uc3m | Universidad Carlos III de Madrid

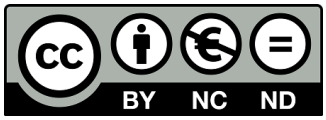
Grado en Ingeniería Informática

2019-2020

Apuntes

Principios de desarrollo de software

Jorge Rodríguez Fraile[1]



[1]Universidad: 100405951@alumnos.uc3m.es | Personal: jrf1616@gmail.com

ÍNDICE GENERAL

1. TEMA 1: ASPECTOS ÉTICOS Y PROFESIONALES RELATIVOS A LA PROFESIÓN DE LA INGENIERÍA DEL SOFTWARE

- El ingeniero de software es más parecido a un artesano que aun ingeniero.
- **Ingeniero:** Aplica el método científico, resolviendo un problema técnico.
    - **Software:** Parte no tangible (abstracta) de las máquinas.
- **Artesano:** Objetos (en el caso de software, abstractos) imprimiendo un sello personal (cada uno la hace de una manera). Proceso repetible y cuantificable.
- **El día a día de un Ingeniero del Software:**
    - Revisar mediante un café, ritmo sostenible y no trabajar extra.
    - Reunión diaria (Daily Stand Up).
    - Escribir código individualmente, dirigiendo el trabajo y comprobando que siga las pruebas.
    - Programar en parejas, aprender y ayudar aun compañero , resolviendo problemas.
    - Revisión de cambios, lo que no funciona o cumple las pruebas se desecha, por ello se hace trabajo dirigido por pruebas, mediante pequeños incrementos.
- **Ingeniería del Software:** Una disciplina relativa a todos los aspectos de la producción de software. No se puede tener el control total, ya que la mayoría de proyectos de software fallan.
    - **Profesión:** Para que sea una profesión debe de haber una educación especial, que debe estar certificada por una institución como son EURACE en Europa o ABET en EE. UU. También hay empresas que certifican profesionales como Microsoft o Scrum Master y colegios profesionales como el IEEE y ACM en Europa, o el CPIICM en Madrid, y se busca que otorgue ventajas o responsabilidades exclusivas, para estar más reglados.
    - Long life learning: También hay cursos para la formación continuada que otorgan certificación.
    - Debe ser capaz de desarrollar tareas que una persona no cualificada no puede llevar a cabo.

- **Código Ético:** Conjunto de principios/valores sobre los que regir nuestro comportamiento profesional, son más como recomendaciones, no son reglas por los que no son de obligado cumplimiento, ni se nos dice que está bien y que está mal. Se utiliza la mezcla de los códigos éticos de ACM y IEEE. Van a primar por el beneficio de los miembros afectados.

    - No hay algoritmos simples para estas decisiones. Nunca es sencilla la decisión.
    - Los principios pueden colisionar entre ellos y habrá que elegir los más importantes.
    - Los 8 principios más importantes:
        - **Interés público:** Tenemos que certificar o autorizar software del que tenemos confianza de que no falla, pero nunca podremos estar completamente seguros. Avisar de peligros actuales o potenciales para el público por su uso.
        - **Cliente y empleador**: Debe satisfacer sus intereses, siempre que sea consistente con el interés público. Debe ser honesto ante la práctica. Mantener la privacidad.
        - **Producto:** Deben desarrollar un producto profesional y hacer las pruebas, depuración y revisiones adecuadas del software, exhaustivamente.
        - **Juicio:** Mantener la integridad e independencia en su juicio profesional, no dejarse llevar en prácticas financieras engañosas (decir que era error del cliente y no nuestro).
        - **Gestión:** Promover un entorno de decisión ético, bien remunerados y no se puede castigar a aquel que exprese sus dudas éticas.
        - **Profesión:** Progresar en integridad y reputación de la profesión(no dar una imagen equivocada o mala), promover el conocimiento público de la profesión y ser meticuloso al establecer las características del software.
        - **Colegas:** Ser justo y proporcionar apoyo a sus colegas, no obligar a que seas necesario. Y acreditar el trabajo de otros adecuadamente. Pedir opiniones y aprender de ellas.
        - **Mi comportamiento(Yo):** Participar en programas de aprendizaje continuado para mejorar mis capacidades para desarrollar software seguro y fiable de la mejor manera documentándolo correctamente.

2. TEMA 2: PRÁCTICAS FACILITADORAS DEL DESARROLLO ÁGIL DE SOFTWARE.

- **Programación en parejas:** es la práctica por la que todo el código desarrollado es escrito por dos desarrolladores sentados frente a una única máquina de trabajo. Un miembro es el «conductor» que es el que controla el ratón y teclado, y el otro es el «observador» que controla los defectos y va pensado en alternativa y pruebas. Ambos roles se van intercambiando, y deben tener niveles de experiencia similares. Y debe haber respeto y no hacer constantes correcciones.

  - Se finaliza más rápido, se corrigen los errores más rápidos.
  - Más personas conocen el funcionamiento del código y han realizado pruebas.
  - Los diseños son de mayor calidad.
  - Posibles pasos:
    - Preparación: Definir como se va a llevar a cabo.
    - Trabajo individual.
    - Cierre del trabajo en pareja: Poner el código funcional y explicar como se lo hemos hecho, y ejecutar las pruebas correspondientes, para comprobar las funcionalidades. Ver si sigue la normativa.

- **Propiedad colectiva de Código:** Un mismo código que puede ser modificado por varios programadores simultáneamente.

  - Todos deben seguir el mismo Estándar de Código.
  - El código que hay integrado ha sido probado y se almacenan las pruebas.
  - Cuenta con un adecuado control de versiones.
  - Funcionamiento:
    - Se descarga el código a modificar del sistema de control.
    - Se descargan las pruebas necesarias para ese código.
    - Desarrollamos la nueva funcionalidad y realizamos las pruebas necesarias.
    - Si el código pasa las pruebas y cumple el estándar de codificación, lo subimos al sistema de control.
  - Facilita la difusión del conocimiento.
  - El código sigue estándares y tiene mayor calidad, y este mecanismo permite detectar errores y corregirlos.
  - El objetivo no es corregir el código de otros sin propósito o cuestionarlo.

- No se debe modificar el mismo código simultáneamente varias personas.

- **Normativas de Código:** Conjunto de reglas y recomendaciones sobre como se debe escribir, estructurar y definir el código, para que en un futuro se pueda escribir sobre el y saber que y como lo hace. Sin gastar tiempo en un futuro en reescribir el código.

  - Ayuda a construir programas correctos, entendibles y fáciles de mantener.
  - Las normas deben cumplirse obligatoriamente y las recomendaciones se aplican siempre en general, pero puede haber excepciones.
  - **Aspectos básicos** a tener en cuenta en una normativa:
    - Nombre y Codificación de Ficheros:
    - Organización de Ficheros: La aparición de la leyenda de derechos de autor, ...
    - Comentarios.
    - Secuencias.
    - Reglas para asignar nombre: Determinadas reglas de mayúsculas o minúsculas para determinado tipo de variable, archivo o parte del código. Camel, Pascal, solo mayúsculas o minúsculas... Nombres en ingles, cortos pero representativos.
  - **Aspectos avanzados:**
    - Gestión de errores y excepciones: Si hay un error sacar una excepción, dar información auxiliar sobre el error y que aparezcan en la aplicación principal.
    - Seguridad.
    - Patrones de diseño.
    - Rendimiento.
    - Globalización.
    - Fiabilidad de mantenimiento.
    - Otras buenas prácticas.

- **Integración continua y automatizada:** Cada vez que se genera una nueva función o porción de código se integra con el código que ya se había generado y probado anteriormente. Se construye incrementalmente la funcionalidad, en vez de hacer todo por separado y juntarlo más tarde. Y cada vez que se integre una parte se debe probar la totalidad.

  - Antes de introducir nuevas funcionalidades se debe comprobar que funcione correctamente el código previo, por lo que hay que dar a conocer el estado de la integración.

- No se harán nuevas integraciones en poco tiempo, excepto si las pruebas se pueden hacer rápido o se codifica en parejas.
- **Características**:
  - Localización centralizada del código fuente.
  - Un único comando para compilar y enlazar los ejecutables.
  - Soporte para automatizar las pruebas.
  - Todos pueden acceder a un ejecutable confiable del sistema.
- **Beneficios**:
  - Se reducen los riesgos técnicos.
  - Se reduce el pesado proceso de integrar todo en un solo momento, de esta manera se empieza a integrar desde el inicio.
  - Los errores se resuelven en el primer momento que se producen.
- **Desventajas**:
  - El coste de poner a punto todo el sistema, configurar, mantener e integrar la plataforma.
  - Mantenimiento de los scripts de configuración.
  - Dificultad de incluir código preexistente.
- Hay herramientas que facilitan esta tarea como Jenkins, Bamboo, Cascade...
- **Cuando se está llevando a cabo una integración correcta?**
  - La versión más actual está en el repositorio, nadie tiene una visión más actualizada.
  - La versión del repositorio supera todas las pruebas sin errores, y estas están registradas para que cualquiera las pueda ejecutar.
  - Todo el mundo conoce el estado del código, para saber si pueden trabajar con el o no.
  - Ejecutable en repositorio de código.
  - El proceso está automatizado y no necesitará intervención humana.

3. TEMA 3: PRINCIPIOS DEL DESARROLLO DIRIGIDO POR PRUEBAS.

- Probar todo lo que puede llegar a fallar, utilizando pruebas automatizadas.
- **Principios básicos:**
  - El código se comparte y se puede modificar rápidamente, para ello hay que asegurarse de que no falle. Hay que probar todas las clases.
  - Escribir las pruebas antes que el código. Prueba un poco, codifica un poco. Las pruebas deben mantenerse, no sé usar y tirará, y almacenarse con el código fuente.
  - Todo el código que está en el repositorio debe estar probado, y debe funcionar cuando nos lo descargamos y en cada paso que realicemos debemos ejecutar las pruebas. Y cuando hemos terminado y todo funciona las subimos junto al código fuente.
  - Solo se publica código que ha superado todas las pruebas. Eso aumenta la percepción de seguridad.
- La unidad básica para probar es el **método** y se llaman **Pruebas unitarias.**
- **Niveles de pruebas de software:**
  - **Pruebas unitarias:** Prueban las clases y métodos. Verifican la unidad más pequeña de software, el método. XUNIT
    - El nombre debe recordar a la clase que se va a probar.
    - Primero escribir la prueba, y si el código falla, corregir el código fuente y repetir la prueba. Cuando se superan las pruebas se puede registrar el código y las pruebas.
    - **Tipo de pruebas**:
      - **Funcionales o Caja negra**: No se conoce la estructura que quiere probar. Se centra en las entradas y salidas.
      - **Estructurales o Caja blanca**: Se conocen la estructura y se pueden probar todos los caminos. Se centra en la estructura interna.
  - **Pruebas de integración:** Probar la relación entre las clases. XUNIT y Maven.
    - Primero se escribe los casos de prueba de nuevas funcionalidades a desarrollar.
    - El código no la supera todavía, porque no está escrito, y debemos escribirlo teniendo una idea precisa del código funcional. El código debe ser el más simple posible que permita superar las pruebas codificadas.

        - El código se refactoriza para que cumpla las reglas y recomendaciones del estándar.
        - Tras comprobar que todo funciona se puede publicar el código con las pruebas.
        - **Estrategias**:
            - **Top-Down**: Se empieza por el más complejo y se continúa con las que dependen de el. Se desciende por la jerarquía.
            - **Bottom-Up**: Empieza por la clase base, y va subiendo en las que dependen de el.
    - **Pruebas de sistema:** Formalización y automatización de casos de pruebas.
    - **Pruebas de aceptación:** Las que se llevan al cliente a aceptar el código.

- **Dificultades y recomendaciones:**

    - Cuesta el cambio de cultura, cuando no se está acostumbrano a escribir primero las pruebas ente que el código.
    - Necesidad de cambio en las rutinas del equipo.
    - Trabajar el código en pequeños incrementos, que pueden resolverse en poco tiempo.

- **Beneficios**:

    - Permite centrarse en los requisitos que se debe satisfacer antes de empezar a escribir el código.
    - Mantener el código simple y fácil de probar entender y modificar, ya que está dividido en pequeños pasos con sus propias pruebas.
    - Proporciona documentación acerca de como funciona el sistema que estamos intenso desarrollar y que se encuentra registrado en el código fuente.

- LEER PREGUNTAS FRECUENTES EN TEMA 3.

- **Hay herramientas de grabación y reproducción** que graban una secuencia de pasos en la interfaz de usuario y determina los resultados que se deben conseguir después de cada paso. Para cada paso hay que definir lo que debe encontrar, y cuando en un paso no se cumple ha fallado la prueba.

- **Hay herramientas para ejecutar pruebas de sistema** como JUnit, que tiene un entorno que permite automatizarlas, pero también se pueden hacer mediante ficheros o hojas de cálculos con los resultados esperados y un programa que los lea y haga la prueba para cada entrada/

4. TEMA 4: PRUEBAS FUNCIONALES PA Y BVA

- **Tipos de errores:**
  - Cálculo.
  - Lógica: Definición incorrecta de una condición.
  - Entrada/Salida: Descripción incorrecta, mala conversión o formato inadecuado.
  - Transformación de datos: Incorrecto acceso o transición de datos.
  - Interfaz: Comunicación incorrecta con otros componentes.
  - Definición de datos.
- Para realizar las pruebas del software es necesario acceder a especificaciones del componente, el código fuente y código objeto. Eso nos permite ver todas las posibles combinaciones entre los elementos que hay que probar.
- **Pruebas Unitarias:** Verifican la unidad más pequeña de software, el método.
  - **Pruebas funcionales:** No conocemos el código fuente, ya que hemos escrito las pruebas, pero todavía no hemos escrito el código. Por lo que no se pueden probar todos los casos.
  - **Pruebas estructurales:** Conocemos el código fuente, por lo que podemos probar todos los casos.
- **Pruebas de Caja Gris**: Se tiene acceso a la estructura interna de datos y algoritmo con el propósito de definir los casos de prueba. Útiles para identificar clases de equivalencia y valores límite. No tenemos el código, pero tenemos idea de como funciona.
- **Pruebas funcionales o de Caja Negra:** Tratan de reducir el número de casos de prueba a un nivel fácil de gestionar. Manteniendo una cobertura razonable. Se pueden usar clases de equivalencia.
  - **Clases de equivalencia:** Agrupa varias pruebas con valores que se procesan de la misma manera o deberían proporcionar el mismo resultado. Todos prueban el mismo procesamiento, si una prueba detecta un error el resto también lo hará. Hay que considerar:
    - **Clases válidas:** Casos de procesamiento normal del método. Un caso de prueba puede considerar varias válidas, se pueden englobar.

- **Clases inválidas**: Casos relacionados con situaciones de error. Por cada inválida a una clase de equivalencia.
- **Definir un caso de prueba:**
  - Identificados.
  - Valores de entrada, indicar un valor para cada parámetro de entrada y claves de equivalencia.
  - Resultados esperados.
- **Reglas identificar clases de equivalencia:**
  - **Rangos de valores continuos:** Identificar el límite inferior, superior y N particiones válidas.
  - **Valores discretos de un rango de valores permisibles:** Una clase válida y dos inválidas, una posibilidad inferior y otra superior.
  - Si el dato **no es un intervalo numérico:** Una clase válida para cada valor válido y otra no válida para el resto.
  - **Numero de valores de entrada:** Identificar el número mínimo y máximo, y elegir una clave válida y dos inválidas.
  - Otra aproximación para utilizar clases de equivalencia consiste en considerar las salidas.
- **Aplicabilidad y Limitaciones:**
  - Reduce significativamente el número de casos de prueba.
  - Es un sistema apropiado para valores incluido en rangos o en conjuntos preestablecidos.
  - Entradas o salida que se puedan particionar de acuerdo a requisito o precondiciones.

- **Valores en los límites:** Son muy importante, gran fuente de problemas. Primero hay que encontrar las clases de equivalencia.
  - Probabilidad de que los defectos sean más frecuentes en los valores límite.
  - Considera valores en los límite del intervalo, justo antes, en y justo después.
  - **Procedimiento**:
    - Identificar las clases de prueba.
    - Identificar los límites de cada clase de equivalencia.
    - Generar los casos de prueba para cada valor límite considerando las reglas.

        - **Reglas para identificar valores límite:**
            - **Valores límite para un rango continuo de entradas:** Considerar un valor antes, en y después del límite inferior, y un valor en y después del límite superior.
            - **Valores límite para un rango discreto de entrada**: Considerar el primero, segundo, penúltimo y último. O el más pequeño, el siguiente, el último y su anterior.
            - **Valores límite de las salidas producidas:** Aplicar la regla anterior pero con salidas.
        - **Aplicabilidad y limitaciones:**
            - Dificultad para formalizar el concepto de valores marginal y límite.
            - Este análisis es más intuitivo y requiere heurística(para tener un método).
            - Reduce significativamente el número de pruebas.
            - Está dirigido para valores dentro de rangos o conjuntos.
            - La entrada o salida se deben poder partición a y los limites identificar.

- **Análisis Sintáctico:** Solo se aplica para entradas, y cuando se pueden modelar como gramáticas.

    - Permite reducir el número de casos de prueba a un nivel fácil de gestiona mientras se mantiene una cobertura razonable.
    - **Aplicaciones y limitaciones:**
        - Reducen el número de casos de prueba, que se generan y ejecutan.
        - Está dirigido para **entradas que se pueden modelar como gramáticas**.
        - Se puede utilizar tanto para pruebas unitarias como de integración.
        - En pocos casos a nivel de sistema y no se recomienda para pruebas de aceptación.
    - **Procedimiento**:
        - Definición de la gramática.
        - Creación del árbol de derivación.
        - Identificación de los casos de prueba.
        - Automatización de los casos de prueba.
    - **Definir una gramática:**
        - Debe ser de tipo 2 o tipo 3: Regular e independiente de contexto.
        - Un único símbolo no terminal a la izquierda.
        - No existan símbolos Lambda.

    - Las gramáticas recursivas son problemáticas porque el árbol de derivación asociado sería infinito.

- **Creación del árbol de derivación**: Se hace usando a gramática del paso anterior.
    - Cada símbolo terminal o no terminal será un nodo diferente.
    - Los nodos se numeran empezando por 1.
    - Debe de diferenciarse por niveles que nodos son terminales y no terminales.

- **Identificación de los casos de prueba:**
    - Se obtienen del análisis del árbol y se dividen en dos partes: entradas válidas e inválidas.
    - Para **identificar las entradas válidas**:
        - Se producen casos de prueba de tal forma que todos los nodos no terminales estén cubiertos.
        - Se repite el anterior hasta cubrir al menos una vez todos los nodos terminales.
    - Para **identificar entradas inválidas**:
        - Hay demasiadas por lo que se consideran una muestra significativa de las mismas.
        - Para los nodos no terminales se procede a su omisión y su adición. La adición de nodos puede producir un gran número de casos de prueba, siendo semánticamente más difícil de generar.
        - Para los nodos terminales debe procederse también a su modificación. Se simula mediante errores tipográficos, siendo aconsejable no someter a pruebas grandes combinaciones de errores. La explosión combinatoria sería enorme y la prueba poco realista.
    - RECORTAR Y MIRAR LAS DOS ÚLTIMAS DIAPOSITIVAS.

5. TEMA 5: PRUEBAS ESTRUCTURALES

Acceso en Drive a las diapositivas

6. TEMA 6: REFACTORING

Acceso en Drive a las diapositivas Acceso en Drive a las diapositivas

7. TEMA 7: DISEÑO SIMPLE Y PATRONES

Acceso en Drive a las diapositivas