Diseño y Desarrollo de Software

Licenciatura en Ciencias de la Computación

Módulo 4: Evolución del software

Mantenimiento, reutilización, reingeniería, sistemas legados

Profesora titular de la cátedra: Marcela Capobianco
Profesores interinos: Sebastián Gottifredi y Gerardo I. Simari

Depto. de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur – Bahía Blanca, Argentina

1er. Cuatrimestre de 2019

Evolución del software

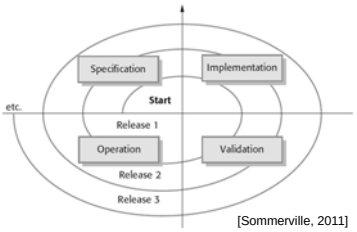
- El desarrollo del software no se *detiene* cuando el sistema se entrega.
- Luego de su puesta en funcionamiento, el sistema inevitablemente sufrirá *cambios*:
  - Cambios en el negocio (*business*)
  - Expectativas de los usuarios (cambios funcionales y no funcionales)
  - Se encuentran errores
  - Cambios en la plataforma de hardware y software
- Los sistemas no evolucionan en *aislación*.

Evolución del software

- Las organizaciones invierten mucho dinero en sistemas, y se tornan rápidamente *dependientes* de ellos.
- Los sistemas son *bienes* del negocio, y se debe invertir en *mantener* su valor (como cualquier bien).
- En general, se *invierte* más en mantenimiento que en desarrollo de sistemas nuevos: entre 65 y 90% del total.
- Los sistemas útiles tienen una *larga vida*:
  - Sistemas militares / control aéreo: pueden durar 30 años o más.
  - Sistemas de negocio (sueldos, etc.): 10 años o más.

Evolución del software

- A través del tiempo, los *requerimientos* del sistema cambian junto con los cambios en el negocio y entorno.
- Es útil entonces pensar en el proceso de ingeniería de SW como un proceso de *espiral*:



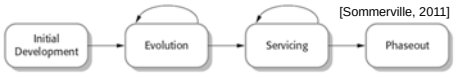
[Sommerville, 2011]

Evolución del software

- El desarrollo de SW “empaquetado/enlatado” (MS Word, Adobe Photoshop, etc.) suele seguir este método.
- El SW particular es diferente, ya que la compañía que lo encargó suele tomar el *control*.
- Puede haber *discontinuidades* en el espiral:
  - La compañía contrata a *terceros* para seguir el desarrollo y no pasa todos los documentos de requerimientos y diseño.
  - Las compañías se unen o reorganizan, y *heredan* software.
- En estos casos, el proceso de evolución se llama “*mantenimiento*”.

Evolución del software

Otro modelo de evolución es más lineal:



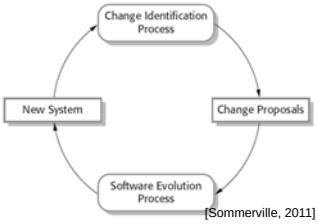
[Sommerville, 2011]

- El período de *evolución* involucra grandes cambios en respuesta a cambios en la especificación; el sistema se va *degradando*.
- *Servicio*: se realizan cambios pequeños, mientras se consideran opciones de reemplazo para el sistema.
- *Salida*: Se utiliza el sistema, pero no se hacen más cambios.

Procesos de evolución

- Los procesos que guían la evolución del SW *varían* dependiendo del tipo de sistema y compañía:
  - Puede ser un proceso muy *informal*, en el que los cambios se solicitan en conversaciones entre usuarios y desarrolladores.
  - Puede ser un proceso *formalizado*, con producción de documentación estructurada en cada etapa.
- Las *propuestas de cambio* son conductores del proceso de evolución.
- Los procesos de *identificación* de cambios y evolución del sistema son cíclicos, como muestra la siguiente figura:

Procesos de evolución

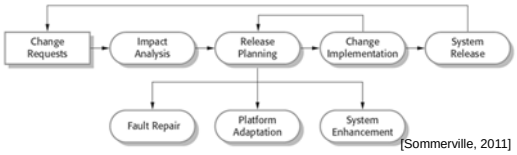


[Sommerville, 2011]

- Las propuestas de cambio son parte integral del proceso llamado "*administración de la configuración*".
- Algunos aspectos de este tema fueron cubiertos en el módulo de sistemas de gestión de versiones.

Procesos de evolución

- El proceso de *evolución* sigue el siguiente esquema:



[Sommerville, 2011]

- Se analiza el *costo* e *impacto* de los cambios propuestos; si se aceptan, se planea una nueva entrega del sistema.
- Durante esta planificación, se evalúa qué implementar en la *próxima versión*.

Procesos de evolución

- Este proceso de implementación de cambios se asemeja a una *iteración/sprint* de la implementación.
- Diferencia importante: la primera iteración puede involucrar un proceso de *comprensión* del sistema:
  - ¿Cómo está estructurado?
  - ¿Cómo entrega funcionalidad?
  - ¿Cómo puede afectar el cambio propuesto al resto del sistema?

Procesos de evolución

Idealmente, la implementación de cambios debe *modificar* la especificación, diseño e implementación:

[Sommerville, 2011]

Procesos de evolución

- Puede ocurrir que un cambio tenga que ver con problemas que deben solucionarse *urgentemente*:
  - Una *falla seria* que no permite la operación normal.
  - Cambios implementados provocaron *efectos inesperados*.
  - Cambios no anticipados en el *negocio* (legislación, etc.)
- En estos casos, puede ser necesario seguir un camino más *corto* para implementar el cambio...

[Sommerville, 2011]

Procesos de evolución

- Estos cambios de emergencia tienen un *peligro* inherente:

  Se soluciona el problema pero no se refleja en la documentación.

- Otro peligro: se opta por la reparación *más rápida* en vez de la mejor.
- Esto acelera el *envejecimiento* del software, también llamado "*erosión*".

Dinámica de la evolución

- Se le llama "dinámica de la evolución" al estudio de los *cambios* en los sistemas.
- Desde los años '70 y hasta fines de los '90, Lehman y Belady realizaron *estudios empíricos* para comprender mejor la evolución del SW.
- De estos estudios surgió un conjunto de conclusiones conocidas como las "*Leyes de Lehman*".
- Sus proponentes sostienen que aplican a todo sistema de software de gran tamaño embebidos en el mundo real (conocidos como "*de tipo E*").

Leyes de Lehman

1) *Ley del cambio continuo*:

"Un sistema que se usa en un entorno del mundo real debe cambiar; si no, se hará progresivamente menos útil en ese entorno."

2) *Ley de la complejidad creciente*:

"A medida de que cambia un sistema en evolución, su estructura tiende a hacerse más compleja."

Se deben utilizar más recursos para preservar y simplificar la estructura.

Leyes de Lehman

3) *Ley de la evolución de los grandes sistemas*:

"La evolución de los sistemas es un proceso auto-regulado."

Atributos tales como tamaño, tiempo entre versiones y la cantidad de errores reportados es una "*invariante aproximada*" para cada versión del sistema.

4) *Ley de la estabilidad organizacional*:

"A través de su vida, la proporción de desarrollo de un sistema es aproximadamente constante e independiente de los recursos dedicados."

Leyes de Lehman

5) *Ley de la conservación de la familiaridad*:

"A través de la vida de un sistema, el cambio incremental en cada versión es aproximadamente constante."

6) *Ley del crecimiento continuo*:

"La funcionalidad que ofrecen los sistemas debe crecer continuamente para mantener la satisfacción de los usuarios."

Leyes de Lehman

7) *Ley de la calidad decreciente*:

"La calidad de los sistemas decaerá a menos de que sean modificados para reflejar los cambios en su entorno operacional."

8) *Ley del sistema de retroalimentación*:

"Los procesos de evolución incorporan múltiples sistemas de retroalimentación multi-agente. Deben ser tratados como sistemas de retroalimentación para alcanzar un mejoramiento significativo del producto."

Mantenimiento del software

Mantenimiento del software

- Por "*mantenimiento*" generalmente nos referimos al proceso de cambiar un sistema *luego* de su entrega.
- El término usualmente se aplica cuando el grupo que realiza las actividades es *diferente* al que lo desarrolló.
- Existen tres *tipos* de mantenimiento:

  1) *Reparación* de fallas:

  – Errores de codificación: suelen ser fáciles de corregir;

  – Errores de diseño: son más complicados porque pueden afectar muchas componentes.

  – Errores en los requerimientos: los peores, dado que pueden disparar un rediseño extensivo a todo al sistema.

Mantenimiento del software

- Existen tres tipos de mantenimiento (cont.):

  2) *Adaptación* al entorno: cuando algún aspecto del entorno (hardware, software, etc.) cambia.

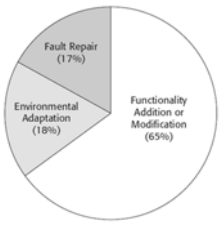
  3) *Agregado* de funcionalidad: cuando cambian los requerimientos del sistema.

  Los cambios suelen ser *mayores* en este tipo de mantenimiento que en los otros.

- En la práctica, la distinción entre los diferentes tipos puede no ser tan clara.

Mantenimiento del software

- Recordemos: el mantenimiento puede ocupar entre el 65% y el 90% del esfuerzo total.
- Afortunadamente, la mayoría de este tiempo no es invertido en el arreglo de *fallas*:



[Sommerville, 2011]

Mantenimiento del software

- Dadas estas estadísticas, suele ser beneficioso invertir parte del *esfuerzo* de diseño e implementación en la reducción del costo de *cambios futuros*:
  – Hacer que el software sea fácil de entender y cambiar
  – Especificaciones precisas
  – Administración de la configuración
- Esencialmente, hay que aceptar los costos *aparentemente* inútiles a corto plazo para luego disfrutar de sus efectos.
- Lamentablemente, los costos a corto plazo son *medibles*, pero los de largo plazo lo son sólo *bajo incertidumbre*.

Mantenimiento del software

Suele ser más costoso agregar funcionalidad *luego* de que un sistema está en operación; principalmente por:

1) Estabilidad del equipo:

Es común que el equipo de desarrollo se disuelva luego de la entrega; el nuevo equipo de invertir tiempo en *comprender* todo lo necesario del sistema.

2) Malas prácticas de desarrollo:

Es común que el contrato de mantenimiento sea otorgado a otra compañía. Junto con (1), este factor se traduce en una falta de *motivación* por entregar código mantenible.

Mantenimiento del software

Suele ser más costoso agregar funcionalidad *luego* de que un sistema está en operación; principalmente por (cont.):

3) Habilidad del equipo:

El equipo de mantenimiento puede no tener experiencia en el dominio; la actividad de mantenimiento tiene una "*mala imagen*" entre los ingenieros de SW y suele asignarse a los más junior.

Los sistemas antiguos pueden estar escritos en lenguajes *obsoletos*, que deben aprenderse.

Mantenimiento del software

Suele ser más costoso agregar funcionalidad *luego* de que un sistema está en operación; principalmente por (cont.):

4) Edad y estructura del sistema:

La estructura se *degrada* con los cambios, haciendo que el código sea más difícil de entender y cambiar.

El código tal vez fue desarrollado con prioridad en *optimizaciones* sobre legibilidad.

La *documentación* puede haberse perdido o ser inconsistente.

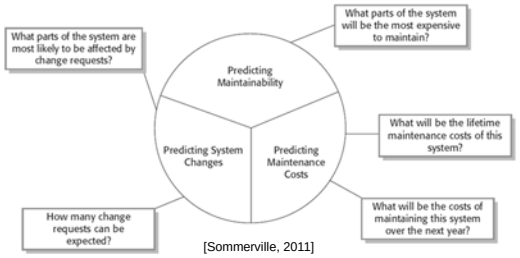
Falta de administración de la configuración: lleva a pérdida de tiempo buscando las *versiones* de las componentes.

Mantenimiento del software

- Los problemas (1) a (3) surgen de que muchos consideran que desarrollo y mantenimiento son actividades *disjuntas*.
- El mantenimiento suele verse como una actividad *secundaria* que no merece muchos recursos.
- El problema (4) es relativamente más fácil de encarar:
  - Reingeniería de software
  - Transformaciones de arquitectura
  - Refactoreo de código

Predicción de mantenimiento

- Los gerentes y administradores no quieren *sorpresas*.
- Se debe intentar *predecir* qué cambios se propondrán, qué partes afectarán y el costo general del mantenimiento:



[Sommerville, 2011]

Predicción de mantenimiento

Este tipo de predicción requiere una comprensión de la *relación* entre el *sistema* y su *entorno*:

- La cantidad y complejidad de *interfaces*: cuanto más y más complejas son las interfaces, más probable es que surjan cambios en ellas.
- La cantidad de requerimientos inherentemente *volátiles*: los requerimientos que reflejan políticas y procedimientos organizacionales son más volátiles.
- Los *procesos* de negocio en los que se usa el sistema: cuantos más procesos usan el sistema, surgen más y mayores cambios.

Predicción de mantenimiento

- Las *métricas* pueden darnos una mano en esta tarea de predicción:
  - Cantidad de pedidos de mantenimiento *correctivo*.
  - Tiempo promedio usado para análisis de *impacto*: refleja la cantidad de componentes afectadas por el pedido de cambio.
  - Tiempo promedio usado para *implementar* un cambio: puede estar correlacionado con el anterior.
  - Cantidad de pedidos de cambio *pendientes*.
- Un *incremento* en cualquiera de estas métricas puede indicar una *reducción* en la mantenibilidad del sistema.

Reingeniería de software y

administración de sistemas legado

Reingeniería de software

- La *reingeniería* apunta a mejorar a los sistemas legado.
- Sistemas legado ("*legacy*"):
  - Sistemas antiguos que todavía son *útiles* y hasta pueden ser *críticos* para el negocio.
  - Pueden estar implementados con lenguajes o tecnología *obsoletos*, o depender de sistemas *difíciles* de mantener.
  - Su estructura puede estar *erosionada* y su documentación ausente o desactualizada.
  - Aun así, puede *no convenir* reemplazarlos (costos, riesgos, poco uso, etc.).

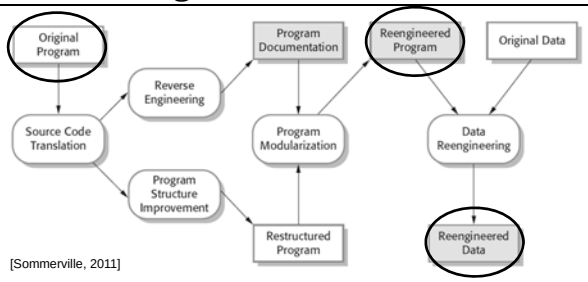
Reingeniería de software

- Las actividades de reingeniería pueden involucrar:
  - Documentar nuevamente al sistema
  - Refactorear la arquitectura
  - Traducción de código a un lenguaje moderno
  - Modificar y actualizar la estructura y valores de los datos utilizados por el sistema.
- Importante: La *funcionalidad* no es afectada por la reingeniería.
- Se debe *evitar* hacer grandes cambios arquitecturales.

Reingeniería de software

- La reingeniería trae dos *beneficios* principales en comparación con el reemplazo:
  - Menor *riesgo*: desarrollar nuevamente un sistema crítico conlleva un alto riesgo: errores, demoras, etc.
  - Menor *costo*: la reingeniería suele utilizar menor cantidad de recursos.
- A continuación veremos un esquema general del proceso de reingeniería.
- No todos los pasos son siempre necesarios.

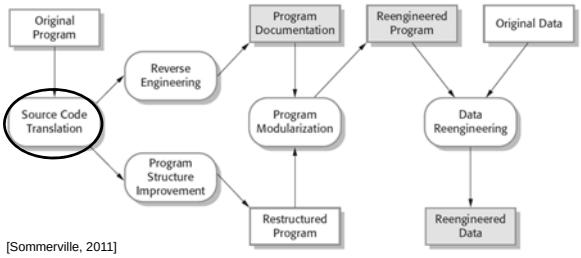
Reingeniería de software



[Sommerville, 2011]

- El proceso comienza con un sistema legado.
- El objetivo es lograr un sistema/datos funcionalmente *equivalente* pero mejorado.

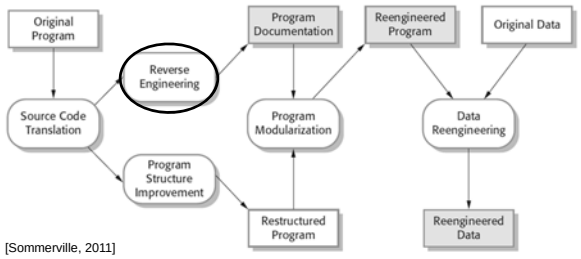
Reingeniería de software



[Sommerville, 2011]

- Con ayuda total o parcial de herramientas, el código se *traduce* de su lenguaje original a otro más deseable.

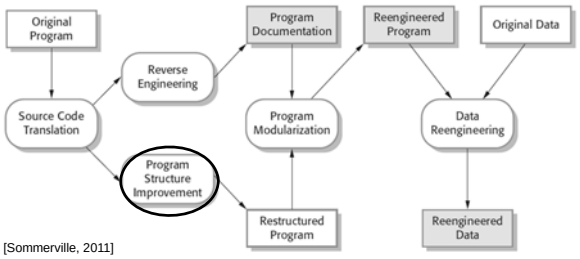
Reingeniería de software



[Sommerville, 2011]

- Se analiza el *código* para extraer información necesaria; ayuda a *documentar* su organización y funcionalidad.
- Nuevamente se usan herramientas automatizadas.

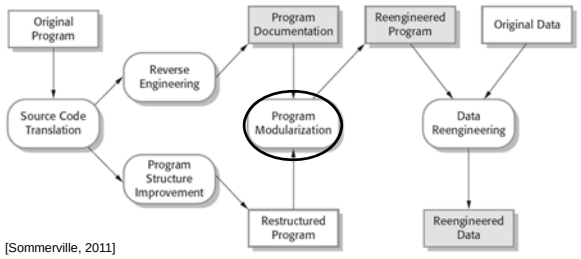
Reingeniería de software



[Sommerville, 2011]

- Se analiza la estructura de *control* y se la modifica para mejorar su legibilidad.
- Suele ser sólo parcialmente automático.

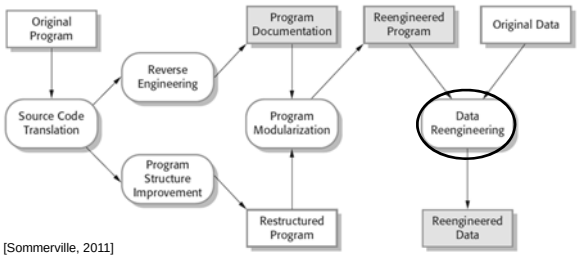
Reingeniería de software



[Sommerville, 2011]

- Se *agrupan* partes relacionadas y se remueven *redundancias*. Puede involucrar refactoreo de *arquitectura*.
- Es un proceso manual.

Reingeniería de software



[Sommerville, 2011]

- Se cambian los *datos* procesados para reflejar los cambios en el código: esquemas de BD, limpieza de datos, etc.
- Existen herramientas de automatización parcial.

Reingeniería de software

- Los *costos* del proceso dependen de la cantidad de trabajo a realizar:
  - La aplicación de *herramientas* reduce los costos.
  - La necesidad de *reestructurar* (módulos, arquitectura) encarece, debido a los procesos manuales.
- Límites:
  - No es posible realizar cambios *radicales* (paradigma, por ej.).
  - Si bien se mejora, probablemente no se logren los mismos *beneficios* que con un sistema totalmente nuevo.

Mantenimiento preventivo: Refactoreo

- Refactoreo: proceso de realizar mejoras a un sistema para reducir su *degradación* debido a los cambios:
  - Mejorar su *estructura*
  - Reducir su *complejidad*
  - Mejorar su *legibilidad*
- Suele asociarse con el desarrollo OO, pero se puede aplicar a *cualquier* tipo de desarrollo.
- Dado que no se cambia la funcionalidad, se dice que el refactoreo es "*mantenimiento preventivo*".

Reingeniería vs. Refactoreo

Tanto la reingeniería como el refactoreo tienen el objetivo de *mejorar* al software, pero no son lo mismo.

- Reingeniería:
  - Se lleva a cabo cuando el sistema ha estado en mantenimiento por un tiempo y los *costos* aumentan.
  - Se usan *herramientas* automáticas para partir de un sistema legado y obtener un nuevo sistema que sea *más mantenible*.
- Refactoreo:
  - Proceso *continuo* durante el desarrollo y evolución.
  - El objetivo es evitar la *erosión* que aumenta los costos.

Administración de sistemas legado

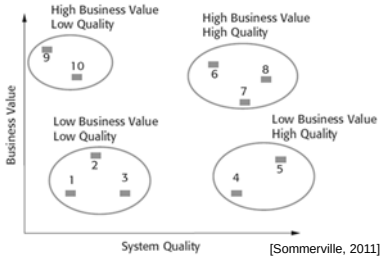
- Muchas compañías tienen un conjunto de sistemas legado que usan, con *poco presupuesto* para mantenerlos.
- Para decidir cómo obtener la mejor relación costo-beneficio, deben decidir entre:
  1) Deshacerse del sistema: no contribuye al negocio, y los procesos ya no dependen de él.
  2) No cambiar el sistema y seguir con el *mantenimiento regular*: aún se necesita y está en un estado relativamente estable.

Administración de sistemas legado

- Para decidir cómo obtener la mejor relación costo-beneficio, deben decidir entre (cont.):
  3) Hacer *reingeniería*: la calidad ha sido degradada y se siguen procesando nuevos cambios.
  4) Reemplazar todo o parte del sistema con uno *nuevo*: el sistema actual no puede continuar en operación, o pueden usarse sistemas "enlatados" con muy bajo costo.
- Estas opciones *no son excluyentes*, dado que se pueden aplicar diferentes opciones a distintas componentes.

Administración de sistemas legado

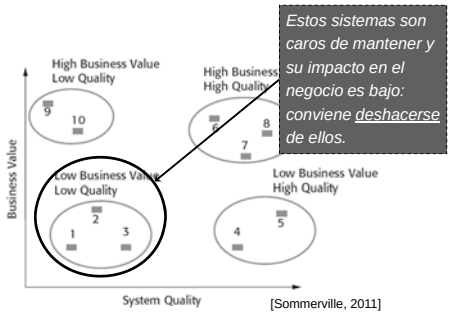
Supongamos que tenemos un conjunto de sistemas legado, y analizamos su calidad y valor para el negocio:



[Sommerville, 2011]

Administración de sistemas legado

Supongamos que tenemos un conjunto de sistemas legado, y analizamos su calidad y valor para el negocio:

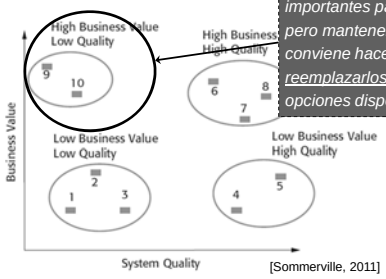


Estos sistemas son caros de mantener y su impacto en el negocio es bajo: conviene deshacerse de ellos.

[Sommerville, 2011]

Administración de sistemas legado

Supongamos que tenemos un conjunto de sistemas legado, y analizamos su calidad y valor para el negocio:



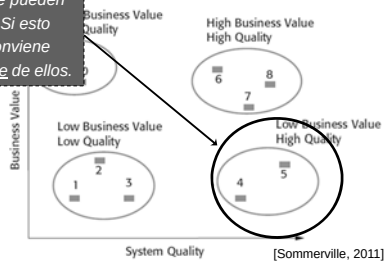
Estos sistemas son importantes para el negocio, pero mantenerlos es caro; conviene hacer reingeniería o reemplazarlos si hay opciones disponibles.

[Sommerville, 2011]

Administración de sistemas legado

Supongamos que tenemos un conjunto de sistemas legado, y analizamos su calidad y valor para el negocio:

Mientras no sea costoso, se pueden mantener. Si esto cambia, conviene deshacerse de ellos.

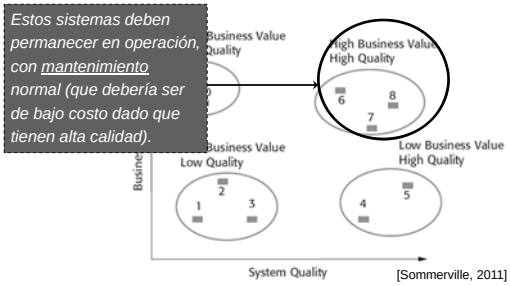


[Sommerville, 2011]

Administración de sistemas legado

Supongamos que tenemos un conjunto de sistemas legado, y analizamos su calidad y valor para el negocio:

Estos sistemas deben permanecer en operación, con mantenimiento normal (que debería ser de bajo costo dado que tienen alta calidad).



[Sommerville, 2011]

Administración de sistemas legado

Para evaluar el *valor* de un sistema legado, se debe analizar una serie de puntos:

- El *uso* del sistema: qué tan frecuente es, cuál es el impacto del uso, y quiénes son los usuarios.
- Los *procesos* que soporta: puede ocurrir que un sistema inflexible fuerce procesos ineficientes.
- *Confiabilidad*: Un bajo nivel de confiabilidad con impacto alto es una señal clara de bajo valor para el negocio.
- Las *salidas*: Si se pueden generar de otra manera fácilmente, o se usan infrecuentemente, tiene bajo valor.

Administración de sistemas legado

Los *factores* que influyen en el valor de un sistema legado se pueden dividir en dos:

- del *entorno* en el que funciona
- del *sistema* en sí

Factores del entorno	Factores del sistema
Estabilidad del proveedor	Comprensibilidad
Proporción de fallas	Documentación
Edad	Datos
Performance	Performance
Necesidades de soporte	Lenguaje de programación
Costo de mantenimiento	Administración de la configuración
Interoperabilidad	Datos de testing
	Habilidades del equipo

Reutilización de software

Reutilización de componentes

- La idea de basar el desarrollo en la *reutilización* de componentes data de fines de los años ‘60, pero recién a principios de este siglo comenzó a ser la norma.
- Los *beneficios* potenciales son claros:
  - Menor *costo* de producción y mantenimiento
  - Producción más *rápida*
  - Mejor *calidad*
- Uno de los mayores impactos lo ha tenido el movimiento del *software libre*.
- Los *estándares* ayudan a desarrollar software reutilizable.

Reutilización del software

- Las unidades de software que se reutilizan pueden ser de diferentes *tamaños*:
  - Sistema: un sistema *completo*, incorporándolo sin cambios a otros o configurándolo para otros clientes.
  - Componentes: *subsistemas* de diferentes tamaños.
  - Objetos y funciones: basado en *librerías*.
- El último tipo es el más común, y se practica desde hace décadas.
- Forma complementaria: "*reutilización de conceptos*", tales como ideas, algoritmos, formas de trabajo, etc.

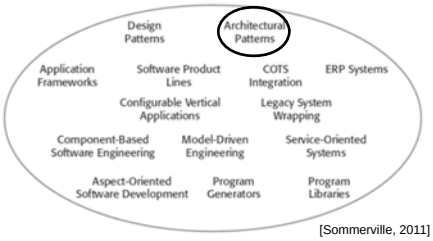
Reutilización del software: *Beneficios*

- Incremento de la *confiabilidad*: debería ser más confiable dado que ha sido usado y testeado en otros sistemas.
- Reducción del *riesgo* de proceso: el costo ya se conoce, mientras que el de desarrollar algo nuevo es incierto.
- Uso efectivo de *especialistas*: éstos encapsulan su conocimiento en componentes reutilizables.
- Conformidad a los *estándares*: se facilita la implementación bajo estándares, lo cual también ayuda a la confiabilidad.
- Desarrollo *acelerado*: reduce los tiempos de desarrollo y validación.

Reutilización del software: *Problemas*

- Mayores *costos* de mantenimiento en el caso en que el código fuente no está disponible.
- Falta de soporte en *herramientas*: en algunos casos no contemplan la reutilización.
- Síndrome de "*no se inventó aquí*": por falta de confianza o por ego, los equipos pueden no confiar en el SW.
- Creación, mantenimiento y uso de *librerías*: incorporar estas actividades en los procesos puede ser costoso.
- Encontrar, usar y adaptar *componentes*: puede ser costoso y/o infructífero si al final no se consiguen.

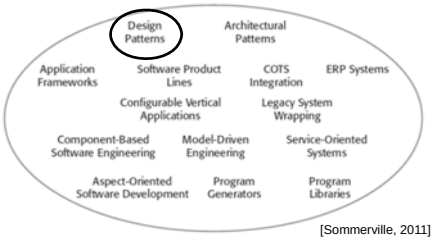
Posibilidades para la reutilización



[Sommerville, 2011]

- Arquitecturas *estándar* de software que soportan tipos comunes de aplicaciones.
- Permiten aprovechar las ventajas conocidas.

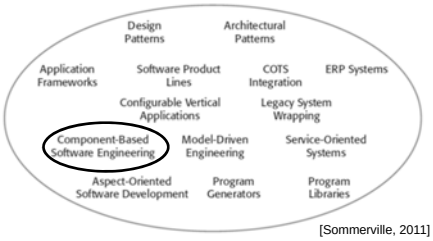
Posibilidades para la reutilización



[Sommerville, 2011]

- Abstracciones *genéricas* que ocurren en muchos casos.
- Se representan como *patrones* de diseño con objetos e interacciones concretas y abstractas.

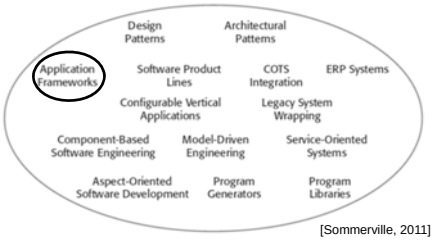
Posibilidades para la reutilización



[Sommerville, 2011]

- Los sistemas se desarrollan integrando *componentes*.
- Existen estándares de *modelos* basados en componentes.

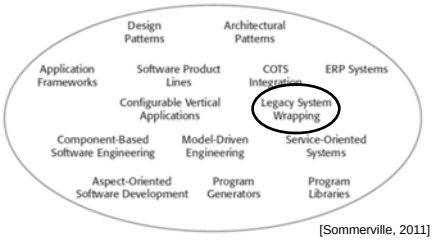
Posibilidades para la reutilización



[Sommerville, 2011]

- Conjuntos de *clases* abstractas y concretas que se adaptan y extienden para crear sistemas.

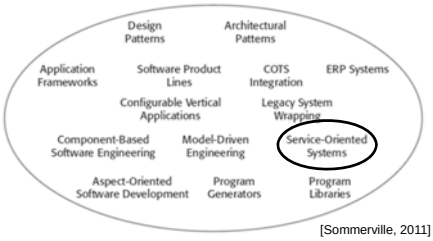
Posibilidades para la reutilización



[Sommerville, 2011]

- Los sistemas legado se "*envuelven*" con interfaces nuevas entre ellos y los sistemas nuevos.

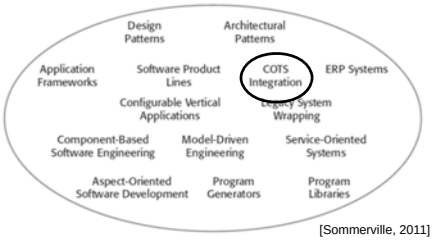
Posibilidades para la reutilización



[Sommerville, 2011]

- Se enlazan *servicios*, que pueden ser provistos por terceros.

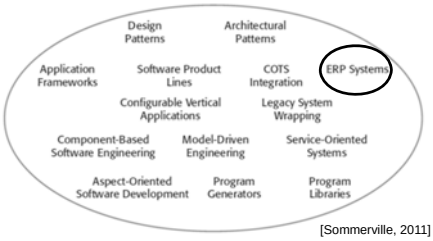
Posibilidades para la reutilización



[Sommerville, 2011]

- COTS: Commercial Off-The-Shelf ("*enlatados*")
- Se configuran e integran sistemas ya *existentes*.

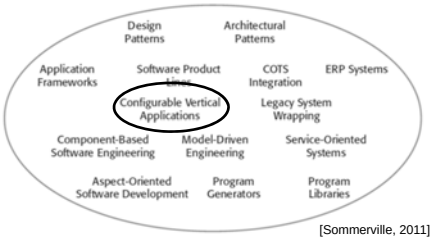
Posibilidades para la reutilización



[Sommerville, 2011]

- ERP: Enterprise Resource Planning
- Sistemas de *gran escala* que encapsulan funcionalidades genéricas de negocio; se *configuran* para cada compañía.

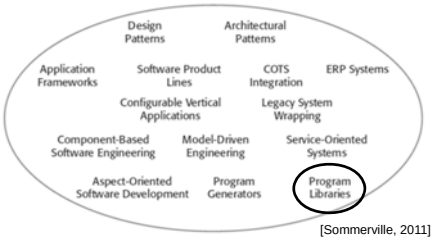
Posibilidades para la reutilización



[Sommerville, 2011]

- Sistemas *genéricos* pero para *dominios específicos* que se pueden configurar para las necesidades del cliente.

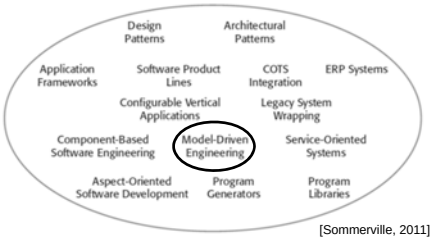
Posibilidades para la reutilización



[Sommerville, 2011]

- *Librerías* de funciones y clases que implementan funcionalidades usadas frecuentemente.

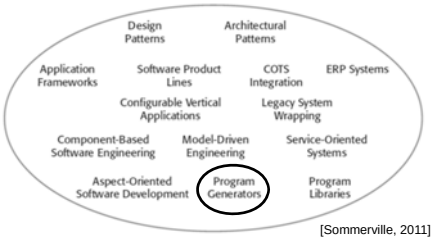
Posibilidades para la reutilización



[Sommerville, 2011]

- Los sistemas se diseñan con modelos *independientes* de la implementación.
- El código se *genera* a partir de los modelos.

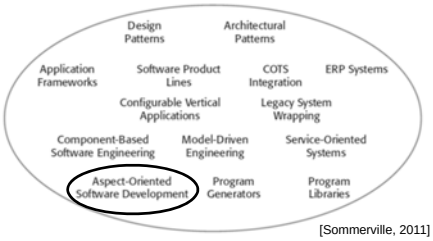
Posibilidades para la reutilización



[Sommerville, 2011]

- Los *generadores* incluyen conocimiento acerca de un tipo de aplicación.
- También generan código a partir de *modelos* provistos.

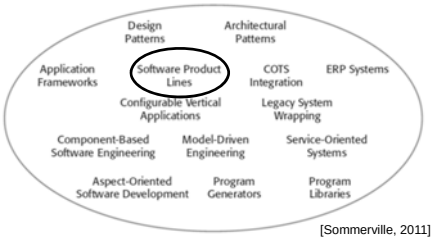
Posibilidades para la reutilización



[Sommerville, 2011]

- Se basa en identificar diferentes *aspectos* (lógicamente) independientes y *unirlos* para lograr el sistema.
- Separa la *funcionalidad* secundaria de la *lógica* de negocio.

Posibilidades para la reutilización



[Sommerville, 2011]

- Se generaliza un *tipo* de aplicación en una arquitectura común.
- Se *adapta* luego a diferentes clientes.

Factores para planificar la reutilización

1) La *planificación* del desarrollo:

   Si se tiene poco tiempo, intentar reutilizar sistemas enlatados antes que componentes individuales.

2) El *tiempo de vida* esperado del software:

   Si el sistema será usado por mucho tiempo, enfocarse en la *mantenibilidad*.

   Si el código fuente de las componentes a reutilizar no está disponible, conviene evitarlas en estos casos.

Factores para planificar la reutilización

3) Habilidades y experiencia del *equipo*:

   Las tecnologías aplicadas en la reutilización son complejas y se requiere tiempo para comprender y usarlas efectivamente.

4) Qué tan *crítico* es el software y sus requerimientos NF:

   Si el software es crítico, las garantías de *confiabilidad* pueden ser difíciles de lograr si no se tiene el código fuente.

   Si el software debe tener alta *performance*, el uso de generadores de código no suele funcionar.

Factores para planificar la reutilización

5) El *dominio* de aplicación:

   Algunos dominios tienen un conjunto ya maduro de productos configurables. Si este es el caso, conviene explorarlos como primera opción.

6) La *plataforma* sobre la cual correrá el sistema:

   Algunos modelos de componentes son *específicas* a una plataforma, como por ejemplo Microsoft .NET.

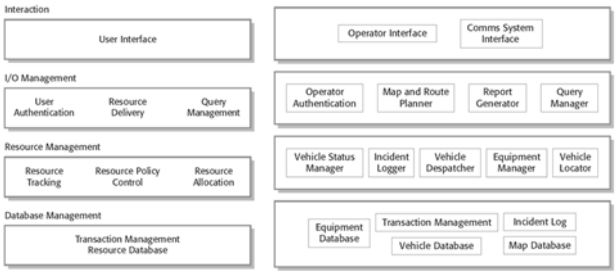
Ejemplo de línea de productos de SW

Consideremos un sistema de *línea de productos* diseñado para el manejo de vehículos de emergencia:

- Los operadores reciben llamadas acerca de incidentes, deciden qué tipo de vehículo enviar, y lo despachan.
- Los desarrolladores quieren tener versiones del sistema para policía, bomberos y ambulancias.

- Este es un ejemplo de una arquitectura de "*manejo de recursos*".
- Las componentes de esta arquitectura genérica pueden ser *instanciadas* para crear un sistema particular como el de los vehículos.

Reutilización de una arquitectura genérica



[Sommerville, 2011]

Líneas de productos de SW

- Para adaptar el sistema a cada cliente, se pueden modificar las *componentes* individuales.
- Los pasos básicos que se deben tomar al extender una línea de productos para crear un nuevo sistema son:
  - Obtener los *requerimientos* del cliente: como siempre, pero mostrando las *opciones disponibles* para adaptar.
  - Elegir la opción que mejor se ajusta.

Líneas de productos de SW

- Para adaptar el sistema a cada cliente, se pueden modificar las *componentes* individuales.
- Los pasos básicos que se deben tomar al extender una línea de productos para crear un nuevo sistema son (cont.):
  - *Renegociar* los requerimientos: puede ser necesario para minimizar los cambios necesarios.
  - *Adaptar* el sistema existente.
  - Entregar el nuevo sistema: *documentar* bien para permitir la posibilidad de reutilizarlo en el futuro.

Líneas de productos de SW

La característica básica es que se pueden *reconfigurar*:

- Al momento de *diseñar*: los desarrolladores modifican el núcleo de un sistema genérico existente, seleccionando o adaptando componentes.

  Posibilita *modificaciones* mayores (hasta del código fuente) y mayor flexibilidad.

- Al momento de puesta en funcionamiento (*deployment*): El sistema genérico puede ser configurado al incluir detalles del *entorno* operacional específico en archivos de configuración.

  Incluye *selección* de componentes, definiciones de *flujos* de trabajo y especificación de *parámetros*.

Referencias

[Sommerville, 2011] I. Sommerville: "*Software Engineering*", 9th Edition. Addison-Wesley, 2011.

[Sommerville, 2015] I. Sommerville: "*Software Engineering*", 10th Edition. Addison-Wesley, 2015.