Gestión de la evolución del software. El eterno problema de los *legacy systems*

Alfredo Rodríguez

Antonio Márquez

Miguel Toro

Departamento de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla
Email:amarser@teleline.es

Abstract.

La mayor parte de los grandes sistemas de información que están hoy funcionando en las empresas del país fueron desarrollados en los años ochenta. La irrupción de las tecnologías relacionadas con Internet, el paradigma de objetos, los componentes distribuidos y la nueva mentalidad empresarial que intenta ofrecer mejores servicios a sus clientes y durante más tiempo, han provocado que la información que permanecía en los viejos sistemas y que es totalmente aprovechable, sea objeto de diversos tratamiento para su recuperación. En este documento se presenta un razonamiento de las soluciones para intentar acabar con los *legacy systems* desde una perspectiva general y desde nuestra particular visión del problema.

Introducción

La situación de la ingeniería del software en los años 80 sólo permitía una arquitectura física y lógica restringida a lo que ofrecía los grandes fabricantes de software y hardware (sobre todo IBM, que suministraba ambos componentes consiguiendo así una dependencia total del cliente). Estos sistemas son sometidos a un mantenimiento estresante y, normalmente, indocumentado, que desemboca en una degradación de la aplicación y, por ende, en un servicio deficiente para el usuario.
Las necesidades de evolución y adaptación a los nuevos requerimientos tecnológicos y de negocio empujan al sistema a una nueva situación a la que no es posible llegar a través del mantenimiento clásico. En una primera aproximación, la solución a estos problemas se puede presentar a través de dos caminos: un nuevo desarrollo que incorpore nuevas tecnologías y funcionalidades o por medio de la aplicación de reingeniería al *legacy system*.
Actualmente, los presupuestos de las empresas no permiten desarrollos de gran envergadura en costo y tiempo. Los *legacy systems* constituyen una fuente valiosa de conocimiento del sistema a partir de los cuales, y una vez analizados, se puede decidir el camino a tomar para actualizar el sistema: reingeniería, abandono o una solución híbrida.

Qué

Ante una situación como la descrita anteriormente, las organizaciones tienen que tomar una determinación que se resume en un cambio del sistema para adecuarlo a las nuevas necesidades. Este cambio puede concretarse en una de las opciones siguientes:
- Reingeniar el sistema
- Abandonar el sistema y sustituirlo por otro nuevo
- Optar por una solución híbrida entre las dos anteriores

Como denominador común de todas las opciones anteriores, la organización debe plantearse incluir dentro del proyecto correspondiente una buena gestión de la evolución del software que se produzca, para no volver a caer en la misma situación actual.

Reingeniar

La definición dada por el Reengineering Center del Software Engineering Institute de la Universidad Carnegie Mellon es

> *Reingeniería es la transformación sistemática de un sistema existente a una nueva forma para realizar mejoras de la calidad en operación, capacidad del sistema, funcionalidad, rendimiento o capacidad de evolución a bajo coste, con un plan de desarrollo corto y con bajo riesgo para el cliente.*

En esta definición se enfatiza que el hecho de que la reingeniería es la mejora de sistemas existentes de modo que la inversión resulte muy rentable y que, de todas formas, dicha mejora podría ser obtenida a través de un nuevo desarrollo. Si la reingeniería no tiene un coste bajo, no está acabada en poco tiempo, no tiene poco riesgo o no ofrece un valor añadido para el cliente, hay que considerar la posibilidad de un nuevo desarrollo.
Los criterios para su aplicación se basan tanto en técnicas heurísticas como la edad del código o el coste del personal de mantenimiento, como en modelos de coste más sofisticados. Si hubiese que reingeniar varios *legacy systems*, habría que recopilar sus similitudes dentro de una misma arquitectura y tratarlos como si fueran una familia de sistemas y no aplicar soluciones aisladas a cada uno de ellos.

Abandonar

Un sistema se abandona [7] cuando no es capaz de seguir el ritmo de las necesidades del negocio y su reingeniería no es posible o no se puede hacer con un coste efectivo. Normalmente, estos sistemas adolecen de documentación, de actualización y de capacidad de expansión.
El abandono de un sistema comporta riesgos:

- Es una actividad que consume muchos recursos y que implica a los mantenedores actuales del sistema. Estos recursos estarán ocupados en el mantenimiento del sistema y, seguramente, no conocerán las nuevas tecnologías que van a ser aplicadas en el desarrollo del nuevo sistema.
- Abandonar el sistema no significa empezar un desarrollo desde cero. No se puede pensar que los datos que contiene el legacy van a desecharse y que los valores que conservan los expertos del sistema van a perderse. El abandono del sistema debe venir precedido por una análisis exhaustivo que permitan decidir el método que se va a utilizar para la migración de la información del sistema actual al nuevo, incluyendo datos, funciones de negocio y arquitectura del sistema.

Sobre las técnicas utilizadas para la resolución de problemas ligados a la migración se puede consultar en [8, 21].

Gestionar la evolución

Durante el proceso de cambio del sistema hay que prever cuál va a ser la gestión de su evolución posterior, sobre todo para que la situación que se vive ahora no vuelva a repetirse o, por lo menos, sea menos traumática. La gestión de la evolución debe consistir en dar una respuesta rápida, preparada y eficiente a los cambios que se produzcan en el entorno ya sean estos de índole tecnológico o de gestión del propio negocio.

Cómo

La evolución de un sistema es un concepto amplio; abarca desde una simple modificación para corregir un “bug” de un programa hasta una reimplantación completa del sistema. Las actividades de evolución que se pueden realizar sobre un sistema son de tres tipos: mantenimiento, reingeniería y abandono. La figura 1 ilustra cómo son aplicadas cada una de las actividades de evolución durante el ciclo de vida de un sistema.

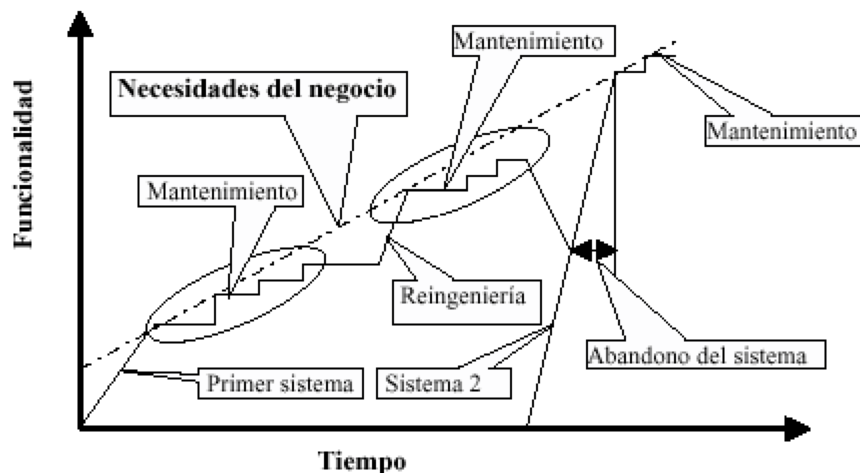

Fig. 1. Ciclo de vida de un Sistema de Información

Mantenimiento

Históricamente, el mantenimiento se puede considerar como la primera forma de evolución de los sistemas. La mayoría de las empresas que realizaron sus grandes desarrollo hace 25 años, necesitan, sobre todo si la aplicación sigue resultando efectiva, la realización de estas actividades.

El ANSI/IEE da la siguiente definición de mantenimiento:

> *Las modificaciones de los productos software después de su entrega para corregir fallos, mejorar rendimiento u otros atributos o adaptar el producto a un cambio de entorno.*

Una definición similar es dada por ISO/IEC:

> *Un producto software soporta una modificación en el código y su documentación asociada para la solución de un problema o por la necesidad de una mejora. Su objetivo es mejorar el software existente manteniendo su integridad.*

El mantenimiento representa el porcentaje más alto del coste de todo el ciclo de vida de un sistema [16]. Lientz y Swanson proponen dividir en cuatro los tipos de mantenimiento:

- Adaptativo para adaptar el software a un cambio de entorno
- Perfectivo para suplir nuevos requerimientos
- Correctivo para solucionar errores puntuales de programas
- Preventivo para prever la aparición de problemas

Según el estudio de Lientz y Swanson, los dos primeros consumen el 75% de los recursos y el correctivo el 21%. Otros estudios proporcionan resultados similares. Todos muestran que, es la incorporación de nuevos requerimientos de usuario la causa principal de la evolución y el mantenimiento del software.

Si los cambios se pudieran anticipar en el diseño, se podrían prever en forma de parametrizaciones. El problema fundamental es que, el elevado número de cambios que el software soporta a lo largo de su ciclo de vida, hace imposible una previsión en el diseño del sistema. Por tanto, el mantenimiento debe tener una consideración especial porque:

- Consume una gran parte del coste total del ciclo de vida del software
- Imposibilita un cambio rápido y fiable en el software haciendo perder oportunidades de negocio
- Dificulta la adopción de nuevas tecnologías
- Ocasiona daños en la integridad del sistema

Por ello, es previsible que la investigación en el campo del mantenimiento del software vaya en aumento en los próximos años, complementada por la reingeniería de sistemas.

Aunque algunos autores enfatizan el ciclo de mantenimiento hasta el punto de asimilarlo a la evolución de un sistema (*staged model* de Bennett [3]), en la comunidad del software se admite que el mantenimiento es sólo una estrategia de microevolución.

Reingeniería o modernización

Posterior a las técnicas de microevolución del mantenimiento, que tienden a moverse a los niveles de abstracción más bajos del sistema, surge la reingeniería para cubrir la necesidad de una evolución más amplia del sistema. La reingeniería trabaja todos los niveles de abstracción (desde la implementación hasta el diseño) para conseguir realizar cambios más drásticos preservando los valores de negocio del sistema [2].
La reingeniería de sistemas puede clasificarse según los niveles de conocimientos requeridos para llevar a cabo el proyecto [26]. La reingeniería que requiere conocimientos a bajos niveles de abstracción (código fuente) se llama *ingeniería inversa* o *modernización de caja blanca* y aquella que sólo requiere el conocimiento de las interfaces del sistema se llama *reingeniería* propiamente dicha o *modernización de caja negra*.

Ingeniería inversa o modernización de caja blanca

Inicialmente, los problemas de evolución aparecen en sistemas complejos con un número elevado de líneas de código (normalmente COBOL) y escasa o nula documentación. Ante la necesidad de recuperación del diseño, la arquitectura y la funcionalidad del sistema desde el código para modernizar el lenguaje de programación o el soporte de los datos, surge la ingeniería inversa. Actualmente conocida como *modernización de caja blanca (White-Box Modernization)* [12], se fundamenta en la identificación de los componentes del sistema y sus relaciones para conseguir una representación a niveles mayores de abstracción.
Chikofsky y Cross [10] la definen en su taxonomía como *el análisis de un sistema para identificar sus componentes actuales y las dependencias que existen entre ellos, para extraer y crear abstracciones de dicho sistema e información de su diseño.* Esta definición implica, por tanto, la creación de una representación a un nivel más alto del examinado (diseño desde implementación).
Una vez que el código es analizado y conocida su funcionalidad puede realizarse la reestructuración del sistema o del código. Chikofsky y Cross [10] definen la *reestructuración* como *la transformación desde una forma de representación a otra en el mismo nivel de abstracción, preservando las características externas del sistema (funcionalidad y semántica).*
La *redocumentación* es también una forma de ingeniería inversa. Es el proceso mediante el que se produce documentación retroactivamente desde un sistema existente [24]. Si la redocumentación toma la forma de modificación de comentarios en el código fuentes, puede ser considerada una forma suave de reestructuración. Sin embargo, puede ser considerada como una subárea de la ingeniería inversa porque la documentación reconstruida es usada para ayudar al conocimiento del programa. Se piensar en ella como una transformación desde el código fuente a pseudocódigo y/o prosa, esta última considerada como más alto nivel de abstracción que la primera.
Aunque la aparición de multitud de herramientas facilitan las labores de la ingeniería inversa, es la labor humana (*humanware*) la decisiva a la hora de completar el estudio del sistema.

Reingeniería o modernización de caja negra

Con el paso del tiempo, se producen tres cambios en el desarrollo de sistemas que afectan a los aspectos de su evolución:

- La importancia del reflejo de la arquitectura en la documentación del sistema
- La aparición del paradigma de objetos
- Los componentes distribuidos

Aunque en un principio se pensó que el paradigma de objetos sería la solución al problema de los *legacy*, se vio pronto que los problemas se agravaron al adaptar sistemas no concebidos para este paradigma a lenguajes como el C++ haciendo mal uso de mecanismos como encapsulación y, sobre todo, de la herencia. Al fin y al cabo, estamos ante el mismo problema de siempre: el paradigma puede tener beneficios si al desarrollarlo se aplican técnicas de ingeniería. En caso contrario, vuelven a aparecer de nuevo los problemas del mantenimiento del sistema.
En el caso de desarrollos orientados a objeto, la falta de experiencia del personal, el empleo de varios lenguajes para el desarrollo de un mismo sistema y la no incorporación de nuevas

tecnologías (UML, CORBA), proporcionan el caldo de cultivo para la construcción de sistemas con las mismas deficiencias que los actuales. Tenemos ante nosotros un nuevo tipo de *legacy system* que ya es necesario someter a técnicas de reingeniería para su mantenimiento y evolución.
FAMOOS [13], es un proyecto que recoge las premisas y particularidades de la tecnología para la reingeniería orientada a objeto. En él se proponen como metas para esta reingeniería la descomposición del sistema en subsistemas, el aumento del rendimiento, hacer el sistema más portable, extraer su diseño e incorporar tecnologías como UML y CORBA. Como en los *legacy* tradicionales, considera que los problemas a la hora de obtener la arquitectura del sistema, se producen por una documentación insuficiente, falta de modularidad con un alto grado de acoplamiento entre clases y funcionalidad duplicada implementada con diferentes implementaciones. La falta de experiencia en la construcción de programas produce una nula o mala utilización de la herencia, operaciones que se definen fuera de la clase correspondiente, violación de la encapsulación y clases mal utilizadas (escribir C++ con estilo C). Para que reingeniería orientada a objeto pueda ser aplicada ha de cumplir unos requisitos: diseño independiente del lenguaje de implementación, desarrollo escalable y herramientas que soporten las técnicas aplicadas.
La reingeniería se ejecuta en seis pasos:

- Análisis de requerimientos que identifiquen las metas perseguidas por la reingeniería
- Captura del modelo por medio del conocimiento del legacy system y su documentación utilizando patrones de ingeniería inversa
- Detección del problema con la ayuda de herramientas de inspección, métrica y software de visualización
- Análisis del problema seleccionando la estructura que puede resolverlo
- Reorganización seleccionando la transformación óptima para el legacy
- Propagación del cambio con metodología de reingeniería

Para reingeniar estos sistemas se emplean técnicas *métricas*, de *visualización de programas*, de *abstracción* y *refactoría* del código. Tanto para reingeniería como para ingeniería inversa, se crean patrones para la resolución de problemas relacionados con estas técnicas [7].
La aparición de técnicas de modelado arquitectónico orientada a objeto (OMT [20]) y su documentación asociada (UML) facilitaron el desarrollo de técnicas de diseño con el paradigma de objetos.
En base al conocimiento del sistema, los datos, las funcionalidades y las interfases, se desarrollan nuevas técnicas de reingeniería no basadas en el conocimiento del código sino en el examen del comportamiento de las entradas y salidas del sistema [13, 18], desarrollando nuevos patrones de reingeniería y sentando las bases de la reingeniería basada en *wrapping*.
Idealmente, *wrapping* [26] es una reingeniería en las que sólo se analizan las interfases (las entradas y salidas) del *legacy* ignorando los detalles internos. Constituye la *reingeniería de caja negra (Black-Box Reengineering)*. Esta solución no es aplicable siempre y a veces requiere el concurso de la ingeniería inversa para el conocimiento interno del sistema.
La reingeniería basada en *wrapping* puede realizarse a nivel funcional, de datos o de interfase. En cada una de ellas se emplean técnicas que se describen a continuación.

Wrapping de interfases de usuario
La interfase de usuario es la parte más visible del sistema y su modernización implica, normalmente, mayor facilidad de operación para el usuario.
Una de las técnicas utilizadas es la de *screen scaping* [9] que consiste en envolver interfases basadas en texto con un entorno gráfico basado en GUI o en HTML. No resuelven en absoluto el problema del legacy

Wrapping de datos
El *wrapping* de datos permite acceder a los datos del *legacy* usando una interfase diferente de la diseñada inicialmente. Se pueden emplear las siguientes técnicas:

- *Gateway* de base de datos: emplea interfases propietarias en la base de datos del *legacy* e interfases estándares (ODBC,JDBC) como puentes hasta los nuevos lenguajes de implantación
- Integración con XML aprovechando los mecanismos de integración de intercambio de datos promocionado por la integración de aplicaciones del B2B.
- Replicación de la base de datos utilizada para descentralizar el almacenamiento masivo de los *mainframes* de modo que instancias locales de bases de datos modernas [17] replican partes de la base de datos centralizada.

- Capa de persistencia construida específicamente para cada sistema [1, 19]. Permite el acceso de aplicaciones desarrolladas con tecnología orientada a objeto a datos organizados de cualquier forma (bases de datos relacionales, archivos VSAM, etc.)

***Wrapping* funcional**
El *wrapping* funcional encapsula datos y funcionalidades del *legacy*. Se emplean una de las técnicas siguientes:
- Integración con CGI que permite el acceso al *legacy* usando *Common Gateway Interface* mediante servidores Web o HTTP. La integración con CGI [22] proporciona un medio de acceso a *legacy systems* implantados en mainframes con monitores de teleproceso. Está constituido por una interfase gráfica que sustituye la antigua interfase de usuario (como la técnica *screen scraping*) y tiene acceso directo con la funcionalidad y los datos del sistema.
- Object Oriented Wrapping (OOW). La tecnología de objetos distribuidos (DOT) es la combinación de la orientación a objeto con la distribución (middelware objeto). Su mejor representante es CORBA de OMG [25], plataforma que combina un lenguaje de especificación de orientado a objeto, métodos de llamada remota robustos, protocolos de interoperabilidad y otros servicios. El modelo conceptual del wrapping orientado a objeto es muy simple: las aplicaciones, los datos de negocio y los servicios son representados como objetos. Esta técnica está lejos de ser sencilla. Dos son las dificultades técnicas más relevantes: la traslación de la semántica monolítica de los legacy a un sistema orientado a objeto (para ello es imprescindible un buen conocimiento del dominio de la aplicación) y la integración de los servicios de infraestructura (seguridad, transacciones y persistencia).
- Wrapping orientado a componentes. Utiliza el modelo de componentes distribuidos [11]. Se considera la aproximación más prometedora. Los estándares del mercado de componentes más utilizados son DNA de Microsoft, CORBA3 y Enterprise JavaBeans de Sun Microsystem. Este último, por las implicaciones de Sun, merece mención especial. Con EJB, el wrapping se lleva a cabo en tres pasos: separar la interfase del legacy en módulos agrupados por unidades lógicas, construir un punto de contacto único mediante el que se establece una comunicación entre el EJB y una función concreta del legacy implementada mediante un bean o un agente de servicio externo y proporcionar una guía para una sustitución de las funciones del sistema incremental y no traumática.

Abandono

Para decidir el proceso de abandono del sistema es necesario descubrir el núcleo del sistema (funcionalidades y datos) utilizando análisis para averiguar el nivel de abstracción en el que hay que moverse y que técnica es más adecuada.
A esta búsqueda de recursos valiosos del *legacy* se le denomina *mining legacy system assets* [4], utilizando el término *adapting legacy system assets* para los niveles más altos de abstracción.
En un sistema puede aplicarse esta técnica si tiene disponibles un mínimo de recursos entre los que deben estar las descripciones de su arquitectura, modelos de dominio, documentación del diseño, programas de prueba, datos de prueba y su documentación, especificaciones de la interfase, herramientas, código y procesos. Es importante conocer los *tradeoff* de la arquitectura y del diseño, además de las restricciones de ingeniería y del conocimiento del dominio de aplicación.
En general, son necesarios cuatro pasos para minar los recursos de un sistema:
- Recogida preliminar de información
- Tomar decisiones con respecto a los recursos a minar y las estrategias básicas a utilizar y las opciones técnicas a aplicar
  - Realizar un análisis técnico detallado de los componentes del sistema y sus relaciones e interfaces
  - Llevar a cabo la rehabilitación de los recursos seleccionados

Dentro de estos cuatro pasos, la tarea más complicada corresponde a decidir las estrategias y las opciones técnicas que se van a aplicar a los recursos seleccionados. Option Analysis for Reengineering (OAR) [6, 15] proporciona ayuda para tomar dos decisiones fundamentales:
- Determinar el nivel de análisis adecuado al problema en los niveles de código, funcional y de arquitectura.
- Determinar la estrategia de reingeniería a aplicar

OAR utiliza el modelo de herradura (*horseshoe*) (figura 2) [28, 5] para establecer el contexto donde tomar estas decisiones. En el modelo se representan tres niveles de abstracción:

- Representación código-estructura que contiene código fuente y artefactos como árboles de sintaxis y gráficos de ejecución. Dentro de este nivel existen dos subniveles: cadenas de texto y estructuras de código.
- Representación a nivel de función que describe las relaciones entre programas (llamadas entre ellos), datos (relación entre funciones y datos) y archivos (relaciones agrupadas de funciones y datos).
- Representación a nivel de concepto que agrupa a los artefactos de los niveles de código y funciones en subsistemas.

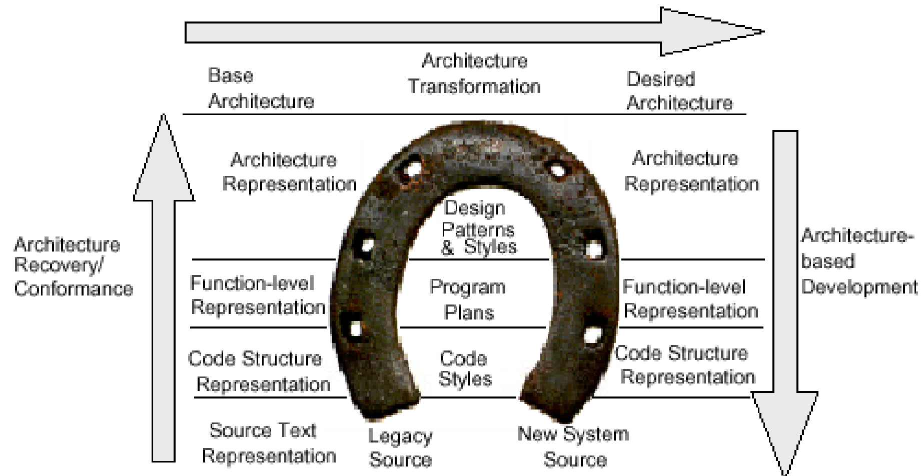

Fig. 2. Modelo de herradura

Gestionar la evolución

El planteamiento de la evolución de los sistemas debe hacerse tanto para sistemas reingeniados como los nuevos desarrollos. Conceptualmente se debe preservar la estructura del sistema incorporando los cambios del entorno.
Para ello hay que establecer un marco de trabajo disciplinado capaz de incorporar los cambios en el sistema en el momento más adecuado preservando su arquitectura y asegurando la transmisión continua de conocimiento. Esto requiere un fuerte compromiso entre la ingeniería del software y la ingeniería de negocio de la empresa y, posiblemente, un cambio en el entorno académico para asumir la inclusión en los currículos de los nuevos paradigmas.

Cuándo y cuánto

El peso de lo económico prevalece sobre el resto de los factores a la hora de elegir un camino hacia la evolución.
El abandono de los *legacy* es una práctica cada vez habitual en favor de la asunción de *products line*. Sin embargo, desde el punto de vista de la evolución no es un trabajo fácil ya que requiere, al menos, técnicas de ingeniería inversa, tal como ha demostrado el hecho de que el SEI haya retomado el *horseshoe model* para realizar trabajos de migración de sistemas. Si este trabajo no se lleva a cabo correctamente, se perderá el conocimiento del sistema y con ello, la posibilidad de vuelta atrás.

Conclusiones y futuros trabajos

En nuestra opinión, la experiencia demuestra que, posiblemente los *legacy* tengan una larga vida por delante a causa de una mala gestión de la evolución continua de los sistemas.
Las tendencias actuales de adquisiciones de productos externos entrañan altos riesgos. Bajo la apariencia atractiva para los gestores de la empresa que encuentran el modo de bajar costos y

eludir responsabilidades, se esconde la pérdida del conocimiento del sistema, de la capacidad de gestión y la inmediatez de la respuesta ante nuevos usos y estrategias de mercado.
Por lo tanto, reivindicamos un nuevo enfoque en la Ingeniería del Software que incluya en los sistemas los mecanismos necesarios para garantizar su continua evolución así como un cambio en los diseños curriculares de las Facultades de Informática para incluir la formación de este escenario. Los puntos siguientes concretan los cambios que, a nuestro parecer, tendrían que formar parte del currículo del ingeniero de sistemas:

- Identificar el rol de reingeniero como parte del currículo del ingeniero de software.
- Creación de la especialidad de reingeniería dentro las Escuelas de Ingeniería del Software.
- Incluir en el currículo de la especialización las materias que permitan desarrollar las habilidades siguientes:
    - Conocimiento de métodos de análisis y diseño, arquitectura, lenguajes de programación y sistemas de archivo utilizados por los sistemas candidatos a ser recuperados.
    - Fundamentos y prácticas del mantenimiento de sistemas viejos.
    - Migraciones de datos desde legacy systems.
    - Adaptación de la Ingeniería del Software para la evolución correcta y continua de sistemas actualmente en desarrollo, de modo que su futuro mantenimiento sea más fácil y barato.
    - Aplicación de las tecnologías de ingeniería de software a la reingeniería.
    - Conocimiento y utilización de técnicas y herramientas de ingeniería inversa tanto de código como de datos con especial incidencia en el conocimiento continuo del programa.
    - Sistemas en Evolución Continua.
- Estar preparado para asimilar e incluir los nuevos cambios tecnológicos en cuanto a su incidencia en las técnicas de reingeniería.
- Reingeniar los sistemas actuales de gestión de la comunidad universitaria de modo que reflejen los nuevos cambios tecnológicos.
- Proporcionar al estudiante las bases para que pueda aplicar estas habilidades a futuros cambios tecnológicos.
- Establecer la credencial correspondiente de cara, sobre todo, a la salida profesional del graduado.

Referencias

[1] Ambler, S. *The Design of a Robust Persistence Layer For Relational Databases*. 1999. http://www.ambysoft.com/persistenceLayer.pdf

[2] Arnold, R.S.; *Software Reengineering*, IEEE Computer Society Press, 1993.

[3] Bennett, K.H., Rajlich, V.T., A new perspective on software evolution: the staged model, IEEE, 1999.

[4] Bergey J., O'Brien L., Smith D., *Mining Existing Assets for Software Product Lines.* (CMU/SEI-2000-TN-008). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, Mayo 2000.

[5] Bergey J., Smith D., Weiderman N., *DoD Legacy System Migration Guidelines.* (CMU/SEI-99-TN-013). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, Septiembre, 1999.

[6] Bergey, J.; Smith, D.; Weiderman, N.; & Woods, S.N. *Options Analysis for Reengineering (OAR): Issues and Conceptual Approach.*(CMU/SEI-99-TN-014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, Octubre, 1999.

[7] Bisdal, J., Lawless, D., Wu, B., Grimson, J., Wade, V., Richardson, R., O'Sullivan, D. *An Overview of Legacy Information System Migration*, Proceedings of the 4 th Asian-Pacific Software Engineering and International Computer Science Conference (APSEC 97, ICSC 97), 1997.

[8] Brodie, M. and Stonebraker, M. *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*. Morgan Kaufmann Publishers, 1995.

[9] Carr, D. *Web-Enabling Legacy Data When Resources Are Tight*. Internet World (August 10 1998).

[10] Chikofsky E. y Cross J., *Reverse engineering and design recovery: A taxonomy*. IEEE Software, 7(1):13{17, January 1990.

[11] Comella-Dorda S., Lewis G., Place P., Plakosh D., Seacord R., *Incremental Modernization of Legacy Systems*. 2001. (CMU/SEI-2001-TN-006). Pittsburgh, Pa.: SEI, Carnegie Mellon University.

[12] Comella-Dorda S., Wallnau K., Seacord R., Robert J., *A Survey of Legacy System Modernization Approaches*, Software Engineering Institute, Carnegie Mellon University, 2000, (CMU/SEI-2000-TN-003)

[13] ESPRIT program project. *The FAMOOS Object-Oriented Reengineering Handbook*, 1999

[14] Gamma et al, *Design Pattern*, Addison Weley, 1995

[15] Kazman R., O'Brien L., Verhoef C., *Architecture Reconstruction Guidelines*. (CMU/SEI-2001-TR-026), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, Agosto 2001

[16] Lientz B. P., Swanson E. B. *Software Maintenance Management*. Addison Wesley, Reading, MA, 1980.

[17] Oracle Corp. *Oracle 8I Concepts*. http://technet.oracle.com/doc/server.805/a58227/ch_repli.htm.
[18] Pooly R., *Software Reengineering Pattern*, 1998
[19] Rodríguez, A.A., Márquez, A. *Persistencia*. Trabajo del curso Diseño de Software basado en métodos formales. Mayo 2000. LSI. Facultad de informática. Sevilla.
[20] Rumbaugh J., Blaha M.; Premerlani W.; Eddy F., Lorensen W., *Modelado y diseño orientado a objetos. Metodologóa OMT*. Prentice Hall 1998
[21] Seng, J., and Tsai, W. *A Structure Transformation Approach for Legacy Information Systems- A Cash Receipts/Reimbursement Example*, Proceedings on the 32º Hawaii International Conference on System Sciences, 1999.
[22] Shklar, L. *Web Access to Legacy Data*. http://athos.rutgers.edu/~shklar/web-legacy/summary.html.
[23] Tilley S., Storey M., *Report of the STEP '97 Workshop on Net-Centric Computing*. 1997. (CMU/SEI-97-SR-016). Pittsburgh, Pa.: SEI, Carnegie Mellon University.
[24] Tilley, Scott R. & Smith, Dennis B. *Perspectives on Legacy Systems Reengineering (draft)*. Reengineering Center, Software Engineering Institute, Carnegie Mellon University. 1995.
[25] Wallnau K., Weiderman N., Northrop L., *Distributed Object Technology With CORBA and Java: Key Concepts and Implications*, Software Engineering Institute, Carnegie Mellon University, 1997, (CMU/SEI-97-TR-004).
[26] Weiderman N., Bergey J., Smith D., Tilley S., *Approaches to Legacy System Evolution*, Software Engineering Institute, Carnegie Mellon University, 1997, (CMU/SEI-97-TR-014)
[27] Weiderman, N; Northrop, L.; Smith, D.; Tilley, S.; & Wallnau, K., *Implications of Distributed Object Technology for Reengineering* ,(CMU/SEI-97-TR-005 ADA326945). Pittsburgh, Pa.: SEI, Carnegie Mellon University.
[28] Woods, S., Carriere, S.J., Kazman, R. *A Semantic Foundation for Architectural Reengineering and Interchange*, 391-398. Proceedings of the International Conference on Software Maintenance (ICSM-99). Oxford, England, August 30-September 3, 1999. Los Alamitos, Ca.: IEEE Computer Society, 1999.