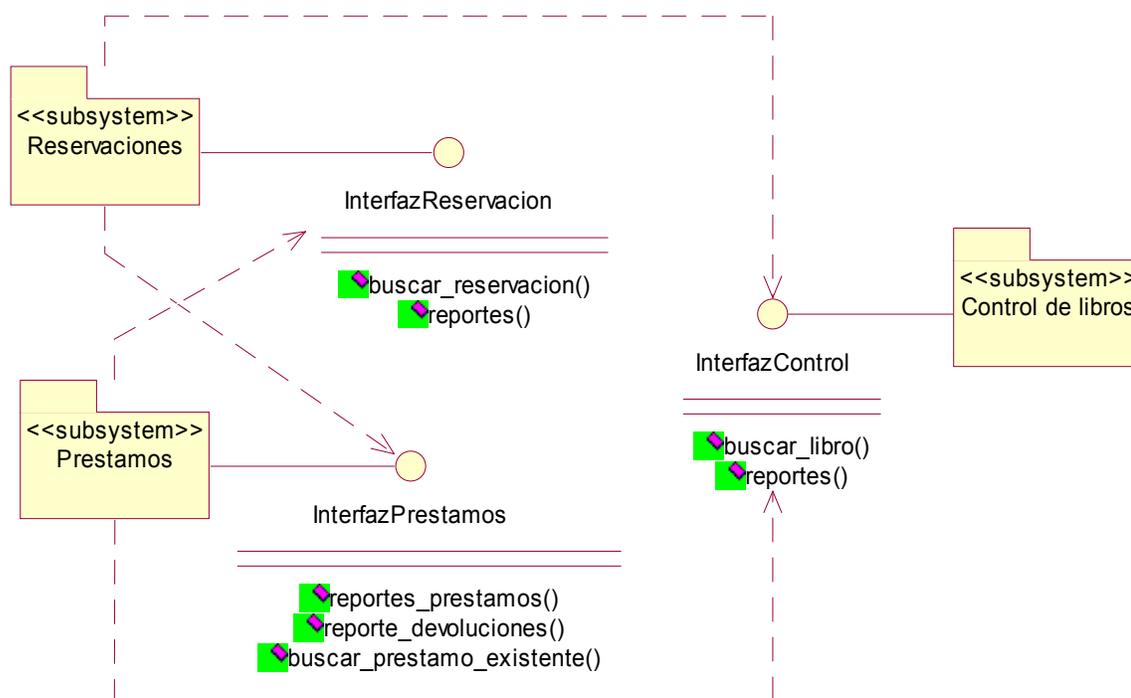


Fig. 3.4: Diagrama de paquetes del sistema de préstamo de libros de una biblioteca ⁴

3.2.4 Diagrama de interacción

La vista de los diagramas de interacción modela las secuencias de intercambio de mensajes entre objetos. Un diagrama de interacción puede ser de dos tipos: diagrama de secuencia o diagrama de colaboración. El primero se organiza de acuerdo al tiempo y último de acuerdo al espacio [Schmuller 00].

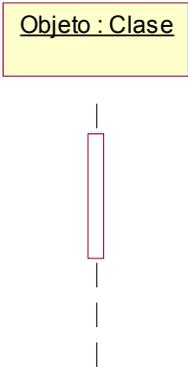
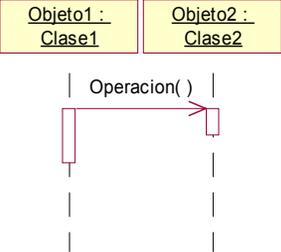
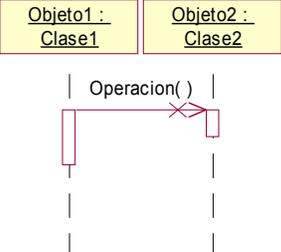
3.2.4.1 Diagrama de secuencia

Un diagrama de secuencia muestra un conjunto de mensajes que se envían de un objeto a otro a través del tiempo. El propósito de estos diagramas es mostrar la secuencia del comportamiento de la especificación de realización de un caso de uso. En el diagrama de secuencia los mensajes corresponden a las operaciones de las clases, es decir, cuando los objetos solicitan la realización de cierta operación a otros objetos o a ellos mismos.

El diagrama de secuencia consta de tres elementos principales: objetos, mensajes y el tiempo. Estos se describen en la siguiente tabla 3.6. [Figueroa 97], [Rumbaugh 00] y [Schmuller 00]

⁴ Ejemplo basado en el sistema de biblioteca de la Universidad Tecnológica de la Mixteca.

Tabla 3.6: Elementos principales del diagrama de secuencia

	<p>Objeto</p> <p>Los objetos son instancias de las clases, y se colocan en la parte superior del diagrama, formados de izquierda a derecha, se representan mediante un rectángulo con el nombre del objeto y de la clase dentro. En el diagrama de secuencia comúnmente son las instancias de los actores quienes comienzan con el envío de mensajes y reciben la respuesta final.</p> <p>Los objetos constan además, de una línea vertical punteada que desciende del objeto, llamada “línea de vida”, que es la representación del tiempo de existencia del objeto. Cuando se hace un envío de un mensaje, se dibuja un rectángulo sobre la línea de vida, llamada “activación”, que representa el tiempo de ejecución de un procedimiento.</p>
<p><i>sin imagen</i></p>	<p>Mensaje</p> <p>Un mensaje representa el envío de información de un objeto a otro, con el fin de que el objeto emisor haga una llamada a una operación del objeto receptor. Un mensaje pasa de la línea de vida de un objeto a la de otro, pudiendo un objeto enviarse un mensaje a sí mismo. Un mensaje puede ser síncrono o asíncrono.</p>
	<p>Mensaje asíncrono</p> <p>Un mensaje asíncrono es el caso más simple de envío de mensajes, se da cuando un objeto envía un mensaje y no espera por una respuesta para continuar con otros envíos, se representa como una flecha con la punta formada por dos líneas.</p>
	<p>Mensaje síncrono</p> <p>Un mensaje síncrono es cuando el objeto que envía el mensaje, espera la respuesta a tal mensaje antes de continuar con otros envíos, y se representa como una flecha con la punta rellena. En algunas herramientas como Rational Rose se representa como una flecha con la punta formada por dos líneas antecedida por los símbolos ><.</p>
<p><i>Sin imagen</i></p>	<p>Tiempo</p> <p>En el diagrama de secuencia, el tiempo es el recorrido de los mensajes en dirección vertical. El tiempo se inicia en la parte superior y avanza hacia la parte inferior, los mensajes se llevan a cabo por su posición en el diagrama, siendo los que estén más arriba los que se lleven primero a cabo. El diagrama de secuencia tiene dos dimensiones: la dimensión horizontal que muestra la disposición de los objetos, y la dimensión vertical que muestra el paso del tiempo.</p>

La Fig. 3.5 muestra el diagrama de secuencia para el curso normal de los eventos del caso de uso “Reservar libro en línea” en un sistema de préstamo de libros en una biblioteca.

En esta figura se puede observar el flujo de los eventos en cuanto al tiempo, ya que siguiendo las flechas que representan los envíos de mensaje se observa el orden en que las diferentes operaciones deben ser llamadas, desde que se realiza una búsqueda de un libro, hasta completar una reservación.

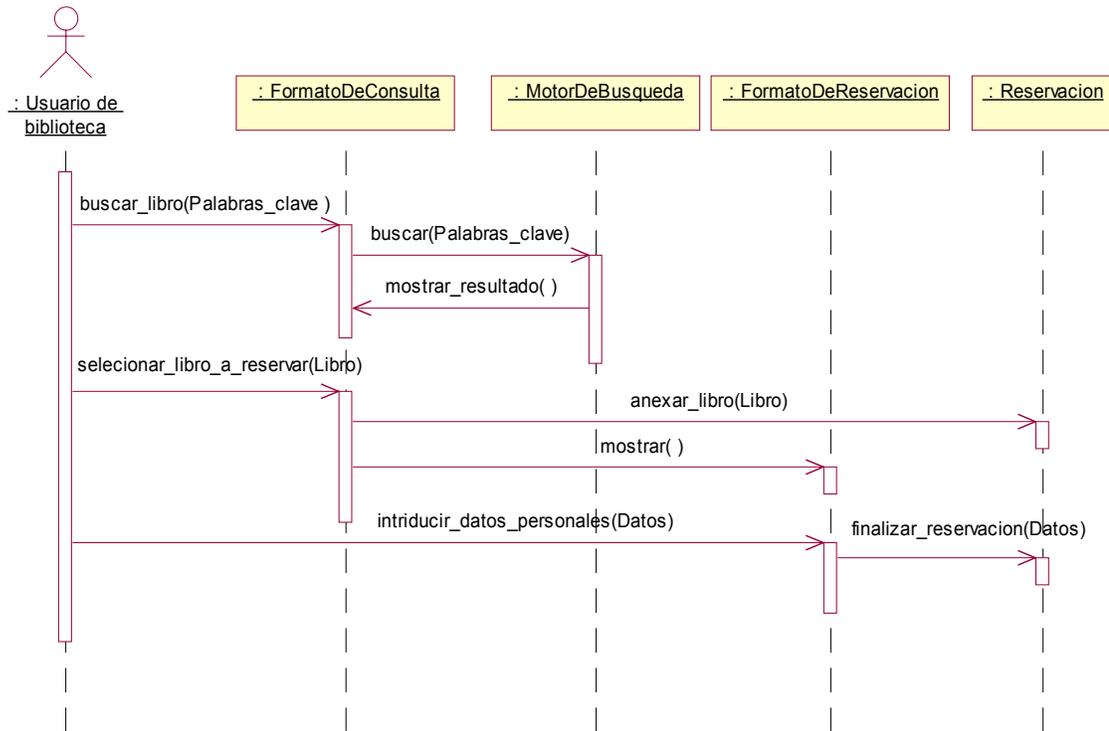


Fig. 3.5: Diagrama de secuencia para el caso de uso “Reservar libro en línea”⁵

3.2.4.2 Diagrama de colaboración

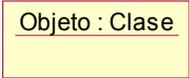
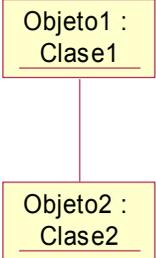
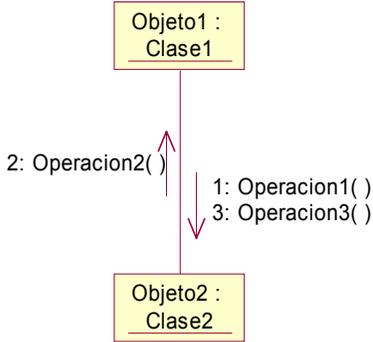
Un diagrama de colaboración es otra forma de representar la interacción entre objetos. Al igual que el diagrama de secuencia muestra los mensajes que se envían un objeto a otro, pero la diferencia radica en que los diagramas de secuencia pueden mostrar el contexto de la operación, es decir, muestra la forma en cómo se vinculan los objetos entre sí y se pueden visualizar ciclos durante la ejecución.

Básicamente un diagrama de colaboración representa la misma información que un diagrama de secuencia, por lo se puede convertir un diagrama de colaboración en un uno de secuencia y viceversa.

⁵ Diagrama basado en el sistema de biblioteca de la Universidad Tecnológica de la Mixteca.

El diagrama de colaboración consta de tres elementos principales: objetos, enlaces o vínculos, y mensajes, los cuáles se describen en la tabla 3.7. [Figueroa 97], [Rumbaugh 00] y [Schmuller 00]

Tabla 3.7: Elementos principales del diagrama de colaboración.

	<p>Objeto</p> <p>Al igual que en el diagrama de secuencia, los objetos se representan como un rectángulo con el nombre del objeto y de la clase dentro. También se suele representar a los actores como objetos que intervienen en el envío de mensajes y se representan con una figura en forma de persona con su nombre debajo, equivalente al diagrama de casos de uso.</p>
	<p>Enlace o vínculo</p> <p>El enlace es una trayectoria de conexión entre dos instancias de clases; este enlace es una instancia de una asociación del diagrama de clases. Se representa como una línea continua que une a dos objetos.</p>
	<p>Mensajes</p> <p>Los mensajes representan el envío de información de un objeto a otro. Un mensaje puede ir de un objeto a otro o bien puede ser dirigido al mismo objeto; al igual que en el diagrama de secuencia el mensaje puede ser síncrono o asíncrono.</p> <p>El envío de mensajes se representa mediante una flecha cerca del enlace, la flecha va dirigida del objeto emisor al objeto receptor. Cerca de esta flecha se escriben todos los mensajes al objeto receptor; cada uno de estos mensajes va acompañado por un número que indica el orden dentro de la interacción.</p>

La Fig. 3.6 muestra el diagrama de colaboración para el curso normal de los eventos del caso de uso “Reservar libro en línea” en un sistema de préstamo de libros en una biblioteca.

En esta figura se puede observar el flujo de los eventos en cuanto al espacio, ya que se aprecian cómo están relacionados los objetos participantes. Por ejemplo, se puede ver que el usuario de biblioteca solicita dos operaciones a la pantalla del formato de consulta y una a la pantalla del formato de reservación. Aunque este diagrama pierde visualmente la secuencia del envío de mensajes, se puede seguir esta secuencia mediante los números que poseen cada una de las actividades, nótese que la operación número 1 es “buscar_libro” y la última operación (número 8) es la operación “finalizar_reservacion”, que corresponden al inicio y final del diagrama de secuencia de la Fig. 3.5.

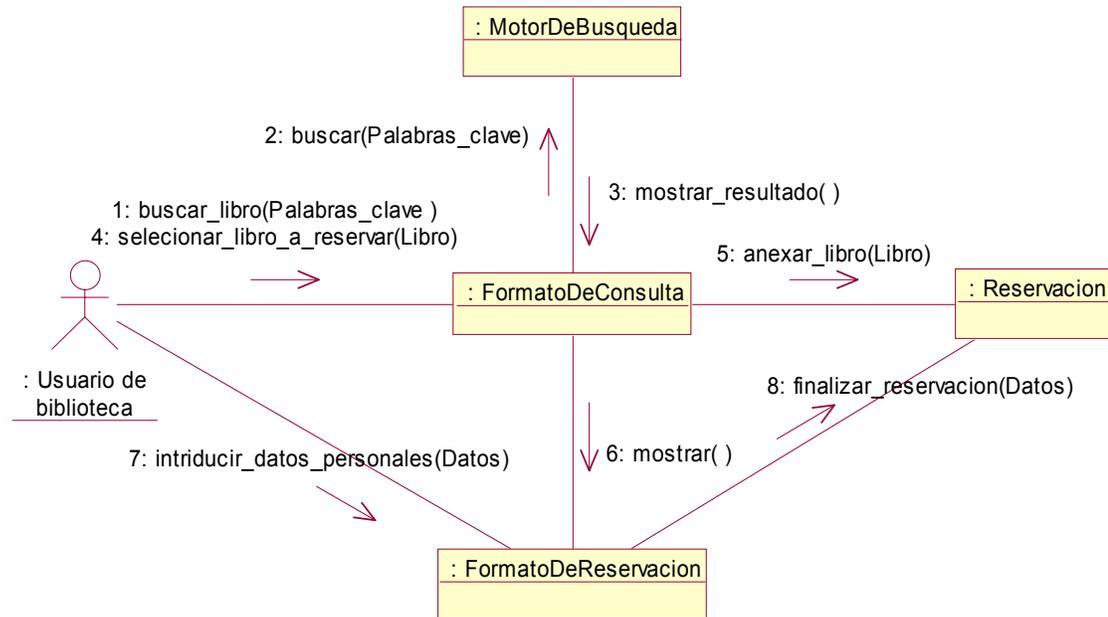


Fig. 3.6: Diagrama de colaboración para el caso de uso “Reservar libro en línea”⁶

Una vez explicados los diagramas de UML que se utilizaron para el desarrollo del proyecto, se revisa un poco al Proceso Unificado, el cual utiliza estos diagramas para el modelado del sistema a lo largo del ciclo de vida del software.

⁶ Diagrama basado en el sistema de biblioteca de la Universidad Tecnológica de la Mixteca

Capítulo 4: El Proceso Unificado de Desarrollo de Software

4.1 Introducción

Como ya se ha mencionado, actualmente la tendencia de los sistemas es construir software cada vez más grande y poderoso, y no sólo eso sino que además los clientes quieren que éste se construya lo más rápido posible. Por tal motivo, los desarrolladores se enfrentan a los problemas del análisis y diseño del software que tienen mucho que ver con las dificultades al momento de coordinar las múltiples actividades que envuelve un proyecto de desarrollo de software.

4.1.1 ¿Qué es el Proceso Unificado?

El Proceso Unificado, es un proceso de desarrollo de software, que Jacobson lo define como “el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema de software”. [Jacobson 00] Pero el UP es más que eso; es una guía de trabajo que se puede adaptar a diferentes tipos de sistema de software, diferentes campos de acción y diferentes tamaños de proyecto.

El UP es un proceso de ingeniería de software que tiene como objetivo proveer una disciplina para la asignación de tareas y responsabilidades dentro de una organización de desarrollo. Todo ello con el objeto de asegurar la producción de software de gran calidad que satisfaga las necesidades de sus usuarios finales, dentro de un tiempo y presupuesto previsible. [Rational 98].

Las características que definen al UP son:

- Está dirigido por casos de uso
- Está centrado en la arquitectura y
- Es iterativo e incremental.

El estar dirigido por casos de uso significa que el proceso se centra en lo que debe hacer el sistema, comenzando por especificar sus requisitos. Los casos de uso representan los requisitos funcionales del sistema y el conjunto de todos estos casos de uso forman lo que se le conoce como modelo de casos de uso, el cual contiene la descripción de todo el sistema. Los casos de uso además de ayudar a especificar los requisitos, sirven de guía durante todo el proceso, ya que se toman como base para el diseño, la implementación y las pruebas. Los casos de uso son el punto de partida del UP

Al estar centrado en la arquitectura, el UP describe al sistema en varios puntos de vista, es decir, la forma en cómo ven al sistema las personas implicadas en él. La arquitectura es la forma que tendrá el sistema y sirve como base para comprender cómo quedará una vez terminado, al principio se crea una arquitectura borrador, es decir, una comprensión general de lo que debe hacer el sistema; más adelante esta arquitectura se irá reforzando conforme se vayan analizando los casos de uso.

Es un proceso iterativo e incremental ya que divide el proyecto en pequeños mini-proyectos, donde a cada uno de estos se les conoce como iteraciones, las cuales una vez terminadas significan un incremento en el proyecto, es decir, una iteración representa una serie de pasos para lograr un objetivo (el objetivo del mini-proyecto), entonces conforme estos se vayan logrando el crecimiento del sistema se irá incrementando.

Estas tres características son de igual importancia, ya que en conjunto son la fortaleza del UP. “La arquitectura proporciona la estructura sobre la cual guiar las iteraciones, mientras que los casos de uso definen los objetivos y dirigen el trabajo en cada iteración”. [Jacobson 00]. Más adelante se explica a detalle el porqué estas características hacen tan poderoso al UP

4.1.2 Antecedentes del Proceso Unificado

El UP es considerado por sus creadores como un proceso completo para el desarrollo de software, respaldado por más de tres décadas de desarrollo y de aplicaciones prácticas. A continuación se presenta una síntesis de la evolución del UP.

El UP tiene sus orígenes en 1967 en el modelo de Ericsson, el cual era similar al de un desarrollo basado en componentes, cuyo creador fue Ivar Jacobson, mismo que siguió mejorando su método durante los años siguientes, hasta que en 1987 abandona a Ericsson y funda Objectory AB, organización dedicada al desarrollo del proceso llamado Objectory, que significa “fabrica de objetos”.

En ese entonces ya existían los lenguajes de descripción y especificación, tal es el caso del SDL (*Specification and Description Languages* – Lenguaje de Especificación y Descripción) que era considerado como un estándar para el modelado de objetos. Este lenguaje fue influenciado por el método de Ericsson, y se utilizó como lenguaje de modelado de Objectory. Objectory creció desde la primera versión en 1988 hasta la versión 3.8 en 1995.

En 1995 Rational Software Corporation compró a Objectory AB, a partir de donde surgió la idea de unificar los procesos de desarrollo. En 1996 surge Objectory de Rational 4.1 que contenía la experiencia y práctica de Rational. En ese año UML estaba en fase de desarrollo y se incorporó como el lenguaje de modelado de Objectory 4.1.

A finales de 1997, Rational se fusionó con otras empresas de ingeniería de software para mezclar sus experiencias en el área de procesos de desarrollo para mejorar a Objectory. En Junio de 1998 Rational publicó el RUP (*Rational Unified Process* - Proceso Unificado de Rational) 5.0. A partir de donde se tiene un Proceso Unificado para soportar todo el ciclo de vida de un sistema de software, el cual se sigue mejorando con la ayuda de muchas empresas y desarrolladores.

Actualmente Rational Software Corporation es parte de IBM.

4.2 La vida del Proceso Unificado

El proceso unificado está basado en 4 fases que se repiten a lo largo del tiempo de vida del software (ver Fig. 4.1), cada una de estas fases a su vez, se dividen en un conjunto de iteraciones que se repiten a lo largo de una serie de ciclos, donde al término de cada ciclo se obtiene una versión del software listo para ser entregado al cliente.

Una iteración puede pasar por los 5 flujos de trabajo fundamentales del Proceso Unificado. La Fig. 4.1 muestra estos cinco flujos de trabajo y cómo se llevan a cabo durante las cuatro fases.

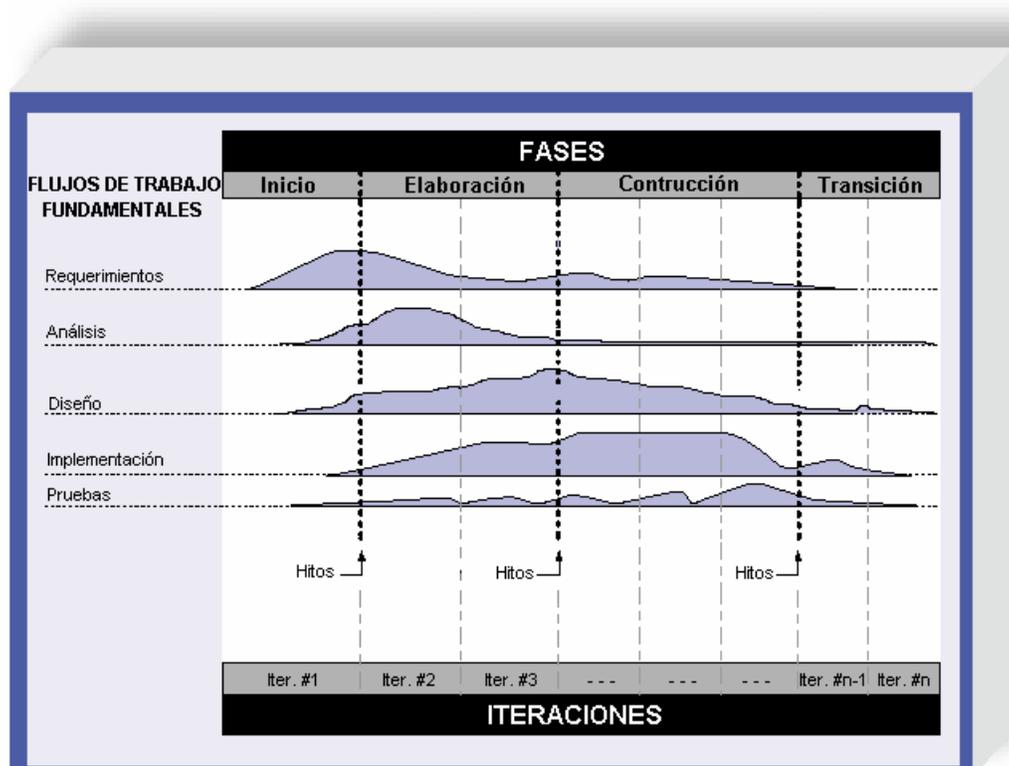


Fig. 4.1: Relación entre los cinco flujos del trabajo fundamentales y las cuatro fases del Proceso Unificado⁷

Los 5 flujos de trabajo fundamentales del Proceso Unificado son:

- Requisitos (*Requirements*)
- Análisis (*Analysis*)
- Diseño (*Design*)
- Implementación (*Implementation*)
- Pruebas (*Test*)

⁷ Figura editada, basada en el original en [Rational 00]

Cada uno de estos flujos de trabajo será explicado en el capítulo 5 junto con el desarrollo de UMLGEC++.

Las 4 fases del Proceso Unificado son:

- Inicio (*Inception*)
- Elaboración (*Elaboration*)
- Construcción (*Construction*)
- Transición (*Transition*)

Cada una de estas fases se descompone en un conjunto de iteraciones, y termina siempre con hito, el cual tiene una serie de objetivos que varía dependiendo de la fase, donde el objetivo más importante en cada una es la posible toma de decisiones antes de pasar a otra fase.

4.2.1 Fase de inicio

En esta fase se obtiene básicamente la descripción del problema a resolver mediante los casos de uso. Se describen las principales funciones que el sistema debe desempeñar, su arquitectura y un plan de tiempo y costo del proyecto.

4.2.2 Fase de elaboración

En esta fase se definen y especifican a detalle los casos de uso encontrados en la fase anterior y con ello se diseña la arquitectura del sistema creando diferentes vistas del sistema. Al término de esta fase se planifican las actividades y los recursos necesarios para la realización del proyecto.

4.2.3 Fase de construcción

Durante la fase de construcción se lleva a cabo la construcción del producto, se parte de la arquitectura creada en la fase de elaboración y se llevar el trabajo hasta tener el sistema completo. Al término de esta fase, el sistema realizará las funciones especificadas por los casos de uso definidos en la fase de elaboración, pero es probable que en esta fase el sistema tenga errores, los cuales se irán detectando y resolviendo en la fase de transición.

4.2.4 Fase de transición

Esta fase se puede considerar como una fase de prueba en la que un número de usuarios verifican el sistema e informan de los posibles errores o defectos que éste pueda tener. Una vez encontradas las fallas los desarrolladores resuelven aquellas que sean críticas para el cumplimiento de los requerimientos del sistema, y con ello poder entregar una versión del producto.

4.3. Elementos del proceso unificado

Un proceso define *quién* está haciendo *qué*, *cuándo* lo hace y *cómo* hacerle para alcanzar un objetivo. [Rational 98] Dentro del Proceso Unificado, estas preguntas son representadas en cuatro elementos:

- Trabajadores: El quién
- Actividades: El cómo
- Artefactos: El qué
- Flujos de trabajo: El cuándo

Trabajadores

Los trabajadores definen los puestos que las personas pueden adoptar, es decir, un trabajador es un papel que un individuo puede desempeñar durante el desarrollo del software.

Cada trabajador tiene un conjunto de responsabilidades y lleva a cabo un conjunto de actividades en el desarrollo de software. Un trabajador puede representar a un conjunto de personas que trabajan juntas. Para el caso del desarrollo del presente proyecto, no se hace énfasis en la diferencia entre los trabajadores, dado que todos los papeles los desempeña una sola persona.

Actividades

Una actividad es una parte de trabajo que un trabajador realiza, produciendo con ello un resultado significativo, lo que representa una unidad de trabajo con límites bien definidos para facilitar la asignación de tareas.

Artefactos

Un artefacto es una pieza de información que es producida, modificada o utilizada en un proceso, en si, es cualquier tipo de información que los trabajadores pueden usar.

Flujos de trabajo

Un flujo de trabajo es un conjunto de actividades, y es el modo en el que se describe un proceso de desarrollo. Ya se han mencionado los 5 flujos principales del Proceso Unificado.

4.4 Características del Proceso Unificado

El Proceso Unificado se sostiene sobre tres ideas básicas: casos de uso, arquitectura, y desarrollo iterativo e incremental.

4.4.1 Un proceso dirigido por casos de uso

El objetivo del proceso unificado es guiar a los desarrolladores en la implementación y distribución eficiente de sistemas que se ajusten a las necesidades de los clientes, por tanto, es necesario que se encuentre la forma de capturar éstas necesidades de forma que puedan comunicarse fácilmente a todas las personas implicadas en el proyecto [Jacobson 00]. Posteriormente, se debe diseñar una implementación funcional que se ajuste a esas necesidades. Por último se debe de verificar que se han cumplido todas las necesidades del cliente mediante pruebas del sistema. Debido a todo esto es que el Proceso Unificado se describe como una serie de flujos de trabajo a lo largo de toda la vida del software.

La Fig. 4.2 muestra los flujos de trabajo fundamentales y los modelos del Proceso Unificado. En ella se puede observar que los desarrolladores inicialmente deben capturar los requisitos del cliente mediante los casos de uso y representarlos en el modelo de casos de uso. Después se debe analizar y diseñar el sistema que cumpla con los casos de uso, para ello primero se crea un modelo de análisis, en seguida uno de diseño, después uno de despliegue o distribución que especifica la ubicación de los componentes del sistema. Posteriormente un modelo de implementación, que es en donde se encuentra el código de los componentes del sistema. Por último se crea un modelo de prueba utilizado para verificar los resultados del sistema con los casos de uso.

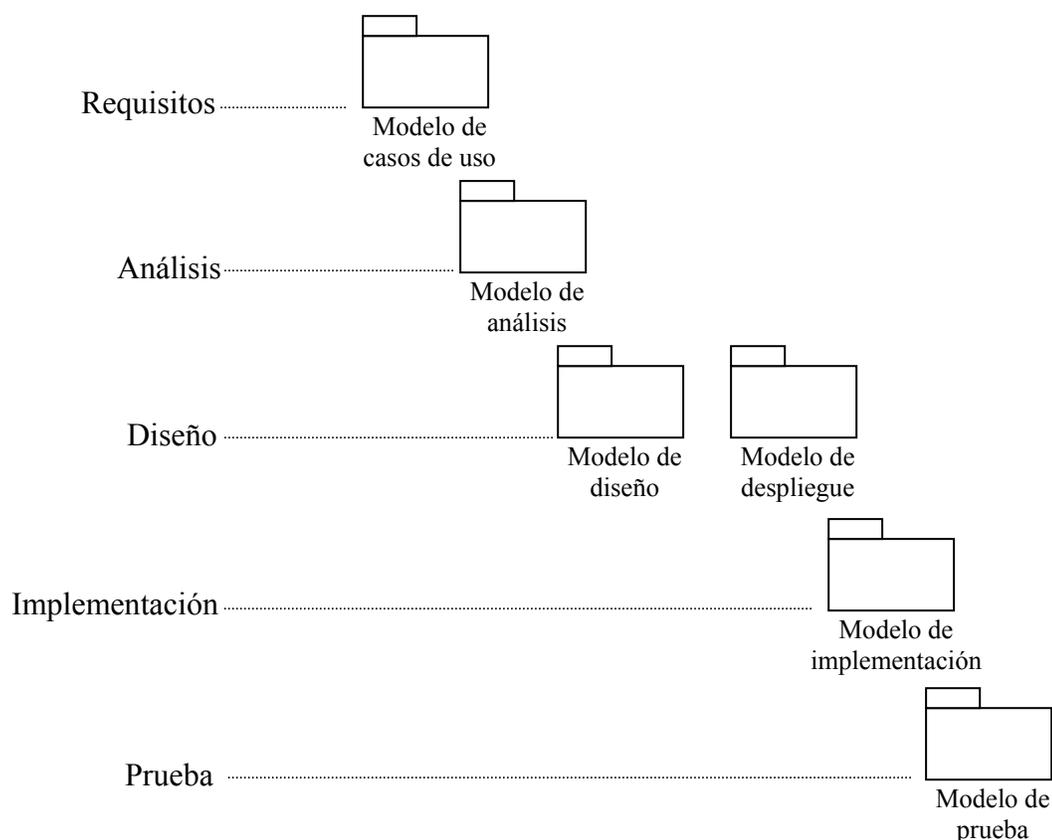


Fig. 4.2: *Los flujos de trabajo fundamentales del UP y los modelos que desarrollan.* ⁸

⁸ Figura obtenida de [Jacobson 00]

La captura de requisitos tiene dos objetivos principales: Encontrar los verdaderos requisitos, y representarlos de forma adecuada a cada una de las personas involucradas en el proyecto.

Los mejores casos de uso son aquellos que añaden el mayor valor al propósito que persigue el sistema. El modelo de casos de uso ayuda a delimitar el sistema definiendo todo lo que debe hacer, con ello se planifica y controla muchas de las tareas que los desarrolladores realizan, también ayudan a idear la arquitectura.

En conclusión, los casos de uso dirigen el proceso y enlazan todas las actividades del proceso de desarrollo. Durante la captura de requisitos, éstos se pueden representar en forma de casos de uso. Durante el análisis y el diseño se crean realizaciones de casos de uso en términos de clases y subsistemas. Y se hacen pruebas que verifican que los casos de uso se han implementado correctamente.

4.4.2 Un proceso centrado en la arquitectura

Se ha dicho que los casos de uso guían el proceso de desarrollo para desarrollar un sistema, pero solos no son suficientes, se necesita de una arquitectura para tener una clara representación del sistema en su totalidad, necesaria para controlar el desarrollo.

La arquitectura describe los elementos del modelo que sea más útil para la guía de trabajo con el sistema, a través de todo el ciclo de vida. Precisamente, el objetivo de la fase de elaboración es establecer una arquitectura sólida de forma que sea de gran utilidad para la fase de construcción.

Un sistema de software es una entidad que debe ser representada en diferentes perspectivas para que su diseño pueda ser apreciado por todos los involucrados en su desarrollo. Las perspectivas son las vistas del modelo del sistema (véase la sección 3.2), y son todas estas vistas las que representan a la arquitectura.

4.4.2.1 Necesidad de la arquitectura

En un sistema de software grande y complejo se requiere de una arquitectura para que todos los desarrolladores tengan una visión en común del sistema. La arquitectura es necesaria para comprender el sistema, organizar el desarrollo, fomentar la reutilización y hacer evolucionar el sistema [Jacobson 00]. Mientras mayor sea la organización del proyecto, mayor será la sobrecarga de comunicación entre los desarrolladores para coordinarse.

Una buena arquitectura ofrece una plataforma estable, ésta se crea a partir de la creación de los componentes reutilizables, diseñados de tal forma que puedan ser utilizados conjuntamente. El UML ayuda a este proceso, ya que crea componentes específicos que pueden estar disponibles para su reutilización.

4.4.2.2 Casos de uso y arquitectura

Se ha visto que los casos de uso interaccionan con la arquitectura. Si el sistema proporciona los casos de uso correctos, los usuarios no tendrán mayor problema para llevar a cabo sus objetivos

usando el sistema. Para lograr obtener esos casos de uso es necesario construir una arquitectura que nos permita implementar los casos de uso de una forma económica en todo momento [Jacobson 00].

Se ha dicho que los casos de uso dirigen a la arquitectura, pero también se ha dicho que se utiliza el conocimiento de la arquitectura para obtener nuevos casos de uso. Dicho de otra forma, la arquitectura guía los casos de uso.

Entonces existe una duda en cuanto a qué es lo primero que hay que resolver, los casos de uso o la arquitectura; la respuesta es que hay que construir primeramente una arquitectura básica a partir de la comprensión del dominio en que se desarrollará el sistema, crear los casos de uso sin ser detallados, a partir de ahí, se seleccionan algunos casos de uso y se hace que la arquitectura se adapte a ellos, con esto se logra que se fortalezca más la arquitectura; posteriormente se seleccionan otros casos de uso y se sigue el mismo proceso.

4.4.3 Un proceso iterativo e incremental

Un proceso de desarrollo eficaz debe tener una serie de hitos, los cuales proporcionarán los criterios suficientes para saber en que momento se puede pasar de una fase a otra dentro del ciclo de vida del sistema. Dentro de cada fase, el proceso unificado pasa a través de una serie de iteraciones, las cuales conducen a esos criterios.

Los criterios fundamentales dentro de cada una de las etapas del proceso unificado son las siguientes: En la fase de inicio el criterio es la viabilidad; en la fase de elaboración es la capacidad de construir el sistema de forma económica; en la fase de transición es tener un sistema que alcance una operatividad final.

Ya se han revisado dos de las tres características del Proceso Unificado, que son el estar dirigido por casos de uso y estar centrado en la arquitectura. Estas dos características deben tener un equilibrio en cuanto a la forma y función del desarrollo. Este equilibrio se consigue con el tiempo a lo largo de una serie de iteraciones

La tercera característica del proceso proporciona la estrategia para desarrollar un producto de software en pasos pequeños y manejables:

- Planificar un poco.
- Especificar, diseñar, e implementar un poco.
- Integrar, probar, y ejecutar un poco cada iteración.

4.5 Los flujos de trabajo fundamentales

Existe otro grupo de flujos de trabajo que no se han mencionado debido a que no son necesarios en todos los proyectos, como en éste, se les llama flujos de soporte y se utilizan para proyectos a gran escala. También, antes de la fase de captura de requisitos existe una fase llamada Modelado de Negocios; esta fase se aplica cuando el proyecto está ligado a procesos de negocios en los que

una cantidad grande de gente tenga interacción con el sistema [Fernández 01], por tanto, dado que la herramienta no tiene nada que ver con procesos de negocios y no interactúan con ella un número considerable de personas, no se llevó a cabo este flujo de trabajo.

Los flujos de trabajo no se llevan a cabo una sola vez durante el Proceso Unificado, estos se repiten durante las iteraciones que se tengan a largo de las 4 fases, pero tampoco quiere decir que por cada vez que se termine un ciclo se hayan llevado a cabo los 5 flujos de trabajo. Los flujos de trabajo se utilizan siempre y cuando sean necesarios para determinada iteración; por ejemplo, como se observó en la Figura 4.1 en la primera iteración de la fase de inicio se pueden utilizar sólo los primeros 3 flujos de trabajo y en una iteración de la fase de elaboración tienen lugar los cinco.

En el siguiente capítulo se explica el seguimiento de los 5 flujos de trabajo fundamentales del Proceso Unificado para el desarrollo de la herramienta CASE, explicando los avances obtenidos a través de cada una de las iteraciones del UP hasta llegar a la versión ejecutable.

Capítulo 5: Desarrollo de la herramienta UMLGEC++

5.1 Introducción

La herramienta UMLGEC++ se desarrolló bajo el paradigma de la Programación Orientada a Objetos, dado que es sabido que este paradigma tiene muchas ventajas sobre otros, como la Programación Estructurada. Además, es razonable pensar que una herramienta creada para ayuda en la Programación Orientada a Objetos fue desarrollada bajo la misma técnica.

Durante el ciclo de vida de la herramienta se siguió el Proceso Unificado, y como lenguaje de modelado al UML. Se ha discutido en el Capítulo 3 acerca de las características del UML y del porqué es considerado como un buen lenguaje de modelado para el análisis y diseño de sistemas de software; así mismo, en el Capítulo 4, se ha examinado el Proceso Unificado, con lo que se resume que este Proceso es el más idóneo para trabajar con el UML, principalmente por haber sido creado para trabajar de la mano con el UML y haber sido impulsado por los mismos autores del UML.

El presente capítulo muestra el seguimiento del desarrollo de UMLGEC++ siguiendo el ciclo de vida del Proceso Unificado.

5.2 Requerimientos

El propósito fundamental de la captura de requisitos es obtener una descripción correcta de lo que debe de hacer el sistema y delimitar su alcance, es decir, qué debe y qué no debe hacer.

Los requisitos juegan un papel importante durante el ciclo de vida del software. Durante la fase de inicio, se identifican la mayoría de los casos de uso para delimitar el sistema y detallar los más importantes. En la fase de elaboración se obtiene el resto de casos de uso y se planean las tareas que se tienen que llevar a cabo para la elaboración del sistema. Si se detectaran algunos otros, se capturan e implementan en la fase de construcción. Finalmente en la fase de transición, sólo hay captura de requisitos si se tuviera que modificar algún caso de uso.

El artefacto obtenido es este flujo de trabajo es el modelo de casos de uso que incluyen los casos de uso y los actores, de los cuales se ha hablado en las secciones 3.2.1 y 4.4.1.

5.2.1 Evaluación de herramientas CASE

Antes de comenzar con la especificación de requerimientos, se debía tener en claro qué es lo que la herramienta debía hacer, para ello fue necesario hacer una evaluación de algunas versiones de herramientas CASE que permiten trabajar con UML, las cuáles contienen entre sus funciones la creación de diagramas de clase, algo equivalente a lo que se pretende desarrollar.

Las herramientas para ser evaluadas fueron aquellas que los autores de la bibliografía utilizada las citan como herramientas de ejemplo, y algunas otras de las que fue posible obtener una versión ejecutable. Al final se contó con las siguientes herramientas:

- GdPro 5.0
- Objecteering 4.3.1a
- Rational Rose 2001
- Rhapsody 2.2
- Visual UML 2.7

Los ambientes de diseño de estas herramientas eran completamente desconocidas antes de llevar a cabo la evaluación, con lo que al ir analizando cada una de ellas, se fueron considerando los siguientes criterios:

- Facilidad de uso del ambiente: Considerando la interfaz del usuario, ¿Que tan fácil es adaptarse al ambiente que proporciona la herramienta para trabajar?
- Facilidad de uso al momento de crear un diagrama de Clases: Nuevamente con la interfaz del usuario, ¿Qué tan cómodo y rápido resulta crear un diagrama de clases dentro de la herramienta?
- Semejanza en la notación que maneja a la notación estándar de UML: Tomando en cuenta la versión 1.3 de UML, ¿Que tan semejante es la notación para diagramas de clases que maneja la herramienta con la notación estándar?
- Total de la notación UML soportada: También con la versión 1.3 de UML, ¿Qué porcentaje de la notación del UML para diagramas de clases está implementada en la herramienta?
- Asistencia al usuario: ¿La herramienta cuenta con ayuda suficiente para su uso?
- Revisor de sintaxis: ¿La herramienta cuenta con revisión de sintaxis en momento de edición y qué tan eficiente es?

Es necesario aclarar que este análisis fue aplicado sólo a la parte de creación de diagramas de clase y no fue llevado a cabo mediante un proceso formal, ya que no es un objetivo de la tesis el determinar el nivel de usabilidad de cada herramienta. Se limitó únicamente a los criterios de los posibles usuarios del sistema; en este caso, el tesista fue el encargado de valorarlas. También es importante resaltar que éstas herramientas en su mayoría son versiones gratuitas, por lo que algunas tienen ciertas restricciones como fecha de expiración y funciones deshabilitadas.

En la tabla 5.1 se encuentran los resultados de las evaluaciones de las herramientas. Cada columna del cuadro representa los productos evaluados con su calificación respectiva de 0 a 10 (mostrado entre paréntesis), siendo 10 la calificación más satisfactoria. Los renglones representan los elementos comparados y evaluados, los cuales tiene asignado un porcentaje que representa la importancia que se le dio durante la evaluación.

Tabla 5.1: *Tabla comparativa de herramientas CASE con soporte a UML*

Herramienta Característica	GdPro 5.0	Objecteering 4.3.1a ⁹	Rational Rose 2001	Rhapsody 2.2	Visual UML 2.0
Facilidad de uso en general (20%)	De las interfaces más difíciles de entender. Tiene demasiadas barras de herramientas y ventanas, algo incómodo son los iconos que tienen demasiados colores. (7)	Herramienta fácil de entender. Contiene una barra de herramientas en donde se representan las vistas y otra donde se muestran los componentes de la vista activa. Contiene además un ventana tipo explorador (8)	Buena interfaz, una de las más fáciles de usar durante esta evaluación, contiene una ventana que sirve como explorador donde se agrupan los componentes de acuerdo a las vistas de UML (9)	Interfaz sencilla y simple, contiene una serie de iconos que representan los diferentes tipos de diagramas de UML que se deseen crear. (9)	Contiene una interfaz ordenada y simple. Contiene un panel tipo explorador para examinar los diagramas y componentes que se vayan creando. (9)
Facilidad para crear un diagrama de clases (20%)	Una dificultad que tiene en una primera instancia, es que no permite trazar libremente las relaciones, ya que impone una cierta dirección de las líneas. (7)	Edita clases dentro y fuera del diagrama. (7)	Edita clases y sus relaciones dentro y fuera del diagrama, contiene características como el ajuste automático del tamaño de los componentes, además tiene mucha flexibilidad para el trazo de líneas. (10)	Edita clases sólo fuera del diagrama, su interfaz tiene gran parecido con Rose. Carece de características, como el ajuste automático de componentes. Flexible para el trazo de líneas. (9)	Edita clases dentro y fuera del diagrama. Contiene una barra de componentes con la notación del UML y con otras imágenes que no lo son, lo que permite insertar otro tipo de dibujos. (9)
Notación (20%)	La notación es un poco parecida a la estándar, tiene algunos componentes como clases y asociaciones. Permite agregar otro tipo de componentes que no son del UML (8)	La notación presenta algunas diferencias con la versión 1.3 (probablemente por la versión de la herramienta), en cuanto a las líneas de relación. (7)	La más cercana a la notación estándar, toda la notación que contiene es parte del UML. Y aunque no toda la notación está en la barra de herramientas, es posible encontrarla dentro de las pantallas de configuración. (9)	En su barra de herramientas, contiene casi toda la notación de UML para diagrama de clase, pero carece de compatibilidad al mostrar los estereotipos de las clases y las asociaciones. (8)	Notación muy parecida a la estándar. Permite configurar las líneas de relación y agregar otro tipo de componentes que no son propios del UML (8)
Asistencia al usuario (10%)	Contiene ayuda por temas, un tutorial, y ligas a soporte en línea. (10)	No se revisó (0)	Contiene ayuda por temas, ayuda interactiva, y ligas a soporte en línea. (10)	La versión Demo que se evaluó sólo trae ligas a ayuda en línea (8)	Contiene ayuda por temas y ligas a soporte en línea (8)
Revisor de sintaxis (30%)	Sólo corrige sintaxis antes de la generación de código y permite dejar líneas sin conexión (8)	No se revisó (0)	Corrige sintaxis en tiempo de edición, antes de la generación de código y como tarea independiente. (10)	Corrige sintaxis en tiempo de edición, antes de la generación de código y como tarea independiente (10)	Corrige sintaxis en tiempo de edición, antes de la generación de código y como tarea independiente (10)

⁹ Debido a los problemas con la licencia de esta herramienta, no fue posible evaluar en su totalidad sus características.

Una vez revisadas las herramientas arriba mencionadas, se obtuvo la siguiente gráfica de acuerdo al puntaje obtenido. La herramienta Objectteering no se incluye en la gráfica, dado que no se pudo evaluar completamente.

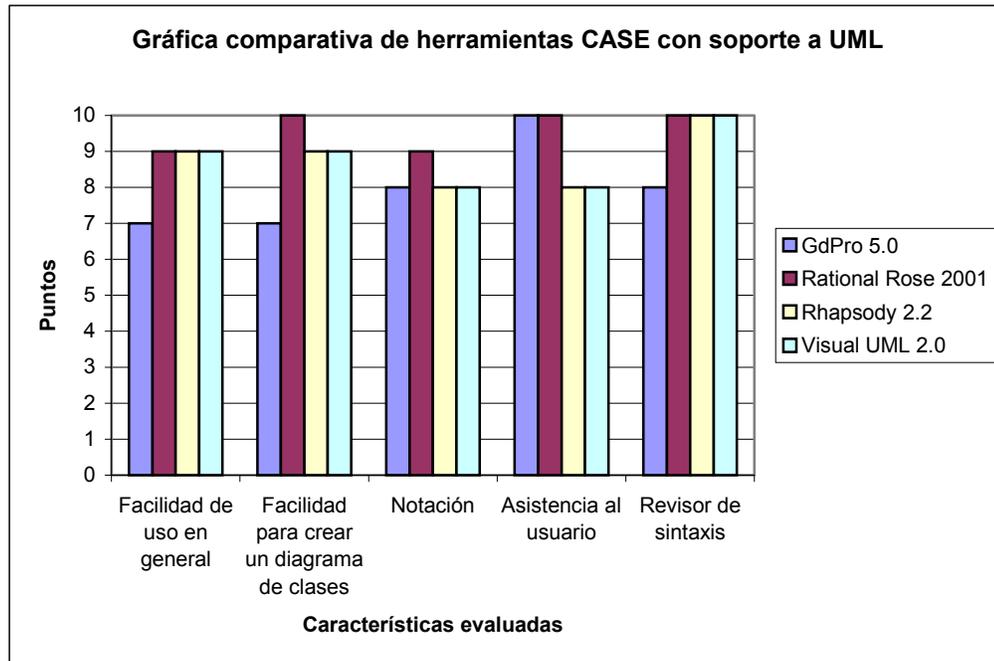


Fig. 5.1: Gráfica comparativa de herramientas CASE con soporte a UML

De aquí, sacando el puntaje promedio de cada herramienta, se obtiene la siguiente lista:

Herramienta	Puntaje
1. Rational Rose 2001	9.6
2. Rhapsody 2.2	9
3. Visual UML 2.7	9
4. GdPro 5.0	7.8

Rational Rose fue la herramienta que cumplió en mayor grado con los criterios establecidos, debido a que su interfaz es bastante entendible y su pantalla de edición de diagramas de clases es de las más amigables, ya que cuenta con ayuda suficiente y valida algunos puntos de sintaxis al momento de estar creando un diagrama. Además de que la notación que maneja es muy similar a la estándar y aunque tiene algunas pequeñas diferencias, contiene casi toda la notación de UML tanto para diagramas de clase como para el resto de diagramas. Es por esto que todos los diagramas UML para el desarrollo de UMLGEC++ fueron hechos en Rational Rose.

La herramienta Rhapsody tiene un comportamiento muy semejante al de Rational Rose en cuanto a su interfaz y ventanas de edición, pero tiene un poco más de diferencias en la notación comparada con la estándar. Cuenta también con validación de sintaxis al momento de la edición.

La interfaz de Visual UML aunque es clara y simple, es un poco más rigurosa la forma de hacer un diagrama de clases lo más completo posible como se podría hacer en las dos herramientas anteriores, por lo demás es semejante.

Para hacer uso de Objectteering fue necesario hacer un tedioso proceso de instalación debido a la licencia, y aunque al final se logró ejecutar la herramienta, el periodo de expiración de la licencia impidió la evaluación de algunas características.

GdPro tienen una interfaz complicada comparada con el resto de las herramientas. La forma de crear un diagrama no resulta del todo cómoda, ya que el configurar un componente de un diagrama de clases se logra después de una serie de pasos, lo que resulta muy tardado.

5.2.2 Encontrar actores y casos de uso

En esta actividad se delimita el sistema, se idea quién y qué actores interactúan con el sistema, y qué funcionalidad se espera del sistema, esto mediante los casos de uso y su descripción.

Como resultado de la evaluación de las herramientas CASE, UMLGEC++ tomó parte del comportamiento de las dos primeras herramientas arriba mencionadas, debido a la sencillez de uso de su interfaz y de su edición de diagramas de clase.

Tomando como principales fuentes de información a éstas herramientas, se procedió a la identificación de los requerimientos que UMLGEC++ cubre.

El proceso de identificación de los casos de uso y actores se llevó a cabo obteniendo una descripción de qué es lo que se desea que la herramienta haga, a continuación se muestra esta descripción.

UMLGEC++ permite crear diagramas de clase de UML en su versión 1.3, en donde estará presente la notación mencionada en la sección 3.2.2 “Diagrama de clases”.

Dentro de la herramienta el usuario podrá:

1. *Colocar dentro de una pantalla de edición cualquiera de los componentes representados en una barra de componentes de UML, los cuáles pueden ser: clases, notas, componentes de relación como conexión a nota, asociación, generalización, agregación y composición.*
2. *En un componente de clase agregar atributos y operaciones, para lo cual se utilizarán pantallas de configuración para darle un formato a cada atributo u operación. Así como una pantalla de configuración general de una clase que contendrá toda la información de una clase de UML: nombre, atributos, operaciones y estereotipo.*
3. *Hacer relaciones entre clases como son: asociación, agregación, composición, generalización y dependencia, permitiendo editar toda la sintaxis que marca el UML.*
4. *Sobre el componente de asociación, establecer un nombre, especificar los nombres y su multiplicidad a cada uno de los roles.*
5. *Generar el código fuente en lenguaje C++ correspondiente a la estructura estática.*
6. *Crear, guardar, y abrir un archivo con la información de un diagrama.*

7. *Manipular los diversos componentes para acomodar el diagrama en la pantalla.*
8. *Usar el portapapeles: copiar, cortar y pegar los componentes de clase y nota.*
9. *Imprimir el diagrama*
10. *Deshacer los últimos cambios hechos sobre el diagrama. Los cambios que se pueden deshacer son: mover una clase o una nota, borrar cualquier componente del diagrama.*

Del texto anterior se destaca tan sólo un actor que es el *usuario*, los casos de uso identificados se pueden ver en la Tabla 5.2

5.2.3 Priorizar casos de uso

Priorizar los casos de uso es una buena estrategia que aunque no es parte del estándar de UML, ayuda a establecer prioridades en el análisis y diseño de los casos de uso, es decir, se hace una clasificación de aquellos casos de uso que son fundamentales para el funcionamiento de la herramienta (primarios), aquellos que pueden ser no tan importantes (secundarios) y aquellos que pueden no llevarse a cabo (opcionales). La tabla 5.2 muestra la lista de los requerimientos y su clasificación.

Tabla 5.2: *Requisitos de UMLGEC++*

Clasificación de los Requisitos
<p>Primarios:</p> <ol style="list-style-type: none"> 1. Crear componente de clase 2. Crear componente de nota 3. Crear componente de relación (asociación, herencia) 4. Abrir archivo 5. Especificar nombres de rol a un componente de asociación 6. Especificar multiplicidad a una asociación 7. Agregar atributos a una clase 8. Agregar operaciones a una clase 9. Crear componente de conexión hacia una nota 10. Crear un nuevo archivo 11. Guardar archivo como 12. Guardar archivo 13. Generar código 14. Mover componente individual (clase y nota) <p>Secundarios:</p> <ol style="list-style-type: none"> 15. Cortar componente individual 16. Copiar componente individual 17. Pegar componente individual 18. Borrar componente 19. Imprimir el diagrama <p>Opcionales</p> <ol style="list-style-type: none"> 20. Deshacer los últimos cambios 21. Acercar (Zoom in) 22. Alejar (Zoom out)

5.2.4 Detallar casos de uso

En esta actividad se detallan los casos de uso, haciendo una descripción del flujo de eventos por los que pasan, es decir: cómo comienzan, terminan e interactúan con los actores.

El Proceso Unificado propone un formato para la descripción de los casos; comúnmente se habla de dos niveles de casos de uso: Los casos de uso de alto nivel y los casos de uso expandidos [Fernández 01], los primeros describen en pocas palabras el propósito del caso de uso. El formato de los casos de uso de alto nivel es el siguiente:

Casos de uso: <Nombre del caso de uso>
Actores: <Lista de actores>
Descripción: <Breve descripción textual>

Los casos de uso expandidos describen más a detalle el proceso del caso de uso que los de alto nivel, la principal diferencia radica en que el caso de uso expandido contiene una sección llamada “flujo de eventos” donde se detalla paso a paso la comunicación que se da entre los actores y el sistema. Al proceso en donde todo ocurre como se espera fuera el comportamiento más común, se le llama *flujo normal* de los eventos. También dentro de esta descripción se muestran los posibles *flujos alternos* que puedan existir, es decir, desviaciones en el flujo normal.

La Fig. 5.2 muestra el formato del caso de uso expandido con la información del primer caso de uso “crear componente de clase” (el resto de los casos de uso se pueden encontrar en el CD que acompaña la tesis). En esta figura se puede observar cómo se detalló el caso de uso paso a paso, desde que el usuario desea insertar un nuevo componente de clase sobre la ventana de edición del diagrama, hasta que el sistema crea satisfactoriamente el componente seleccionado. También se aprecian los flujos alternos que se llevan a cabo en caso de que el usuario no proporcione un nombre para el componente y otro para el caso de que exista una clase con el mismo nombre proporcionado.

5.2.5 Estructurar el modelo de casos de uso

Una vez detallados todos los casos de uso, se crea el diagrama de casos de uso y se crean los casos de uso reales, que al igual que los casos de uso expandidos, describen el proceso paso a paso, pero incluyendo un contexto más apegado a las tecnologías a utilizar.

El diagrama de casos de uso se muestra en la Fig. 5.3, en el se puede observar la forma en que se definieron las relaciones entre los diferentes casos de uso, basándose obviamente, en las descripciones de los mismos. Por mencionar las relaciones más notables:

- El caso de uso “Generar código” *usa* a “Guardar archivo” después de terminar la generación de código para almacenar el código generado.
- El caso de uso “Guardar archivo como” extiende al caso de uso “Guardar archivo”, ya que “Guardar archivo como” lleva a cabo otras actividades a parte de las realizadas en “Guardar archivo”.

Se puede observar también que se generalizó el caso de uso “Crear Componente”, así como también se han especializado los casos de uso de copiar, pegar, cortar un componente.

<u>Caso de uso 1: Crear componente de clase</u>	
Breve descripción: Colocar un componente de clase en el diagrama. El usuario selecciona el componente de clase y lo coloca en el diagrama donde crea conveniente.	
Flujo básico:	
Acción del actor	Respuesta del sistema
1.- El caso de uso inicia cuando el usuario selecciona un componente de clase	2.- El sistema indica de forma visual el componente que se ha seleccionado
3.- El usuario coloca el componente en la ventana de edición en donde crea conveniente.	4.- El sistema crea el componente visual en la ventana de edición
	5.- El sistema pide un nombre para el componente proponiendo uno generado automáticamente
6.- El usuario introduce el nombre del componente. En caso de que el usuario no introduzca nada se ejecuta el flujo alternativo 1.	7.- En caso de exista otro componente con el nombre proporcionado se ejecuta el flujo alternativo 2.
	8.- El sistema muestra visualmente el nombre en el componente creado.
Flujos alternos	
1.- El sistema asigna el nombre generado. Regresa al caso de uso	
3.- Si el existe otro componente con el mismo nombre el sistema manda un mensaje indicando que existe una clase con ese nombre y se repite el caso de uso desde el paso 5.	
Precondiciones	
1.- El usuario debe tener abierta una ventana de edición	
Poscondiciones	
1.- La ventana de edición contiene al menos un componente de clase	
2.- El diagrama presenta cambios	

Fig. 5.2: *Formato expandido del caso de uso “Crear componente de clase”*

Dentro de esta misma actividad se pueden elaborar los casos de uso reales – los casos de uso reales son aquellos que al igual que los casos de uso expandidos, describen el proceso paso a paso, pero incluyendo un contexto más apegado a las tecnologías a utilizar –. Aunque estos casos de uso se recomienda se pospongan para etapas posteriores, se decidió elaborar algunos en esta etapa, ya que sus descripciones en el formato expandidos facilitaban su comprensión.

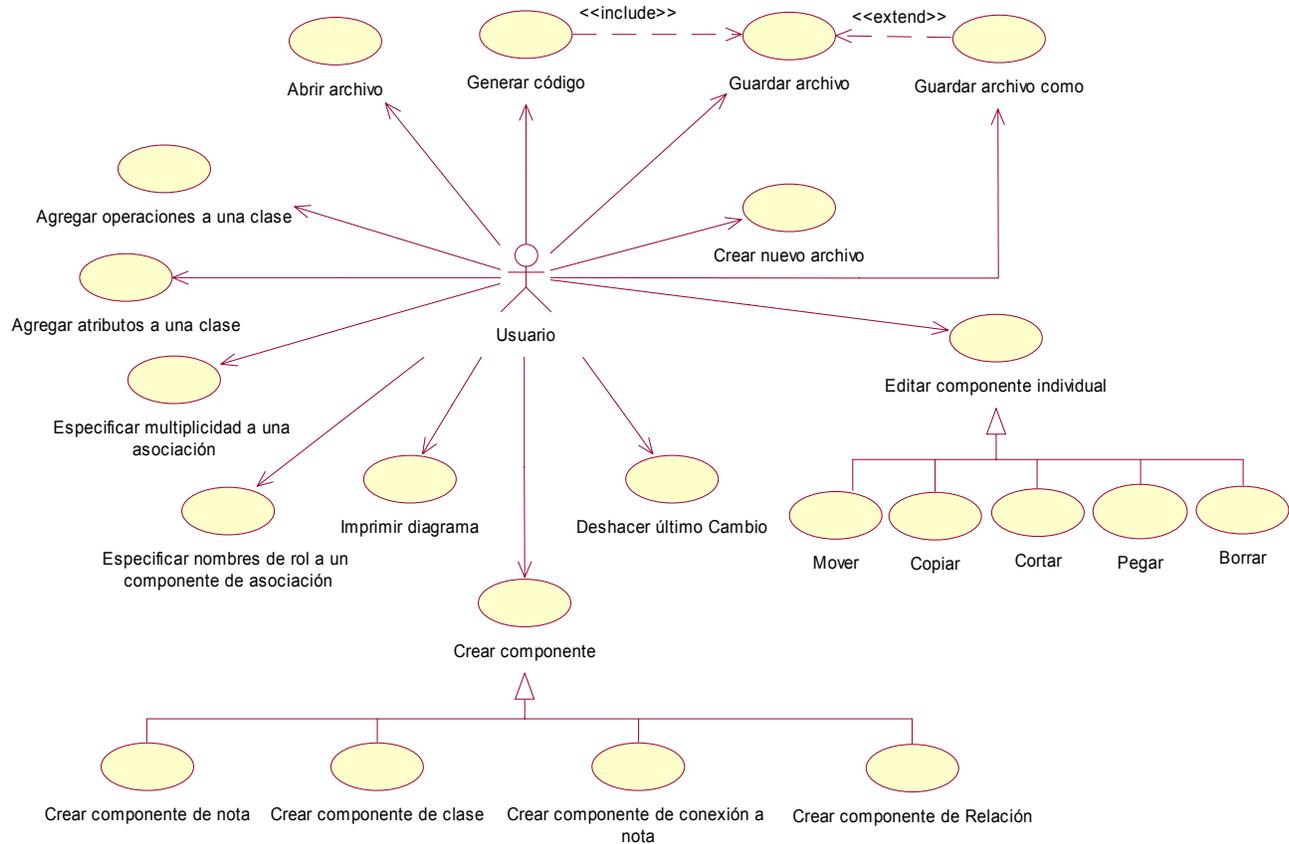


Fig. 5.3: Diagrama de casos de uso de UMLGEC++

La Fig. 5.4 muestra el formato del caso de uso real con la información del primer caso de uso “crear componente de clase” (el resto de los casos de uso reales se pueden encontrar en el CD que acompaña la tesis). En esta figura se puede observar que el formato del caso de uso real es prácticamente el mismo que el caso de uso expandido (ver Fig. 5.2), los que los hace diferentes es el contexto utilizado.

Se observa que prácticamente son los mismos pasos que se siguieron en la Fig. 5.2, pero ya se hace mención de algunos términos técnicos como “click”, “botón”, “barra de componentes”, “campo de edición”, etc., es decir, un lenguaje más apegado al de los desarrolladores.

Caso de uso real: Crear componente de clase

INTRODUCCIÓN

El presente caso de uso tiene como propósito colocar un componente de clase en el diagrama. El usuario selecciona el componente de clase y lo coloca en el diagrama donde crea conveniente.

FLUJO DE EVENTOS

Flujo básico:

Acción del actor

1.- El caso de uso inicia cuando el usuario hace click izquierdo en el botón “botonClase” de la barra de herramientas.

4.- El usuario coloca el puntero del ratón en la ventana de edición en donde crea conveniente y da un click izquierdo sobre ella. En caso de que el usuario coloque el componente fuera de la ventana de edición no se hace nada.

9.- El usuario introduce el nombre del componente en el campo correspondiente al nombre y oprime la tecla ENTER o da click en el botón “Aceptar”.

En caso de que el usuario oprima la tecla ESC o se salga del campo, se asigna al componente el nombre generado.

Respuesta del sistema

2.- El sistema identifica el componente seleccionado.

3.- Se marca el botón “botonClase” y se cambia el puntero del ratón con una imagen que represente tal componente

5.- Si el cursor corresponde a la clase. Se desmarca el botón “botonClase” y se cambia el puntero de ratón al estándar.

6.-Se crea una instancia de objeto “clase” y se visualiza en la ventana de edición.

7.- Se genera un nombre automáticamente de acuerdo al número de componentes clase que tiene la ventana de edición.

8.- Se pide un nombre para el componente creado, proponiendo el generado.

10.- Se agrega el componente a la lista de clases de la ventana de edición, lo que hará que se revise si existe un componente con el mismo nombre.

11.- Si el componente está repetido se ejecuta el flujo alternativo 1 “Nombre de componente existente”.

12.- Fin del caso de uso

Fig. 5.4: *Formato real del caso de uso “Crear componente de clase”*

5.3 Análisis

En la etapa de análisis se analizan los requisitos que se describen en la captura de requisitos, se refinan y se estructuran. Esto con el propósito de obtener una comprensión más precisa de los requisitos y una descripción de los mismos que sea fácil de mantener y que ayude a construir la arquitectura del sistema entero. Se puede utilizar un lenguaje más formal para señalar detalles relativos a los requisitos del sistema, lo que se conoce dentro del Proceso Unificado como “refinar los requisitos”, con esto se facilitan, su comprensión, su preparación, su modificación y, en general, su mantenimiento.

En este flujo se obtienen los siguientes artefactos:

- Modelo de análisis: Contiene la realización de los casos de uso y las clases de análisis.
- Realización de casos de uso: Es una colaboración dentro del modelo de análisis que describe cómo se lleva a cabo y cómo se ejecuta un caso de uso determinado en términos de las clases y de sus objetos de análisis en interacción [Jacobson 00].
- Clases de análisis: Representan una abstracción de una o varias clases y/o subsistemas del diseño del sistema definiendo su comportamiento mediante responsabilidades y atributos.
- Paquete de análisis: Los paquetes de análisis son utilizados para organizar los artefactos del análisis en piezas manejables.

5.3.1 Análisis de la arquitectura

El propósito de esta actividad es expresar el modelo de análisis y la arquitectura mediante paquetes del análisis y requisitos especiales. Hay que definir en primer término, las convenciones de modelado que se adoptarán durante las próximas actividades, en este caso las convenciones definidas fueron:

- Todos los paquetes se deben mostrar en un diagrama principal, mostrando el contenido de cada uno de ellos en diagramas diferentes.
- Los nombres de las clases empezarán con una letra T mayúscula, y cada una de las palabras que formen el nombre, comenzarán con la primera letra también en mayúscula, pe. *TVentanaDeEdicion*.
- Los nombres de los atributos y operaciones empezarán con minúscula, pero el resto de palabras que formen el nombre empezarán con mayúscula, por ejemplo: *abrirArchivo()*

Durante este análisis se lleva a cabo una identificación inicial de paquetes, que son paquetes generales en que podría dividirse el sistema. Los paquetes que aquí se identifiquen, serán modificados y refinados en la fase de diseño. En primera instancia se consideró el modelo de capas típico [Henderson 99]:

5.3.1.1 Modelo de capas típico

- Capa de presentación: La presentación de información de puntos externos al sistema. Este paquete podría contener todas aquellas clases que tengan que ver con la interfaz del usuario.
- Capa de reglas de negocio: Procesos que implementan e imponen el contexto de los datos dentro del uso especificado de negocios. En este paquete estarán todas aquellas clases que controlan las funciones del sistema.
- Capa de almacenamiento: Procesos del sistema de almacenamiento de datos. En este paquete estarán todas las clases que almacenen información relevante para el sistema.

5.3.2 Análisis de casos de uso

En ésta actividad se identifican las clases de análisis y se les asigna un prototipo, también se describe el comportamiento de un caso de uso mediante clases de análisis, para lo cuál se hace uso de los diagramas de colaboración, generando así, las realizaciones de casos de uso.

Se analizan entonces, las descripciones de los casos de uso realizados previamente, con el fin de identificar las clases que intervienen en el flujo de eventos de los casos de uso.

Inicialmente se completó la descripción del flujo de evento, ya que en una primera instancia existían detalles no detectados en el desarrollo de los casos de uso reales. También se detallaron algunas cajas negras que se tenían en un principio. La descripción que se muestra en la Fig. 5.4 es ya una descripción refinada del caso de uso.

5.3.2.1 Identificación inicial de clases

Enseguida se procedió a la identificación inicial de clases; para ello se hizo uso de la estrategia de frases nominales, que consiste en identificar sustantivos o pronombres en las descripciones textuales del dominio del problema, los cuales pueden convertirse en clases o en atributos. Una vez que se obtuvo una lista con las clases candidatas, se les asignó un prototipo: *boundary*, *control* o *entity*.

En el modelo de análisis las clases de análisis comúnmente encajan en uno de los tres estereotipos básicos: *boundary*, *control*, *entity*, que en su traducción al español significan interfaz, control y entidad respectivamente. [Fernández 01]

Las clases de tipo *boundary* son aquellas que se utilizan comúnmente para modelar la interacción entre el sistema y sus actores; las clases de tipo *control* son aquellas que generalmente representan una coordinación, transiciones y control de objetos; y las clases de tipo *entidad* son utilizadas para modelar información que sea duradera en el sistema, como el almacenamiento de información.

Como ejemplo, considérese la Fig. 5.4, donde se muestra el texto del caso de uso “Crear componentes de clase”, del cual se obtuvieron las siguientes frases nominales:

- Usuario
- Botón
- Barra de herramientas
- Clase
- Puntero del ratón
- Ventana de edición
- Cursor
- Número de clases
- Nombre
- Lista de clases

De aquí, se obtuvieron las siguientes conclusiones:

El usuario es un actor y las clases identificadas serían:

“Botón”, “Clase”, “Barra de herramientas”, y “Ventana de edición” son componentes que forman parte de la interfaz del usuario, por tanto se les asigna el prototipo de clases *boundary*. “Puntero del ratón” y “cursor” son lo mismo y se consideran como un atributo de la “Ventana de Edición” y no como una clase, ya que esta palabra representa sólo el valor que define un tipo de puntero. Lo mismo sucede con “Número de clases”, ya que representa un valor puro.

Para el caso de “Lista de clases” representa al conjunto de todas lo componentes de clase que la ventana pueda tener, en esta fase preliminar se considera como un atributo de “Ventana de Edición” y no como una clase.

“Nombre” representa un valor del componente de clase, por tanto es un atributo de éste.

No hay clases que representa únicamente almacenamiento de información, por lo tanto, no se tiene en este caso de uso clases de tipo *entity* y dado que no hay necesidad de coordinar comunicación entre clases de tipo *entity* con *boundary*, no es necesario tener una clase de *control*.

La siguiente figura muestra el diagrama de clases inicial de este caso de uso.

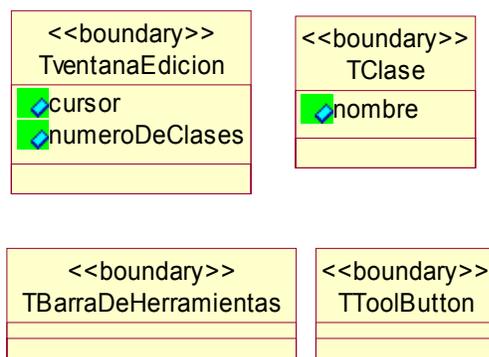


Fig. 5.5: Diagrama inicial de clases del caso de uso “Crear componente de clase”

El UML y Rational Rose – la herramienta en donde fue modelada UMLGEC++ – proporcionan tres iconos para los estereotipos de clases, mismos que serán utilizados en los diagramas de interacción. La siguiente figura muestra estos iconos.

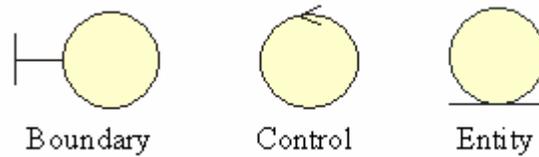


Fig. 5.6: Iconos de UML para representar los estereotipos de clases

5.3.2.2 Descripción de las interacciones entre objetos del análisis

Una vez terminados los diagramas de las clases necesarias para realizar un caso de uso, se describieron como interactúan sus correspondientes objetos de análisis. Para ello se utilizaron los diagramas de interacción.

Dado que básicamente los diagramas de secuencia y colaboración muestran información similar, se optó por crear inicialmente los diagramas de secuencia, y a partir de ellos, se crearon los diagramas de colaboración. Las consideraciones que se siguieron para crear el diagrama de secuencia fueron:

1. Se toman como base los casos de uso reales.
2. Se revisa paso a paso cada uno de los eventos, analizando cada uno de los verbos que aparecen, con el fin de detectar un posible envío de mensaje, que puede significar que un objeto le pide a otro objeto que haga alguna tarea.
3. El sistema puede ejecutar operaciones usando únicamente objetos *boundary* y *entity*, pero casos de uso más complejos o simplemente si se desea hacer el sistema más tolerante a cambios, se necesitan de una o más clases de *control*. Se crea entonces, al menos una clase de tipo *control* por caso de uso para coordinar el envío de mensajes entre los objetos y separar los *boundary* de los *entity*.

Se creó un diagrama de secuencia por cada flujo de eventos que contenga un caso de uso y un solo diagrama de colaboración por caso de uso, lo que permitió hacer más clara la realización del caso de uso. En este tipo de diagramas se observaron los mensajes que se envían entre los objetos, aunque en esta etapa, los mensajes no se asocian precisamente a operaciones finales, debido a que hasta este momento no se habían especificado formalmente en las clases. Las figuras 5.7 y 5.8 muestran los diagramas de interacción del flujo normal de eventos, nuevamente del primer caso de uso.

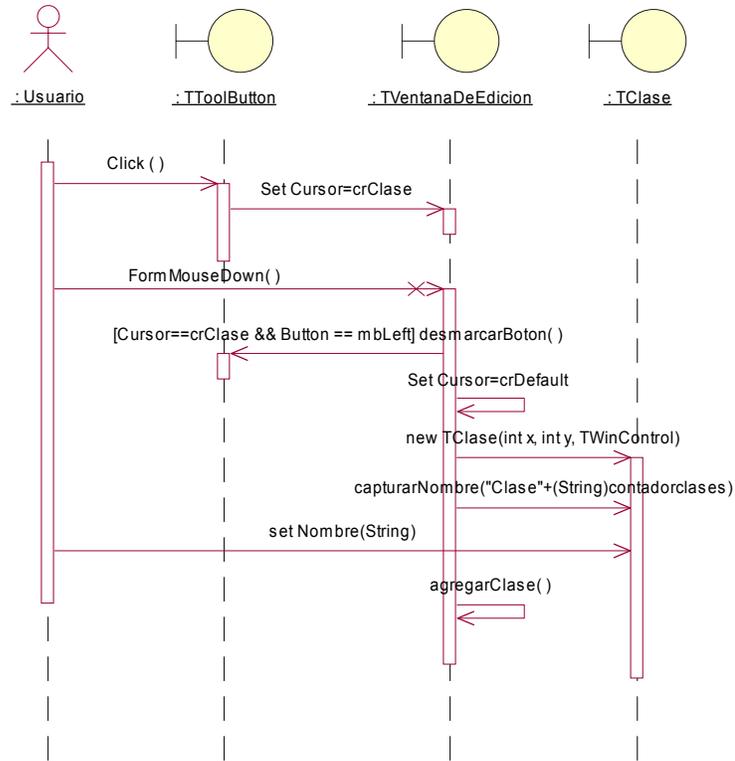


Fig. 5.7: Diagrama de secuencia del flujo normal del caso de uso “Crear componente de clase”

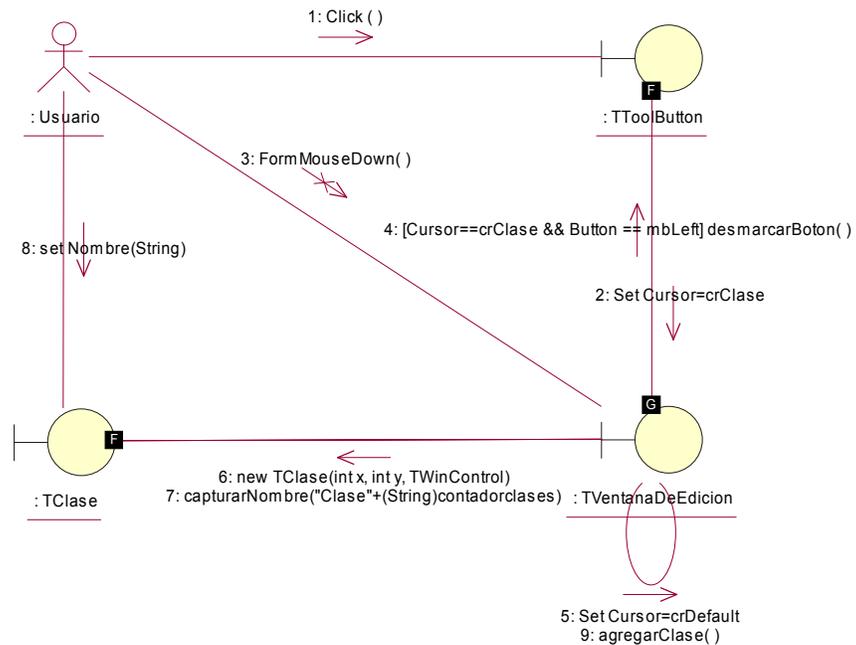


Fig. 5.8: Diagrama de colaboración del flujo normal del caso de uso “Crear componente de clase”

En los diagramas anteriores se observa que ambos muestran de forma visual básicamente la misma información contenida en la descripción textual de los casos de uso reales.

5.3.3 Análisis de clases

Al analizar una clase se debe de identificar sus responsabilidades, atributos, generalizaciones, asociaciones y agregaciones con multiplicidad. Hasta este momento ya se tenía la identificación inicial de clases y los mensajes con los que los objetos se relacionan en cada caso de uso, entonces se fortalecieron los diagramas de clases que se tenían, agregando las responsabilidades, atributos, generalizaciones y asociaciones correspondientes.

5.3.3.1 Identificación de responsabilidades

Las responsabilidades hasta este nivel son derivados de los mensajes en los diagramas de interacción. Una responsabilidad es una orden que se le puede solicitar a un objeto para que haga algo, que puede ser, ejecutar alguna acción o proporcionar el conocimiento que contiene [Jacobson 00]. Entonces por cada objeto en el diagrama de interacción se analizó cada uno de los mensajes que recibe, los cuáles algunos llegaron a ser parte de una responsabilidad de dicho objeto.

Las responsabilidades de las clases se documentan dentro de la clase como parte de su descripción o bien, se les da un nombre más claro a las operaciones de las clases para describir las responsabilidades. En este caso se optó por darles un nombre lo más claro posible a las funciones, con el fin de que cada nombre represente la responsabilidad que la clase tiene.

5.3.3.2 Identificación de atributos

Un atributo especifica una propiedad de una clase y no tiene mayor responsabilidad que contener sus propios datos. Durante el análisis los tipos de atributos pueden indicarse, aunque no es necesario que sean del algún tipo de lenguaje de programación.

Durante el desarrollo del sistema y hasta este momento, se habían hallado algunos atributos que eran obvios, el resto se obtuvo de analizar las responsabilidades de las clases que se necesitaron para satisfacer los requerimientos de información de los casos de uso.

La Fig. 5.9 muestra la representación gráfica de dos clases, en las cuáles se pueden observar sus atributos y responsabilidades identificadas hasta esta etapa. Los atributos fueron obtenidos a partir de identificar qué información debía conocer cada objeto, así como sus características.

En este caso, la clase “TClase” debe almacenar información acerca de su nombre, atributos y operaciones, la clase “TVentanaDeEdicion” debe saber cuantos componentes pudieran existir en un diagrama, y una variable que le indica si el diagrama ha sufrido cambios desde la última vez que se guardó.

En cuanto a las responsabilidades, se observa que básicamente son funciones basadas en los mensajes enviados entre objetos en los diagrama de interacción.

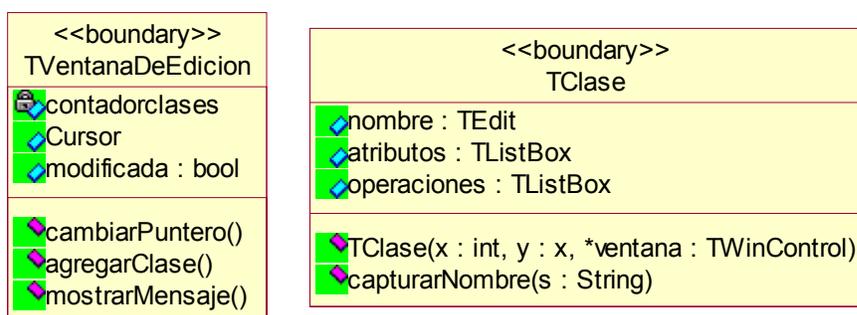


Fig. 5.9: Clases con atributos y responsabilidades

5.3.3.3 Identificación de asociaciones y agregaciones

Antes de la identificación de asociaciones, se tomó en cuenta la visibilidad entre objetos.

5.3.3.3.1 Visibilidad

La visibilidad es la capacidad de un objeto de ver a otro o hacer referencia él. Los diagramas de colaboración describen gráficamente los mensajes entre objetos. Para que un objeto emisor envíe un mensaje a un objeto receptor, éste tiene que ser visible al emisor. [Fernández 01]

Existen 4 tipos comunes de visibilidad:

- Visibilidad de atributo
Existe visibilidad de atributo de A a B cuando B es atributo de A.
- Visibilidad de parámetro
Existe visibilidad de parámetro de A a B cuando B se transmite como parámetro a un método de A.
- Visibilidad local
Existe visibilidad local de A a B cuando se declara que B es un objeto local dentro de un método de A.
- Visibilidad global
Existe visibilidad global de A a B cuando B es global para A.

En UML existe una notación opcional que representa la visibilidad entre objetos dentro de los diagramas de colaboración. Ésta es, colocando en cada extremo de los mensajes el prototipo que indica la visibilidad. La Fig. 5.10 muestra cómo se representa gráficamente en Rational Rose.

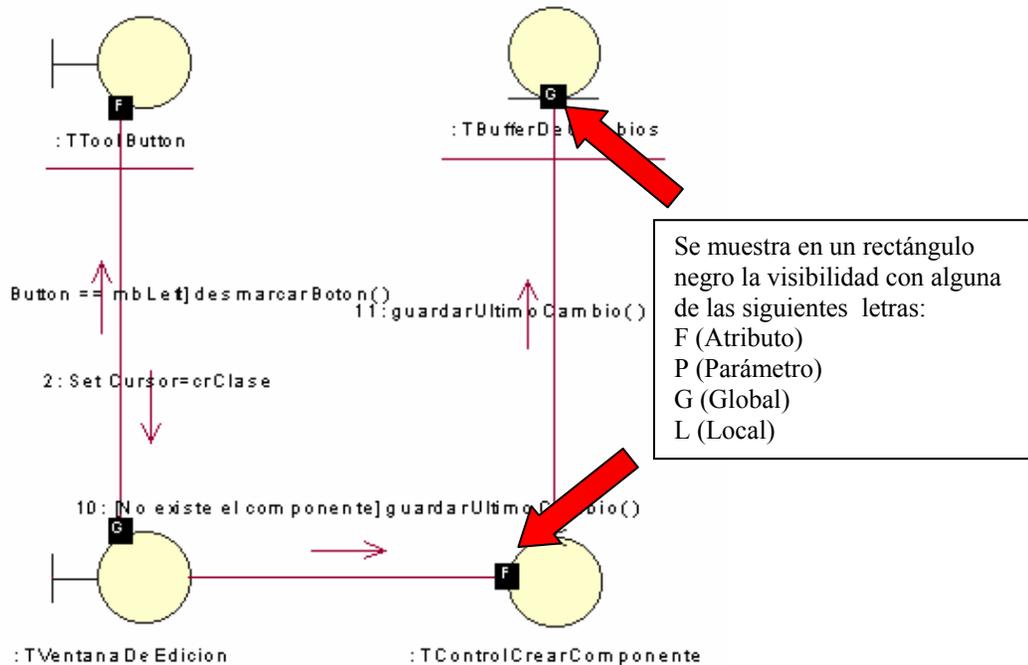


Fig. 5.10: Representación gráfica de la visibilidad entre objetos en Rational Rose

5.3.3.2 Asociaciones y agregaciones

Se estudiaron los enlaces empleados en los diagramas de colaboración para determinar qué asociaciones son necesarias, tomando en cuenta que cuando un objeto tiene visibilidad de atributo con otro, ésta se convierte automáticamente en una asociación. También se definió la multiplicidad de las asociaciones, los nombre de los roles, y en su momento, asociaciones reflexivas y clases de asociación.

Para identificar una asociación se tomó en cuenta lo siguiente [Fowler 99]:

- Un objeto A es parte física o lógica de un objeto B
- Un objeto A está física o lógicamente contenido en un objeto B
- Un objeto A está registrado en B

Las agregaciones es un caso particular de asociación, y debe utilizarse cuando los objetos representan [Fernández 01]:

- Conceptos que contienen físicamente a otros.
- Conceptos que están compuestos uno de otro
- Conceptos que forman una colección conceptual de objetos.

La agregación maneja otra variedad conocida como composición, en ésta, el componente puede pertenecer a un todo, y se espera que las partes vivan y mueran con el todo.

La figura 5.11 muestra el diagrama de clases final para el caso de uso “Crear componente de clase”, en ella se pueden apreciar las asociaciones, agregaciones y composiciones identificadas.

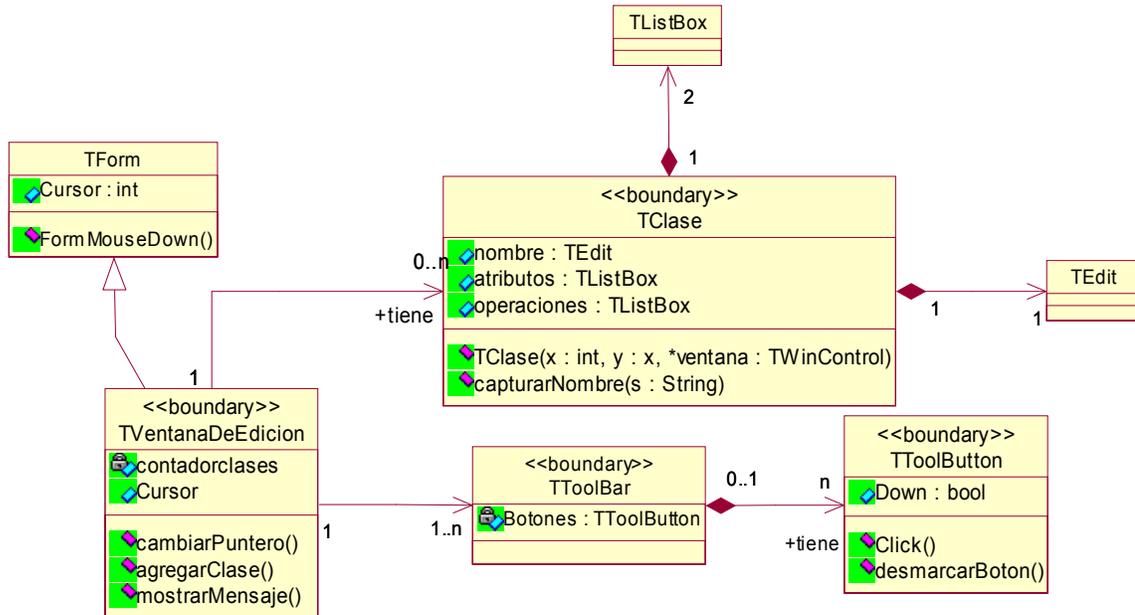


Fig. 5.11: Diagrama de clases del caso de uso “Crear componente de clase”

5.3.3.4 Identificación de generalizaciones

Las generalizaciones deben utilizarse durante el análisis para extraer comportamiento compartido y común entre varias clases de análisis diferentes. En esta etapa, sólo se indicaron las generalizaciones obvias, el resto se identificaron en la etapa de diseño.

5.4 Diseño

En la etapa de diseño se modela el sistema y se le da forma para que soporte todos los requisitos. El modelo de análisis proporciona una comprensión detallada de estos requisitos, y una estructura que se debe conservar y reforzar.

Una vez terminado el flujo de análisis, se adaptaron los resultados arrojados a las restricciones impuestas por los requerimientos funcionales y no funcionales y el ambiente de implementación – Que para el proyecto fue C++ Builder –, en otras palabras, se hace un refinamiento del análisis.

En este flujo se obtienen los siguientes artefactos:

- Modelo de diseño: Describe la realización física de los casos de uso, en este modelo los casos de uso son realizados por las clases de diseño y sus objetos, representados mediante realizaciones de casos de uso de diseño.

- Realización de casos de uso: Es una colaboración dentro del modelo de análisis que describe cómo se realiza un caso de uso específico, y cómo se ejecuta en términos de clases de diseño y sus objetos.
- Clases de diseño: Son una abstracción similar de una clase o construcción en la implementación del sistema, es decir, el lenguaje utilizado debe ser el mismo que el lenguaje de programación. Contienen también la visibilidad de los atributos, ya sea pública, privada o protegida.
- Paquete de diseño: Los paquetes de diseño son utilizados para organizar los artefactos del diseño en piezas manejables.

5.4.1 Diseño de la arquitectura

El propósito de esta actividad es expresar los modelos de diseño y arquitectura mediante la identificación de subsistemas y sus interfaces, así como las clases de diseño. Por tanto la primera tarea del diseño fue identificar los subsistemas

5.4.1.1 Identificación de subsistemas

En la actividad de análisis de la arquitectura se tomó en cuenta una posible clasificación de los paquetes haciendo uso del modelo de capas típico. Entonces basándose en esa primera clasificación se tiene que:

Tomando en cuenta que todas aquellas clases con el prototipo *boundary* por sus características pertenecen a la capa de presentación; las de prototipo *control* pertenecen a la capa de reglas de negocio; y las de prototipo *entity* pertenecen a la capa de almacenamiento.

Se puede decir entonces que se tenían hasta el momento 3 posibles subsistemas gracias a que se hicieron subsistemas horizontales a partir del diagrama de clases. Esto es, que las clases *boundary* y *entity* relacionadas se colocaron en subsistemas separados.

Se optó por esta forma y no por la opción de subsistemas verticales, que nos dice que las clases *boundary* y *entity* relacionadas se ponen en el mismo subsistema [Fernández 01], ya que al analizar el diagrama de clases global de UMLGEC++ (Ver Apéndice B), se ve que existen clases *boundary* que no tienen relación con entidades *entity*, sin embargo colaboran con varias clases de su mismo tipo. Por tanto, si se crearan subsistemas verticales, todas estas clases tendrían que delegar varias de sus responsabilidades hacia las interfaces de los subsistemas en donde se encuentren todas aquellas clases con las que tiene relaciones, con esto lo que se obtiene es bastante dependencia entre subsistemas, que en etapas posteriores es algo que se quiere evitar.

Como ya se ha mencionado, se necesita reducir lo más posible la dependencia entre subsistemas. Revisando nuevamente el diagrama de clases global, se observa que las clases correspondientes a los componentes gráficos como “Clase”, “Nota”, “Asociación”, etc. están fuertemente acopladas debido a varias asociaciones que existen entre ellas. Entonces se decidió crear un nuevo subsistema llamado “Componente”, con esto, dado que todas las clases que forman parte de un componente son de tipo *boundary*, se restó responsabilidad al subsistema “ClasesBoundary”.

Además dentro del subsistema “Clases Control” se incluyó la clase de tipo *entity* “TPortapapeles”, ya que esta clase sólo es accesada por la clase de *control* “TControlEdicion”, y con esto se obtiene como beneficio eliminar una responsabilidad a la interfaz del subsistema “ClasesEntity”. Se tienen entonces, cuatro subsistemas que se muestran en la figura 5.12

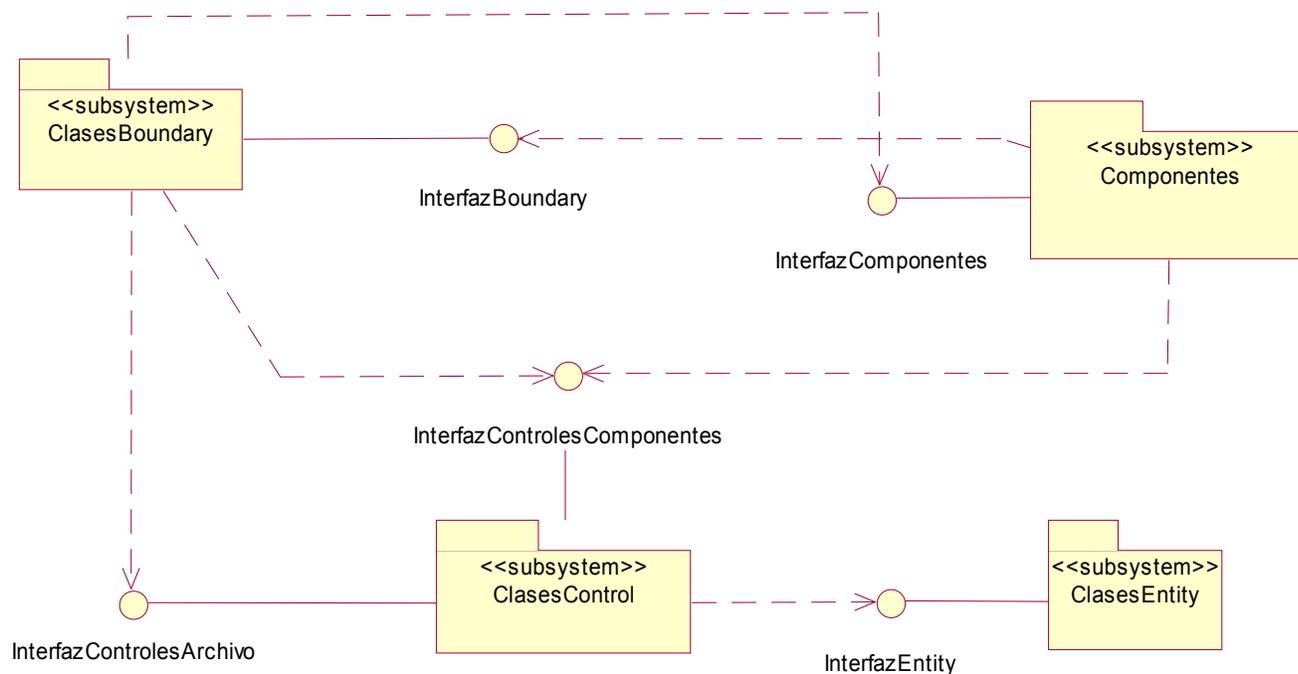


Fig. 5.12: *Subsistemas de UMLGEC++*

Las clases contenidas en cada subsistema se muestran en el Apéndice C.

5.4.1.2 Identificación de interfaces

Una interfaz es un elemento de modelado que define una colección de operaciones que son utilizadas para especificar un servicio de una clase o paquete [Fowler 98], para el caso del proyecto, las interfaces proporcionan las operaciones de las clases internas de un subsistema, que necesitan ser accesadas por otras clases en otros subsistemas.

Inicialmente, se creó una interfaz por cada paquete, la cual debía contener todas las responsabilidades públicas de cada clase dentro del paquete que tenían que ser utilizadas externamente, se buscaron responsabilidades similares entre las clases, como por ejemplo la responsabilidad “borrar”, que es una responsabilidad de la mayoría de las clases dentro del subsistema “Componentes”, entonces la operación borrar() de la interfaz del subsistema representa las responsabilidades de cada componente de ser borrado.

La Fig. 5.12 muestra el diagrama de paquetes de la herramienta, en donde se puede apreciar los subsistemas, así como sus correspondientes interfaces. Se puede apreciar que cada subsistema

tiene una interfaz, excepto por el subsistema “Clases Control”, que contiene dos. La razón es porque se quiso diferenciar las responsabilidades que corresponden a las clases de control sobre el manejo de archivos con las que corresponden al manejo de componentes.

5.4.2 Diseño de casos de uso

El propósito de esta actividad es identificar las clases de diseño y las instancias de los subsistemas. Así como la describir el comportamiento de un caso de uso mediante paquetes, interfaces y clases de diseño, logrando con esto, la refinación de las realizaciones de casos de uso.

5.4.2.1 Identificación de clases de diseño participantes

En esta parte se identificaron las clases de diseño que se necesitaban para refinar las realizaciones de cada caso de uso, para ello se estudiaron las clases de análisis que participan en cada realización de casos de uso y se refinaron de tal forma que contuvieran una descripción más apegada a la forma de implementación.

También se identificaron algunas clases que eran necesarias para cumplir los objetivos de los caso de uso y que no fueron detectadas en la fase de análisis.

5.4.2.2 Descripción de las interacciones entre objetos del diseño

Una vez que se tienen las clases de diseño necesarias para refinar una realización de caso de uso, se describen cómo interactúan sus correspondientes objetos de diseño. Para ello se hizo uso nuevamente de los diagramas de interacción.

Con esto lo que se obtuvo es básicamente los diagramas de interacción que se tenían en la fase de análisis, pero ahora refinados con objetos de diseño que son las instancias de las interfaces de los subsistemas de diseño y de los actores. Las consideraciones que se siguieron para crear el diagrama de secuencia fueron:

1. Se toman como base las realizaciones de casos de uso, las clases de diseño, los subsistemas y las interfaces.
2. Se revisa paso a paso cada uno de los eventos, decidiendo que objetos de diseño son necesarios para realizar cada paso.

Al igual que en el análisis, se creó un diagrama de secuencia por cada flujo de eventos que contenga un caso de uso y un solo diagrama de colaboración por caso de uso. En este tipo de diagramas se observaron los mensajes que se envían entre los objetos de diseño. Las figuras 5.13 y 5.14 muestran los diagramas de interacción del flujo normal de eventos del caso de uso “Crear componente de clase”

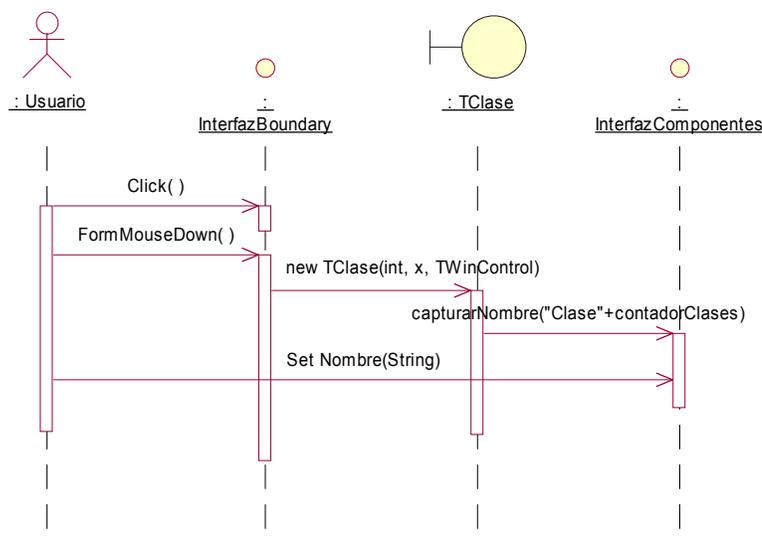


Fig. 5.13: Diagrama de secuencia de diseño del flujo normal del caso de uso “Crear componente de clase”

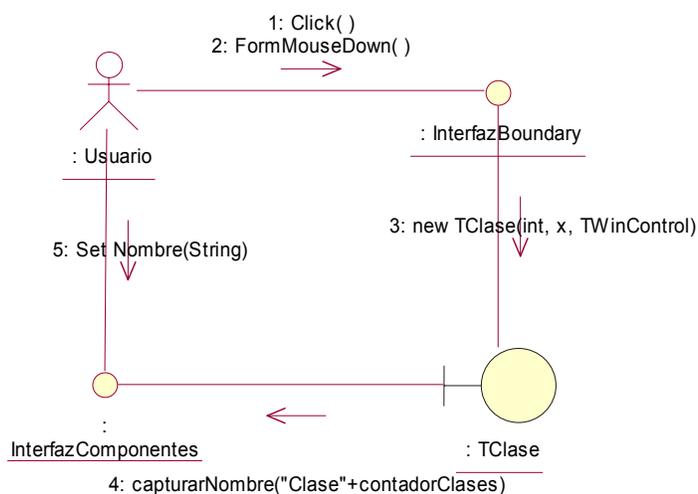


Fig. 5.14: Diagrama de colaboración de diseño del flujo normal del caso de uso “Crear componente de clase”

Se puede observar en ambas figuras, que se sigue básicamente el mismo flujo de eventos que en las figuras 5.7 y 5.8 respectivamente, pero ahora la secuencia de mensajes es por medio de objetos de diseño. Se nota también que muchas de las llamadas a las operaciones los objetos no se muestran, ya que están encapsuladas dentro de las responsabilidades de cada interfaz porque el encargada de comunicarse con su subsistema.

5.4.3 Diseño de una clase

El objetivo de diseñar una clase es crear una clase de diseño que cumpla el papel en las realizaciones de casos de uso y los requisitos no funcionales que se aplican a estos, esto es, que se debe tener una clase fortalecida, lista para poder ser codificada dentro de un lenguaje de programación.

Además, se pueden agregar clases adicionales que sean necesarias para llevar a cabo cierta tarea, agregando sus relaciones correspondientes.

5.4.3.1 Identificación de operaciones

Las operaciones de las clases en esta etapa se describen utilizando la sintaxis del lenguaje de programación en donde se pretende implementar el sistema. Esto incluye:

- Especificar los parámetros, con tipo y valores por omisión.
- Especificar el tipo de valor de retorno.
- Especificar la visibilidad de cada operación: *public*, *protected*, *private*.

5.4.3.2 Identificación de atributos

También en esta etapa, se refinan los atributos de la clase de análisis y se identifican algunos otros requeridos por la clase diseño, y se describen utilizando la sintaxis del lenguaje de programación. Esto incluye:

- Especificar el tipo de dato del atributo.
- Especificar la visibilidad de cada atributo.

5.4.3.3 Identificación de asociaciones y agregaciones

Al igual que los objetos de análisis, los objetos de diseño interactúan unos con otros, esto es, que existe una relación entre ellos que requiere ser modelada. Entonces, se tomaron como base las relaciones identificadas en la fase de análisis y se refinaron de la siguiente manera:

- Se considera las asociaciones y agregaciones del modelo de análisis que pueden evolucionar al modelo de diseño, y se identifican relaciones adicionales que sean necesarias.
- Se refinan la multiplicidad de las asociaciones, nombre de asociación, nombre de rol; donde estos últimos evolucionan como atributos de la clase al momento de su codificación.

5.4.3.4 Identificación de generalizaciones

Las generalizaciones deben ser adaptadas al lenguaje de programación a utilizar. Para este caso no hubo problema, dado que C++ permite la herencia de clases. Entonces dentro de esta etapa el

paso más importante fue identificar oportunidades de reuso en el lenguaje de programación a utilizar. Por ejemplo: En la Fig. 5.15 se observa que la clase TVentanaDeEdicion hereda las características de la clase TForm, donde esta última es una clase del IDE de C++ Builder, que se utiliza para crear ventanas estándares y poder manipularlas, por tanto, aprovechamos esta ventaja del lenguaje de programación.

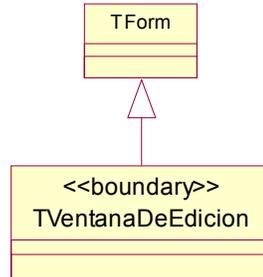


Fig. 5.15: Ejemplo de reuso con la clase TForm de C++ Builder 5.0

5.4.4 Diseño de un subsistema

El propósito de esta actividad es garantizar que los subsistemas identificados sean lo más independiente posible uno del otro, así como garantizar que cumplen con sus propósitos. Por tanto, se dio mantenimiento a las dependencias entre subsistemas, verificando que todos los elementos dentro de un sistema estuvieran correctamente asociados a las interfaces de los subsistemas de los cuales depende, así como reubicar algunas clases de diseño con el propósito de reducir lo más posible la dependencia entre paquetes.

También se revisó mediante las refinaciones de realizaciones de casos de uso, que efectivamente cada subsistema cumple con sus objetivos y que proporcionan las interfaces necesarias.

5.5 Implementación

La implementación es el flujo de trabajo donde se convierte el resultado del diseño en código fuente y en un sistema ejecutable.

Terminado el flujo de diseño, se tiene todo listo para la creación de un sistema ejecutable. Basándose en la arquitectura del diseño se lograría la implementación de ésta en archivos de código fuente en C++.

En este flujo se obtienen los siguientes artefactos:

- Modelo de implementación: Es la implementación de los elementos del diseño en componentes como archivos de código fuente, ejecutables, etc.; y una descripción de la organización de tales componentes de acuerdo al entorno de implementación y el o los lenguajes de programación utilizados.

- **Componente:** Un componente es la parte física de los elementos del diseño, que son las clases, paquetes, interfaces, etc., un componente puede ser un archivo de código fuente, un archivo ejecutable, una librería, una tabla de base de datos, u otro documento.
- **Subsistema de implementación:** Son la evolución de los paquetes de del diseño, organizan los componentes de implementación en piezas más manejables. Se representa por medio de un mecanismo de empaquetamiento, que depende del ambiente de programación, por ejemplo. Un *paquete* en Java, un *directorio de archivos* en C++, o bien un *proyecto* en el caso de C++ Builder.

5.5.1 Implementación de la arquitectura

La primera parte de la implementación fue identificar los componentes más significativos de la herramienta, para comenzar con su implementación, ya que la implementación de algunas clases podría ser menos complicada que otras, como el caso de la implementación de la clase “Estructura del Diagrama” que incluye la generación de código (la parte fundamental de la herramienta). Para esto se basó en gran parte en la clasificación de los casos de uso que se hizo en el punto 5.2.3.

Entonces aquellos componente de diseño que aparecieron dentro de los casos de uso primarios, serían los primeros en ser implementados, siguiendo con los secundarios y finalmente con los opcionales.

5.5.2 Implementación de un subsistema

El objetivo de esta actividad es asegurar que un subsistema cumple con su papel en cada construcción: esto es, asegurar que los requisitos implementados en la construcción y aquellos que afecten al sistema son implementados correctamente por componentes o por otros subsistemas dentro del subsistema.

En la implementación de los subsistemas, se definió la forma en la que estarían implementados en archivos los subsistemas de diseño. Revisando la figura 5.12, se tienen 4 subsistemas, los cuáles se implementaron de la siguiente forma:

Todo el sistema se creó bajo un proyecto en C++ Builder 5.0, al que se llamó “Case.bpr”.

La Fig. 5.16 muestra la forma en que fueron transformadas las clases de diseño de los subsistemas en componentes de implementación.

Se observa en esta figura, que no para cada clase de diseño fue necesario un componente de implementación, esto no es necesariamente una regla, sino simplemente se hizo para tener en un solo archivo aquellas clases que están más relacionadas, además de que la modularización del lenguaje C++ Builder lo permite; y por el contrario no se colocaron todas las clases de un subsistema en un componente de implementación para no tener un archivo demasiado grande y que no dificulte su claridad al momento del mantenimiento.

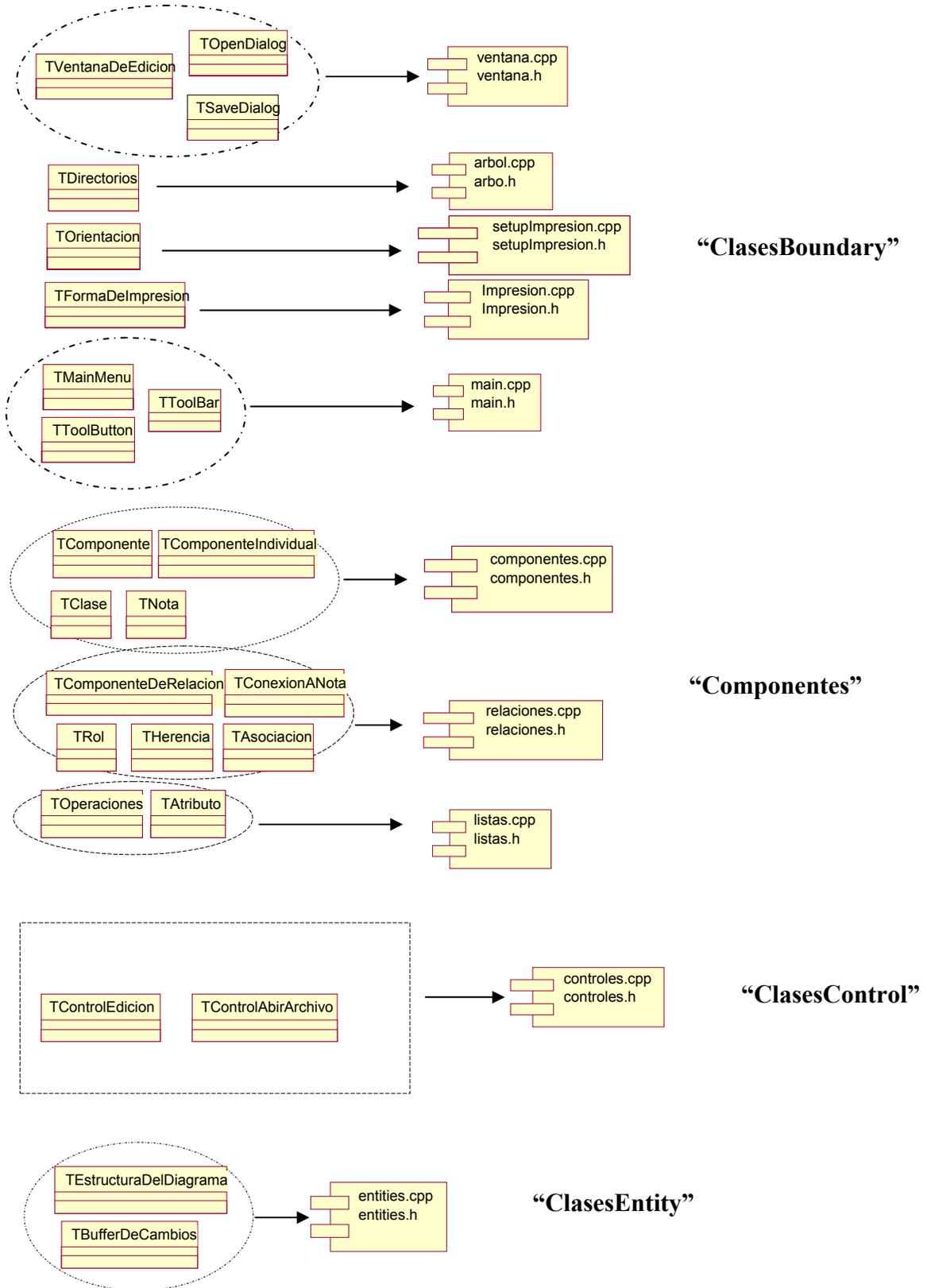


Fig. 5.16: Evolución de las clases de diseño en componente de implementación

5.5.3 Implementación de una clase

Una vez definida la prioridad y la forma de implementación, se procedió a la creación del código fuente de los componentes de implementación.

En las siguientes figuras se puede apreciar la forma en que se transformaron las clases de diseño en componentes de implementación.

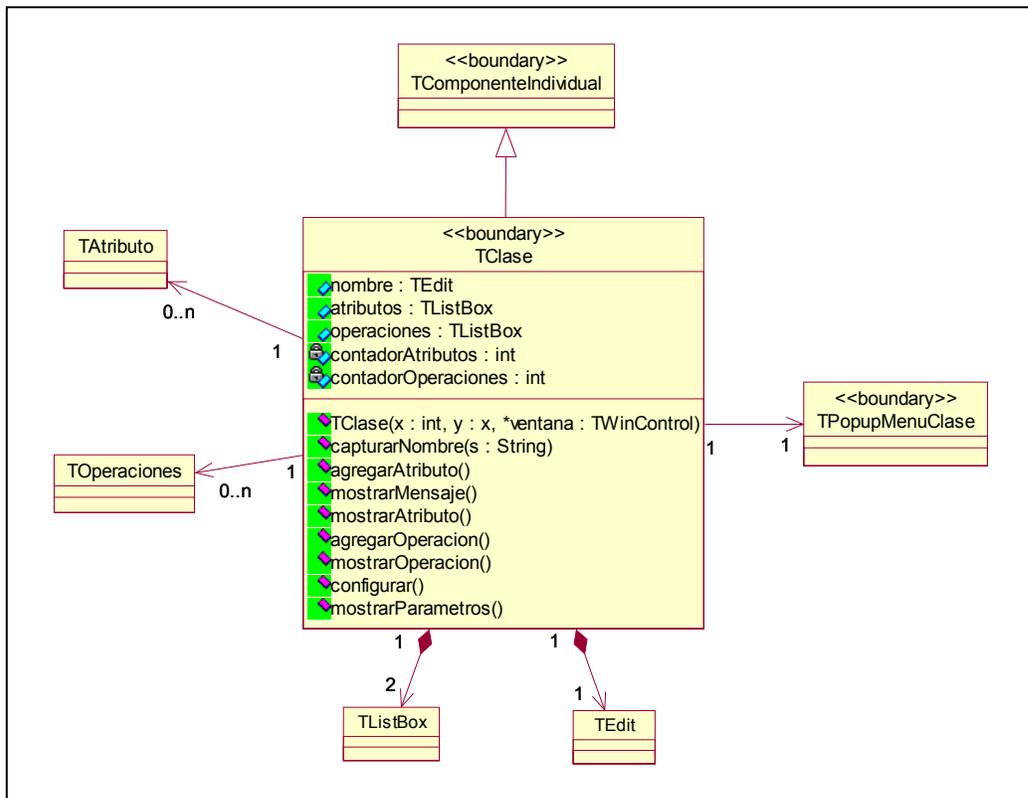
5.5.3.1 Generación de código a partir de una clase de diseño

Con la generación del código fuente se logra obtener una estructura estática de la herramienta, que incluye la signatura de los atributos y las operaciones.

Una parte muy importante en esta actividad es definir la forma en que se implementarían las asociaciones y agregaciones. Y lo que se estableció para la codificación de la herramienta fue lo siguiente:

- Para una asociación entre objetos A y B, donde A hace referencia a B, será representada como un atributo en el objeto A y el nombre del atributo será el nombre del rol del extremo de B (que en caso de no tener, sería “objetoB” por omisión).
- La multiplicidad del extremo del objeto B, y dependiendo si es una agregación o composición, definirá si el atributo debe ser implementado como un apuntador (por referencia) o como una copia del objeto (por valor). Entonces se tienen los siguientes casos:
 - Para una *asociación simple* con multiplicidad igual a 1, el atributo es implementado como un apuntador al objeto B.
 - Para una *asociación simple* con multiplicidad mayor a 1, el atributo es implementado como una lista de apuntadores de objetos B. (se creó una clase TLista, ya que además de contener la colección de apuntadores, se necesitó de la implementación de funciones específicas para trabajar con estos apuntadores)
 - Para una *agregación* con multiplicidad igual a 1, el atributo es implementado como un apuntador al objeto B.
 - Para una *agregación* con multiplicidad mayor a 1, el atributo es implementado como una lista de apuntadores de objetos B.
 - Para una *composición* con multiplicidad igual a 1, el atributo es implementado como un atributo del tipo del objeto B (por valor).
 - Para una *composición* con multiplicidad mayor a 1, no se dio el caso.

Como ejemplo véase la Fig. 5.17, que muestra el código generado a partir de la clase “TClase”, en la que se puede apreciar la implementación de la herencia, asociaciones y composiciones.



Implementación

```

class TClase:public TComponenteIndividual
{
public:
    TEdit *nombre;
    TListBox *atributos,*operaciones;
    TListaAtributos *listaAtributos;
    TListaOperaciones *listaOperaciones;
    TPopupMenu *submenuClase;
    TClase(int x, int y, TWinControl *ventana);
    ~TClase();
    void capturarNombre(String s);
    bool mostrarAtributo(String n,String t,String val,int v);
    bool mostrarOperacion(String n,String t,int v,TStringGrid *par);
private:
    bool arrastre,presionado;
    int contadorAtributos;
    int contadorOperaciones;
    //Funciones
    void __fastcall configurar(TObject *Sender);
    void __fastcall agregarAtributo(TObject *Sender);
    void __fastcall agregarOperacion(TObject *Sender);
    void __fastcall mostrarParametros(TObject *Sender);
};
  
```

Código fuente
C++

Fig. 5.17: Implementación de clase "TClase" en C++ Builder

5.5.3.2 Implementación de operaciones

El trabajo restante, para que la herramienta trabaje, es la implementación del cuerpo de las funciones. La implementación de una operación (método), puede estar basada en un algoritmo o un pseudocódigo en aquellos casos en que el método sea muy grande, o requiera de un análisis cuidadoso para su correcto funcionamiento. Tal es el caso de las funciones “guardar()” y “generarCódigo()” y la revisión de sintaxis, las cuáles se analizan a continuación.

5.5.3.3 Métodos fundamentales de la herramienta

5.5.3.3.1 El método “guardar()”

Una parte importante de la herramienta es que una vez que se haya creado un diagrama, éste podrá ser abierto en alguna otra ocasión sin perder información, es decir, que el diagrama debe aparecer en pantalla exactamente igual a como el usuario lo había diseñado, así como contener la misma información previamente agregada al diagrama.

Para que esto fuera posible, fue necesario analizar todos los datos de los objetos que representan los componentes del diagrama de clase, y decidir cuáles de ellos eran necesarios y cuáles no, para que permitiera guardar toda la información fundamental y eliminar aquella que puede ser obtenida a partir de otra.

Entonces, se creó un código que pudiera almacenar y leer estructuras con información de cada componente involucrado en un diagrama de clases, como son: las clases, las relaciones y notas. Esta información incluye datos sobre la posición y tamaño visual, información propia de cada componente como el nombre, atributos, etc., así como relaciones entre componentes.

En la Fig. 5.18 se puede observar parte del código propuesto para el almacenamiento en archivo de un diagrama (El código completo de la estructura se encuentra en el Apéndice D). En esta figura se observan algunos de los datos que se consideraron importantes y que tenían que ser almacenados para su recuperación posterior, por ejemplo, cómo ya se había mencionado para las clases se almacena su nombre, la coordenada *x*, la coordenada *y*, los atributos, etc.

Este formato está basado un pequeño análisis del formato .mdl, que es el formato los archivos de la herramienta Rational Rose, no se hizo un análisis a detalle ya que esta información no está disponible al público por ser parte de un producto comercial.

En la Fig. 5.19 se observa ahora, parte del código final, en donde ya se hace uso de *símbolos* (el código final y el valor de los símbolos se encuentran en el Apéndice E), para identificar cada tipo de dato, ya que ésta es la forma en que es almacenada la información en un archivo, al que se le asignó la extensión “.uml”.

```

DIAGRAMA "<Nombre del diagrama>"
<Número de componentes>
Objeto Clase <ID Clase>
<Nombre_clase1> //Nombre de la clase
{
    X   <Posición x>
    Y   <Posición y>
    Ancho <Ancho en pixeles>
    Ver_Atributos   <True o False>
    Ver_Operaciones <True o False>
    Alto_Etiqueta <Alto del módulo de las etiquetas>
    Alto_Atributos <Posición del módulo de los atributos>
    Alto_Operaciones <Posición del módulo de las
operaciones>
    Atributos //Lista de atributos
    {
        {
            Nombre   <Nombre atributo1>
            Visibilidad <Tipo de visibilidad>
            Tipo <Tipo de dato>
            Valor inicial <valor>
        }
        {
            Nombre   <Nombre atributo2>
            Visibilidad <Tipo de visibilidad>
            Tipo <Tipo de dato>
            Valor inicial <valor>
        }
        .
        .
        Más atributos
    }
}

```

Fig. 5.18: Propuesta de código para almacenamiento en archivo

```

TI <Título del diagrama>
X[NUMERO DE COMPONENTES]
{
    OC <ID>
    <Nombre Clase1>
    X[102]
    Y[232]
    W[80]
    VA T
    VO T
    H[10] //Alto Nombre
    H[30] //Alto Atributos
    H[40] //Alto Operaciones
    AT
    NA <Nombre atributo1>
    V 1
    T <Tipo de dato>
    VI <Valor inicial>

    NA <Nombre atributo2>
    V 1
    T <Tipo de dato>
    VI <Valor inicial>
    .
    .
    .
    OP
    NO<Nombre operacion1>
}

```

Fig. 5.19: Código final para almacenamiento en archivo

Así con estos símbolos, se sabe el orden en el que deben venir los datos del archivo, que es cómo sigue:

1. Datos del diagrama
2. Implementación y datos gráficos de las clases
3. Implementación y datos gráficos de las asociaciones
4. Implementación y datos gráficos de la generalización (herencia)
5. Implementación y datos gráficos de las notas
6. Implementación y datos gráficos de las relaciones de notas

Conforme se vayan creando los componentes en un diagrama de clases, se irán guardando en el orden que marca la lista anterior, es decir, siempre aparecerá los datos de las clases, antes que las asociaciones – éste es un caso especial ya que sirve de mucho para la revisión de sintaxis –, las asociaciones aparecerán antes que las generalizaciones, y así sucesivamente. Este último orden es muy útil para el momento de abrir un archivo.

La función del método “guardar()” es crear este código con la información del diagrama creado y guardarlo en archivo, que a su vez es recuperada por la función “abrir()” que de igual forma lee el código, lo interpreta y crea nuevamente el diagrama tal y como el usuario lo guardo la última vez.

5.5.3.3.2 El método “generarCódigo()”

Es la parte principal de la herramienta. Las características más significativas de este módulo son que se genera código en lenguaje C++ y que el cuerpo del código generado es la estructura estática de un sistema, es decir, se generan las estructuras de las clases que participan en el sistema modelado y las relaciones entre ellas, adoptando buenas prácticas de programación como la declaración de funciones inline y uso de la STL de C++.

Para lograr esto, se analizaron nuevamente los datos de los componente del diagrama para tomar aquellos que nos permitieran generar el código fuente deseado. También se creo la estructura que deben tener los archivos generados.

La generación de código incluye las siguientes prácticas de programación:

- Se genera un archivo “.cpp” y un “.h” por cada clase involucrada en el diagrama.
- Para tener acceso a los atributos, se generan operaciones “get” y “set” con la visibilidad correspondiente al atributo, y todos los atributos son colocados en la sección privada.
- Para tener acceso a los objetos involucrados en una asociación, se generan operaciones get() y set() con la visibilidad correspondiente a la asociación, y todas las asociaciones son colocadas en la sección privada.
- Para las asociaciones, si la multiplicidad es menor o igual que 1 no se usan “plantillas” de la STL de C++ y se procede de la siguiente manera:
 - Si es asociación normal o agregación, es declaración por referencia (apuntador)
 - Si es composición, es declaración por valor (sin apuntador)
- Si la multiplicidad es mayor que 1 se usan el template <list> de la STL de C++.

El proceso que se sigue para generar las estructuras de los archivos de código fuente es el siguiente. Por cada clase:

1. Agregar comentarios de cabecera al archivo (.h y .cpp)
2. Crear implementación de clase con herencia si lo hubiera (.h)
3. Crear constructor por omisión (.h y .cpp)
 - 3.1. Si existen atributos con datos iniciales, inicializarlos en el constructor (.cpp)
4. Crear constructor de copia (.h y .cpp)
 - 4.1. Si existen atributos con datos iniciales, inicializarlos también aquí (.cpp)
5. Crear destructor (.h y .cpp)
6. Crear operación de asignación = (.h y .cpp)
7. Crear operación de igualdad == (.h y .cpp)
8. Crear operación de desigualdad != (.h y .cpp)
9. Crear operaciones públicas
 - 9.1. Operaciones públicas definidas por el usuario (.h y .cpp)
 - 9.2. Operaciones get() y set() para atributos públicos (.h)
 - 9.3. Operaciones get() y set() para asociaciones públicas (.h).
10. Crear operaciones protegidas
 - 10.1. Operaciones protegidas definidas por el usuario (.h y .cpp)
 - 10.2. Operaciones get() y set() para atributos protegidos (.h)
 - 10.3. Operaciones get() y set() para asociaciones protegidas (.h)
11. Crear operaciones privadas
 - 11.1. Operaciones privadas definidas por el usuario (.h y .cpp)
 - 11.2. Operaciones get() y set() para atributos privados (.h)
 - 11.3. Operaciones get() y set() para asociaciones privadas (.h)
12. Cerrar el archivo .cpp
13. Crear la implementación en un área privada
 - 13.1. Declaración de atributos públicos
 - 13.2. Declaración de atributos protegidos
 - 13.3. Declaración de atributos protegidos
 - 13.4. Declaración de asociaciones públicas
 - 13.5. Declaración de asociaciones protegidas
 - 13.6. Declaración de asociaciones privadas
14. Crear métodos “inline”
 - 14.1. métodos get() y set() para atributos
 - 14.2. métodos get() y set() para asociaciones.
15. Cerrar el archivo .h

En la Fig. 5.20 se puede observar la estructura con que los archivos .h son generados. Se pueden observar las diferentes secciones que lo forman por medio de los comentarios que se agregan automáticamente al generarse.

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesista: Irving Alberto Cruz Matias
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\Diagrama1.uml
//Generado el Sábado, 26 de Abril de 2003, 06:36 PM
//Clase: Clase0

#ifdef Clase0_h
#define Clase0_h 1

class Clase0
{
    //Declaraciones públicas
    public:
        //----- Constructores generados por omisión
        Clase0();
        Clase0(const Clase0 &right);
        //----- Destructor generados por omisión
        ~Clase0();
        //----- Operación de asignación generado
        Clase0 & operator=(const Clase0 &right);
        //----- Operaciones de igualdad generadas
        int operator==(const Clase0 &right) const;
        int operator!=(const Clase0 &right) const;

        //----- Operaciones definidas por el usuario

        //----- Operaciones Get y Set para atributos

        //----- Operaciones Get y Set para asociaciones

    //Declaraciones protegidas
    protected:
        //----- Operaciones definidas por el usuario

        //----- Operaciones Get y Set para atributos

        //----- Operaciones Get y Set para asociaciones

    //Declaraciones privadas
    private:
        //----- Operaciones definidas por el usuario

        //----- Operaciones Get y Set para atributos

        //----- Operaciones Get y Set para asociaciones

    private: //Implementación
        //----- Atributos establecidos como públicos

        //----- Atributos establecidos como protegidos

        //----- Atributos establecidos como privados

        //----- Asociaciones establecidas como públicas

        //----- Asociaciones establecidas como protegidas

        //----- Asociaciones establecidas como privadas

};

//----- Métodos Get y Set para atributos (declaraciones inline)

//----- Métodos Get y Set para asociaciones (declaraciones inline)

#endif
//-----FIN DEL ARCHIVO-----

```

Fig. 5.20: Estructura del cuerpo generado para un archivo .h

En la Fig. 5.21 se ve la estructura con que los archivos .cpp son generados. Se pueden observar los cuerpos vacíos de las funciones creadas por omisión y definidas por el usuario.

```
//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesisista: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\Diagrama1.uml
//Generado el Sábado, 26 de Abril de 2003, 06:36 EM
//Clase: Clase0

#include "Clase0.h"

Clase0::Clase0()
{
    //-----Cuerpo del constructor
}

Clase0::Clase0(const Clase0 &right)
{
    //-----Cuerpo del constructor de copia
}

Clase0::~Clase0()
{
    //-----Cuerpo del destructor
}

Clase0 & Clase0::operator=(const Clase0 &right)
{
    //-----Cuerpo de la operación de asignación
}

int Clase0::operator==(const Clase0 &right) const
{
    //-----Cuerpo de la operación de igualdad
}

int Clase0::operator!=(const Clase0 &right) const
{
    //-----Cuerpo de la operación de desigualdad
}

//-----Operaciones definidas por el usuario

//-----FIN DEL ARCHIVO-----
```

Fig. 5.21: Estructura del cuerpo generado para un archivo .cpp

5.5.3.3 Análisis de sintaxis

El analizador de sintaxis se implementó basándose en las características de cada elemento del diagrama de clases de UML, los cuáles se analizaron en la sección 3.2.2. Cada elemento determina los datos que puede contener y las relaciones en las que puede participar.

Los datos que pueden contener cada elemento se capturan mediante ventanas de configuración que validan la captura de estos datos. La Fig. 5.22 muestra la pantalla de configuración de una asociación, donde se puede apreciar que los valores de los datos están representados por diferentes elementos como radio botones y cajas de comprobación que filtran la entrada de los datos.



Fig. 5.22: Ventana de configuración de una asociación en UMLGEC++

Las relaciones en las que cada elemento puede participar están limitadas por la notación de UML, esta característica se modeló desde el detalle de los casos de uso, y fue evolucionando a través de los diagramas de interacción hasta llegar a su implementación. Revisando por ejemplo, la creación de una asociación. La Fig. 5.23 que muestra el diagrama de secuencia del flujo normal del caso de uso “Crear componente de relación”.

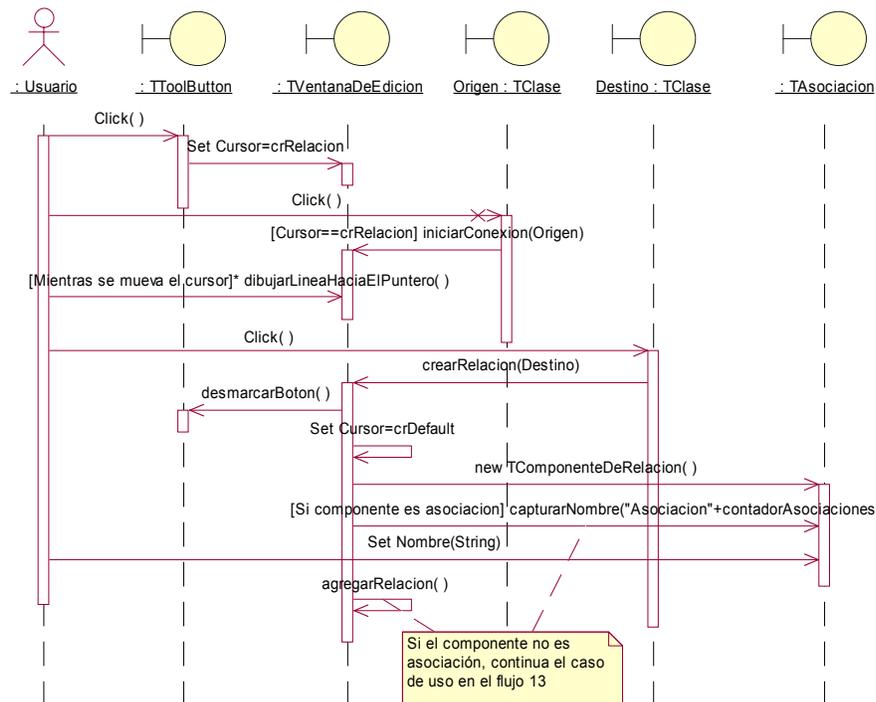


Fig. 5.23: Diagrama de secuencia del flujo normal del caso de uso “Crear componente de relación”

En esta figura se observa que hay un mensaje síncrono donde se hace clic sobre la clase origen, de la cual partirá la asociación. Este mensaje espera a que el usuario haga clic sobre la clase

destino, ya que de otro modo el caso de uso no continuará, por ejemplo, si el usuario hiciera clic sobre un nota, el sistema ignorará dicha acción, ya que no hay ningún diagrama de interacción que modele tal secuencia de mensajes.

También se valida que algunos componentes no estén repetidos dentro de un mismo diagrama, por ejemplo, no puede haber dos clases con el mismo nombre en diagrama de clases. La Fig. 5.24 muestra el flujo alterno del caso de uso de crear un componente de clase, que se lleva a cabo cuando se intenta proporcionar un nombre a una clase que ya pertenece a otra. Se observa que se pide el nombre hasta que éste no esté repetido, ya que tampoco una clase puede quedarse sin un nombre.

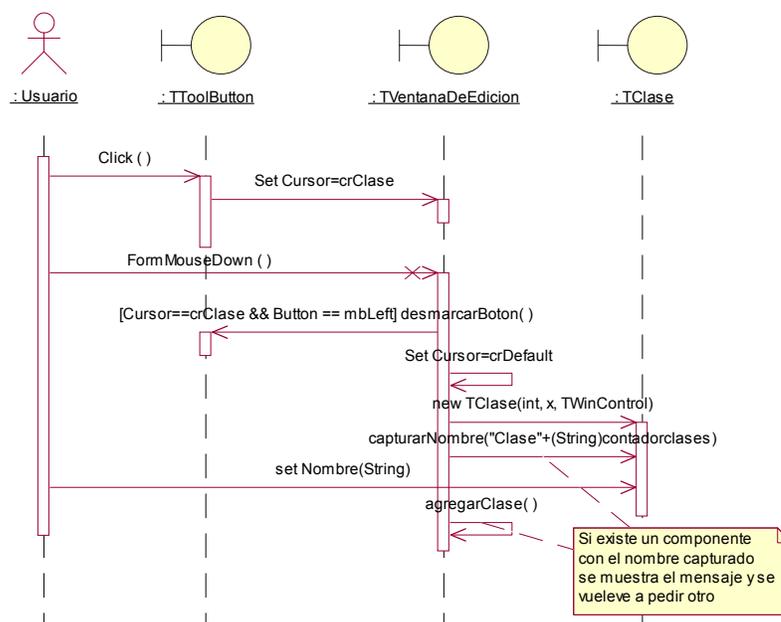


Fig. 5.24: Diagrama de secuencia del flujo alterno “Nombre existente” del caso de uso “Crear componente de clase”

5.6 Pruebas

En la etapa de pruebas, se demuestra la funcionalidad de la herramienta, para ello se necesita el diseño de un sistema de prueba.

5.6.1 Diseñar pruebas

Como sistema de prueba se consideró crear un ejemplo que abarcara todos los componentes y módulos de la herramienta. Se diseñaron y realizaron varias pruebas, aunque aquí sólo se mostrará la más relevante debido a las características del problema.

El sistema de prueba se obtuvo de un problema publicado por primera vez por Grady Booch [Trutra 98], especialmente para mostrar cómo un problema no tan complicado, ni tan simple, podría tener un diseño sencillo y cómo sólo un poco de código necesita ser escrito manualmente.

El sistema consiste en lo siguiente:

Supongamos que tenemos varios cuartos (en una casa, un edificio, etc.), en alguna parte en nuestro diseño, y tenemos que diseñar un sistema de calefacción, de tal manera que en cualquier momento que un cuarto desee un poco de calor, el sistema proveerá ese calor, y cuando el cuarto no lo necesite más, el sistema dejará de mandar calor a ese cuarto.

*Naturalmente, el sistema de calefacción contendrá un horno, el cual será la fuente de calor, y un regulador de flujo de calor, el cual controlará el flujo de calor, éste enviará el calor a un cuarto a través del sistema de calefacción, o detendrá el calor al cuarto, según sea el caso. El calor será enviado a cualquier cuarto usando agua caliente: el cuarto que reciba el calor, abrirá la válvula de agua caliente.*¹⁰

5.6.2 Realizar pruebas

Contando ahora con el sistema de prueba, se obtuvo el diseño de clases y se modeló en la herramienta CASE.

La Fig. 5.25 muestra el diseño del sistema modelado en la herramienta. Se observan las clases participantes y todas sus relaciones de asociación, y composición.

Estando en esta pantalla se pueden llevar a cabo cualquiera de los casos de uso definidos desde los requerimientos, donde el más importante es, por supuesto, la generación de código.

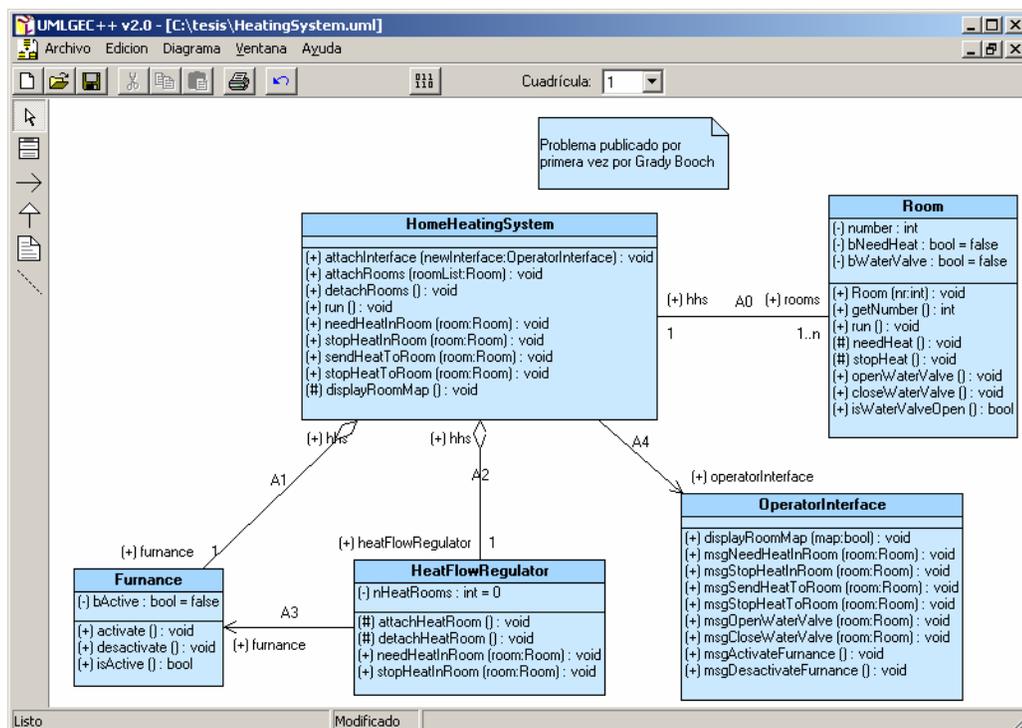


Fig. 5.25: Diagrama de clases del sistema “Home Heating System”

¹⁰ Ejemplo traducido del original expuesto en [Trutra 98]

5.7 Resultados obtenidos

UMLGEC++ es una herramienta de desarrollo visual que permite generar código fuente, de acuerdo a las especificaciones del modelo. El código fuente generado del diagrama de clases es sólo la estructura estática de una aplicación. La especificación del ambiente deberá ser aumentada por el programador de la aplicación, aumentando el código del cuerpo para cada función.

A partir del diagrama modelado en UMLGEC++, se selecciona la opción “Generar código C++”, lo que dará como salida los archivos de código fuente, los cuales se guardarán en el directorio especificado por el usuario. Ver Fig. 5.26.

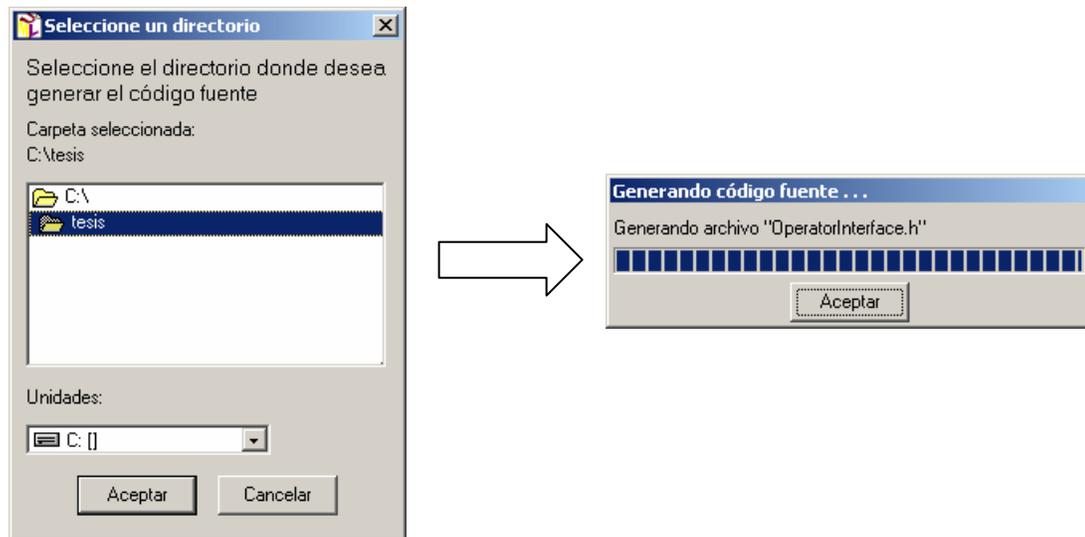


Fig. 5.26: Proceso de generación de código fuente del sistema “Home Heating System”

La Fig. 5.27 muestra los archivos de código fuente, como son generados por UMLGEC++.

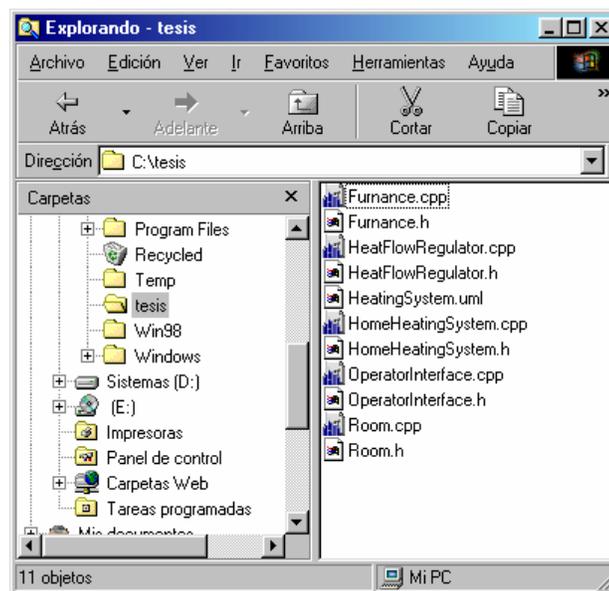


Fig. 5.27: Archivos creados a partir del diagrama del sistema “Home Heating System”

Ahora sólo resta editar los archivos para completar manualmente con código C++ para compilarlo en C++ Builder 5.0 y tener una aplicación 100% ejecutable. El código fuente generado y complementado se encuentra en el Apéndice F.

Se creó un nuevo proyecto en C++ Builder con los archivos generados. En el momento de la edición se ven las ventajas de contar con la estructura del sistema, y sobre todo de la implementación de la STL para manejar las asociaciones con multiplicidad mayores a 1; en C++ esto codifica comúnmente con un arreglo de datos, una lista de apuntadores o se tenía que hacer la implementación de una lista, según las características del problema.

En este caso en particular, se toma ventaja de la STL por ejemplo, en la implementación de la asociación que existe de la clase “HomeHeatingSystem” a la clase “Room” donde existe una multiplicidad de 1 o más del lado de la clase “Room” (Ver Fig. 5.22), porque se tiene que manejar múltiples instancias de la clase “Room”, pero gracias a que UMLGEC++ implementa automáticamente la plantilla “list” para multiplicidades mayores a 1, se obtienen todas las ventajas de una lista de datos.

La declaración de esta asociación se encuentra el archivo “HomeHeatingSystem.h” (Fig. 5.28)

```

private: //Implementación
//----- Asociaciones establecidas como públicas

//Asociación: A0
//Rol: Room::rooms -> Multiplicidad:1..n (Asociación)
list<Room> *rooms;

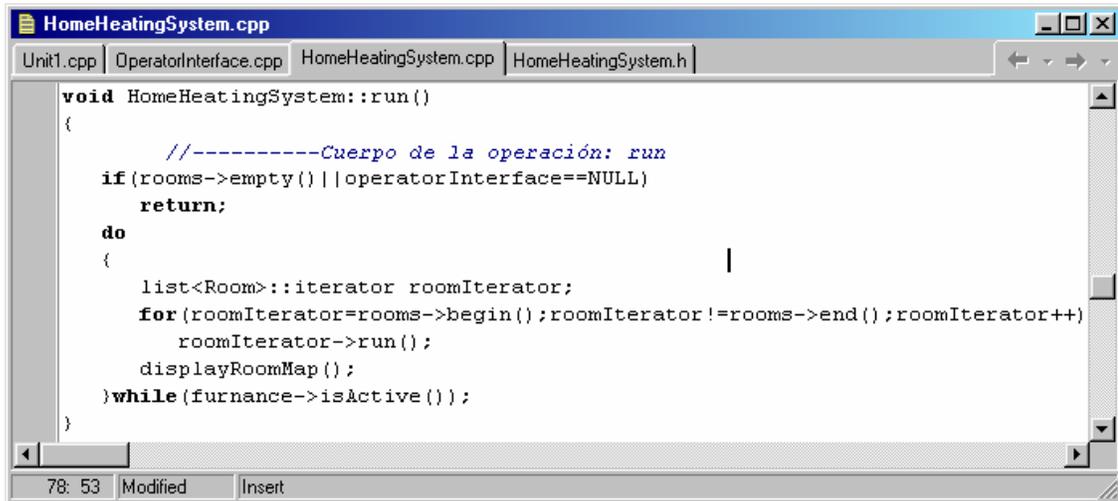
//Asociación: A1
//Rol: Furnance::furnance -> Multiplicidad:1 (Composición)
Furnance *furnance;

```

Fig. 5.28: Sección de implementación de asociaciones de la clase “HomeHeatingSystem”

El uso de esta asociación se puede notar en la sección de código mostrada en la Fig. 5.29, donde se nota que se puede recorrer la lista mediante el uso de un iterador (*iterator*) para este tipo de plantillas, lo que facilita mucho el acceso a los datos que contiene la lista, los cuales se pueden recorrer, insertar, extraer, eliminar, modificar, etc. Además de que se cuenta con las funciones principales de una lista.

Otra de las ventajas notables de usar UMLGEC++ es cuando se generan los atributos y asociaciones de una clase, éstas se declaran en una sección privada de clase, y el acceso a ellas es mediante funciones “get” y “set” implementadas como funciones inline, con esto se separa un poco la implementación de las clases, ya que la única forma de acceder a los datos de una clase es mediante las funciones “get” y “set” de cada atributo o asociación.



```

HomeHeatingSystem.cpp
Unit1.cpp | OperatorInterface.cpp | HomeHeatingSystem.cpp | HomeHeatingSystem.h
void HomeHeatingSystem::run()
{
    //-----Cuerpo de la operación: run
    if (rooms->empty() || operatorInterface==NULL)
        return;
    do
    {
        list<Room>::iterator roomIterator;
        for (roomIterator=rooms->begin(); roomIterator!=rooms->end(); roomIterator++)
            roomIterator->run();
        displayRoomMap();
    } while (furnance->isActive());
}
78: 53 Modified Insert

```

Fig. 5.29: Fragmento de código que muestra el uso de la STL.

Finalmente se compiló el código terminado y se ejecutó la aplicación. La Fig. 5.30 muestra el sistema en ejecución.

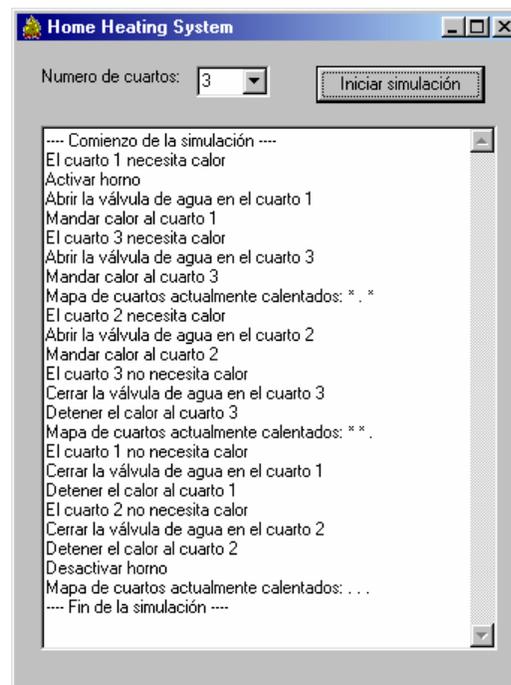


Fig. 5.30: Ejecución del sistema “Home Heating System”

Con el programa en ejecución se observa que efectivamente se obtuvo una aplicación funcional a partir del código generado por UMLGEC++.

5.8 Comparativa de UMLGEC++ frente a otras herramientas

Para la evaluación de UMLGEC++ se aplicaron los mismos criterios que a las otras herramientas case en la sección 5.2.1. Los encargados de evaluar a UMLGEC++ aparte del tesista, fueron algunos alumnos de la Universidad Tecnológica de la Mixteca y empleados del área de desarrollo de software de la misma universidad, quienes valoraron la herramienta mediante un cuestionario de evaluación que mezclaba los criterios ya mencionados.

La tabla 5.3 muestra los resultados de las evaluaciones de UMLGEC++. Se presentan los elementos comparados y evaluados, los cuales tiene asignado un porcentaje que representa la misma importancia que se le dio a las otras herramientas durante la evaluación.

Tabla 5.3: *Evaluación de UMLGEC++*

Característica	UMLGEC++
Facilidad de uso en general (20%)	UMLGEC++ proporciona una interfaz clara que permite a los usuarios adaptarse de manera fácil al ambiente de trabajo que presenta. Contiene una barra de herramientas donde se encuentran los componentes que se pueden utilizar. (8)
Facilidad para crear un diagrama de clases (20%)	Permite editar las características de los diferentes componentes mediante ventanas de configuración y algunas otras directamente en el diagrama, como el caso de de las asociaciones. Y aunque no permite el trazo libre de las relaciones, éstas se mueven junto con sus datos. Implementa también características como el ajuste automático del tamaño de los componentes y la posibilidad de deshacer los cambios sobre el diagrama. (9)
Notación (20%)	Toda la notación que UMLGEC++ contiene es parte de los diagramas de clase UML, y aunque no toda la notación está en la barra de herramientas, es posible encontrarla dentro de las ventanas de configuración. (9)
Asistencia al usuario (10%)	Contiene un archivo de ayuda organizado por temas. (8)
Revisor de sintaxis (30%)	Validación de la sintaxis en tiempo de edición. (8)

La Fig. 5.31 muestra la gráfica comparativa de UMLGEC++ frente a las demás herramientas.

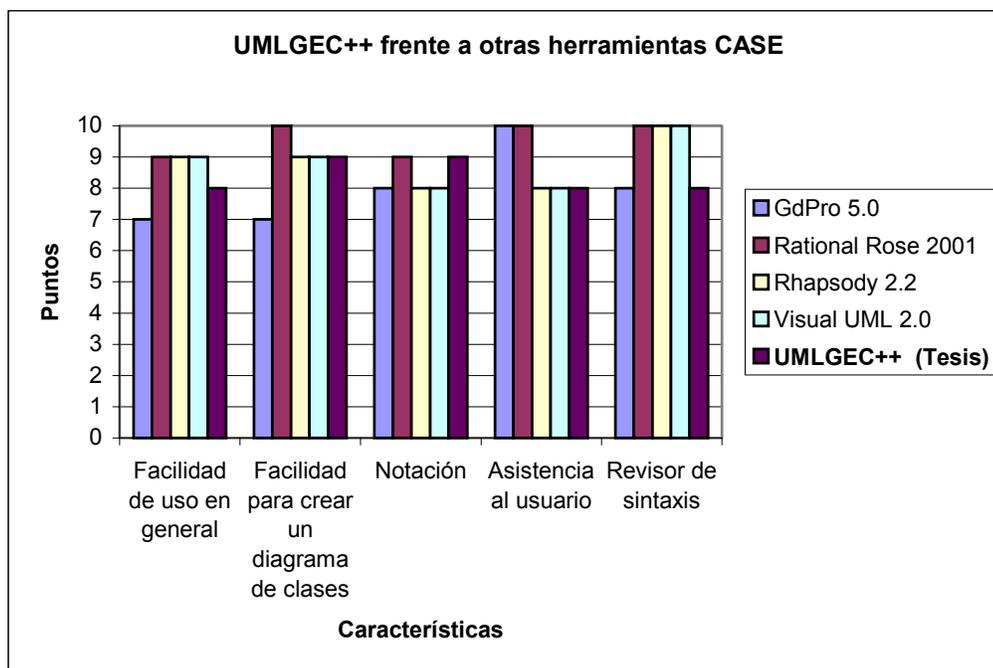


Fig. 5.31: *Gráfica comparativa de UMLGEC++ frente a otras herramientas CASE con soporte a UML*

De aquí, se sacó el puntaje promedio y se agregó a siguiente la lista de los puntajes de las otras herramientas.

Herramienta	Puntaje
1. Rational Rose 2001	9.6
2. Rhapsody 2.2	9
3. Visual UML 2.7	9
4. UMLGEC++	8.4
5. GdPro 5.0	7.8

De esta comparación se deduce que, aunque UMLGEC++ no obtiene el mayor puntaje, si satisface las necesidades de ciertos usuarios, con lo que se asegura que la herramienta será de utilidad para quienes estén empezando con el análisis y diseño orientado a objetos.

Conclusiones y trabajos a futuro

El contenido presentado en este documento di una perspectiva del seguimiento de la Ingeniería del Software con el UML y el Proceso Unificado para crear una herramienta CASE para UML, la cual, con la correcta compilación y ejecución de las pruebas que se han hecho se puede decir que se han cumplido con los objetivos propuestos, y no sólo eso, sino que el proyecto queda abierto para poder ser ampliado con nuevas funciones, y ser adaptado a algunas otras necesidades.

El proceso de desarrollo de UMLGEC++ se facilitó con el uso del UML y el Proceso Unificado, ya que a lo largo de desarrollo del proyecto, se iba obteniendo una vista más clara del sistema, lo que permitió en determinado momento hacer cambios y actualizaciones más fácilmente.

Otra de las ventajas de seguir un proceso formal y un lenguaje igualmente bien definido, es que al final es de gran utilidad, ya que se crea una documentación fortalecida con una notación estándar que puede ser retomada después, y ampliar o modificar la construcción siguiendo cualquier metodología de desarrollo.

La idea básica detrás de UMLGEC++ es – cómo se dijo desde un principio – asistir la creación de diagramas de clase de UML, que permita al desarrollador crear código en lenguaje C++ eficiente y de alta calidad, lo que se pudo comprobar con las pruebas realizadas, como el caso del ejemplo en la sección 5.6 donde se observa que teniendo una vez diseñado el diagrama de clases, basta con modelarlo en UMLGEC++ para poder generar el código fuente de las estructuras de las clases y sus relaciones, lo que anteriormente tomaba tiempo al tener que codificarlo directamente y en revisar las relaciones entre archivos.

Otra ventaja que se obtiene al hacer uso de UMLGEC++ es que el desarrollador podrá tomar ventaja del uso de la declaración de funciones *inline* y de la STL de C++, y de que al hacer un cambio en el sistema, sólo se hace en el diagrama y no pierde tiempo en cambiar todas estas implementaciones en el código.

El objetivo general del proyecto es proveer un material de apoyo a la materia de Programación Orientada a Objetos II impartida en la Universidad Tecnológica de la Mixteca, y dado que en el campus predomina el uso de plataforma Windows, UMLGEC++ fue compilada en Builder C++ para poder ser ejecutado en cualquier versión de Windows. Por otra parte, la herramienta es de ambiente gráfico, ámbito donde C++ Builder es altamente poderoso y crea aplicaciones de poco espacio de almacenamiento y de un alto desempeño de ejecución, esto se puede comprobar al momento de utilizar la herramienta y utilizar los principales módulos como el generador de código.

A pesar de todas estas ventajas de C++ Builder, el principal inconveniente de éste como herramienta de desarrollo, es que no se crea una aplicación multiplataforma, sin embargo, existen (aunque no se probó) emuladores para Linux que corren aplicaciones Windows y emuladores para Macintosh que permiten ejecutar Windows.

Vale la pena mencionar que el desarrollo de una herramienta CASE que soporte casi toda la metodología UML puede llevar demasiado tiempo para crearse y se necesitaría además de varios desarrolladores, lo cual no significa que no se pueda llevar a cabo.

Gracias a que se cuenta con artefactos creados través del ciclo de desarrollo, como documentos de texto, diagramas y archivos de código fuente, se pueden considerar dentro de los desarrollos a futuro de esta herramienta los siguientes puntos:

- Aumentar la capacidad de soportar otros diagramas de UML, como los diagramas de secuencia, colaboración, de casos de uso, de estados, etc., y con ello soportar más parte del ciclo de desarrollo del software y por tanto poder generar un código más completo.
- Generación de código en otros lenguajes. Aunque la herramienta está implementada en lenguaje C++, y genera código en el mismo lenguaje, se puede crear diferentes plantillas de generación de código para otros lenguajes, como Java, Pascal, u otro lenguaje de programación, de forma similar a como se expuso en la sección 5.5.3.3.2.
- Compatibilidad con otras herramientas CASE. La estructura de almacenamiento en archivo tiene como base el formato de los archivos .mdl de Rational Rose, aunque no es compatible con éste. Pero la estructura para guardar un archivo puede ser modificada agregando o quitando información con el fin de ser compatible con algunas otras herramientas CASE existentes.

Bibliografía

[Booch 99] Grady Booch, *UML in Action*, Communications of the ACM 26 October 1999/Vol.42, No.10.

[Coleman 97] Panel Session. Moderator: Derek Coleman, Panelists: John Artim, Viktor Ohnjec, Erick Rivas, Jim Rumbaugh, Rebecca Wtis-Brock., *UML: the language of blueprints for software?*, Conference on Object Oriented programming systems languages and applications October 5-9, 1997.

[CSI 01] Consejo Superior de Informática de España, *Herramientas de Ayuda al Desarrollo*, <http://www.map.es/csi/silice/Sgcase.html>, Fecha de acceso: 5 de Enero de 2001.

[Fernández 01] Carlos Alberto Fernández y Fernández, *Programación Orientada a Objetos II – Notas de clase*, Instituto de Electrónica y Computación – Universidad Tecnológica de la Mixteca., 2001.

[Figuroa 97] Pablo Figuroa, *Elementos notacionales de UML 1.0*, PhD student of the Computing Science Department at the University of Alberta, Canada <http://www.cs.ualberta.ca/~pfiguero/soo/uml/>, 1997.

[Fowler 98] Martin Fowler, *Why Use the UML?*, 1998 Software Development magazine.

[Fowler 99] Martin Fowler, *UML Gota a Gota*, Editorial Addison Wesley Longman de México, S.A. de C.V. México 1999.

[Garmire 99] David Garmire, *Lecture Notes on CASE-Tools : Together/J*, Carnegie Mellon University, <http://stars.globalse.org/stars1/Lectures/CASE/CaseTools1.pdf>, 23 September 1999

[Henderson 99] Peter Henderson & Robert Walters, *System Design Validation Using Format Models*, 10th IEEE International Workshop in Rapid System Prototyping, Clearwater, USA, June 99

[Hetherington 99] Tom Hetherington, *UML Tool Review*, Applied Research Laboratories - University of Texas, <http://www.arlut.utexas.edu/~tomh/UML/>, Last modified: Feb 24 1999.

[Hogan 99] Debbi Hogan, *A Study of Case Tools*, University of Missouri - St. Louis (Information Systems Analysis), <http://www.umsl.edu/~sauter/analysis/dfd/CaseTool.html>, Fall 1999.

[IEEE 93] IEEE, *Standards Collection: Software Engineering*, IEEE Standard 610.12-1990, IEEE 1993.

[INEI 97] Instituto Nacional de Estadística e Informática del Perú, *Metodologías informáticas – Herramientas para el desarrollo de sistemas de información*, <http://www.inei.gob.pe/cpi/bancopub/libfree/lib615/>, Julio de 1997.

- [INEI 99] Instituto Nacional de Estadística e Informática del Perú – Sub-Jefatura de Informática. Dirección Técnica de Desarrollo Informático. *Herramientas CASE*, <http://www.uap.edu.pe/samples/demo/Libro-5103.pdf>, Noviembre de 1999.
- [Jacobson 00] I. Jacobson , G. Booch y J. Rumbaugh, *El Proceso Unificado de Desarrollo de Software*, Editorial: Pearson Educación, S.A., Madrid, 2000.
- [Kruchten 96] Philippe Kruchten, *A Rational Development Process*, Rational Software Corp, Vancouver,BC, 1996.
- [Lokan 99] Chris Lokan & Gary Millar, *Course Notes of Information Systems: Methods for IS Development*, School of Computer Science - University of New South Wales <http://www.cs.adfa.edu.au/teaching/studinfo/is1/Lectures/topic19.html> , 26 August 1999.
- [McClure 89] Carma McClure, *The CASE experience*, BYTE magazine of April 1989.
- [Pressman 93] Roger S. Pressman, *Ingeniería del Software: Un enfoque práctico*, Editorial: McGraw Hill España, 1993.
- [Rational 98] Rational Software Corporation, *Rational Unified Process - Best Practices for Software Development Teams*, Rational White Paper 1998.
- [Rational 00] Rational Software Corporation, *Rational Unified Process – Process Made Practical*, Rational White Paper 2000.
- [Rumbaugh 00] J. Rumbaugh, I. Jacobson y G. Booch, *El Lenguaje Unificado de Modelado – Manual de Referencia*, Editorial: Pearson Educación, S.A. Madrid, 2000.
- [Schmuller 00] Joseph Schmuller, *Aprendiendo UML en 24 horas*, Editorial: Pearson Educación, S.A. México, 2000.
- [SEI 99] Software Engineering Institute (SEI), *Computer-Aided Software Engineering (CASE) Environments*, http://www.sei.cmu.edu/legacy/case/case_what.html , Last Modified: 14 April 1999.
- [Stobart 96] Simon Stobart (Senior Lecturer in Computing – University of Sunderland, UK), *The CASE Tool Home Page*, <http://osiris.sunderland.ac.uk/sst/case2/welcome.html> , Last modified: March 1996.
- [Terry 90] B. Terry & D. Logee, *Terminology for Software Engineering and Computer-Aided Software Engineering*, Software Engineering Notes. April 1990.
- [Trutra 98] Cosmín Trutra, *Visual Modeling with Rational Rose and UML*, Department of Computer Science – University of Toronto, <http://www.cs.toronto.edu/~cosmin/rosetutor/> , December 1998.

Apéndice A. Glosario

Introducción

En ésta sección se recopila y definen los principales términos usados durante el desarrollo del presente proyecto. En este suplemento se definen términos relacionados con UML y el Proceso Unificado, así como de términos de programación Orientada a Objetos.

Términos

actividad: La ejecución de una acción.

arquitectura: Conjunto de decisiones significativas acerca de la organización de un sistema de software, la selección de los elementos estructurales a partir de los cuales se compone el sistema y las interfaces entre ellos, junto con sus comportamiento tal y como se especifica en las colaboraciones entre esos elementos, la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente mayores, y el estilo arquitectónico que guía esta organización: estos elementos y sus interfaces, sus colaboraciones y su composición. La arquitectura del software se interesa no sólo por la estructura y el comportamiento, sino también por las restricciones y compromisos de uso, funcionalidad, funcionamiento, flexibilidad al cambio, reutilización, comprensión, economía y tecnología, así como por aspectos estéticos.

CASE: *Computer Aided Software Engineering* – Ingeniería del Software Asistida por Computadora.

caso de uso: Es una descripción de un tipo de secuencias de acciones. Incluyendo variaciones, que un sistema lleva a cabo y que conduce a un resultado observable de interés para un actor determinado.

ciclo: Ciclo de vida del software que cubre las cuatro fases del Proceso Unificado.

diagrama: La presentación gráfica de un conjunto de elementos, usualmente representado como un grafo conectado de vértices y arcos.

estereotipo: Una extensión del vocabulario del UML, que permite la creación de nuevos tipos de bloques de construcción que se derivan de otros existentes pero que son específicos a un problema particular.

fase: Periodo de tiempo entre dos hitos principales de un proceso de desarrollo.

flujo de trabajo : Realización de un caso de uso o parte de él. Puede describirse en términos de diagramas de actividad que incluye a los trabajadores participantes, las actividades que realiza y los artefactos que producen.

hito : Punto en el que han de tomarse decisiones importantes. Cada fase acaba en un hito principal en el cual los gestores han de tomar decisiones cruciales de continuar o no el proyecto, y decidir sobre la planificación, presupuestos y requisitos del mismo. Se consideran a los hitos principales como puntos de sincronización en los que coinciden una serie de objetivos bien definidos, se completan artefactos, se toman decisiones de pasar o no a la fase siguiente.

instancia: Una manifestación concreta de una abstracción; una entidad sobre la que pueden aplicarse un conjunto de operaciones y que tiene un estado que almacena los efectos de las operaciones; un sinónimo de objeto.

iteración: Conjunto de actividades llevadas a cabo de acuerdo a un plan (de iteración) y unos criterios de evaluación, que lleva a producir una versión, ya sea interna o externa.

mensaje: Una especificación de una comunicación entre objetos que lleva información con la expectativa de que de ella se seguirá alguna actividad; la recepción de una instancia de mensaje es normalmente considerada una instancia de un evento.

metamodelo: Un metamodelo o modelo es la representación de algo en cierta forma, para el caso de la Ingeniería del Software un modelo es un diagrama que representa la definición de la notación.

método: La implementación de una operación.

modelo: Una abstracción de un sistema cerrado semánticamente. (Ver también metamodelo)

modelo de dominio: Modelo que captura los aspectos más importante de objetos en el contexto del sistema. Es un caso especial de un modelo de negocios.

modelo de negocios: Técnica que provee una metodología para modelar procesos de negocio basada en la utilización del UML.

modelo de casos de uso: Modelo formado por actores, casos de uso y relaciones entre ambos, modelo que describe lo que el sistema debería hacer por sus usuarios y bajo que restricciones.

OMG: *Object Management Group* – Grupo Administrador de Objetos.

OO: *Object Oriented* – Orientado a Objetos.

requisito: Condición o capacidad que debe cumplir un sistema

requisito funcional: Requisito que especifica una acción que debe ser capaz de realizar el sistema, sin considerar restricciones físicas, requisito que especifica comportamiento de entrada / salida de un sistema.

requisito no funcional: Requisito que especifica propiedades del sistema, como restricciones del entorno de implementación, rendimiento, dependencias de plataforma, extensibilidad o fiabilidad.

responsabilidad: Un contrato u obligación de un tipo o clase.

rol: Comportamiento específico de una entidad que participa en un contexto particular.

UML: *Unified Modeling Language* – Lenguaje Unificado de Modelado.

UP: *Unified Process* – Proceso Unificado, comercialmente conocido como RUP (*Rational Unified Process*) propiedad de Rational Software Corporation.

versión: Conjunto de artefactos relativamente completo y consistente – que incluye posiblemente una construcción – entregado a un usuario interno o externo; entrega de tal conjunto.

versión externa: Versión expuesta a los clientes y usuarios, externos al proyecto y sus miembros.

versión interna: Una versión no expuesta a los clientes y usuarios, sino sólo de forma interna, al proyecto y sus miembros.

vista: Proyección de un modelo, que es visto desde una perspectiva dada o un lugar estratégico, y que omite las entidades que no son relevantes para esta perspectiva.

Apéndice C. Clases contenidas dentro de los subsistemas

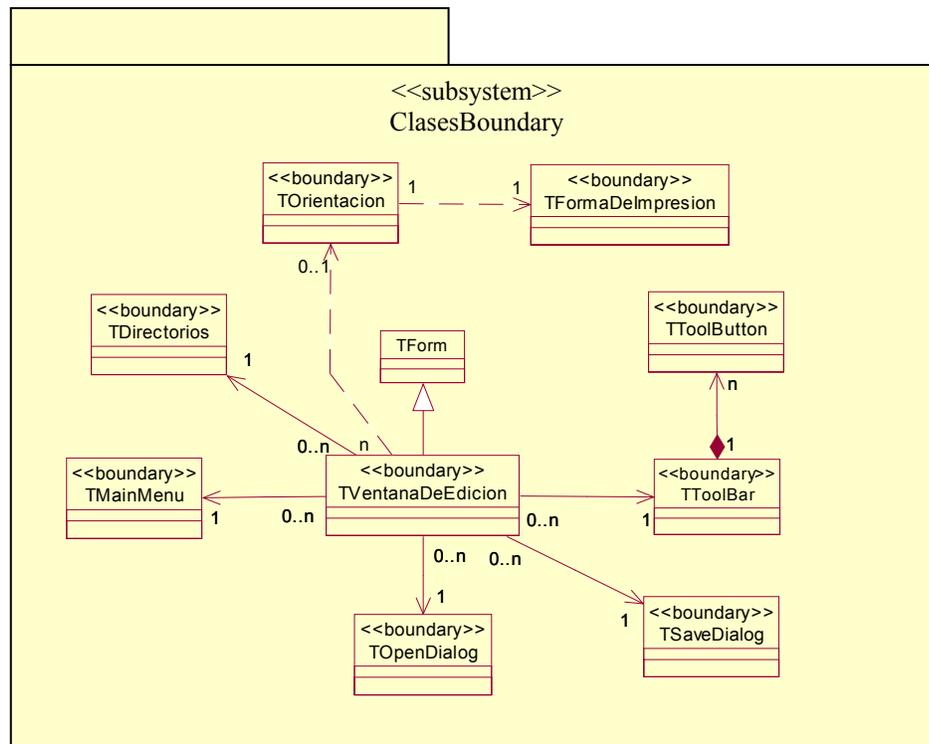


Fig. C.1: Clases contenidas dentro del subsistema “Clases Boundary”

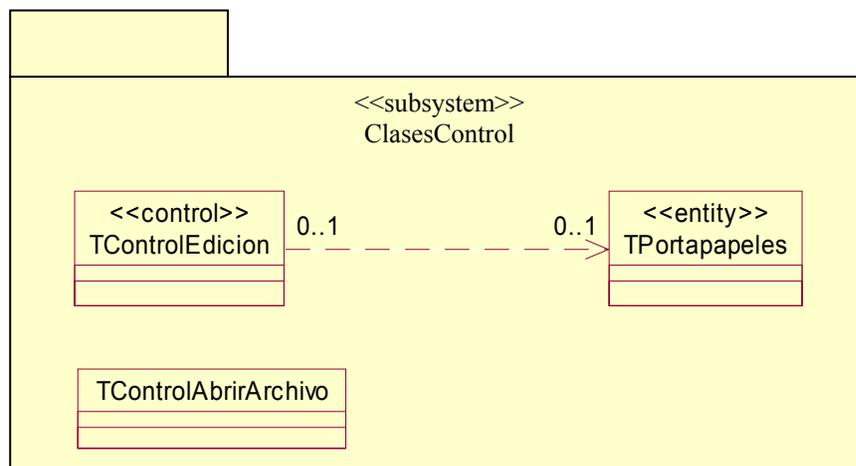


Fig. C.2: Clases contenidas dentro del subsistema “ClasesControl”

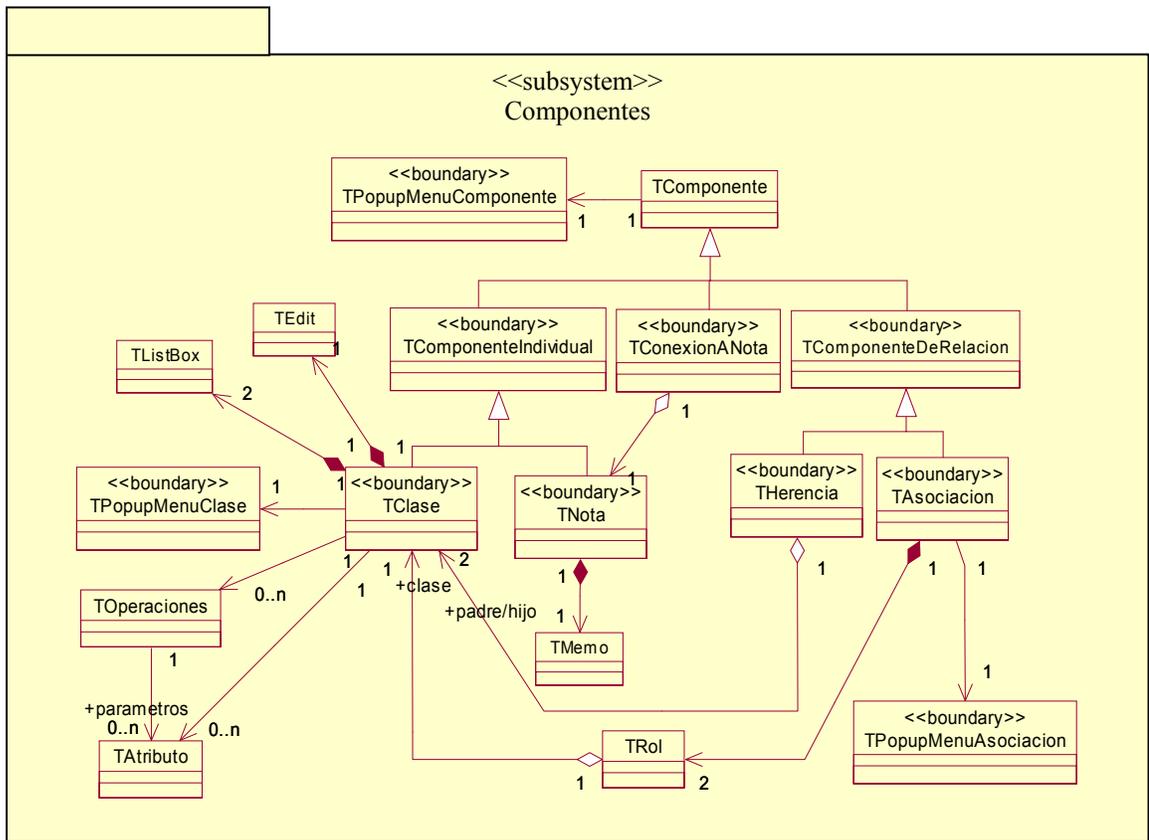


Fig. C.3: Clases contenidas dentro del subsistema "Componentes"

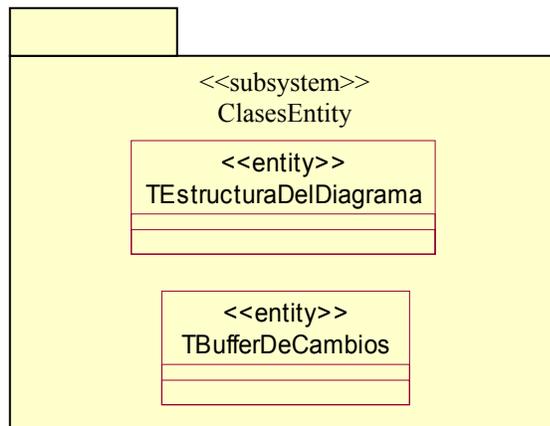


Fig. C.4: Clases contenidas dentro del subsistema "ClasesEntity"

Apéndice D. Estructura para el almacenamiento de archivo

A continuación se muestra la estructura de alto nivel que se creó para almacenar un archivo:

```

DIAGRAMA "<Nombre del diagrama>"
<Número de componentes>
Objeto Clase <ID Clase>
<Nombre_clase1> //Nombre de la clase
{
    X <Posición x>
    Y <Posición y>
    Ancho <Ancho en pixeles>
    Ver_Atributos <True o False>
    Ver_Operaciones <True o False>
    Alto_Etiqueta <Alto del módulo de las etiquetas>
    Alto_Atributos <Posición del módulo de los atributos>
    Alto_Operaciones <Posición del módulo de las operaciones>
    Atributos //Lista de atributos
    {
        {
            Nombre <Nombre atributo1>
            Visibilidad <Tipo de visibilidad>
            Tipo <Tipo de dato>
            Valor inicial <valor>
        }
        {
            Nombre <Nombre atributo2>
            Visibilidad <Tipo de visibilidad>
            Tipo <Tipo de dato>
            Valor inicial <valor>
        }
        .
        .
        Más atributos
    }
    Operaciones //Lista de las operaciones de la clase
    {
        {
            Nombre <Nombre operacion1>
            Visibilidad <Tipo de visibilidad>
            Tipo <Tipo de dato que devuelve>
            Parámetros
            {
                {
                    Nombre <Nombre parametro1>
                    Tipo <Tipo de dato>
                    Valor inicial <valor>
                }
                {
                    Nombre <Nombre parametro2>
                    Tipo <Tipo de dato>
                    Valor inicial <valor>
                }
                .
                .
                Más parámetros
            }
        }
        {
            Nombre <Nombre operacion2>
            Visibilidad <Tipo de visibilidad>

```

```

        Tipo <Tipo de dato que devuelve>
        Parámetros
        .
        .
        Más operaciones
    }
}
.
.
Más clases

Objeto Asociacion <ID asociación>
<Nombre asociación1>
{
    Tipo <Tipo de asociación>
    Clase <ID de la clase asociada al RolA>
    Clase < ID de la clase asociada al RolB>
    RolA
    {
        Etiqueta <Nombre del rolA>
        Visibilidad <Tipo de visibilidad>
        Multiplicidad <Multiplicidad>
        Navegable <True o False>
        Agregado <True o False>
        Compuesto <True o False>
    }
    RolB
    {
        Etiqueta <Nombre del rolB>
        Visibilidad <Tipo de visibilidad>
        Clase <ID de la clase a la cual está conectando>
        Multiplicidad <Multiplicidad>
        Navegable <True o False>
        Agregado <True o False>
        Compuesto <True o False>
    }
}
.
.
Más asociaciones

Objeto Generalización <ID generalizacion1>
{
    Visibilidad <Tipo de visibilidad>
    CP <ID de la clase padre>
    CH <ID de la clase hijo>
}
Objeto Generalización <ID generalizacion2>
{
    Visibilidad <Tipo de visibilidad>
    CP <ID de la clase padre>
    CH <ID de la clase hijo>
}
.
.
Más generalizaciones

Objeto Nota <ID nota>
{
    X <coordenada>
    Y <coordenada>
    ALTO <height>
    ANCHO <width>
    [
        línea 1
        línea 2
    ]
}

```

```
      .  
      .  
    ]  
  }  
  .  
  .  
  Más notas  
  
Objeto conexión a nota  
{  
    N <ID de la nota a la cual está conectando>  
    C <ID del componente al cual está conectando el otro extremo>  
}  
Objeto conexión a nota  
{  
    N <ID de la nota a la cual está conectando>  
    C <ID del componente al cual está conectando el otro extremo>  
}  
.  
.  
Más conexiones a nota  
  
/*****FIN DEL ARCHIVO*****/
```

Apéndice E. Código para el almacenamiento de archivo

A continuación se muestra el código tal de cómo es realmente almacenada la estructura de un diagrama en un archivo. Al final de este código se encuentra la lista de los valores ASCII de los símbolos por los cuáles son reemplazados.

```

TI <Título del diagrama>
X[NUMERO DE COMPONENTES]
{
    OC <ID>
    <Nombre Clase1>
    X[102]
    Y[232]
    W[80]
    VA T
    VO T
    H[10] //Alto Nombre
    H[30] //Alto Atributos
    H[40] //Alto Operaciones
    AT
    NA <Nombre atributo1>
    V 1
    T <Tipo de dato>
    VI <Valor inicial>

        NA <Nombre atributo2>
    V 1
    T <Tipo de dato>
    VI <Valor inicial>
    .
    .
    .
    OP
    NO<Nombre operacion1>
    V 1
    T <Tipo de dato que devuelve>
    P
        NP <Nombre parametro1>
        T <Tipo de dato>
        VI <Valor inicial>

        NP <Nombre parametro2>
        T <Tipo de dato>
        VI <Valor inicial>
        .
        .
        .
    NO<Nombre operacion2>
    V 2
    T <Tipo de dato que devuelve>
    P
        NP <Nombre parametro1>
        T <Tipo de dato>
        VI <Valor inicial>

        NP <Nombre parametro2>
        T <Tipo de dato>
        VI <Valor inicial>
        .
        .
        .

```

```

.
.
OC <ID>
<Nombre Clase2>
  X[102]
  Y[232]
  W[80]
  VA T
  VO T
  H[10] //Alto Nombre
  H[30] //Alto Atributos
  H[40] //Alto Operaciones
AT
  NA <Nombre atributo1>
  V 1
  T <Tipo de dato>
  VI <Valor inicial>
  NA <Nombre atributo2>
  V 1
  T <Tipo de dato>
  VI <Valor inicial>
.
.
.
OP
  NO<Nombre operacion1>
  V 1
  T <Tipo de dato que devuelve>
  P
    NP <Nombre parametro1>
    T <Tipo de dato>
    VI <Valor inicial>

    NP <Nombre parametro2>
    T <Tipo de dato>
    VI <Valor inicial>
    .
    .
    .

  NO<Nombre operacion2>
  V 2
  T <Tipo de dato que devuelve>
  P
    NP <Nombre parametro1>
    T <Tipo de dato>
    VI <Valor inicial>

    NP <Nombre parametro2>
    T <Tipo de dato>
    VI <Valor inicial>
    .
    .
    .
.
.
OA <ID>
<Nombre asociación1>
  TA 1
  C <ID de la clase asociada ROL A>
  C <ID de la clase asociada ROL B>
RA
  NR<Nombre del rolA>
  V 1
  MU 2
  NV T

```

```

                AG F
                CO F
RB
    NR<Nombre rolB>
    V 2
        MU2
        NVT
        AGF
        COF

OA <ID>
<Nombre asociación2>
    TA 1
    C <ID de la clase asociada ROL A>
    C <ID de la clase asociada ROL B>
RA
    NR<Nombre rolA>
        V 0
            MU 2
            NV T
            AG F
            CO F
    RB
        NR<Nombre rolB>
        V 0
            MU 2
            NV T
            AG F
            CO F
    .
    .
    .
    OG <ID>
    V 1
    C<ID de la clase padre>
    C<ID de la clase hija>

    OG <ID>
    V 1
    C<ID de la clase padre>
    C<ID de la clase hija>
    .
    .
    .
ON <ID>
    X[38]
    Y[332]
    H[34]
    W[12]
    [
        línea 1
        línea 2
        .
        .
    ]

ON <ID>
    X[348]
    Y[232]
    H[48]
    W[48]
    [
        línea 1
        línea 2
        .
        .

```

```

]
.
.
.
OCN
N <ID de la nota a la cual está conectando>
C <ID del componente al cual está conectando el otro extremo>

OCN
N <ID de la nota a la cual está conectando>
C <ID del componente al cual está conectando el otro extremo>
.
.
}

```

Símbolos

Las palabras de la estructura de arriba serán reemplazadas por su código binario correspondiente según la siguiente lista:

TI=1 ☉	X=2 ☉
Y=3 ▼	H=4 +
W=5 ♁	OC=6 ⬆
VA=7 ▪	VO=8 ■
AT=9 ○	NA=32 (espacio)
V=11 ♂	T=12 †
VI=14 ⚢	OP=15 ○
NO=16 ►	P=17 ◀
NP=18 †	OA=19 !!
TA=20 ¶	RA=21 §
NR=22 —	C=23 †
MU=24 †	NV=25 ↓
AG=26 →	CO=27 ←
RB=28 L	OG=29 ↔
ON=30 ▲	OCN=31 ▼
N=33 ;	{='{'
}='}'	<='<'
>='>'	[='['
]=']'	.=35

En esta lista no se incluye el carácter 10 y 13 por ser el código correspondiente a la tecla *Enter*.

Apéndice F. Código fuente completo del sistema *Home Heating System*

La implementación de la interfaz gráfica fue hecha en C++ Builder en los archivos Unit.h y Unit.cpp, los cuales fueron escritos manualmente. El resto de los archivos fueron generados mediante la herramienta UMLGEC++, estos archivos fueron completados manualmente con código que se muestra con letras en **negritas** en este documento.

INTERFAZ GRÁFICA (C++ Builder)

Unit1.h

```
//-----
#ifndef Unit1H
#define Unit1H
//-----
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//-----
class TPrincipal : public TForm
{
__published:           // IDE-managed Components
    TLabel *Label1;
    TComboBox *numRooms;
    TMemo *listMsg;
    TButton *Button1;
    void __fastcall Button1Click(TObject *Sender);
private:               // User declarations
public:                // User declarations
    __fastcall TPrincipal(TComponent* Owner);
};
//-----
extern PACKAGE TPrincipal *Principal;
//-----
#endif
```

Unit1.cpp

```
//-----
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include "HomeHeatingSystem.h"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TPrincipal *Principal;

void execute(int nr);

//-----
__fastcall TPrincipal::TPrincipal(TComponent* Owner)
: TForm(Owner)
{
}
//-----
void __fastcall TPrincipal::Button1Click(TObject *Sender)
{
    listMsg->Clear();
    int nr;
    try
    {
        nr= numRooms->Text.ToInt();
    }
    catch (EConvertError &e)
    {
        nr = 0;
    }
    if (nr > 0)
        execute(nr);
    else
        listMsg->Lines->Add("; Número no válido !!, introduzca un número válido");
}
}
```

```
//-----
void execute(int nr)
{
    HomeHeatingSystem *hhs = new HomeHeatingSystem();
    list<Room> *rooms = new list<Room>;
    for (int i = 0; i < nr; i++)
    {
        rooms->push_back(Room(i+1)); // Room numbers are in range 1..nr
    }
    OperatorInterface *operatorInterface = new OperatorInterface();
    hhs->attachInterface(operatorInterface);
    hhs->attachRooms(rooms);
    Principal->listMsg->Lines->Add("---- Comienzo de la simulación ----");
    hhs->run();
    Principal->listMsg->Lines->Add("---- Fin de la simulación ----");
}

```

CÓDIGO GENERADO EN UMLGEC++ Y COMPLETADO MANUALMENTE

HeatingSystem.h

```
//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesis: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: HomeHeatingSystem

#ifndef HomeHeatingSystem_h
#define HomeHeatingSystem_h 1

//Room
#include "Room.h"
class Room;
//Furnance
#include "Furnance.h"
class Furnance;
//HeatFlowRegulator
#include "HeatFlowRegulator.h"
class HeatFlowRegulator;
//OperatorInterface
#include "OperatorInterface.h"
class OperatorInterface;
//Platilla <list> de la STL
#include <list.h>

class HomeHeatingSystem
{
    //Declaraciones públicas
public:
    //----- Constructores generados por omisión
    HomeHeatingSystem();
    HomeHeatingSystem(const HomeHeatingSystem &right);
    //----- Destructor generado por omisión
    ~HomeHeatingSystem();
    //----- Operación de asignación generado
    HomeHeatingSystem & operator=(const HomeHeatingSystem &right);
    //----- Operaciones de igualdad generadas
    int operator==(const HomeHeatingSystem &right) const;
    int operator!=(const HomeHeatingSystem &right) const;

    //----- Operaciones definidas por el usuario
    void attachInterface(OperatorInterface *newInterface);
    void attachRooms(list<Room> *roomList);
    void detachRooms();
    void run();
    void needHeatInRoom(Room *room);
    void stopHeatInRoom(Room *room);
    void sendHeatToRoom(Room *room);
    void stopHeatToRoom(Room *room);

    //----- Operaciones Get y Set para atributos

    //----- Operaciones Get y Set para asociaciones

    //Asociación: A0
    //Rol: Room::rooms
    const list<Room> *get_rooms() const;
    void set_rooms(list<Room> *valor);

    //Asociación: A1
    //Rol: Furnance::furnance
    const Furnance *get_furnance() const;
    void set_furnance(Furnance *valor);
}

```

```

//Asociación: A2
//Rol: HeatFlowRegulator::heatFlowRegulator
const HeatFlowRegulator *get_heatFlowRegulator() const;
void set_heatFlowRegulator(HeatFlowRegulator *valor);

//Asociación: A4
//Rol: OperatorInterface::operatorInterface
const OperatorInterface *get_operatorInterface() const;
void set_operatorInterface(OperatorInterface *valor);

//Declaraciones protegidas
protected:
//----- Operaciones definidas por el usuario
void displayRoomMap();

//----- Operaciones Get y Set para atributos

//----- Operaciones Get y Set para asociaciones

//Declaraciones privadas
private:
//----- Operaciones definidas por el usuario

//----- Operaciones Get y Set para atributos

//----- Operaciones Get y Set para asociaciones

private: //Implementación
//----- Atributos establecidos como públicos

//----- Atributos establecidos como protegidos

//----- Atributos establecidos como privados

//----- Asociaciones establecidas como públicas

//Asociación: A0
//Rol: Room::rooms -> Multiplicidad:1..n (Asociación)
list<Room> *rooms;

//Asociación: A1
//Rol: Furnance::furnance -> Multiplicidad:1 (Agregación)
Furnance *furnance;

//Asociación: A2
//Rol: HeatFlowRegulator::heatFlowRegulator -> Multiplicidad:1 (Agregación)
HeatFlowRegulator *heatFlowRegulator;

//Asociación: A4
//Rol: OperatorInterface::operatorInterface -> Multiplicidad:<No especificada> 1 por omisión (Asociación)
OperatorInterface *operatorInterface;

//----- Asociaciones establecidas como protegidas

//----- Asociaciones establecidas como privadas

};

//----- Métodos Get y Set para atributos (declaraciones inline)

//----- Métodos Get y Set para asociaciones (declaraciones inline)

inline const list<Room> *HomeHeatingSystem::get_rooms() const
{
    return rooms;
}

inline void HomeHeatingSystem::set_rooms(list<Room> *valor)
{
    rooms = valor;
}

inline const Furnance *HomeHeatingSystem::get_furnance() const
{
    return furnance;
}

inline void HomeHeatingSystem::set_furnance(Furnance *valor)
{
    furnance = valor;
}

inline const HeatFlowRegulator *HomeHeatingSystem::get_heatFlowRegulator() const
{
    return heatFlowRegulator;
}

inline void HomeHeatingSystem::set_heatFlowRegulator(HeatFlowRegulator *valor)
{
    heatFlowRegulator = valor;
}

inline const OperatorInterface *HomeHeatingSystem::get_operatorInterface() const
{

```

```

        return operatorInterface;
    }

inline void HomeHeatingSystem::set_operatorInterface(OperatorInterface *valor)
{
    operatorInterface = valor;
}

#endif
//-----FIN DEL ARCHIVO-----

```

HeatingSystem.cpp

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesista: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: HomeHeatingSystem

#include "HomeHeatingSystem.h"

HomeHeatingSystem::HomeHeatingSystem()
{
    //-----Cuerpo del constructor
    //-----Cuerpo del constructor
    furnace = new Furnance();
    furnace->set_hhs(this);
    heatFlowRegulator = new HeatFlowRegulator();
    heatFlowRegulator->set_hhs(this);
    heatFlowRegulator->set_furnance(furnance);
}

HomeHeatingSystem::HomeHeatingSystem(const HomeHeatingSystem &right)
{
    //-----Cuerpo del constructor de copia
}

HomeHeatingSystem::~HomeHeatingSystem()
{
    //-----Cuerpo del destructor
}

HomeHeatingSystem & HomeHeatingSystem::operator=(const HomeHeatingSystem &right)
{
    //-----Cuerpo de la operación de asignación
}

int HomeHeatingSystem::operator==(const HomeHeatingSystem &right) const
{
    //-----Cuerpo de la operación de igualdad
}

int HomeHeatingSystem::operator!=(const HomeHeatingSystem &right) const
{
    //-----Cuerpo de la operación de desigualdad
}

//-----Operaciones definidas por el usuario

void HomeHeatingSystem::attachInterface(OperatorInterface *newInterface)
{
    //-----Cuerpo de la operación: attachInterface
    operatorInterface = newInterface;
}

void HomeHeatingSystem::attachRooms(list<Room> *roomList)
{
    //-----Cuerpo de la operación: attachRooms
    rooms = roomList;
    list<Room>::iterator roomIterator;
    for(roomIterator=rooms->begin(); roomIterator!=rooms->end(); roomIterator++)
        roomIterator->set_hhs(this);
}

void HomeHeatingSystem::detachRooms()
{
    //-----Cuerpo de la operación: detachRooms
    delete rooms;
}

void HomeHeatingSystem::run()
{
    //-----Cuerpo de la operación: run
    if(rooms->empty() || operatorInterface==NULL)
        return;
    do
    {

```

```

        list<Room>::iterator roomIterator;
        for(roomIterator=rooms->begin();roomIterator!=rooms->end();roomIterator++)
            roomIterator->run();
        displayRoomMap();
    }while(furnance->isActive());
}

void HomeHeatingSystem::needHeatInRoom(Room *room)
{
    //-----Cuerpo de la operación: needHeatInRoom
    heatFlowRegulator->needHeatInRoom(room);
}

void HomeHeatingSystem::stopHeatInRoom(Room *room)
{
    //-----Cuerpo de la operación: stopHeatInRoom
    heatFlowRegulator->stopHeatInRoom(room);
}

void HomeHeatingSystem::sendHeatToRoom(Room *room)
{
    //-----Cuerpo de la operación: sendHeatToRoom
    room->openWaterValve();
    operatorInterface->msgSendHeatToRoom(room);
}

void HomeHeatingSystem::stopHeatToRoom(Room *room)
{
    //-----Cuerpo de la operación: stopHeatToRoom
    room->closeWaterValve();
    operatorInterface->msgStopHeatToRoom(room);
}

void HomeHeatingSystem::displayRoomMap()
{
    //-----Cuerpo de la operación: displayRoomMap
    list<bool> *roomMap = new list<bool>;
    list<Room>::iterator roomIterator;
    for(roomIterator=rooms->begin();roomIterator!=rooms->end();roomIterator++)
        roomMap->push_back(roomIterator->isWaterValveOpen());
    operatorInterface->displayRoomMap(roomMap);
}

//-----FIN DEL ARCHIVO-----

```

Room.h

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesista: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: Room

#ifndef Room_h
#define Room_h 1

//HomeHeatingSystem
#include "HomeHeatingSystem.h"
class HomeHeatingSystem;

class Room
{
    //Declaraciones públicas
public:
    //----- Constructores generados por omisión
    Room();
    //Room(const Room &right); //Para evitar problemas con el <list>
    //----- Destructor generado por omisión
    ~Room();
    //----- Operación de asignación generado
    Room & operator=(const Room &right);
    //----- Operaciones de igualdad generadas
    int operator==(const Room &right) const;
    int operator!=(const Room &right) const;

    //----- Operaciones definidas por el usuario
    Room(int nr);
    void run();
    void openWaterValve();
    void closeWaterValve();
    bool isWaterValveOpen();

    //----- Operaciones Get y Set para atributos

    //Atributo: number
    const int get_number() const;
    void set_number(int valor);
}

```

```

//----- Operaciones Get y Set para asociaciones

//Asociación: A0
//Rol: HomeHeatingSystem::hhs
const HomeHeatingSystem *get_hhs() const;
void set_hhs(HomeHeatingSystem *valor);

//Declaraciones protegidas
protected:
//----- Operaciones definidas por el usuario
void needHeat();
void stopHeat();

//----- Operaciones Get y Set para atributos

//----- Operaciones Get y Set para asociaciones

//Declaraciones privadas
private:
//----- Operaciones definidas por el usuario

//----- Operaciones Get y Set para atributos

//Atributo: bNeedHeat
const bool get_bNeedHeat() const;
void set_bNeedHeat(bool valor);

//Atributo: bWaterValve
const bool get_bWaterValve() const;
void set_bWaterValve(bool valor);

//----- Operaciones Get y Set para asociaciones

private: //Implementación
//----- Atributos establecidos como públicos
int number;

//----- Atributos establecidos como protegidos

//----- Atributos establecidos como privados
bool bNeedHeat;
bool bWaterValve;

//----- Asociaciones establecidas como públicas

//Asociación: A0
//Rol: HomeHeatingSystem::hhs -> Multiplicidad:1 (Asociación)
HomeHeatingSystem *hhs;

//----- Asociaciones establecidas como protegidas

//----- Asociaciones establecidas como privadas

};

//----- Métodos Get y Set para atributos (declaraciones inline)

inline const int Room::get_number() const
{
    return number;
}

inline void Room::set_number(int valor)
{
    number = valor;
}

inline const bool Room::get_bNeedHeat() const
{
    return bNeedHeat;
}

inline void Room::set_bNeedHeat(bool valor)
{
    bNeedHeat = valor;
}

inline const bool Room::get_bWaterValve() const
{
    return bWaterValve;
}

inline void Room::set_bWaterValve(bool valor)
{
    bWaterValve = valor;
}

//----- Métodos Get y Set para asociaciones (declaraciones inline)

inline const HomeHeatingSystem *Room::get_hhs() const
{
    return hhs;
}

```

```

inline void Room::set_hhs(HomeHeatingSystem *valor)
{
    hhs = valor;
}

#endif
//-----FIN DEL ARCHIVO-----

```

Room.cpp

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesisista: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: Room

#include "Room.h"
#include "stdlib.h" //Para uso del random()

Room::Room()
    //-----Inicialización de variables
    :bNeedHeat(false),
    bWaterValve(false)
{
    //-----Cuerpo del constructor
}

Room::~Room()
{
    //-----Cuerpo del destructor
}

Room & Room::operator=(const Room &right)
{
    //-----Cuerpo de la operación de asignación
}

int Room::operator==(const Room &right) const
{
    //-----Cuerpo de la operación de igualdad
}

int Room::operator!=(const Room &right) const
{
    //-----Cuerpo de la operación de desigualdad
}

//-----Operaciones definidas por el usuario

Room::Room(int nr)
{
    //-----Cuerpo de la operación: Room
    number = nr;
    bNeedHeat=false;
    bWaterValve=false;
}

void Room::run()
{
    //-----Cuerpo de la operación: run
    boolean bChange = (random(10) >= 5);
    if (!bNeedHeat && bChange)
        needHeat();
    else if (bNeedHeat && bChange)
        stopHeat();
}

void Room::openWaterValve()
{
    //-----Cuerpo de la operación: openWaterValve
    bWaterValve = true;
    hhs->get_operatorInterface()->msgOpenWaterValve(this);
}

void Room::closeWaterValve()
{
    //-----Cuerpo de la operación: closeWaterValve
    bWaterValve = false;
    hhs->get_operatorInterface()->msgCloseWaterValve(this);
}

bool Room::isWaterValveOpen()
{
    //-----Cuerpo de la operación: isWaterValveOpen
    return bWaterValve;
}

```

```

void Room::needHeat()
{
    //-----Cuerpo de la operación: needHeat
    if (!bNeedHeat)
    {
        bNeedHeat = true;
        hhs->get_operatorInterface()->msgNeedHeatInRoom(this);
        hhs->needHeatInRoom(this);
    }
}

void Room::stopHeat()
{
    //-----Cuerpo de la operación: stopHeat
    if (bNeedHeat)
    {
        bNeedHeat = false;
        hhs->get_operatorInterface()->msgStopHeatInRoom(this);
        hhs->stopHeatInRoom(this);
    }
}

//-----FIN DEL ARCHIVO-----

```

Furnance.h

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesis: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: Furnance

#ifndef Furnance_h
#define Furnance_h 1

//HomeHeatingSystem
#include "HomeHeatingSystem.h"
class HomeHeatingSystem;

class Furnance
{
    //Declaraciones públicas
    public:
        //----- Constructores generados por omisión
        Furnance();
        Furnance(const Furnance &right);
        //----- Destructor generado por omisión
        ~Furnance();
        //----- Operación de asignación generado
        Furnance & operator=(const Furnance &right);
        //----- Operaciones de igualdad generadas
        int operator==(const Furnance &right) const;
        int operator!=(const Furnance &right) const;

        //----- Operaciones definidas por el usuario
        void activate();
        void deactivate();
        bool isActive();

        //----- Operaciones Get y Set para atributos

        //----- Operaciones Get y Set para asociaciones

        //Asociación: A1
        //Rol: HomeHeatingSystem::hhs
        const HomeHeatingSystem *get_hhs() const;
        void set_hhs(HomeHeatingSystem *valor);

    //Declaraciones protegidas
    protected:
        //----- Operaciones definidas por el usuario

        //----- Operaciones Get y Set para atributos

        //----- Operaciones Get y Set para asociaciones

    //Declaraciones privadas
    private:
        //----- Operaciones definidas por el usuario

        //----- Operaciones Get y Set para atributos

        //Atributo: bActive
        const bool get_bActive() const;
        void set_bActive(bool valor);
}

```

```

//----- Operaciones Get y Set para asociaciones

private: //Implementación
//----- Atributos establecidos como públicos

//----- Atributos establecidos como protegidos

//----- Atributos establecidos como privados
bool bActive;

//----- Asociaciones establecidas como públicas

//Asociación: A1
//Rol: HomeHeatingSystem::hhs -> Multiplicidad:<No especificada> 1 por omisión (Asociación)
HomeHeatingSystem *hhs;

//----- Asociaciones establecidas como protegidas

//----- Asociaciones establecidas como privadas

};

//----- Métodos Get y Set para atributos (declaraciones inline)

inline const bool Furnance::get_bActive() const
{
    return bActive;
}

inline void Furnance::set_bActive(bool valor)
{
    bActive = valor;
}

//----- Métodos Get y Set para asociaciones (declaraciones inline)

inline const HomeHeatingSystem *Furnance::get_hhs() const
{
    return hhs;
}

inline void Furnance::set_hhs(HomeHeatingSystem *valor)
{
    hhs = valor;
}

#endif
//-----FIN DEL ARCHIVO-----

```

Furnance.cpp

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesista: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: Furnance

#include "Furnance.h"

Furnance::Furnance()
    //-----Inicialización de variables
    :bActive(false)
{
    //-----Cuerpo del constructor
}

Furnance::Furnance(const Furnance &right)
    //-----Inicialización de variables
    :bActive(false)
{
    //-----Cuerpo del constructor de copia
}

Furnance::~Furnance()
{
    //-----Cuerpo del destructor
}

Furnance & Furnance::operator=(const Furnance &right)
{
    //-----Cuerpo de la operación de asignación
}

int Furnance::operator==(const Furnance &right) const
{
    //-----Cuerpo de la operación de igualdad
}

```

```

int Furnance::operator!=(const Furnance &right) const
{
    //-----Cuerpo de la operación de desigualdad
}

//-----Operaciones definidas por el usuario

void Furnance::activate()
{
    //-----Cuerpo de la operación: activate
    bActive = true;
    hhs->get_operatorInterface()->msgActivateFurnance();
}

void Furnance::deactivate()
{
    //-----Cuerpo de la operación: deactivate
    bActive = false;
    hhs->get_operatorInterface()->msgDeactivateFurnance();
}

bool Furnance::isActive()
{
    //-----Cuerpo de la operación: isActive
    return bActive;
}

//-----FIN DEL ARCHIVO-----

```

HeatFlowRegulator.h

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesis: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: HeatFlowRegulator

#ifndef HeatFlowRegulator_h
#define HeatFlowRegulator_h 1

//HomeHeatingSystem
#include "HomeHeatingSystem.h"
class HomeHeatingSystem;
//Furnance
#include "Furnance.h"

class HeatFlowRegulator
{
    //Declaraciones públicas
public:
    //----- Constructores generados por omisión
    HeatFlowRegulator();
    HeatFlowRegulator(const HeatFlowRegulator &right);
    //----- Destructor generado por omisión
    ~HeatFlowRegulator();
    //----- Operación de asignación generado
    HeatFlowRegulator & operator=(const HeatFlowRegulator &right);
    //----- Operaciones de igualdad generadas
    int operator==(const HeatFlowRegulator &right) const;
    int operator!=(const HeatFlowRegulator &right) const;

    //----- Operaciones definidas por el usuario
    void needHeatInRoom(Room *room);
    void stopHeatInRoom(Room *room);

    //----- Operaciones Get y Set para atributos

    //----- Operaciones Get y Set para asociaciones

    //Asociación: A2
    //Rol: HomeHeatingSystem::hhs
    const HomeHeatingSystem *get_hhs() const;
    void set_hhs(HomeHeatingSystem *valor);

    //Asociación: A3
    //Rol: Furnance::furnance
    const Furnance *get_furnance() const;
    void set_furnance(Furnance *valor);

    //Declaraciones protegidas
protected:
    //----- Operaciones definidas por el usuario
    void attachHeatRoom();
    void detachHeatRoom();

    //----- Operaciones Get y Set para atributos

```

```

        //----- Operaciones Get y Set para asociaciones

//Declaraciones privadas
private:
    //----- Operaciones definidas por el usuario

    //----- Operaciones Get y Set para atributos

    //Atributo: nHeatRooms
    const int get_nHeatRooms() const;
    void set_nHeatRooms(int valor);

    //----- Operaciones Get y Set para asociaciones

private: //Implementación
    //----- Atributos establecidos como públicos

    //----- Atributos establecidos como protegidos

    //----- Atributos establecidos como privados
    int nHeatRooms;

    //----- Asociaciones establecidas como públicas

    //Asociación: A2
    //Rol: HomeHeatingSystem::hhs -> Multiplicidad:<No especificada> 1 por omisión (Asociación)
    HomeHeatingSystem *hhs;

    //Asociación: A3
    //Rol: Furnance::furnance -> Multiplicidad:<No especificada> 1 por omisión (Asociación)
    Furnance *furnance;

    //----- Asociaciones establecidas como protegidas

    //----- Asociaciones establecidas como privadas

};

//----- Métodos Get y Set para atributos (declaraciones inline)

inline const int HeatFlowRegulator::get_nHeatRooms() const
{
    return nHeatRooms;
}

inline void HeatFlowRegulator::set_nHeatRooms(int valor)
{
    nHeatRooms = valor;
}

//----- Métodos Get y Set para asociaciones (declaraciones inline)

inline const HomeHeatingSystem *HeatFlowRegulator::get_hhs() const
{
    return hhs;
}

inline void HeatFlowRegulator::set_hhs(HomeHeatingSystem *valor)
{
    hhs = valor;
}

inline const Furnance *HeatFlowRegulator::get_furnance() const
{
    return furnance;
}

inline void HeatFlowRegulator::set_furnance(Furnance *valor)
{
    furnance = valor;
}

#endif
//-----FIN DEL ARCHIVO-----

```

HeatFlowRegulator.cpp

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesista: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: HeatFlowRegulator

#include "HeatFlowRegulator.h"

HeatFlowRegulator::HeatFlowRegulator()

```

```

                //-----Inicialización de variables
                :nHeatRooms(0)
    {
        //-----Cuerpo del constructor
    }

HeatFlowRegulator::HeatFlowRegulator(const HeatFlowRegulator &right)
    //-----Inicialización de variables
    :nHeatRooms(0)
    {
        //-----Cuerpo del constructor de copia
    }

HeatFlowRegulator::~HeatFlowRegulator()
    {
        //-----Cuerpo del destructor
    }

HeatFlowRegulator & HeatFlowRegulator::operator=(const HeatFlowRegulator &right)
    {
        //-----Cuerpo de la operación de asignación
    }

int HeatFlowRegulator::operator==(const HeatFlowRegulator &right) const
    {
        //-----Cuerpo de la operación de igualdad
    }

int HeatFlowRegulator::operator!=(const HeatFlowRegulator &right) const
    {
        //-----Cuerpo de la operación de desigualdad
    }

//-----Operaciones definidas por el usuario

void HeatFlowRegulator::needHeatInRoom(Room *room)
    {
        //-----Cuerpo de la operación: needHeatInRoom
        attachHeatRoom();
        hhs->sendHeatToRoom(room);
    }

void HeatFlowRegulator::stopHeatInRoom(Room *room)
    {
        //-----Cuerpo de la operación: stopHeatInRoom
        hhs->stopHeatToRoom(room);
        detachHeatRoom();
    }

void HeatFlowRegulator::attachHeatRoom()
    {
        //-----Cuerpo de la operación: attachHeatRoom
        if (nHeatRooms == 0)
            furnace->activate();
        nHeatRooms++;
    }

void HeatFlowRegulator::detachHeatRoom()
    {
        //-----Cuerpo de la operación: detachHeatRoom
        nHeatRooms--;
        if (nHeatRooms == 0)
            furnace->deactivate();
    }

//-----FIN DEL ARCHIVO-----

```

OperatorInterface.h

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesista: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: OperatorInterface

#ifndef OperatorInterface_h
#define OperatorInterface_h 1
#include <list.h>
#include "Room.h"
class Room;

class OperatorInterface
{
    //Declaraciones públicas
public:
    //----- Constructores generados por omisión
    OperatorInterface();

```

```

OperatorInterface(const OperatorInterface &right);
//----- Destructor generado por omisión
~OperatorInterface();
//----- Operación de asignación generado
OperatorInterface & operator=(const OperatorInterface &right);
//----- Operaciones de igualdad generadas
int operator==(const OperatorInterface &right) const;
int operator!=(const OperatorInterface &right) const;

//----- Operaciones definidas por el usuario
void displayRoomMap(list<bool> *map);
void msgNeedHeatInRoom(Room *room);
void msgStopHeatInRoom(Room *room);
void msgSendHeatToRoom(Room *room);
void msgStopHeatToRoom(Room *room);
void msgOpenWaterValve(Room *room);
void msgCloseWaterValve(Room *room);
void msgActivateFurnance();
void msgDeactivateFurnance();

//----- Operaciones Get y Set para atributos

//----- Operaciones Get y Set para asociaciones

//Declaraciones protegidas
protected:
//----- Operaciones definidas por el usuario

//----- Operaciones Get y Set para atributos

//----- Operaciones Get y Set para asociaciones

//Declaraciones privadas
private:
//----- Operaciones definidas por el usuario

//----- Operaciones Get y Set para atributos

//----- Operaciones Get y Set para asociaciones

private: //Implementación
//----- Atributos establecidos como públicos

//----- Atributos establecidos como protegidos

//----- Atributos establecidos como privados

//----- Asociaciones establecidas como públicas

//----- Asociaciones establecidas como protegidas

//----- Asociaciones establecidas como privadas

};

//----- Métodos Get y Set para atributos (declaraciones inline)

//----- Métodos Get y Set para asociaciones (declaraciones inline)

#endif
//-----FIN DEL ARCHIVO-----

```

OperatorInterface.cpp

```

//Código generado por la herramienta: UMLGEC++
//Versión 1.0 Proyecto de Tesis para obtener el grado de Ing. en Computación
//Universidad Tecnológica de la Mixteca
//Tesista: Irving Alberto Cruz Matías
//Asesor: M.C.Carlos Alberto Fernandez y Fernandez

//-----
//Archivo del Diagrama: C:\WINNT\tempo\Heating Room System (tesis test)\HeatingSystem.uml
//Generado el Miércoles, 30 de Abril de 2003, 05:30 PM
//Clase: OperatorInterface

#include "OperatorInterface.h"
#include "Unit1.h //La interfaz gráfica

OperatorInterface::OperatorInterface()
{
//-----Cuerpo del constructor
}

OperatorInterface::OperatorInterface(const OperatorInterface &right)
{
//-----Cuerpo del constructor de copia
}

OperatorInterface::~OperatorInterface()
{
//-----Cuerpo del destructor
}

```

```

OperatorInterface & OperatorInterface::operator=(const OperatorInterface &right)
{
    //-----Cuerpo de la operación de asignación
}

int OperatorInterface::operator==(const OperatorInterface &right) const
{
    //-----Cuerpo de la operación de igualdad
}

int OperatorInterface::operator!=(const OperatorInterface &right) const
{
    //-----Cuerpo de la operación de desigualdad
}

//-----Operaciones definidas por el usuario

void OperatorInterface::displayRoomMap(list<bool> *map)
{
    //-----Cuerpo de la operación: displayRoomMap
    String s = "Mapa de cuartos actualmente calentados: "; // "Currently heated room map: ";
    list<bool>::iterator boolIterator; // POR CMIA
    for(boolIterator = map->begin(); boolIterator!=map->end(); boolIterator++)
        if(*boolIterator)
            s += "** ";
        else
            s += ". ";
    Principal->listMsg->Lines->Add(s);
}

void OperatorInterface::msgNeedHeatInRoom(Room *room)
{
    //-----Cuerpo de la operación: msgNeedHeatInRoom
    Principal->listMsg->Lines->Add("El cuarto " + IntToStr(room->get_number()) + " necesita calor");
}

void OperatorInterface::msgStopHeatInRoom(Room *room)
{
    //-----Cuerpo de la operación: msgStopHeatInRoom
    Principal->listMsg->Lines->Add("El cuarto " + IntToStr(room->get_number()) + " no necesita calor");
}

void OperatorInterface::msgSendHeatToRoom(Room *room)
{
    //-----Cuerpo de la operación: msgSendHeatToRoom
    Principal->listMsg->Lines->Add("Mandar calor al cuarto " + IntToStr(room->get_number()));
}

void OperatorInterface::msgStopHeatToRoom(Room *room)
{
    //-----Cuerpo de la operación: msgStopHeatToRoom
    Principal->listMsg->Lines->Add("Detener el calor al cuarto " + IntToStr(room->get_number()));
}

void OperatorInterface::msgOpenWaterValve(Room *room)
{
    //-----Cuerpo de la operación: msgOpenWaterValve
    Principal->listMsg->Lines->Add("Abrir la válvula de agua en el cuarto " + IntToStr(room->get_number()));
}

void OperatorInterface::msgCloseWaterValve(Room *room)
{
    //-----Cuerpo de la operación: msgCloseWaterValve
    Principal->listMsg->Lines->Add("Cerrar la válvula de agua en el cuarto " + IntToStr(room->get_number()));
}

void OperatorInterface::msgActivateFurnance()
{
    //-----Cuerpo de la operación: msgActivateFurnance
    Principal->listMsg->Lines->Add("Activar horno");
}

void OperatorInterface::msgDeactivateFurnance()
{
    //-----Cuerpo de la operación: msgDeactivateFurnance
    Principal->listMsg->Lines->Add("Desactivar horno");
}

//-----FIN DEL ARCHIVO-----

```