Herramientas utilizadas en el proceso de Diseño. Hasta dónde llega su alcance.

Rosana Montes Soldado

Dpto. LSI. Universidad de Granada

Abstract

En el mercado podemos encontrar herramientas que facilitan algún aspecto concreto del ciclo de vida del desarrollo de software. Algunas son herramientas ampliamente utilizadas y conocidas, y otras tiene un uso mas bien local dentro de una empresa. No obstante toda herramienta nos va a permitir desarrollar de forma más facil y comodamente que si lo llevasemos todo en la cabeza o sobre un papel. Nos planteamos pues el alcance de estas herramientas y su facilidad para permanecer en uso en la mayor parte del proceso de desarrolo posible. En concreto veremos la adecuación de la herramienta Rational Rose 2000 de Rational Software Corporation para la fase de Diseño e Implementación, dentro del marco del Lenguaje de Modelado de Software UML.

1 El Proceso Unificado : desarrollo de software

La tendencia actual en el software lleva a la construcción de sistemas más grandes y más complejos. También lo queremos más rapido. Conseguirlo, sin embargo, es difícil. Se necesita un método común, que integre las multiples facetas del desarrollo.

Los autores del Unified Modeling Language (UML) han desarrollado de forma paralela un proceso de desarrollo de software, proporcionando una guía para ordenar las actividades de un equipo, ofreciendo criterios para el control y la medición de los productos que deben desarrollarse.

El Proceso Unificado [1] utiliza el Lenguaje Unificado de Modelado para preparar todos los esquemas de un sistema software. Nosotros utilizaremos aquí Rational Rose para trabajar con UML. Este método creado en 1997 con la colaboración de innumerables compañías (Object Management Group entre ellas) y propiedad de Rational Software Corporation supone un cambio importante con respecto a lo que había por esas fechas.

UML es un lenguaje visual de modelado para construir y documentar sistemas [2]. Consiste en un conjunto de diagramas que son utilizados en diferentes fases de un proyecto software, para modelar diferentes vistas de un sistema (requerimientos, arquitectura, componentes y comportamiento). [3]

Básicamente el proceso se repite a lo largo de una serie de ciclos iterativos que constituyen la vida de un sistema. Cada ciclo se puede ver como una versión del producto para el usuario. Se comienza por conocer lo que los futuros usuarios neesitan y desean: los casos de uso. Este modelo va a relacionarse activamente con el resto de modelos desarrollados:

- Casos de uso - especificado por - Modelo de Análisis
- Casos de uso - realizado por - Modelo de Diseño
- Casos de uso - distribuido por - Modelo de Despliegue
- Casos de uso - implementado por - Modelo de Implementación
- Casos de uso - verificados por - Modelo de Prueba

El modelo contiene instancias de artefactos de UML tales como Actores y Casos de Uso, relacionados por asociaciones y generalizaciones [8]. Tras finalizar el caso de uso del análisis, un diseñador debe crear modelos de clases y diagramas de transición de estados basados en el modelo de casos de uso.

1.1 El producto del desarrollo del software

El producto que se obtiene será para el usuario un conjunto de ejecutables, de código máquina: una descripción binaria entendida por el ordenador. Para el programador el sistema software es un conjunto de código fuente: una descripción que entiende el compilador. Todo depende de quien mire el sistema.

Un sistema son todos los artefactos que se necesitan para representarlo en una forma comprensible por todos los interesados: máquina, trabajadores y clientes.

En este trabajo pretendemos mirar al sistema en su fase de diseño, cuando se utilizan términos como subsistema, clases, diagramas de interacción, diagramas de estados, paquetes, componentes y otros artefactos.

Está organizado de la siguiente manera: En el apartado siguiente comentaremos las metodologías que han dado pie a Proceso Unificado que coinciden con las filosofías defendidas por cada uno de los creadores de UML : Booch - Jacobson - Rumbaugh. En la Sección 3 veremos a groso modo el trabajo que implica el proceso de diseño y la posibilidad de automatizarlo. A continuación conoceremos qué nos permite hacer la herramienta Rational Rose en la fase de diseño. En la Sección 5 comentaremos su adecuación para el diseño y posteriores fases como la implementacion. Finalmente daremos paso a las conclusiones obtenidas y presentaremos líneas de trabajo futuro, en haciendo una propuesta de herramienta de desarrollo.

2 Diseño : con qué metodología.

Aún no podemos decir hasta qué punto el proceso de diseño es suficientemente complejo como para no poder ser formalizado y/o automatizado. Podemos

hablar de metodologías que han tenido mayor repercusión que otras a lo largo de la historia. Esto se puede notar por su uso fuera de los libros. Rational Rose 2000 posiblita ver un modelo desde tres métodos distintos.

2.1 Object Modeling Tecnique (O.M.T.)

El método OMT dado por Rumbaugh, describe un sistema elaborando tres abstracciones o modelos, basándose en el paradigma orientado a objetos. [4]

Modelo de Objetos

Con esta visión se modeliza lo que se está usando en el sistema, lo que tiene. Describe una estructura estática: objetos, clases, metaclases y relaciones. Para descricribir esta estructura se usan diagramas de objetos.

Maneja los siguientes conceptos o abstracciones:

- Herencia (generalización-especialización, extensión-restricción) simple y múltiple.
- Agregación
- Clase abstracta
- Metadatos (datos para describir datos, como lo expresado y almacenado en DDL - Data Description Language - de un sistema de gestión de bases de datos relacional). Las metaclases son metadatos.
- Llaves candidato
- Restricciones

Modelo Dinámico

Se modeliza cuánto cambia el sistema. Describe mediante diagramas de estado los aspectos cambiantes, concernientes al tiempo y a la secuencia de operaciones. Los nodos son estados, los arcos transiciones causadas por eventos. Los eventos son llamadas a operaciones sobre objetos.

Maneja los siguientes conceptos:

- Eventos
- Escenarios
- Condiciones
- Estados
- Operaciones
- Actividad

- Concurrencia, comunicación de eventos, sincronización...

Modelo Funcional

Se modeliza cómo cambia el sistema. Mediante diagramas de flujos de datos se describen las transformaciones de los valores de los datos. Los nodos son procesos y los arcos son flujos de datos.

Maneja los siguientes conceptos:

- Procesos
- Flujos de datos
- Actores o Agentes
- Almacenes de datos
- Flujos de control
- Operaciones
- Restricciones

2.2 Booch

Grady Booch toma cuantro puntos de vista en este método basándose en la estructura lógica, la física, los aspectos estático y dinámico. [5]

- Estructura lógica.

  Diagrama de Clases. Recoge la estructura clasificatoria de los objetos presentes en el sistema. Las relaciones se recogen en forma de flechas, notando uso, herencia o visibilidad.

  Diagrama de Objetos. Muestra los objetos que conforman el sistema.

- Estructura física

  Diagrama de módulos

  Diagrama de procesos

- Estructura dinámica

  Diagramas de transición de estados

- Estructura estática

  Diagramas de temporización

A su vez maneja las siguientes abstracciones:

- Herencia (múltiple)
- Abstracción
- Reusabilidad
- Modularidad
- Extensibilidad
- Interfaces
- Encapsulación
- Generación de código
- Independencia del lenguaje de implementación
- Comunicación entre desarrolladores y usuarios
- Concurrencia

2.3 Objectory y OOSE

Desarrollado por Jacobson en 1992 y propiedad de Object-Oriented Software Enginering. [6]

Distingue las siguientes categorías:

- Arquitectura como conjunto de técnicas elegidas para ser reutilizadas
- Método como los procedimientos explícitos que se siguen
- Proceso

Orienta el ciclo de vida a las etapas de análisis / diseño, implementación y prueba, enfatizando los requerimientos. Las descripciones son diagramales y contemplan a los usuarios.

Clasifica los objetos en:

- Objetos - interfaz: donde tiene particular importancia el almacenamiento.
- Objetos - entidades
- Objetos - control: donde se resalta el comportamiento

Maneja herencia y agregación.

3 Refinamiento del diseño : nos acercamos a la implementación

La capacidad de automatizar un proceso (en este caso el diseño) depende de que tengamos una visión clara de los casos de uso que necesita cada usuario y de qué artefactos necesita manejar. Entiendase artefacto como la solución software a los requisitos de un cliente.

Las herramientas deben ser capaces tanto de soportar la automatización de las actividades repetitivas como de gestionar la información representada por una serie de modelos y artefactos, fomentando y soportando las actividades creativas que en el fondo son el núcleo fundamental del desarrollo.

El proceso de desarrollo debe aprender de las herramientas y las herramientas deben soportar un proceso bien pensado. El problema viene cuando no se dispone de una herramienta que nos guíe en todo el proceso de forma efectiva, o no existe una comunicación entre las distintas herramientas que se han centrado en una fase concreta del proceso de desarrollo.

El proceso y las herramientas vienen en el mismo paquete: las herramientas son esenciales en el proceso y el proceso se ve influido fuertemente por las herramientas. Con poco soporte automatizado, se incrementa el trabajo manual. Todo ello redunda en más trabajo, dificultades en la actualización, mayores tiempos de desarrollo, fisuras en los artefactos, etc.

Podemos hacer un excelente diseño en UML pero si no disponemos de una herramienta que automatice gran parte del proceso hasta la implementación, al final tendremos que trabajar sobre el papel con gran frustración por el tiempo empleado en la especificación de las clases diseñadas.

3.1 Diseño de un Caso de Uso

Los objetivos a groso modo, cuando estamos diseñando un caso de uso son:

1. Identificar las clases del diseño
2. Distribuir el comportamiento del caso de uso entre los subsistemas que intervienen.
3. Definir los requisitos sobre las operaciones, sus interfaces y la implementación en sí.

El punto 1. no es automatizable, pero de él se obtiene un diagrama de clases asociado con la realización. Se utilizará el diagrama de clases para mostrar las dependencias entre los subsistemas y cualquier interfaz que se utilice en la realización del caso de uso.

El punto 2. se puede obtener a partir de diagramas de secuencias, que contienen las instancias de los actores, los objetos del diseño, y las transmisiones de mensajes entre éstos, que participan en el caso de uso. Esta parte si es automatizable y generaría las operaciones necesarias, junto con los argumentos. Esto nos ayudaría a descubrir nuevas clases de diseño.

Por ultimo el punto 3. debe tratarlo el diseñador. Este habrá de incluir los requisitos no funcionales identificados durante el diseño.

3.2 Diseño de una Clase

Nuestro propósito aquí es diseñar una clase que cumpla su papel en las realizaciones de los casos de usos. Esto incluye los siguientes aspectos:

1. Operaciones y atributos.
2. Métodos que realizan las operaciones.
3. Relaciones en las que participa.
4. Correcta realización de una interfaz si esta es requerida.

Como primer paso necesitamos esbozar una o varias clases de diseño: de interfaz, de control o de entidad.

Para cada una de estas, identificaremos las operaciones que dichas clases van a necesitar empleando ya la sintaxis del lenguaje de programación que vayamos a utilizar. Eso implica que habrá que tomar decisiones de visibilidad. Se podría obtener una aproximación de la visibilidad de las operaciones comprobando a qué subsistema pertencen los objetos relacionados por la operación en cuestión. Toda operación sin interrelación con otros objetos u actores sería considerada como privada. Todo esto serían convenciones que el diseñador debería poder modificar en cualquier momento en la herramienta.

El punto 2. se refiere a que un método puede especificar un algorítmo que sea utilizado para realizar una operación. Esta parte normalmente no se realiza en el diseño, sino ya en la implementación, aunque si la herramienta utilizada soporta generación de código integrada para los métodos, como es el caso de Rational Rose, se pueden especificar los métodos directamente en la herramienta.

A la hora de identificar asociaciones y agregaciones partimos de las involucradas en las clases de análisis. De estas podremos refinar la multiplicidad de las asociaciones, modificar distintos roles, refinar la navegabilidad de las asociaciones, pero nada que pueda hacerse de forma automatizada. Esto es trabajo del diseñador. La herramienta en todo caso, puede facilitar la tarea.

3.3 Diseño de un Subsistema

Los objetivos del diseño de un subsistema son:

1. Garantizar que el subsistema es tan independiente como sea posible.
2. Garantizar que el subsistema proporciona las interfaces correctas.
3. Garantizar que el subsistema ofrece una realización correcta de las operaciones tal y como están definidas.

Se debería incluir en el proyecto información de las dependencias de un sistema con respecto a otros. De esta forma si un subsistema es sustituido por otro que posea un diseño interno distinto, no estaríamos obligados a sustituir la interfaz.

Es facil de comprobar por una herramienta que todos los roles definidos son soportados por alguna de las realizaciones de los casos de uso. Tan solo habría que añadir en la especificación de un método, para qué rol colabora.

Para garantizar el tercer punto, la herramienta debería llevar un control sobre las modificaciones de cualquiera de las especificaciones de una clase y de sus operaciones y atributos.

Parece que la parte de diseño de un subsistema, por su caracter de control sobre lo realizado, es más susceptible de ser automatizada.

4 La herramienta : Rational Rose 2000

Los pétalos de UML son la representación textual propuesta por Rational Software Corporation. Se pueden considerar documentos equivalentes a los CDIF (CASE Data Interchange Format) definidos por el estándar ISO/CDIF y utilizados para el intercambio de datos de herramientas CASE. La diferencia está en que los primeros manejan el vocabulario o metamodelo de UML. [7]

Según la propia compañía expresa, su producto ofrece posibilidades completas de modelado utilizando UML. No puedo decir lo mismo, cuando a la hora de hacer una vista del Modelo de Clases no se pueden añadir artefactos tales como Clases Activas, ni calles. Pasa también con otros elementos. En ese sentido la herramienta no va a la par con el lenguaje UML.

Nos centraremos en los aspectos de la herramienta orientados más al diseño y a la implementación, para investigar el problema comentado anteriormente. Estos aspectos son básicamente la generación de código en un lenguaje de programación, la ingeniería inversa y la interfaz con otras herramientas de desarrollo de software: los entornos de programación.

4.1 Especificación de las clases

Los objetos se descubren dentro de la fase de análisis, pero es en el diseño cuando se decide entre diferentes formas de implementarlo, buscando siempre la minimización de los tiempos de ejecución, utilización de memoria, etc.

Rational Rose 2000 ofrece las siguientes posibilidades para la especificación de una clase:

1. Nombre de la clase, estereotipo, nombre de la clase que hereda (en su caso).

2. Control de exportación. Este puede ser: público, privado, protegido e implementación.

3. Operaciones de la clase. Aquí se abre un nuevo apartado para la especificación de la operación.

(a) Nombre de la operación, tipo devuelto, estereotipo y clase a la que pertenece. Argumentos de la operación. Cada argumento puede llevar asociado una especificación del mismo estilo que el de un atributo.

(b) Protocolos empleados, excepciones que pueda lanzar y tipo de concurrencia: secuencial, guardada o síncrona. Método abstracto o no.

(c) Precondiciones, postcondiciones y semántica.

(d) Apartado para la documentación de la operación.

4. Atributos de la clase. Aquí se abre un nuevo apartado para la especificación del atributo.

   (a) Nombre del atributo, tipo devuelto, estereotipo, clase a la que pertenece y valor inicial.

   (b) Estática o derivada.

   (c) Apartado para la documentación del atributo.

5. Documentación. Comentarios a cerca de la propia clase que nos sirvan en la generación del Documento de Diseño.

4.2 Generación de Código

Podemos generar codigo procedente de la informacion contenida en un modelo del Rational Rose. El codigo que se genera para cada componente seleccionado corresponde con la especificación que este tenga, las propiedades de código que se establezcan, así como las propiedades del proyecto en sí. Estas propiedades serán específicas para el lenguaje específico que se elija: C++, Java o Visual Basic.

- C++. En el caso del generador para C++ se utilizan anotaciones (regiones protegidas) empleadas en la actualización del codigo (regeneración no-destructiva). A su vez Rational Rose proporciona una interfaz con el entorno de desarrollo de Microsoft Visual C++.

- Visual Basic. Es posible modelar, generar y utilizar ingenieria inversa con aplicaciones escritas en Microsoft Visual Basic. Incluye herramientas del tipo: Asistente de clases, Asistente de Modelos, Asignacion de componentes, Actualizacion de código y Actualizacion del modelo.

- Java. En este caso el generador también utiliza anotaciones (comentarios) auque no queda muy claro para qué lo emplea.

4.3 Ingeniería Inversa

La ingeniería inversa nos permite cargar modelos Rose a partir de una aplicación escrita en uno de los lenguajes anteriormente citados. Además también procedentes de:

- Corba: Es posible realizar ingenieria inversa de codigo CORBA IDL en elementos de modelado de Rose.

- Oracle8: Rational Rose 2000 permite crear modelos de objetos procedentes de esquemas de bases de datos relacionales. A su vez facilita el descubriminento y la composición de objetos de negocio (business objects). Todo ello implica sacarle un mayor partido al esquema relacional que ya tenemos realizado.

- Objetos COM: Utilizando el Importador de Librerias de Tipo. Todos aquellos componentes COM que cuentan con ficheros TLI de tipos pueden ser directamente importados desde el navegador IExplorer arrastrándolos hasta Rose.

4.4 Interfaz con otras herramientas

Se trata de comprobar los límites entre una herramienta de modelado de sistemas y un entorno de desarrollo. Comentábamos en la Sección 1.1. que el sistema puede ser el modelo o el código fuente, y que todo depende de quién lo mire. Una herramienta ha de dar soporte a todo proceso. No es conveniente pués encontrarnos con "el fin del mundo" de un planeta plano. La transición de una herramienta a otra para continuar con el proceso debe hacerse de la forma más automatizada posible.

Rational Rose 2000 ofrece las siguientes posibilidades:

- Publicador Web. Es posible exportar el modelo en HTML (incluyendo los diagramas) de forma que puedan ser vistos desde un navegador corriente y desde cualquier parte del mundo.

- Rational Rose proporciona una interfaz con el entorno de desarrollo de Microsoft Dev. Studio.

- Repositorio de Microsoft. Permite intercambiar informacion del modelo con aquellas herramientas de modelado que utilicen el Gestor de Componentes Visual de Microsoft - VCM (Visual Component Manager)

- Control de Versiones. Integra sistemas de control de versiones (SCC - Source Code Control) haciendo que los comandos mas frecuentes se encuentren en los menus desplegables del propio Rose.

- Clear Case Add-In. Rational ofrece un sistema de control de versiones denominado Rational Clear Case, que no emplea la API Microsoft SCC comentada anteriormente.

- Integrator de Modelos. Podremos comparar elementos del modelo procedentes de siete ficheros diferentes, ver las diferencias y elegir aquel que queramos añadir al modelo principal.

Tal vez se puede pensar que no es demasiado completo dada la gran variedad de herramientas CASE en el mercado. Se aprecia también una inclinación por las tecnologías de Microsoft. Parece un poco contrario a la filosofía de integración de los atores de UML y el Proceso Unificado.

5 Casos de Estudio : uso de Rational Rose

Hemos visto muy por encima lo que significa y requiere la fase de diseño del ciclo de vida. Algunos de sus procesos pueden ser automatizados y otros no. El caso es que es necesario una herramienta que auque no lo haga todo, sí permita que el diseñador obtenga por su trabajo, un completo y fácil salto hasta la parte de implementación.

Se ha comentado las capacidades de la herramienta Rational Rose 2000 tal y como nos lo presentan sus creadores. Sin embargo no se ha comentado nada de la herramienta tal y como se ve cuando se usa. En un caso de estudio realizado para comprobar el alcance de la herramienta, se ha tomado una aplicación Java como punto de partida. Tras realizar un proceso de ingeniería inversa, se descubre que tan solo se han tomado el número, los nombres de los componentes y las relaciones de uso entre estos. Es necesario crear todos los diagramas.

Sería muy interesante que se obtuviese un Diagrama de Implementación. Estos muestran aspectos de la implementación del sistema, incluyendo la estructura del código fuente y la estructura en tiempo de ejecución (distribución de los ejecutables, por ejemplo), así como el despliegue o configuración que del ejecutable se realiza. Para cada una de las visiones se define un diagrama diferente.

También se realizó un caso de estudio que partiese de una supuesta fase de análisis anterior. Se diseñaron clases y asociaciones y se utilizaron todos los métodos de especificación que el Rational Rose ofrece. Se añadieron todos los comentarios necesarios para enriquecer el Documento de Diseño que esperabamos generar. El resultado no fue del todo satisfactorio tras la generación de código y la generación del informe.

El codigo Java generado contenía tan solo los nombres y tipos de los métodos y atributos indicados en la especificación. Todas las relaciones establecidas en el modelo (salvo herencia) no fueron reflejadas. Mucha de la documentación (precondiciones y postcondiciones de las operaciones, por ejemplo) no fueron incluidas.

El informe generado no dejó de ser un simple listado de los nombres de operaciones y atributos manejados. En todo caso se añadía la visibilidad de cada uno. No se incluyó ni uno solo de los comentarios hechos en los apartados de Documentación de clases, operaciones y atributos.

6 Conclusiones y trabajos futuros

Aunque muchas herramientas CASE aseguran parcialmente una aproximación sistemática al proceso para el cuál se emplea, hay una gran deficiencia en cuanto a la compatibilidad y la comunicación entre diferentes métodos de trabajo y aplicaciones. Tampoco existe una herramienta capáz de acompañar al equipo de desarrollo y al usuario durante todo el el proceso de desarrollo de software. Parece casi inviable la posibilidad de su existencia debido a la no existencia de un método unificado que realmente sea utilizado por todos los desarrolladores, y a la existencia de herramientas que realizan de forma muy adecuada su labor concreta.

En general no hay que confiar en todo lo que dice el propietario de la herramienta empleada, sobre todo en la parte de integración con otras herramientas. Cierto es que cada vez se incluyen más características en estas y que tal vez sea cuestión de tiempo la aparición de la herramienta completa.

En principio y como posible trabajo de futuro, parece mas factible la búsqueda de módulos que conecten distintas herramientas mediante la construcción de una interfaz intermedia, comprensible desde ambos lados.

References

[1] JACOBSON I., BOOCH G., RUMBAUGH J., El Proceso Unificado de Desarrollo de Software, Addison Wesley, 2000.

[2] RUMBAUGH J., JACOBSON I., BOOCH G., Unified Modelling Language, Reference Manual, Addison Wesley, 1997.

[3] GRÜNBACHER P., PARETS J., Capturing and Managing Evolutionary Knowledge in the Software Production Process: A Case Study with WinWin and the UML.

[4] RUMBAUGH J., et Al. Object-Oriented Modeling and Design, Prentice-Hall, 1991.

[5] BOOCH G., Object-Oriented Analysis and Design with Applications, Addison Wesley, 1993.

[6] JACOBSON I. et Al., Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley, 1992.

[7] HERNANDEZ M., FERNANDEZ P., Hacia la automatización en el diseño, Granada 1999.

[8] GRÜNBACHER P., PARETS J., A Framework for the Elicitation, Evolution, and Traceability of System Requirements.