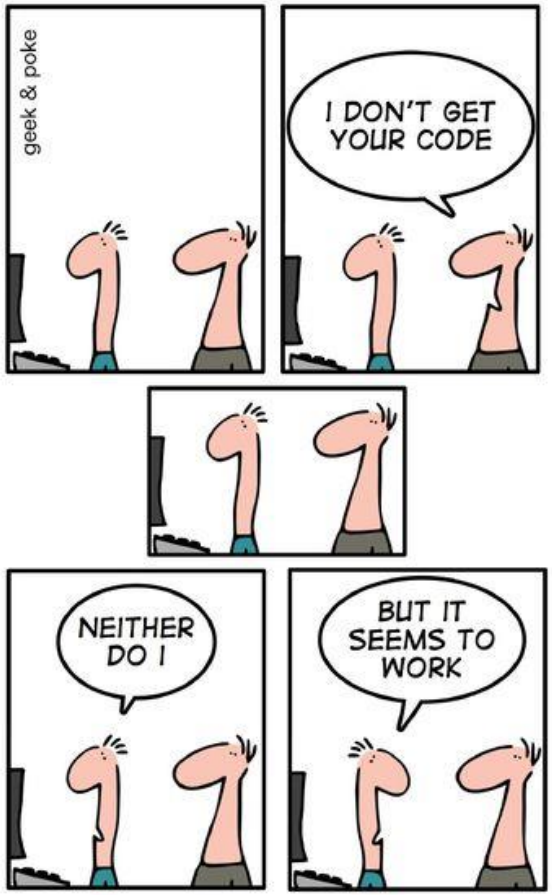
Tema 4
Análisis estático de código

EI1031 – Verificación y Validación
Grado en Ingeniería Informática

Ramón A. Mollineda Cárdenas

análisis estático de código

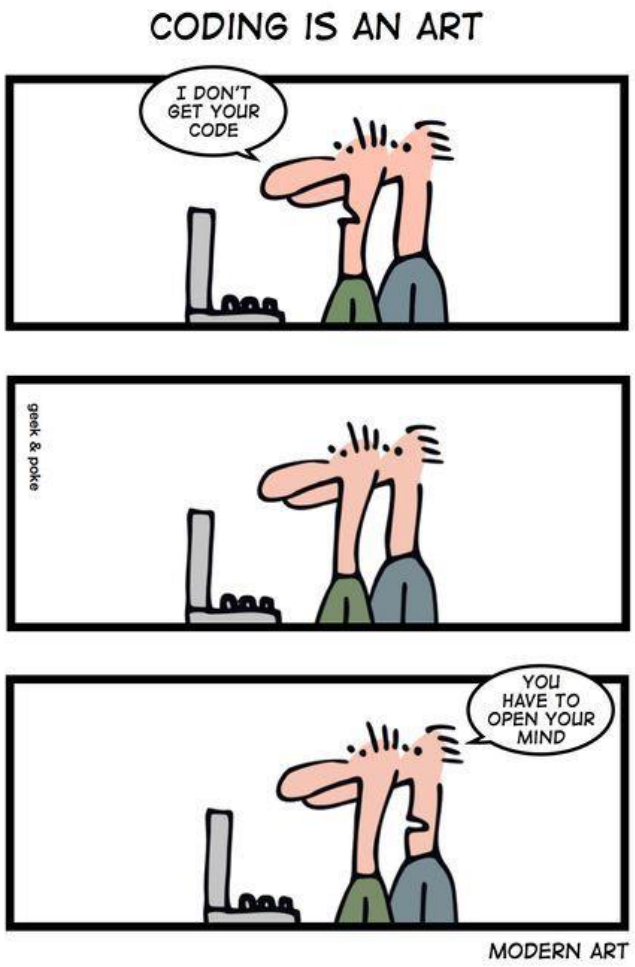
el pan nuestro de cada día …



Inspired by a Slashdot post:
http://tech.slashdot.org/article.pl?sid=08/02/04/1710209

análisis estático de código

el pan nuestro de cada día …



análisis estático de código

otra cita …

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand".

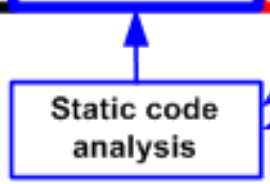
Martin Fowler

introducción

motivación

coste medio de eliminación de defectos según momentos de introducción y detección:

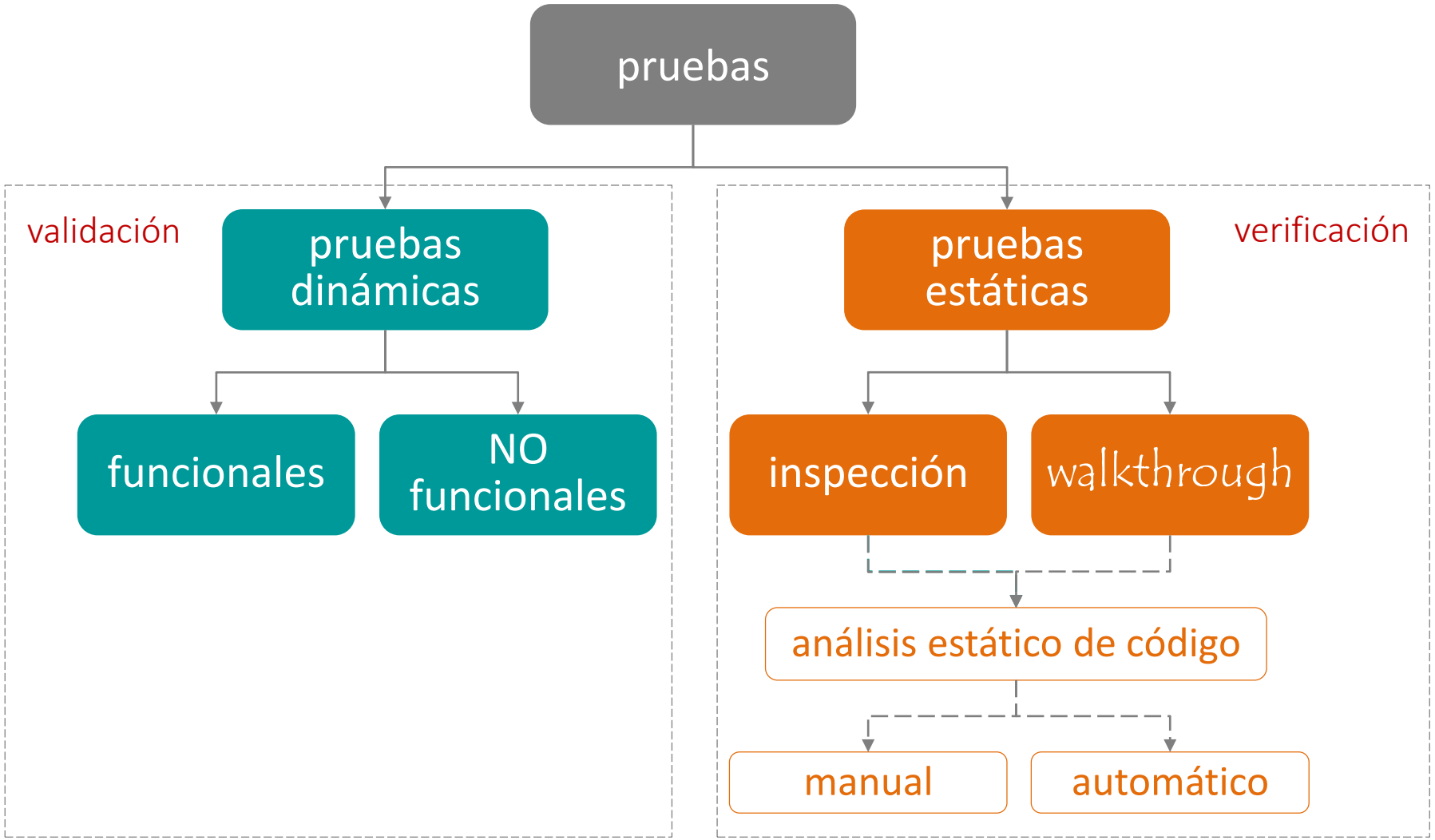
	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25



https://www.viva64.com/en/t/0046/

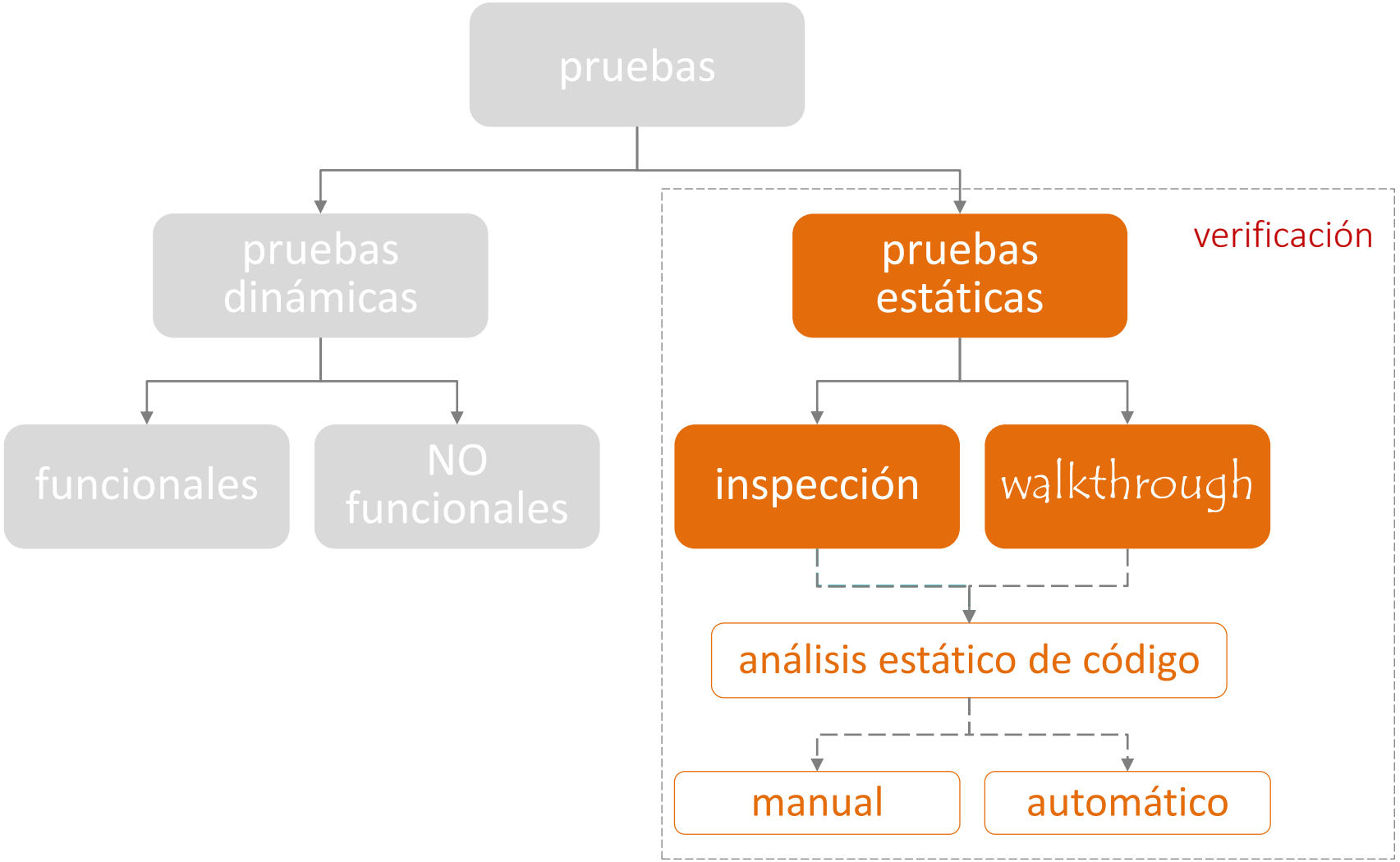
introducción

contexto



introducción

índice de contenidos



lo que ya sabemos …

verificación

verificación:

¿estamos desarrollando el producto correctamente?

objetivos:

- identificar defectos tan pronto como sea posible
- evaluar si productos se ajustan a especificaciones/documentación
- evaluar calidad estructural de productos (código, diseño, pruebas, …)
- identificar oportunidades de mejora

verificación

ámbitos

- diseño: complejidad, buenas prácticas, corrección, etc.
  - abstractness: número de clases abstractas en relación al total
  - coupling (package's responsibility): número de clases de otros paquetes que dependen de clases de un paquete de referencia (r)
  - coupling (package's dependence): número de clases de un paquete de referencia que dependen de clases de otros paquetes (d)
  - instability = d / (r + d); d = 0, paquete estable; d = 1, paquete inestable
- código:

  …

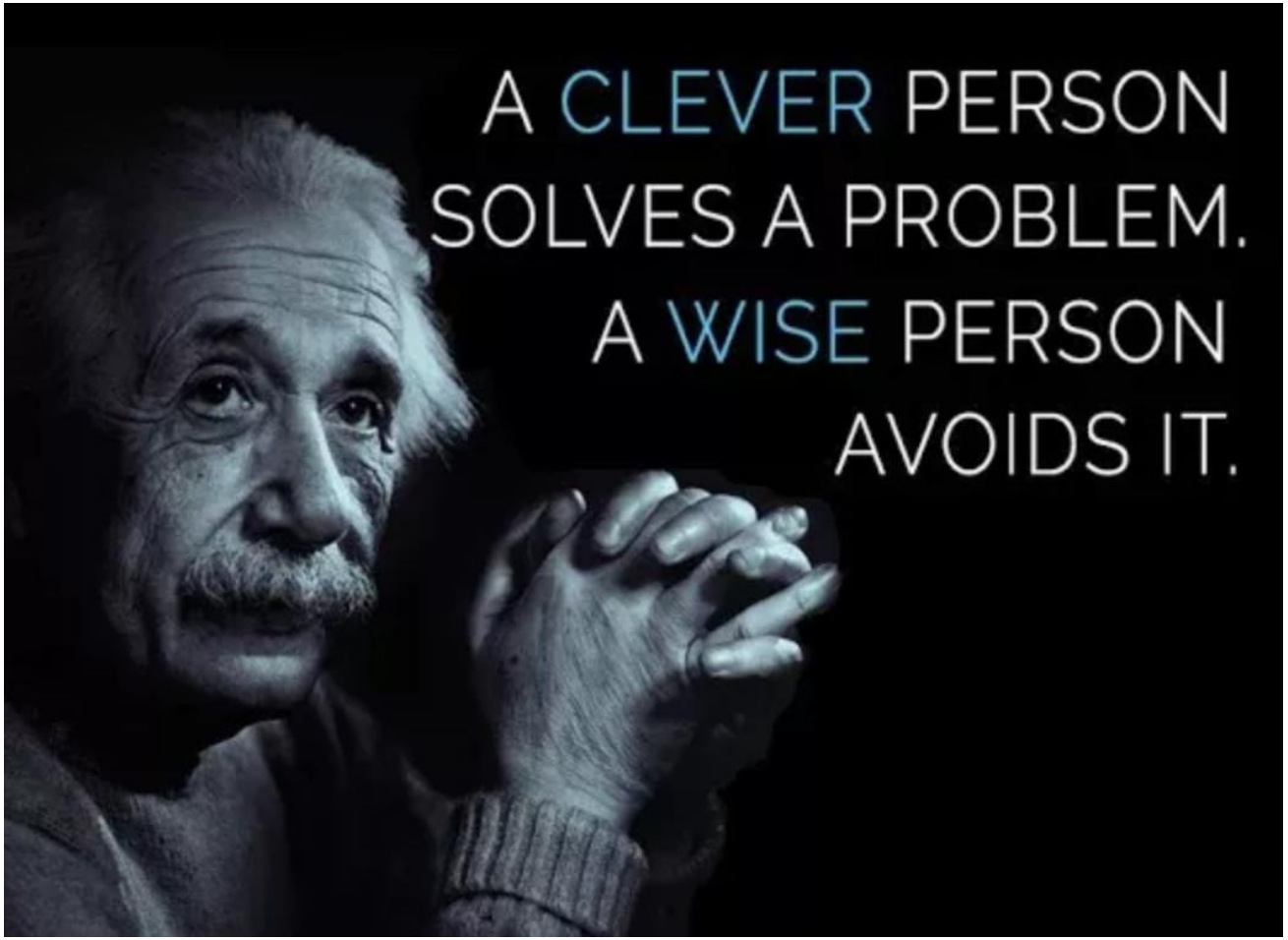
análisis estático de código

generalidades (i): qué es

análisis estático de código es la lectura y análisis sistemático de código fuente, **SIN** ejecución de código.

análisis estático de código

un sabio dijo una vez …



https://i.pinimg.com/originals/27/f9/17/27f9175650dd9775dd724c8055205426.webp

análisis estático de código

generalidades (i): qué es

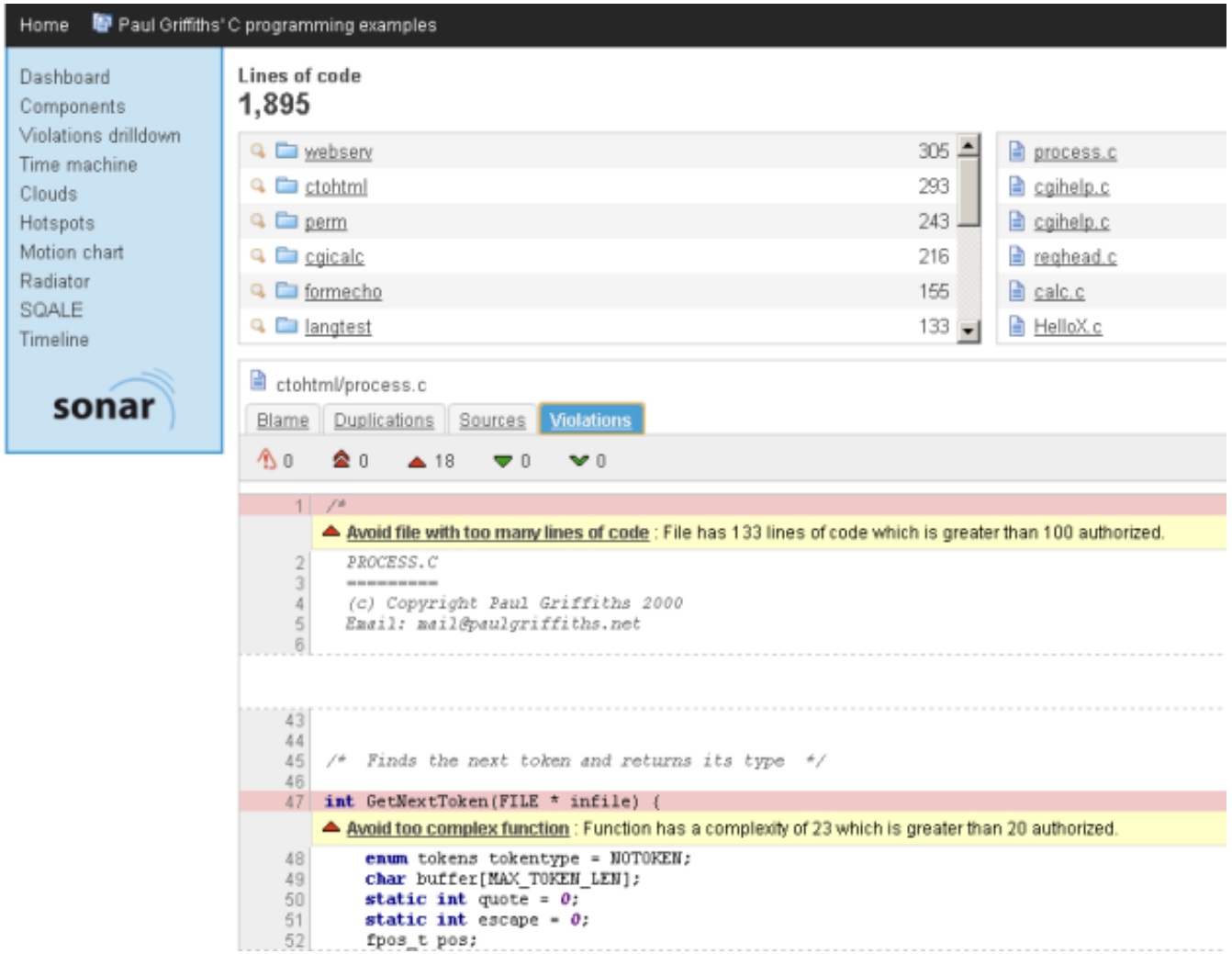
análisis estático de código

generalidades (ii): modos

- manual:
  - realizado por humanos, generalmente un equipo.
- automático:
  - mediante aplicaciones en IDE y procesos de Continuous Integration
  - proporcionan realimentación inmediata
  - detección temprana de vulnerabilidades

análisis estático de código

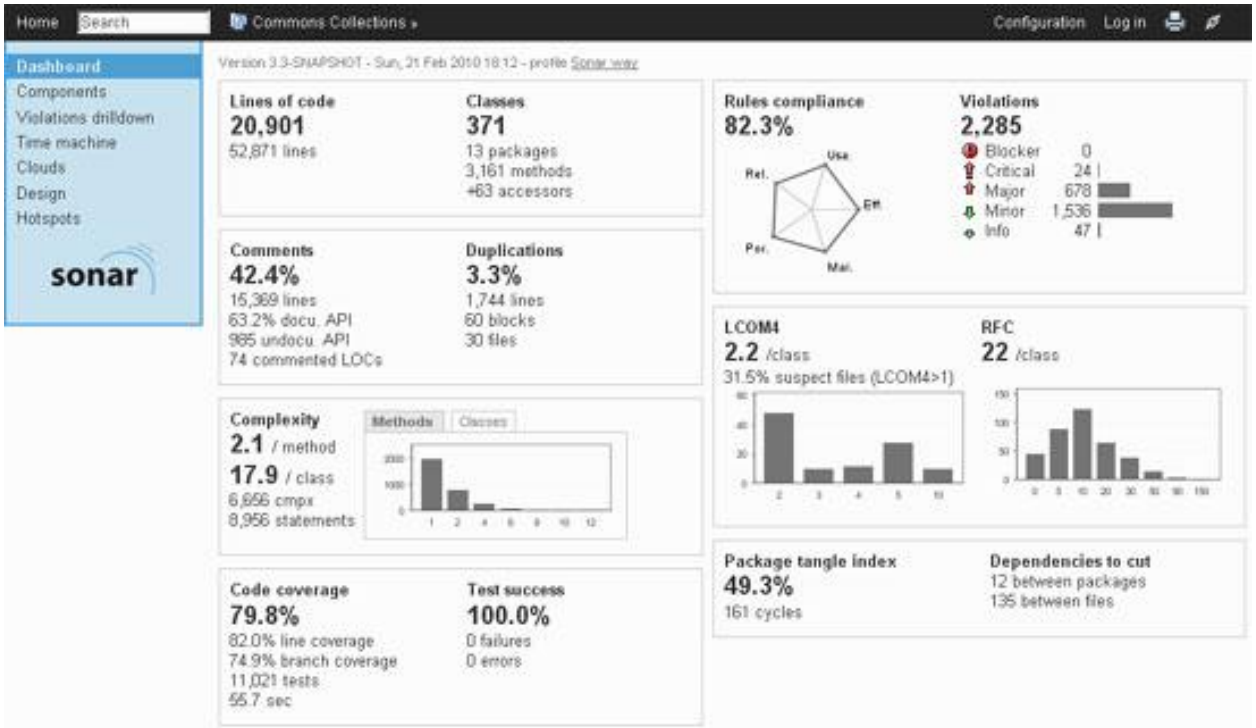
generalidades (ii): ejemplo modo automático



Fuente: Static Code Analysis of LoadRunner C Code. Seven Seconds (enlace).

análisis estático de código

generalidades (ii): ejemplo modo automático



Fuente: Olivier Gaudin, Freddy Mallet. What is Sonar? Methods and tools (enlace).

análisis estático de código

beneficios potenciales (i)

beneficios del análisis estático de código para el código en desarrollo:

- detección temprana de errores de codificación.
- conducen a procesos de depuración (debugging) poco costosos: la detección, localización y análisis del error se realizan conjuntamente.
- al ser procesos analíticos, detectan “lotes” de errores.
- detectan entre el 30 y el 70% de los errores que finalmente se encuentran.
- son complementarios con las pruebas sobre código ejecutable (validación).
- sus beneficios aumentan sobre código modificado.
- proceso de naturaleza más relajada que la evaluación de código en ejecución; es muy efectiva la corrección de errores detectados.

análisis estático de código

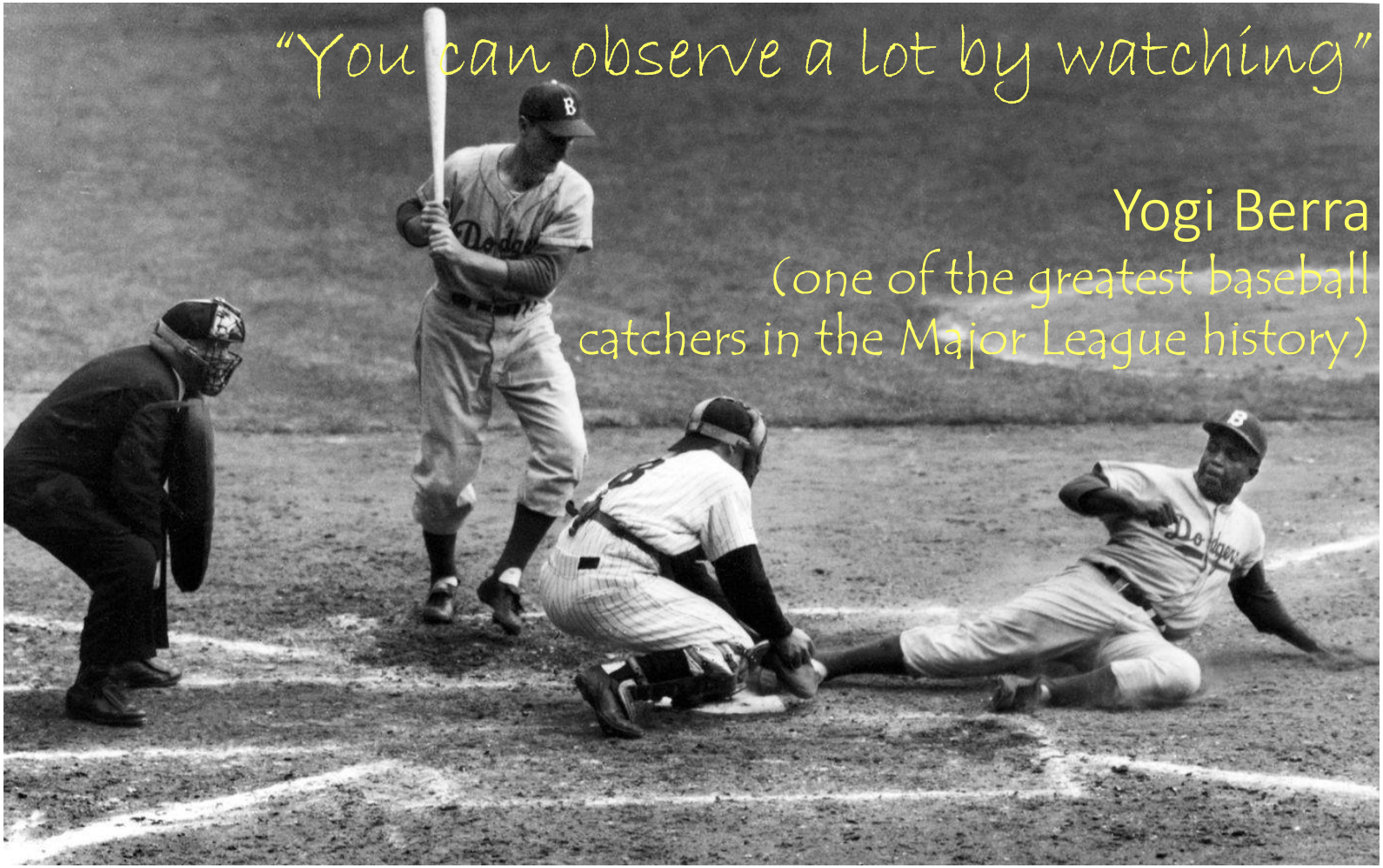
beneficios potenciales (ii)

beneficios del análisis estático de código para el proyecto
(*asume uso de herramientas automáticas*)

- desarrollo más rápido de los proyectos
- mejor código fuente
- permite optimizar estrategias de test (dedicar más recursos a secciones más propensas a error)
- ciclos de desarrollo menos costosos
- menor time-to-market

análisis estático de código

cita



análisis estático de código

métodos (manuales)

inspecciones de código

revisión formal del código; análisis de lógica y estructuras de datos mediante lectura ordenada del código.

walkthroughs (test data walks through the logic)

ejecución mental de casos de prueba; análisis de la lógica del programa a través de la ejecución mental de casos de prueba.

inspecciones de código

generalidades

objetivo:

detección de errores y áreas de mejora (legibilidad, eficiencia, diseño, etc.)

técnica de detección de errores:

lectura sistemática del código en grupo; análisis colectivo del código

composición del grupo:

- moderador: programador experto; planifica y dirige la sesión; distribuye material; registra errores; supervisa corrección (*quality control*)
- programador (del código bajo análisis)
- diseñador (del software al que pertenece el código bajo análisis)
- especialista en pruebas (de software)

inspecciones de código

preparación

pre-condiciones:

- el moderador distribuye especificaciones de diseño y listado del programa.
- los participantes se familiarizan con estos materiales.
- los participantes están familiarizados con los estándares de la organización.

recomendaciones sobre hora, lugar y duración:

- planificar sesión en lugar y momento que eviten interrupciones externas.
- prácticas mentalmente agotadoras; sesiones largas son menos productivas.
- duración óptima de una sesión: 90 – 120 minutos.
- productividad media: 150 sentencias / hora.
- grandes programas deben ser analizados en varias sesiones/inspecciones; en cada una pueden examinarse uno o varios módulos o subrutinas.

inspecciones de código

ejecución

el moderador debe garantizar …

- discusiones productivas.
- cumplimiento de objetivo: detección y análisis de errores, no su corrección.
- acta de la reunión con documentación de errores, propuestas, acuerdos, etc.

actividades de la sesión:

- presentación del código (por su programador), sentencia a sentencia; los participantes intervienen, discuten cuestiones, detectan errores.
- análisis contra checklist de errores y malas prácticas frecuentes.

después de la sesión:

- el programador corrige errores; si es necesario, se planifica nueva sesión.

inspecciones de código

actitudes y prácticas apropiadas

escenario negativo

el programador entiende la inspección como un ataque a su trabajo =>

… el programador adopta postura defensiva =>

… la inspección NO será efectiva.

escenario positivo:

el programador deja su EGO "fuera" de la sesión (egoless attitude) =>

… el programador se muestra humilde, receptivo, actitud constructiva =>

… inspección efectiva =>

… mejora calidad del producto + mejora de capacidades personales.

buena práctica (muy recomendable):

firmar acuerdo de confidencialidad sobre resultados de sesión; de su cumplimiento depende éxito de futuras inspecciones.

inspecciones de código

beneficios colaterales

beneficios colaterales; las inspecciones …

… proporcionan realimentación (feedback) a los programadores sobre su estilo y prácticas de programación, incluyendo esquemas algorítmicos.

… permiten al resto de participantes aprender del estilo de programación y de los errores de otros.

… fomentan el uso de buenas prácticas de programación en la organización.

lista de errores comunes o *error checklist*

introducción

- objetivo: examinar código contra lista de errores y malas prácticas.
- categorías de defectos frecuentes en Java:
  - Especificación y diseño.
  - Declaración e inicialización de atributos, variables, constantes.
  - Definición e invocación de métodos.
  - Definición de clases.
  - Referencias a datos, arrays.
  - Operaciones.
  - Comparaciones.
  - Entrada/Salida.
  - Estructuras de control.
  - Excepciones.
  - Comentarios.
  - Empaquetado.
  - Rendimiento.

lista de errores comunes o error checklist

defectos en especificación y diseño

defectos en especificación y diseño:

- ¿Se implementa la funcionalidad descrita en la especificación?
- ¿Se implementa sólo la funcionalidad descrita en la especificación?
- ¿Contiene el código bonehead (brutish) programming?
  - `Math.pow(x,2)` … ¿o `x*x`?
  - `if (size>0) return true; else return false;`
- ¿Está el código libre de bad smells (código duplicado, métodos largos, clases grandes, información no encapsulada, variables globales, etc.)?
- ¿Existe código repetido que pueda sutituirse por llamadas métodos?
- ¿Existe bajo nivel de acoplamiento entre métodos y clases?
- ¿Se usan las Java class libraries en el momento y lugar adecuados?
- ¿Es el código correcto? (verificar corrección con traza manual).

lista de errores comunes o error checklist

defectos en declaración e inicialización

defectos en declaración e inicialización de atributos, variables, …:

- ¿Es cada variable declarada en el ámbito (scope) apropiado?
- ¿Es cada variable o atributo de tipo correcto?
- ¿Está cada atributo afectado por un modificador de acceso apropiado?
- ¿Se inicializa cada variable, o referencia a objeto, antes de ser usada?
- ¿Es descriptivo el nombre de cada variable o constante? ¿Se ajusta a convenciones de nombres?
- ¿Existen variables o atributos con nombres muy similares?
- ¿Son todas las variables de for-loop control declaradas en el loop header?
- ¿Existen literales en el código que deban ser convertidos en constantes?
- ¿Existen atributos que deban ser variables locales?
- ¿Existen atributos estáticos que no deberían serlo o viceversa?
- ¿Se invoca un constructor cada vez que se desea un objeto nuevo?
- ¿Se usan variables globales en los módulos (e.g. atributo público y estático)?

lista de errores comunes o *error checklist*

defectos en definición e invocación de métodos

defectos en definición e invocación de métodos:

- ¿Coinciden el número, orden, tipos y valores de parámetros de cada llamada con la declaración del método?
- ¿Es correcta cada invocación de un método o debió invocarse otro?
- ¿Son usados correctamente los valores de retorno de los métodos?
- ¿Es descriptivo el nombre de cada método? ¿Se ajusta a convenciones?
- ¿Se comprueba cada valor de parámetro antes de ser usado en un método?
- ¿El sistema de unidades de un parámetro se corresponde con el sistema del argumento enviado? Ejemplo: un ángulo en grados o en radianes, …
- ¿Se modifica algún parámetro cuyo rol era ser sólo un dato de entrada?
- ¿Es correcto cada valor de retorno?
- ¿Está cada método afectado por un modificador de acceso apropiado?
- ¿Existen métodos estáticos que no deberían serlo o viceversa?

lista de errores comunes o *error checklist*

defectos en definición de clases

defectos en definición de clases:

- ¿Tiene cada clase constructores apropiados?
- ¿Existen miembros comunes en subclases que deben estar en la superclase?
- ¿Puede simplificarse la jerarquía de clases?
- ¿Existen roles genéricos no identificados susceptibles de ser modelados como interfaces?

lista de errores comunes o error checklist

defectos en referencias a datos y arrays

defectos en referencias a datos y arrays:

- ¿Existen off-by-one errors (OBOE) al indexar arrays?
- ¿Existen mecanismos de prevención de índices fuera de rango al indexar arrays u otras colecciones?
- ¿Se invoca un constructor cada vez que se desea un nuevo ítem de array?
- ¿Es cada referencia a objeto o array distinta de null?

lista de errores comunes o error checklist

defectos en operaciones

defectos en operaciones:

- ¿Es correcta la precedencia de operadores en cada operación?
- ¿Existen mecanismos de prevención de denominadores nulos?
- ¿Se usa adecuadamente la aritmética de enteros (en particular la división), evitándose redondeos y truncamientos indeseados?
- ¿Han sido simplificadas las expresiones lógicas moviendo las negaciones lo más internamente posible?
- ¿Existen cálculos con tipos de datos mixtos?
- ¿Está el código libre de conversiones implícitas de tipos?
- ¿Es posible overflow o underflow como resultado de un cálculo?
- ¿Se usan paréntesis para evitar ambigüedades en el orden de evaluación?
- ¿Existen operadores '&' y '|' incorrectos en lugar de '&&' y '||'?

lista de errores comunes o error checklist

defectos en comparaciones

defectos en comparaciones:

- ¿Son correctos los operadores de comparación? Es frecuente confundir relaciones como “at most”, “at least”, “greater than”, “not less than”, etc.
- ¿Se realiza cada comparación de objetos (incluyendo strings) con equals y no mediante “==“?
- ¿Es el sentido de cada comparación correcto?
- ¿Existen efectos secundarios indeseados en una comparación?

lista de errores comunes o error checklist

defectos en entradas/salidas

defectos en entradas/salidas (I/O):

- ¿Se abren todos los ficheros antes de ser usados?
- ¿Se cierran todos los ficheros correctamente, incluso en caso de error?
- ¿Se detectan y gestionan adecuadamente las EOF conditions?
- ¿Se gestionan adecuadamente las excepciones de I/O?
- ¿Son comprensibles los mensajes de error? ¿Proponen soluciones?
- ¿Existen errores gramaticales u ortográficos en textos que deben ser impresos o mostrados en pantalla?
- ¿Es correcto el formato de la información de salida?

lista de errores comunes o error checklist

defectos en estructuras de control

defectos en estructuras de control:

- ¿Es cada tipo de bucle la mejor opción?
- ¿Están correctamente definidos todos los bucles, incluyendo expresiones apropiadas de inicialización, incremento y terminación?
- En casos de bucles con múltiples puntos de salida, ¿son todos necesarios?
- ¿Existen anidamientos profundos de bucles? ¿Son necesarios y correctos?
- ¿Existen estructuras de control vacías? ¿Están debidamente comentadas?
- ¿Terminan todos los bucles?
- ¿Es posible que un bucle nunca se ejecute?
- ¿Incluyen todos los cases de cada switch sentencias break o return? En caso de case sin break, ¿está debidamente comentado?
- ¿Incluye cada switch una opción default?

lista de errores comunes o error checklist

defectos en comentarios

defectos en comentarios:

- ¿Existe un comentario de encabezado apropiado para cada método, clase y fichero?
- ¿Existe un comentario apropiado para cada atributo, variable y constante?
- ¿Es el comportamiento de cada método y clase descrito en lenguaje simple?
- ¿Es completo cada comentario, incluyendo DbC y error checking specs?
- ¿Es cada comentario consistente con el código que describe?
- ¿Ayuda cada comentario a entender el código relacionado?
- ¿Faltan comentarios?
- ¿Sobran comentarios?

lista de errores comunes o *error checklist*

defectos en empaquetado

defectos en empaquetado:

- ¿Existe una indentación estándar? ¿Se usa consistentemente?
- ¿Es la longitud de cada método no mayor que 60 líneas aproximadamente?
- ¿Es la longitud de cada módulo no mayor que 600 líneas aproximadamente?
- ¿Existe bajo nivel de acomplamiento entre paquetes?

lista de errores comunes o error checklist

defectos en excepciones

defectos en excepciones:

- ¿Se capturan todas las excepciones relevantes?
- ¿Se adoptan acciones apropiadas en cada catch block?
- Las excepciones capturadas, ¿son suficientemente específicas?

lista de errores comunes o *error checklist*

defectos en rendimiento

defectos en rendimiento:

- ¿Pueden usarse mejores estructuras de datos?
- ¿Pueden usarse algoritmos más eficientes?
- ¿Se organizan las decisiones lógicas de forma que las más simples y probables precedan a las más complejas e improbables?
- ¿Se recalcula un valor que podría guardarse tras un primer cálculo?
- ¿Se usa cada resultado calculado y guardado?
- ¿Puede moverse un cálculo fuera de un bucle?
- ¿Puede desdoblarse un bucle pequeño?
- ¿Podrían integrarse 2 bucles que operan por separado sobre el mismo dato?
- …

walkthroughs

generalidades

objetivo:

detección de errores y áreas de mejora (legibilidad, eficiencia, diseño, etc.).

técnica de detección de errores:

ejecución mental de casos de prueba simples; análisis colectivo del código.

composición del grupo:

- moderador: programador experto; planifica y dirige la sesión; distribuye material; supervisa corrección (quality control).
- secretario: registra errores.
- especialista en pruebas (de software).
- programador (del código bajo análisis)
- otro programador, experto o novel (opcional).

walkthroughs

generalidades (ii)

- diferencia con las inspecciones: técnica de detección de errores.
- semejanzas con las inspecciones: objetivos, estructura de equipo, y criterios de organización.
  - reunión ininterrumpida de 1 o 2 horas de duración.
  - los materiales son repartidos con días de antelación.
  - la actitud de los participantes es muy importante: los comentarios deben hacer referencia al programa, no al programador
  - los beneficios son los mismos: localización y análisis de errores, identificación de regiones propensas a error, educación en técnicas, estilos, etc.

walkthroughs

ejecución

el moderador debe garantizar …

discusiones productivas; cumplimiento de objetivo.

actividades de la sesión:

1. el especialista en pruebas trae un conjunto pequeño de casos simples; el caso de prueba es un mero vehículo para analizar el código.
2. cada caso de prueba es mentalmente ejecutado; cada dato de prueba "se mueve a través de" (walks through) la lógica del programa.
3. el estado del programa (valores de variables) se mantiene en papel o pizarra.
4. discusión sobre elementos del programa; se detectan más errores de analizar el programa que de ejecutar casos de test.

después de la sesión:

el programador corrige errores; si es necesario, se programa nueva sesión.

análisis automático de código estático

herramientas populares para Java

Eclipse:

- PMD: detecta "código muerto", expresiones demasiado complejas, creación de objetos innecesarios, etc. Incluye plugin Eclipse.
- CPD (copy-paste-detector): detecta código duplicado.
- SonarLint: detecta código de mala calidad, errores potenciales.
- CheckStyle: verifica cumplimiento de reglas/estándares de codificación; configurable (Google Java Style, Sun Code Conventions, etc.)
- ejemplo uso de PMD y FindBugs en Java: https://www.youtube.com/watch?v=WptLYujK4Nk

IntelliJIDEA

- análisis estático de código integrado (ver detalles)

análisis automático de código estático

ejemplos de bugs detectables por FindBugs (*)

infinite recursive loop

```
public String resultValue() {
   return this.resultValue();
}
```

(*) *https://www.romexsoft.com/blog/improve-java-code-quality/*

análisis automático de código estático

ejemplos de bugs detectables por FindBugs (*)

Null Pointer Exception

```
Object obj = null;
obj.doSomeThing();

…

if((str == null && obj == null) || str.equals(obj)) {
   //do something
}
```

(*) *https://www.romexsoft.com/blog/improve-java-code-quality/*

análisis automático de código estático

ejemplos de bugs detectables por FindBugs (*)

method whose return value should not be ignored

```
String str = "Java";
str.toUpper();
if (str.equals("JAVA")){
   ...
```

(*) *https://www.romexsoft.com/blog/improve-java-code-quality/*

análisis automático de código estático

ejemplos de bugs detectables por FindBugs (*)

suspicious equal() comparison

```
Integer value = new Integer(10);
String str = new String("10");
if (str != null && !str.equals(value)) {
   //do something;
}
```

(*) *https://www.romexsoft.com/blog/improve-java-code-quality/*

análisis automático de código estático

limitaciones

limitaciones

- muchas vulnerabilidades son difíciles de detectar automáticamente (falsos negativos).
- detección de posibles vulnerabilidades que realmente no lo son (falsos positivos).

análisis estático de código

estándares, estilos (colección de buenas prácticas)

ventajas: mejora …

- legibilidad, calidad del código, reusabilidad, tiempo de desarrollo …

ejemplos de ámbitos sujetos a estándares:

- comentarios de Javadoc para clases, atributos y métodos.
- convenciones de nombres para atributos y métodos.
- límite en número de parámetros de función.
- límite en longitud de línea
- uso de imports
- espacios entre caracteres
- diseño de clases
- tamaños de métodos, clases, paquetes, etc.
- dependencias entre clases, paquetes, etc.

análisis estático de código

¿necesario?

industrias que han incorporado el análisis estático código como medio para mejorar la calidad de código complejo y crítico:

- software médico (embebidos en dispositivos médicos)
- software nuclear, i.e. en Sistemas de Protección de Reactores.
- software de aviación
- "The UK Defense Standard 00-55 requires that Static Code Analysis be used on all 'safety related software in defense equipment'" (Wikipedia, http://www.software-supportability.org/Docs/00-55_Part_2.pdf).

resumen

resumen

- la verificación “humana” del código, a pesar de su buena relación costes/beneficios, es generalmente poco considerada.
- consiste en la lectura de código, con el objetivo de identificar errores o fragmentos de código con baja calidad.
- los métodos Inspección y walkthrough han demostrado ser muy efectivos detectando errores de código.