RETOS Y EXPERIENCIAS EN EL APRENDIZAJE DEL MODELADO ORIENTADO A OBJETOS SOBRE LA BASE DEL TRABAJO PRÁCTICO TUTORADO EN EL LABORATORIO

CHALLENGES AND EXPERIENCES IN THE LEARNING OF OBJECT ORIENTED MODELLING ON THE BASIS OF GUIDED PRACTICAL WORK IN THE LABORATORY

Julio Medina

Departamento de Electrónica y Computadores, Universidad de Cantabria
Av. Los Castros s/n, 39005 Santander (España)

Resumen

Esta comunicación presenta las dificultades encontradas y experiencias recogidas en el aprendizaje del modelado orientado a objetos, como parte de cursos avanzados de programación orientada a objetos e ingeniería del software. Estas disciplinas contemplan no solo el uso de clases magistrales, sino el de clases de laboratorio, que por su agilidad requieren una gran interacción docente-alumno. El empleo del laboratorio en esta disciplina tiene en realidad la particularidad de dotar este proceso de aprendizaje de características que asemejan la docencia de ella al trabajo artesanal tutorado más que al de la aplicación directa de las reglas y modos operacionales de las ingenierías. Se convierten así estas sesiones en verdaderas tutorías/taller del arte del modelado conceptual más que la simple supervisión de la aplicación de un guión preconcebido de experiencias de laboratorio. La experiencia indica que siendo la disciplina a aprender aún un dominio inmaduro y difícilmente aceptable como una ingeniería tradicional, el trabajo de laboratorio tiene esa connotación tanto más artesanal que reglada.

Palabras clave: Ingeniería del Software, Orientación a Objetos, Modelado Conceptual, prácticas de Laboratorio, Tutorías.

Abstract

This paper presents the difficulties and experiences found in the teaching/learning process applied for dealing with object oriented modeling techniques as part of advanced courses of object oriented programming and software engineering.. These disciplines used as teching methods not only lectures but also laboratory experiences, which due to their agility required a strong interaction teacher-student. Some features of the use of laboratory experiences in this discipline made its teaching actually similar to the learning/teaching processes followed by the ancient artisans. This is a tutored learning process that differs significantly from the classical rules- and procedural-based education of engineering. Laboratory sessions become actual tutorial guidance proper of an atelier work for the art of conceptual modeling instead of the simple supervision of the application of rules in a preconceived script for the laboratory experiences. The experience obtained indicates that being the discipline to learn an immature domain, hard to accept as a traditional engineering, the laboratory work will expose finally more an artisan nature more than a rule based one.

Keywords: Software engineering, Object orientation, Conceptual modeling, Laboratory practice, Tutorship.

SECCIÓN-1: INTRODUCCIÓN

El dominio del software tiene algunas características que lo hacen particularmente difícil de enseñar y de aprender. Una de ellas habitualmente enunciada es la necesidad manifiesta de contar con una cierta predisposición al razonamiento lógico y abstracto. Pero la verdadera dificultad estriba en que los conceptos que definen la materia desde su origen hasta su aplicación práctica están continuamente redefiniéndose, dificultándose así un proceso de maduración que en otras ramas de la ingeniería se ha alcanzado con cierto éxito.

Así los conceptos que debieran formar la base de una parte del dominio para otra, mutan y obligan a la redefinición de las reglas de composición de la materia de estudio con mayor velocidad de la que es admisible por el sistema educativo e industrial.

1.1 La crisis del software

Desde 1968 se identifica con la denominación "crisis del software" el conjunto de tópicos relacionados con la problemática asociada con el desarrollo de software, cuando aún la ingeniería de software era una quimera. El término expresa las dificultades del desarrollo de software frente al rápido crecimiento de la demanda del mismo, la complejidad de los problemas a ser resueltos y de las soluciones dadas y la inexistencia de técnicas establecidas para el desarrollo

automatizado de sistemas que funcionaran adecuadamente o pudieran ser validados. Los principales aspectos identificados con esta crisis y que siguen siendo válidos aún son:

- Responsiveness: Frecuentemente los sistemas basados en computador no satisfacen las expectativas que tiene el usuario.
- Reliability: Los programas suelen presentar fallos, y su depuración es muy difícil de garantizar.
- Cost: El costo del software es de evaluación difícil, y es habitual que resulte más caro de lo que se preveía al inicio de su desarrollo.
- Modifiability: Los programas son productos muy rígidos y de difícil modificación. El costo de su mantenimiento es muy alto.
- Timeless: El desarrollo del software requiere siempre más tiempo que el previsto.
- Transportability: Cuando se traslada un software de un equipo a otro, siempre se presentan problemas de adaptación.
- Efficiency: Los programas utilizan solo una fracción pequeña de la capacidad del hardware en que se ejecuta.

El espíritu de la crisis del software se puede resumir en la frase: "Construir una aplicación software es una tarea mucho más compleja de lo que de antemano parece". Un pequeño paquete con algún millar de líneas de código es asequible mentalmente a un programador, sin embargo, un paquete mediano o grande con cientos de miles de líneas, desborda la capacidad de cualquier programador. En estos casos se hace crítica la necesidad de pasar del desarrollo artesanal basado en la capacidad de los programadores al desarrollo basado en equipos de profesionales coordinados por una metodología de desarrollo de conocimiento común.

1.2 Las leyes del dominio

El problema principal de este dominio educativo, que difiere de otras ramas de la ingeniería, es el hecho de que las leyes que rigen los elementos del dominio no son leyes naturales de la física, sino que se asientan en definiciones establecidas por la experiencia de trabajos anteriores. Esto puede resultar en un avance significativo cuando los sub-dominios de aplicación son similares, pero puede ser un fracaso total cuando los avances parciales no escalan a la complejidad y requerimientos del problema a enfrentar. Es así que la variación de lenguajes y primitivas que soportan los servicios de los sistemas operativos, dan lugar a redefiniciones completas de los contenidos docentes y de la cualificación necesaria del tejido industrial.

Esto se ve afectado más aún cuando las variaciones se producen por razones estrictamente comerciales y de lucro sobre una posición de dominio de mercado, sin asidero técnico ninguno que les de soporte (diversas versiones del mismo lenguaje etc.).

SECCIÓN-2: ÁMBITO DE ESTUDIO

El trabajo reportado en esta comunicación se centra en el aprendizaje de una de las metodologías que mayor éxito ha tenido en el dominio de los sistemas de mediana y gran complejidad: La conceptualización de aplicaciones basada en objetos. Ante cualquier problema al que se enfrenta la mente humana, lo primero en que se piensa para resolverlo es en "simplificarlo", ello a menudo implica "describirlo", "representarlo" y eventualmente "seccionarlo", luego identificar los trozos para los que se tiene ya alguna solución y "reusarla" y con los que quedan, bien seguir algún algoritmo o método que le sea de aplicación o ensayar hasta dar con uno, o bien "iterar" el proceso hasta encontrar la representación y partición adecuadas para obtener las soluciones.

La clave para que este procedimiento funcione es la capacidad de abstracción. El mecanismo de la abstracción se basa en la eliminación de parte de las peculiaridades del ente o fenómeno bajo estudio, a fin de que lo que se retiene de él sea más fácil de comprender o manejar, pero que a la vez sea lo suficientemente significativo del comportamiento o fenómeno que se quiere estudiar como para representarlo con precisión a los efectos del tema en estudio.

Cuando las abstracciones de todos los elementos en que se ha particionado un problema, se realizan con un tema de estudio común y de manera consistente, la combinación de todas ellas constituye un modelo de ese sistema dentro del dominio de estudio elegido. No existe un modelo único, ni correcto de una determinada situación, sólo modelos adecuados o no a un cierto fin, pues toda abstracción es por definición incompleta (Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, 1991).

El modelado, como herramienta de abstracción y simplificación de la realidad, es pues una técnica extremadamente útil para enfrentarse a la complejidad, al facilitar tanto la comprensión como la resolución de los problemas. Los sistemas de información en general y las aplicaciones software de tiempo real en particular, precisamente por la complejidad que les caracteriza, son un caso paradigmático de esta afirmación.

El término modelo recibe muy diversos significados en función de su entorno de aplicación y de su forma de representación, incluso dentro de las áreas técnicas, así se pueden encontrar modelos matemáticos, mecánicos, electrónicos, modelos de datos, etc. El más próximo al caso que nos ocupa sin embargo, es el modelo conceptual, su ámbito de estudio proviene de las ciencias del conocimiento. A continuación se presenta una traducción de la definición que dan Borgida, Mylopoulos & Wong (1984):

"... tal modelo consiste en un número de estructuras de símbolos y manipuladores de estructuras de símbolos, los cuales, de acuerdo a una digamos que simple filosofía de aceptación de la cualidad creadora e interpretativa de la mente, se supone que corresponden a conceptualizaciones del mundo realizadas por observadores humanos..."

La creación de modelos conceptuales se ve auxiliada por técnicas de abstracción cuyas raíces están descritas también en métodos epistemológicos para la organización del conocimiento y entre ellas destacan la clasificación, la agregación y la generalización (Borgida, Mylopoulos &

Wong, 1984), formas de abstracción que han probado ser útiles en la descripción de modelos complejos, al punto de que actualmente son la base de la tecnología de conceptualización de sistemas orientada a objetos.

Otras estrategias de modelado conceptual basadas en la abstracción (o las mismas pero vistas de manera distinta), han sido identificadas como herramientas para enfrentar la complejidad. Selic, Gullekson & Ward (1994) describen: la recursividad, el modelado incremental y la reutilización. Booch (1994) destaca prominentemente, junto a la abstracción, la descomposición (divide y vencerás) y la jerarquización (como forma avanzada de clasificación y generalización).

Los sistemas informáticos, aún los que se expresan directamente en un lenguaje de programación, son modelos del mundo real o del entorno del problema real en que operan, modelos que están orientados a satisfacer el conjunto de requisitos que le definen. En cuanto mayor sea el nivel de abstracción con el que éstos son expresados, mayores serán las posibilidades de tales modelos de ser útiles a su propósito en el tiempo. Es decir de adaptarse ante los cambios potenciales en la realidad de la que provienen.

La evolución histórica (Booch, 1994) tanto de los lenguajes de programación como de las estrategias de descomposición y conceptualización del entorno real de los sistemas y consecuentemente de las estrategias de análisis y diseño de sistemas, ha ido transformando casi de forma "darwiniana" la actividad del desarrollo de software: de simplemente definir lo que tiene que hacer el ordenador, a encontrar cuales son las abstracciones esenciales del dominio del problema y la forma más fidedigna de modelarlo. Y la estrategia de abstracción y modelado que más éxito ha tenido en el terreno del software en el curso de los últimos 20 años es el desarrollo orientado a objetos

El paradigma de orientación a objetos tiene una larga lista de ventajas sobre su más próximo antecesor, el desarrollo estructurado por descomposición funcional, pero la más clara está en la estabilidad del modelo conceptual del sistema frente a las siempre cambiantes necesidades del mundo que le rodea. Al modelar un sistema, lo importante en la relación que sostiene con su entorno, es decir lo persistente en ella, no es la forma de respuesta esperada del sistema, que es el criterio de abstracción esencial de la aproximación funcional, lo persistente es el sistema en si (Selic, B., 1999). De esta manera la atención del modelador se centra en la estructura del sistema y no en la de su comportamiento, que es tanto más susceptible de cambiar cuanto más complejo sea el sistema. Por su parte, puesto que la estructura del sistema sigue la estructura fundamental del entorno real en que debe operar, sus necesidades de adaptación dependerán de la naturaleza esencial de esa realidad, que se entiende cambia a un paso relativamente menor.

SECCIÓN-3: MÉTODO

A la hora de definir el estilo de aprendizaje más adecuado a este dominio, se vio que si bien para describir los lenguajes de modelado, una aproximación teórica de clases magistrales funciona parcialmente bien, para ejercitar la descomposición en objetos de modo que éstos sean

finalmente estables a lo largo del proceso de desarrollo, la mejor forma es mediante la tutoría en el laboratorio. Las clases prácticas de aula tienen el problema que sitúan al alumno en una posición demasiado cómoda no comprometida con la resolución del problema y por tanto sin poder de fijación de los conceptos. Las clases de laboratorio por el contrario, en las que el alumno debe presentar una primera versión de su propuesta de trabajo (bien en papel o ya en una herramienta gráfica), discutirla con el profesor y luego ingresarla a la herramienta de ayuda al desarrollo, si dejan en el alumno valiosa información que le permite afrontar problemas sucesivos de complejidad incremental. En este tipo de proceso de aprendizaje el profesor debe a su vez defender su posición y emplear para ello los criterios básicos que justifican la descomposición modular escogida; a menudo empleando como razón última la ayuda a la resolución de alguno de los problemas enunciados en la crisis (permanente) del software.

Puesto que las relaciones entre los conceptos que definen la arquitectura de una aplicación como buena o mala no son estrictamente definibles de forma matemática, el alumno debe justificar sus decisiones basandose en criterios de calidad, tales como:

CORRECTITUD (Correctness): Es el factor de calidad externo fundamental. Es un factor muy fácil de definir, pero solo se puede verificar si las especificaciones son exhaustivas.

ROBUSTEZ (Robustness): La respuesta fuera de la especificación no es de interés, pero la aplicación debe dar respuestas correctas cuando se retorna a condiciones especificadas. El concepto de robuztez es difuso, se suele requerir que nunca se alcancen estados catastróficos.

EXTENSIBILIDAD (Extendebility): Aparentemente es consustancial con el concepto de software, pero en los proyectos grandes no es posible estimar los efectos colaterales de los cambios. Las dos bases de la extensibilidad son: La simplicidad del diseño y el diseño descentralizado.

REUSABILIDAD (Reusability): Es un concepto importante en la industria del software. Debería ser fácil de alcanzar ya que son pocas las operaciones básicas de software, sin embargo aún no es así.

COMPATIBILIDAD (Compatibility): Las claves para conseguir la compatibilidad son: la homogeneidad y la estandarización.

EFICIENCIA (Efficency): Es la capacidad de un producto software de hacer un uso completo de los recursos que le proporciona la plataforma sobre la que se ejecuta.

PORTABILIDAD (Portability) : Es la capacidad de un producto software para ser transferido a diferentes plataformas hardware y entornos software.

VERIFICABILIDAD (Verifiability): Es la facilidad que proporciona un software para detectar y trazar fallos durante las fases de validación y operativa.

INTEGRIDAD (Integrity): Es la capacidad que ofrece un producto software para proteger sus recursos (código, datos, ficheros, etc.) contra accesos y modificaciones no autorizadas.

FACILIDAD DE USO (Ease of use): Es la facilidad que presenta el producto software para ser utilizada, ser gestionada, preparar la información de entrada, interpretar los resultados, etc.

SECCIÓN-4: RESULTADOS

Un resultado muy interesante de esta forma de trabajo es que las competencias que se entrenan y se observan finalmente en los alumnos no se asientan exclusivamente en valoraciones técnicas del estilo de las matemáticas o la física, en las que se inducen y evalúan técnicas procedimentales concretas y a menudo son competencias muy específicas del dominio técnico. Con este tipo de técnicas de aprendizaje, las competencias se enmarcan a menudo en el ámbito de las ciencias sociales y otras transversales como el trabajo en grupo y la defensa y racionalización de argumentos. Así pues la razón fundamental para considerar la ingeniería del software como una disciplina de las ciencias sociales es que el sustrato de las decisiones de gestión, de los propios conceptos del dominio se soporta sobre las decisiones de gestión del proceso de desarrollo, teniendo así que este último se hace responsable en gran parte de la validez y calidad técnica del producto final.

SECCIÓN-5: CONCLUSIONES

El trabajo de laboratorio en la modalidad de tutorías, es extremadamente útil para el aprendizaje de conceptos y competencias propios del dominio de la ingeniería del software, del modelado orientado a objetos y de la programación orientada a objetos que se apoya en ambos. Ello es consistente con las tendencias a la adecuación de metodologías docentes a la casuística del alumnado que propugna el Espacio Europeo de Educación Superior (2010), sin embargo ello tiene un coste también significativamente elevado. Para una asignatura cuatrimestral de 6 créditos ECTS es extremadamente difícil llevar a cabo este trabajo con más de 5 grupos (puestos) por grupo de prácticas de laboratorio. En el caso de asignaturas de master que se desarrollan de forma condensada el problema es aún mayor, encontrándose dificultades para ir a más de 4 grupos de alumnos por grupo de prácticas de laboratorio. La ventaja en la captación de los conceptos de los alumnos de master se compensa con el hecho de que los problemas a resolver deben de ser de mayor entidad, así como el nivel de calidad esperado en las soluciones obtenidas.

REFERENCIAS

Booch, G. (1994). *Object Oriented Analysis and Design with Applications*. ISBN 0-8053-5340-2. USA: The Benjamin/Cummings Publishing Company, Inc.

Borgida, A., Mylopoulos, J., & Wong, H.K.T. (1984). Generalization/Specialization as a Basis for Software Specification. En Brodie, M. L., Mylopoulos, J., & Schmidt, J. W. (Eds), *On Conceptual Modeling*, New York: Springer-Verlag Inc.

Espacio Europeo de Educación Superior. (2010). Descargado el 1 de mayo de 2012 de http://www.eees.es

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-Oriented Modeling and Design*. ISBN 0-13-630054-5. USA: Prentice Hall, Inc.

Selic, B. (1999). Turning Clockwise: Using UML in the Real-Time Domain. *Communications of the ACM*, *42*(10).

Selic, B., Gullekson, G. & Ward, P.T. (1994). *Real-time Object Oriented Modeling*. ISBN 0-471-59917-4. USA: John Wiley & Sons, Inc.