

Clase 1: Introducción

1.1 Sobre el curso 6.170

El presente curso engloba tres cursos en uno:

- Un curso intensivo de programación orientada a objetos.
- Un curso de diseño de software en el medio.
- Un curso sobre construcción de software en equipo.

Énfasis en el diseño. El curso incluye conocimientos de programación, por tratarse de un requisito esencial; así como la realización de un proyecto, ya que la única forma realmente eficaz de aprender una idea es llevándola a la práctica.

En este curso, usted aprenderá:

- Cómo diseñar software: mecanismos robustos de abstracción, patrones de diseño que han demostrado su eficacia en la práctica y métodos de representación de diseños que permitan comunicarlos y hacer críticas.
- Cómo implementar en Java
- Cómo diseñar e implementar adecuadamente para crear un software fiable y flexible.

También aprenderá, sin recurrir a parches:

- A trabajar con la arquitectura del sistema, y no simplemente a escribir código de bajo nivel.
- Cómo no perder tiempo depurando un programa.

1.2 Administración y principios

Presentación del personal del curso:

- Profesores: Daniel Jackson y Rob Miller.
- Ayudantes técnicos (TAs): los conocerá la *próxima* semana en las sesiones de revisión.
- Monitores de prácticas (LAs): los conocerá en los grupos de prácticas.
- Horario: consulte la página web. Los profesores no tienen horario fijo de consulta, pero se mostrarán encantados de poder atender a los estudiantes: tan sólo tendrá que enviarles un correo o pasarse por su despacho.

Materiales:

- Libro de texto de Liskov; léalo siguiendo el programa del boletín de información general.
- Material de clase: normalmente se publica el mismo día de la clase.
- Se recomienda el libro de patrones de diseño *Gang of Four*.
- Otro libro recomendado es “Effective Java”, de Bloch.
- Tutorial de Java: consulte el boletín de información general para más información.

Los libros de texto recomendados son excelentes; le proporcionarán buenas referencias y le ayudarán a convertirse rápidamente en un buen programador en breve. Si compra la oferta le harán un gran descuento. Adquiriendo el paquete completo obtendrá un interesante descuento.

Organización del curso:

- Primera mitad del trimestre: clases, ejercicios semanales, revisiones, prueba.
- Segunda mitad del trimestre: proyecto en equipo. Se dará más información sobre el mismo más adelante.

Hay diferencias en relación con trimestres anteriores: no tiene que preocuparse ahora de quién formará parte de su equipo. Prepárese para un cambio de ayudante técnico a mediados del trimestre.

Revisiones:

- En las sesiones semanales con los ayudantes técnicos, se *revisará* el trabajo de los estudiantes.
- Al comienzo del curso, los ayudantes técnicos le pedirán fragmentos de su trabajo que tomarán como punto de referencia para la revisión.
- El grupo al completo debatirá los temas de manera constructiva y con la colaboración participación de todos.
- Un aspecto esencial del curso es que le ofrece la oportunidad de observar la aplicación práctica de los conceptos tratados en clase.

Iniciación a Java:

- El aprendizaje de Java lo realiza cada estudiante por sí mismo, pero cuenta para ello con nuestra ayuda.
- Utilice el tutorial de Java de Sun y haga los ejercicios.
- Tendrá a su disposición un amplio equipo de ayudantes de prácticas dispuestos a resolver sus dudas.

Colaboración y política sobre protocolos de Internet:

- Consulte la información general.
- En resumen: puede intercambiar ideas con sus compañeros, pero los trabajos escritos, tales como especificaciones, diseños, código, pruebas, explicaciones, etc., debe realizarlos usted personalmente.
- Puede utilizar código de dominio público.
- En el proyecto de equipo, puede colaborar en todo.

Pruebas:

- Dos pruebas centradas en el material explicado en las clases.

Calificaciones:

- El 70% lo constituirá el trabajo individual = el 25% las pruebas y el 45% de los boletines de problemas.
- El proyecto final valdrá el 30% restante, puntuándose por igual a los miembros de un mismo grupo de trabajo.
- La participación en clase supondrá una puntuación adicional del 10%.
- *no se aceptará ningún trabajo entregado fuera de plazo.*

1.3 ¿Por qué es importante la ingeniería de software?

Aportación del software a la economía de EE.UU (datos del año 1996):

- Principal fuente de superávit por exportaciones en la balanza comercial.
- 24.000 millones de dólares de ingresos por exportaciones de software y 4.000 millones gastados en importaciones, arrojan un superávit anual de 20.000 millones de dólares.
- Datos comparativos (también en millones de dólares): agricultura, 26-14-12; industria aeroespacial, 11-3-8; industria química, 26-19-7; industria automovilística, 21-43-(22); productos manufacturados, 200-265-(64).

(Datos tomados de: *Software Conspiracy*, Mark Minasi, McGraw Hill, 2000).

Papel del software en la infraestructura:

- no sólo tiene un papel importante en Internet;
- también en sectores como transportes, energía, medicina y finanzas.

El software se halla cada vez más presente como elemento incorporado a otros mecanismos arraigados. Los automóviles modernos, por ejemplo, poseen entre 10 y 100 procesadores para dirigir todo tipo de funciones, desde el reproductor de música hasta el sistema de frenado.

El coste del software:

- La relación entre la adquisición de software y hardware se aproxima a cero.
- Coste total de la propiedad del software: 5 veces el coste del hardware. El grupo Gartner calcula que el coste de mantenimiento de un PC durante 5 años asciende en la actualidad a 7 dólares por cada K de memoria del ordenador).

¿Qué calidad presenta nuestro software?

- Fallos en el desarrollo.
- Accidentes.
- Software de baja calidad.

1.3.1 Fallos en el desarrollo

Estudio llevado a cabo por IBM en el año 1994:

- El 55% de los sistemas costaron más de lo previsto.
- El 68% excedieron el tiempo previsto para su desarrollo.
- El 88% se tuvo que volver a diseñar por completo.

Sistema de Automatización Avanzada (FAA, 1982-1994)

- El promedio de producción industrial era de 100 dólares/línea, y se preveía pagar 500 dólares/línea.
- Se terminó por pagar 700-900 dólares/línea.
- Trabajos cancelados por valor de 6.000 millones de dólares.

Departamento de Estadística Laboral (1997)

- 2 de cada seis nuevos sistemas que se ponen en funcionamiento sufren cancelaciones.
- Los grandes sistemas tienen aproximadamente un 50% de probabilidad de ser cancelados.
- La media de tiempo empleado en un proyecto se excede en un 50% con respecto al plazo previsto.

- las $\frac{3}{4}$ partes de los sistemas se consideran “fracasos operativos”.

1.3.2 Accidentes

La mayor parte de los expertos coinciden en señalar que “la manera más probable de destruir el mundo es por accidente”. Y aquí es donde entramos en juego nosotros, los profesionales de la informática: “nosotros somos los que provocamos los accidentes”.

Nathaniel Borenstein, creador de MIME en: *Programming as if People Mattered: Friendly Programs, Software Engineering and Other Noble Delusions*, Princeton University Press, Princeton, NJ, 1991.

Caso Therac-25 (1985-87)

- Se trataba de un aparato de radioterapia dotado de controlador de software.
- Se retiró el interbloqueo del hardware, pero el software no tenía dispositivo de interbloqueo.
- El software falló al mantener las constantes vitales: un flujo de electrones o bien un flujo más intenso de radiación mediante placa para generar rayos X.
- A consecuencia de ello se produjeron varias muertes por quemaduras.
- El programador no tenía experiencia en programación concurrente.
- Véase: <http://sunnyday.mit.edu/therac-25.html>

Cabría pensar que se aprendería de la experiencia y que un desastre de este tipo no volvería a suceder jamás. Sin embargo...

- La Agencia Internacional de Energía Atómica anunció una “emergencia radiológica” el 22 de Mayo del 2001 en Panamá.
- 28 pacientes sufrieron sobreexposición: 8 murieron, 3 de ellos como consecuencia directa de la sobreexposición; con la probabilidad de que $\frac{3}{4}$ de los 20 que sobrevivieron desarrollaran “serias complicaciones que en algunos casos, a la larga, podrían resultar mortales”
- Los expertos anunciaron que el equipo de radioterapia “funcionaba perfectamente”; la razón de la emergencia tuvo que ver con la entrada de datos.
- Si los datos se introdujeron en varios bloques protegidos dentro de un lote, la dosis se computó de manera incorrecta.
- Al menos, la FDA llegó a la conclusión de que “la interpretación de los datos del bloqueo de flujo por el software” fue uno de los factores causantes del desastre.
- Visite la web: <http://www.fda.gov/cdrh/ocd/panamaradexp.html>

Ariane-5 (Junio de 1996)

- Agencia Espacial Europea.
- Pérdida absoluta de misiles no tripulados poco después del despegue.
- Causada por una excepción en el código de Ada.
- Ni siquiera se precisó el código defectuoso después del despegue.
- Debido a un cambio en el entorno físico: se infringieron supuestos no documentados.
- Visite la web: <http://www.esa.int/htdocs/tidc/Press/Press96/ariane5rep.html>

En los desastres provocados por fallos de software, son más comunes los accidentes como el del Ariane, que los causados por aparatos de radioterapia. No es muy probable que los errores en el código sean la causa; normalmente, el problema se remonta al análisis de las

necesidades; en este caso, a un error al articular y evaluar presunciones claves sobre el entorno.

Servicio de Ambulancias de Londres (1992)

- Pérdida de llamadas, doble servicio por llamadas duplicadas.
- Mala selección del programador: experiencia insuficiente.
- Visite la web: <http://www.cs.ucl.ac.uk/staff/A.Finkelstein/las.html>

El desastre del Servicio de Ambulancias de Londres se debió en realidad a fallos de gestión. Los informáticos que desarrollaron el software pecaron de ingenuidad y aceptaron una oferta de una empresa desconocida que era bastante peor que las de otras compañías más acreditadas. Cometieron el terrible error de saltar a la red repentinamente, sin pararse a contrastar la ejecución del sistema nuevo y la del ya existente.

A corto plazo, estos problemas se acentuarán debido al uso generalizado del software en nuestra infraestructura cívica. En el informe PITAC se hacía eco de esta realidad, y los argumentos que en él se exponían han servido para incrementar los fondos destinados a la investigación en software:

"La demanda de software ha crecido mucho más rápido que nuestra capacidad para crearlo. Además, el país requiere un tipo de software más práctico, fiable y robusto que el que se está desarrollando hoy en día. Nos hemos hecho peligrosamente dependientes de los grandes sistemas de software, cuyo comportamiento no es del todo comprensible, y que a menudo fallan de forma imprevista".

Investigación en tecnología de la información: Invirtiendo en nuestro futuro
Comité Consultor en Tecnología de la Información del Presidente (PITAC)
Informe al Presidente, 24 de febrero de 1999
Información disponible en <http://www.ccic.gov/ac/report/>

Foro de RIESGOS

- coteja informes de prensa sobre incidentes relacionados con la informática.
- <http://www.catless.ncl.ac.uk>

1.3.3 Calidad del Software

Sistema de medición de la calidad: errores/kloc

- La medición se realiza después de la entrega del software.
- La media en la industria es de aproximadamente 10.
- De alta calidad: 1 o menos.

Sistema Praxis CDIS (1993)

- Sistema de control de tráfico aéreo desde terminales, empleado en el Reino Unido.
- Utilizaba un lenguaje de especificación concreto, muy parecido al de los modelos de objeto que aprenderemos en el curso.
- Sin aumento del coste de la red.
- Tasa de error mucho menor: aproximadamente 0'75 fallos/kloc.
- ¡Incluso se ofrecía garantía al cliente!

Por supuesto, la calidad de un software no se mide únicamente por los errores. Podemos probar un software y depurarlo, eliminando la mayoría de los errores que pueden hacer que falle, pero al final nos encontraremos ante un programa que es imposible utilizar, y que la mayoría de las veces no logra hacer lo que espera porque presenta muchos casos especiales. Para solucionar este problema, es preciso crear calidad desde el principio.

1.4 Importancia del diseño

“¿Sabe usted lo que hace falta para poder producir software de buena calidad? En los Estados Unidos, la calidad de los automóviles mejoró cuando Japón nos mostró otros métodos más eficaces de fabricación. Alguien tendrá que enseñar a la industria del software que éste también puede desarrollarse de un modo más eficiente.”

John Murria, experto en control de calidad del software de FDA, citado en *Software Conspiracy*, Mark Minasi, McGraw Hill, 2000.

¡ Este es probablemente también su caso!

El objetivo que perseguimos en el curso 6.170 es mostrarle que el código o la programación avanzada no lo es todo a la hora de crear software. De hecho, supone sólo una pequeña parte. No piense en el código como parte de la solución; normalmente forma parte del problema. Necesitamos palabras distintas a código para referirnos al software, palabras que sean menos aparatosas, más directas y menos vinculadas a la tecnología, ya que en breve se quedarán obsoletas.

Función del diseño y de los diseñadores:

- Pensar a largo plazo nunca viene mal (¡y es barato!).
- No se puede añadir calidad al final del proceso: hay que contrastar confiando en el testeo; resulta más efectivo y mucho menos costoso.
- Hacer posible la delegación de tareas y el trabajo en equipo.
- Un diseño defectuoso perjudica al usuario: software difícil de utilizar, incoherente y poco flexible.
- Un diseño defectuoso también afecta al programador: interfaces pobres, proliferación de errores y dificultades para añadir nueva funcionalidad.

No deja de ser curioso que los estudiantes de informática suelen resistirse a considerar la creación de software como una labor de ingeniería. Quizás piensen que las técnicas de ingeniería le restarán lo místico a su trabajo o no se adecuarán a su don de innatos mañosos. Quizás piensen que la aplicación de técnicas de ingeniería hace su trabajo menos excitante, o que éstas no se adecúan a sus dones innatos para la programación. Por otro lado, las técnicas que aprendas en el curso 6.170 permitirán que su talento goce de mayor eficacia. (Por el contrario, las técnicas que veremos en este curso ayudan al estudiante a poner en práctica sus condiciones naturales de un modo mucho más eficaz?).

Incluso los programadores profesionales se engañan a sí mismos. Durante un experimento, 32 informáticos de la NASA pusieron en práctica 3 técnicas distintas para probar algunos programas de pequeño tamaño. Se les pedía que evaluaran la proporción de errores que esperaban hallar en cada método. Resultó que sus intuiciones eran erróneas. Creyeron que el testeo en modalidad de caja negra sobre las especificaciones era el más efectivo, pero en realidad la modalidad de lectura de código resultó ser más eficaz (a pesar de que el código no

estaba comentado). Al aplicar el método de pruebas basado en la lectura de código, ¡encontraron los errores en la mitad de tiempo!

Victor R. Basili y Richard W. Selby:

Comparing the Effectiveness of Software Testing Strategies.

IEEE Transactions on Software Engineering. Vol. SE-13, No. 12, diciembre de 1987, págs. 1278 – 1296.

El diseño tiene mucha importancia para el software de infraestructura (como el de control de tráfico aéreo). Incluso en estos casos, muchos mandos altos y medios del sector parecen no darse cuenta de cuánta influencia puede tener lo que enseñamos en el curso 6.170. Échele un vistazo al artículo que John Chapin (un antiguo profesor del curso) y yo escribimos, en el cual explicamos como rediseñamos un componente de CTAS, un nuevo sistema de control de tráfico aéreo, a partir de conceptos que se exponen en el curso:

Daniel Jackson y John Chapin. *Redesigning Air-Traffic Control: An Exercise in Software Design.* IEEE Software, mayo/junio 2000. Disponible en <http://sdg.lcs.mit.edu/dnj/publications>.

1.4.1 La historia de Netscape

Existe un mito acerca del software de los ordenadores personales, según el cual el diseño carece de importancia, ya que lo único que importa en realidad es el tiempo que se tarda en lanzar el producto al mercado. En este sentido, la desaparición de Netscape es una historia sobre la que merece la pena reflexionar.

El originario equipo Mosaic del NCSA (National Center for Supercomputing Applications) de la Universidad de Illinois creó el primer navegador de uso generalizado, pero su trabajo fue rápido y ciertamente mejorable. El equipo fundó Netscape, y entre abril y diciembre del año 1994 se creó Navigator 1.0. Se ejecutó sobre 3 plataformas, y pronto se convirtió en el principal navegador para Windows, Unix y Mac. Microsoft empezó a desarrollar el navegador Internet Explorer 1.0 en octubre del año 1994, y lo lanzó al mercado con Windows 95 en agosto del año 1995.

Durante el periodo de mayor expansión de Netscape, desde 1995 a 1997, los programadores trabajaron duro para lanzar al mercado nuevos productos con nueva funcionalidad, dedicando muy poco tiempo al diseño. La mayoría de las empresas dedicadas al negocio del empaquetado de software (aún) creen que el diseño de un producto puede ser postergado: que una vez conquistada una cuota de mercado y una serie de cualidades convincentes, se puede “volver a considerar” el código y obtener las ventajas de un diseño nítido. Netscape no fue la excepción, aún cuando sus ingenieros eran probablemente más competentes que los de la mayoría de las compañías rivales.

Mientras tanto, Microsoft se había dado cuenta de la necesidad de añadir diseños más sólidos. Creó NT partiendo desde cero, y replanteó la suite de Office para utilizar aplicaciones compartidas. Se apresuró a lanzar al mercado IE (Internet Explorer) para ponerse al nivel de Netscape, pero les llevó bastante tiempo reestructurar IE 3.0. Microsoft considera ahora que este replanteamiento de IE, fue una decisión clave que les ayudó a reducir distancias con Netscape.

El desarrollo de Netscape siguió avanzando. Eran 120 programadores (de 10 que había inicialmente) los que trabajaban en el desarrollo de Communicator 4.0, y habían desarrollado 3 millones de líneas de código 30 veces más que al principio. Michael Toy, director de ventas, afirmó:

“Nos encontramos ante una situación verdaderamente mala... Tendríamos que haber dejado de sacar este código hace un año. Está acabado... Esto es como despertarse de golpe de un sueño... Estamos pagando el precio de las prisas.”

Curiosamente, las razones que llevaron en 1997 a Netscape a pensar en un diseño modular surgieron del deseo de volver a trabajar con equipos pequeños para el desarrollo de productos. Pero si no se dispone de interfaces simples y claras resulta imposible dividir el trabajo en partes independientes unas de otras.

Netscape dejó de lado el proyecto durante 2 meses para reestructurar el navegador, pero este tiempo no fue suficiente. Así que se decidió volver a empezar desde el principio con Communicator 6.0. Pero la versión 6.0 nunca se acabó, y a sus programadores les volvieron a asignar trabajo en la versión 4.0. Mozilla, que era la versión 5.0, se encontraba disponible como versión de libre distribución, pero nadie quería trabajar en código espagueti.

Al final Microsoft ganó la batalla de los navegadores, y AOL se hizo con Netscape. Por supuesto que esta no es la historia completa de cómo el navegador creado por Microsoft llegó a imponerse sobre el de Netscape. Las prácticas empresariales de Microsoft no beneficiaron a Netscape, y la independencia de la plataforma fue un tema crucial desde el principio; Navigator se ejecutó sobre Windows, Mac y Unix desde la versión 1.0, y Netscape se esforzó por mantener la máxima independencia de plataforma en su código. Incluso se pensó en crear una versión pura en Java (“Javagator”), y se crearon muchas herramientas propias en Java (porque en esa época las herramientas de Sun no estaban acabadas). Sin embargo, en 1998 Netscape arrojó la toalla. De todas formas, Communicator 4.0 aún contiene aproximadamente 1’2 millones de líneas de código en Java.

He extraído esta sección de un excelente libro sobre Netscape, la empresa y sus estrategias técnicas. Puede leer la historia completa en:

Michael A. Cusumano and David B. Yoffie. *Competing on Internet Time: Lessons from Netscape and its Battle with Microsoft*, Free Press, 1998. Lea especialmente el capítulo 4, Estrategia de Diseño.

A este respecto, tenga en cuenta que Netscape tardó más de dos años en reconocer la importancia del diseño. Así que no se extrañe si no queda convencido de sus ventajas al terminar el curso; hay cosas que sólo se adquieren mediante la experiencia.

1.5 Consejos

Claves del curso

- No se quedes atrás: ¡el ritmo es rápido!
- Asista a las clases: no todo el material está en los libros.
- Piense por adelantado: no tenga prisa a la hora de codificar.
- Concentre su atención en el diseño, más que en depurar el programa.

Por mucho que se insista, siempre me parece poca la importancia que le doy a que empiece cuanto antes y a que piense por adelantado. Es obvio que no se espera que usted acabe el boletín de problemas el mismo día en que se le entregan pero, a la larga, ahorrará mucho tiempo y obtendrá resultados mucho mejores si empieza a trabajar desde el principio. En primer lugar, se beneficiará del tiempo que le haya dedicado: reflexionará sobre los problemas inconscientemente.

En segundo lugar, sabrá qué otros recursos va a necesitar, y podrá hacerse con ellos con tiempo, cuando aún es fácil conseguirlos. En especial, acuda siempre que lo necesite al personal del curso --¡estamos aquí para prestarle ayuda! Tenemos programado un horario para las prácticas de laboratorio en grupos, y horas de tutorías con los ayudantes técnicos, considerando los plazos de entrega de trabajos, aunque puede contar con nuestra ayuda en cualquier momento, siempre que no sea la noche anterior a la entrega de los boletines de problemas, que es cuando todos ustedes suelen necesitar ayuda.....

No se complique:

“Me cansé de advertir sobre los riesgos de la ambigüedad, la complejidad y la ambición desmedida en los nuevos diseños, pero nadie escuchó mis consejos. He llegado a la conclusión de que existen dos formas de construir un diseño de software: simplificándolo hasta el punto que resulte obvio que no hay en él errores o complicándolo de tal forma que los errores que haya en él no sean obvios”.

Tony Hoare, *Turing Award Lecture*, 1980

Refiriéndose al diseño de Ada, aunque su punto de vista se puede aplicar al diseño de programas en general.

"Cómo evitar complicarse" (KISS), (*keep it simple stupid*).

- Evite pisar terreno resbaladizo: trate de no recurrir a soluciones para iniciados ni a estructuras de datos y algoritmos complejos.
- No emplee las propiedades más complejas de un lenguaje de programación
- Muestre escepticismo ante la complejidad.
- No sea excesivamente ambicioso: reconozca el “*creeping featurism*” (tendencia que se basa en incorporar demasiado al programa en poco tiempo para satisfacer los requerimientos de un nuevo hardware/software) y el “síndrome del sistema sucesor” (tendencia que consiste en convertir un nuevo sistema, sucesor de otro más pequeño, en algo grandioso).
- Recuerde que es fácil crear algo complicado, pero que lo difícil consiste en desarrollar algo que resulte verdaderamente sencillo.

Regla de optimización

- No lo haga
- Déjelo en manos de expertos: no lo haga aún.

(de Michael Jackson, *Principles of Program Design*, Academic Press, 1975).

1.6 Colofón

Notas recordatorias:

- Mañana habrá *clase* normal, y no de repaso.

- Rellene la solicitud de matrícula que se encuentra en línea antes de la medianoche de hoy.
- ¡ Iníciase en Java desde ya!
- El plazo de entrega del Ejercicio 1 es el próximo martes.

Consulte esta dirección:

- http://www.170systems.com/about/our_name.html