INFORME DE PRÁCTICAS

Repositorio de proxecto: https://github.com/andreu-barro/VVS-DFA-JAVA
Participantes no proxecto: F. Javier Moure López, Emma Oitavén Carracedo, Xoan Andreu Barro Torre

Validación e Verificación de Software

1. Descrición do proxecto

Dicha aplicación simula el comportamiento de una máquina de estados

2. Estado actual

Las funciones que se ocupan de la funcionalidad de nuestra aplicación son las siguientes son las que serán evaluadas:

- Clase Alphabet

  void Alphabet()

  void Alphabet(int n)

  void addNewSymbol(Symbol symbol)

  getAlphabet : GenList¡Symbol¿

  getExistingObject(Symbol symbol) : Symbol

- Clase DFA

  DFA(GenList¡State¿states, Alphabet alphabet, State initialState, GenList¡State¿finalStates, GenList¡Transition¿transitions)

  getAllConnectedStates() : GenList¡State¿

  getConnectedDFA() : DFA

  getTransitionsTable() : String

- Clase State

  State(String state)

  getState() : String

- Clase Symbol

  Symbol(String symbol)

  getSymbol() : String

- Clase Transition

  void Transition(State startState, State endState, Symbol symbol)

  getEndState() : String

  getStartState() : String

  getSymbol() : Symbol

- Clase GenList

  void GenList()

  void GenList(int buffer)

  void add(T obj)

  void clearNulls()

  get(int n) : T

  getArray() : Object[]

getBuffer() : int

getExistingObject(T obj) : T

getSize() : int

void remove(int n)

El objetivo es probar que todas estas funciones funcionan correctamente, para ello aplicaremos pruebas de unidad, pruebas dinámicas, pruebas de rendimiento y chequearemos el estilo de programación. Después de aplicarle las herramientas de pruebas y utilizar las herramientas de validación como cobertura y PIT creemos que nuestras funciones son estables y que puede que tengamos una bastantes pruebas realizadas y el código testeado para fiarnos de él.

2.1. Compoñentes avaliados

Si ejecutamos *mvn test org.pitest:pitest-maven:mutationCoverage site* en la raiz del proyecto se generarán reports sobre los tests.

Los genera en target/site/Index.html, ahí navegamos a project reports.

3. Especificación de probas

3.1. Pruebas de unidad

Pruebas de unidad

3.2. Pruebas basadas en propiedades

Pruebas basadas en propiedades

3.3. Validación de calidad de las pruebas

Cobertura) Mutation Testing: PIT)

3.4. Pruebas dinámicas de unidad

Pruebas dinámicas de unidad

3.5. Pruebas no funcionales

Hemos probado todas los métodos de todos las clases que tenemos (Alphabet, DFA, GenList, State, Symbol y Transition)

Nos hemos ayudado de JETM y el modelo que hemos utilizado es este: Pruebas no funcionales

3.6. Pruebas estructurales

CheckStyle)

4. Registro de pruebas

4.1. Pruebas unidad: JUnit

JUnit se utiliza para realizar pruebas unitarias sobre nuestra aplicación, nos sirven para encontrar errores y solventar los problemas en la programación de forma manual. Se realizan pruebas de todas las funciones implementadas en la aplicación. Los casos de prueba se implementan manualmente como parte de funciones de prueba, utilizamos la directiva assertEqual(Expected, Expr).

4.2. Pruebas basadas en propiedades: Quickcheck

QuickCheck es una herramienta para generar automáticamente y ejecutar casos de prueba aleatorios, basados en especificaciones de propiedades. Es decir, ejecutar pruebas con esta herramienta, significa instanciar las propiedades n veces. Las pruebas se detienen al encontrar un caso concreto en el que la propiedad no se cumple (contraejemplo).La ejecución con éxito significa que ninguno de los casos generados incumplió la propiedad Se crean generadores de:

- GeneradorAlphabet
- GeneradorDFA
- GeneradorState
- GeneradorSymbol
- GeneradorTransition
- GeneradorGenList

La programación de generadores para aplicar Quickcheck se realiza en el siguiente paquete: es.udc.fic.vvs.vvsproject.generadorTest

4.3. Validación de calidad de las pruebas: Cobertura

Cobertura(Plugin cobertura) es una herramienta libre (GPL) escrita en Java, que nos permite comprobar el porcentaje de código al que accedemos desde los test. Es decir, Cobertura nos permite saber cuanto código estamos realmente probando con nuestros test. De esta forma Cobertura se convierte en una potente herramienta de trabajo, ya que lo podemos usar como medida de calidad (mientras más código tengamos probado, más garantías tenemos de que podemos hacer refactorizaciones sin peligro). Nuestro objetivo es llegar a cobertura cercana a 100.

4.4. Pruebas dinámicas de unidad: Mockito

Para poder crear un buen conjunto de pruebas unitarias, es necasario que nos centremos exclusivamente en la clase a testear, simulando el funcionamiento de las capas inferiores (pensad por ejemplo en olvidarnos de la capa de acceso a datos, DAO). De esta manera estaremos creando test unitarios potentes que os permitiría detectar y solucionar los errores

que tengáis o que se cometan durante el futuro del desarrollo de vuestra aplicación. Para esta tarea nos apoyaremos en el uso de mock objects, que no son más que objetos que simulan parte del comportamiento de una clase, y más especificamente vamos a ver una herramienta que permite generar mock objects dinámicos, mockito.

Pruebas que se realizarán en el paquete es.udc.vvs.dfa.mockito, sólo se realizarán las pruebas del servidor, debido a que las pruebas de componentes ya son pruebas de unidad.

4.5. Pruebas no funcionales: JETM

JETM permite realizar pruebas de rendimiento y comprobar la velocidad de ejecuciones al implementar unas pruebas con n iteraciones. Se pueden generar test con múltiples iteraciones para detectar problemas de rendimiento.

Pruebas que se realizarán en el paquete es.udc.vvs.dfa.rendimiento

4.6. Validación de calidad de las pruebas (mutación testing): PIT

El método que utilizaban en el sistema para realizar estas mediciones lo denominaba "Programa mutado". Básicamente, el Mutation testing consiste en introducir pequeñas modificaciones en el código fuente de la aplicación, a las que denominaremos mutaciones o mutantes.

Si las pruebas pasan al ejecutarse sobre el mutante, el mutante sobrevive. Si las pruebas no pasan al ejecutarse sobre el mutante, el mutante muere El objetivo es que todos los mutantes mueran, así podremos decir que el test responde a la definición concreta del código y que lo prueba correctamente. Este concepto se basa en dos hipótesis:

- Hipótesis del programador competente: La mayoría de los errores introducidos por programadores Senior consisten en pequeños errores sintácticos.
- Hipótesis del efecto de acoplamiento: Pequeños fallos acoplados pueden dar lugar a otros problemas mayores.

Problemas de mayor orden serán revelados por mutantes de mayor orden, que se crean mediante la unión de multiples mutaciones.

4.7. Pruebas estructurales: CheckStyle

Checkstyle es una herramienta de desarrollo que ayudar a los programadores a escribir código Java para que se adhiera a un estándar de codificación. Automatiza el proceso de comprobación de código Java. Esto lo hace ideal para los proyectos a los que se desea aplicar un estándar de codificación.

Checkstyle es altamente configurable y se puede hacer para apoyar casi cualquier estándar de codificación. De tal manera que se puedan suministrar diferentes estándares de código para su posterior comprobación mediante la herramienta.

Reglas del CheckStyle El conjunto de reglas disponible es muy completo y está clasificado en los siguientes grupos:

- Comentarios Javadoc: facilitar el mantenimiento pasa por comentar el código, pero luego los comentarios también hay que mantenerlos... CheckStyle tiene muchas reglas para los javadoc y es muy flexible. Te permite, por ejemplo, obligar a comentar los nombres de clases, todos los métodos menos los get/set y los atributos públicos.

- Convenciones de nombres: puedes definir una expresión regular para el nombre de todo.
- Cabeceras: expresiones regulares para la cabecera de los ficheros.
- Imports: reglas para los import, como no usar *, imports sin usar, etc.
- Violaciones de tamaño: define un máximo para el tamaño de tus clases, métodos, líneas y número de parámetros de un método. Espacios en blanco: un montón de reglas para definir donde se ponen espacios en blanco y tabuladores en el código.
- Modificadores: establece un orden para los modificadores y evita modificadores innecesarios.
- Bloques: reglas para los bloques de código y sus llaves.
- Problemas en la codificación: Acá hay de todo, desde malas prácticas tipo asignaciones internas y posibles fuentes de bugs como definir un método equals que no es el equals(Object), a cosas más estéticas o poco prolijas, como que el default sea el último elemento en un switch o paréntesis innecesarios.
- Diseño de clases: varias reglas sobre el diseño de interfaces y clases, con especial atención en las excepciones.
- Duplicados: te permite definir un mínimo de líneas para buscar código duplicado en tus clases.
- Métricas: define máximos para métricas como complejidad ciclomática, complejidad de expresiones lógicas, npath, líneas de código seguidas sin comentar y dependencia de clases.
- Misceláneo: variables final, indentación, un buscador de expresiones regulares y varias cosas más.
- J2EE: reglas para EJBs.
- Otros: internos a CheckStyle y activados por defecto.
- Filtros: para eventos de auditoria del propio CheckStyle, no hace falta mirarlos.

Checkstyle es una herramienta de desarrollo para ayudar a los programadores escribir código Java que se adhiere a un estándar de codificación. Para comprobar el estilo, pasamos la herramienta CheckStyle a nuestra aplicación y comprobamos el resultado:

5. Registro de errores

5.1. Pruebas unidad: JUnit

Se puede comprobar que no fallan.

5.2. Pruebas dinámicas de unidad: Mockito

Se puede comprobar que no fallan.

5.3. Validación de la calidad de las pruebas: Cobertura

Estado inicial: IMAGEN1COBERTURA
Estado final: IMAGEN2COBERTURA

5.4. Pruebas no funcionales: JETM

Resultado del rendimiento de JETM: Informe JETM

Con JETM hemos tenido problemas a la hora de que nos genera el site todos los resultados de las pruebas. Sí genera los resultados guardandolos en un xml con todos los resultados, pero no los muestra en la página de resultados del site.

5.5. Validación de calidad de las pruebas (mutación testing): PIT

Realizado el mutation testing, resultados:
Informe PIT

5.6. Pruebas estructurales: CheckStyle

Podemos comprobar los herrores encontrados en el siguiente informe: Informe CheckStyle
Al pasar la herramienta de CheckStyle descubrimos que nuestra aplicación tiene muchos errores de estilo, es decir, que no cumple el estandar de programación java.
Resumen de errores encontrados:

- Faltan comentarios: En la mayoría de clases faltan comentarios javadoc. Se añaden.
- Mala indexación código y espacios: Se reestructura el código para que solventar dichos errores.

5.7. Pruebas estáticas/estructurales: FindBugs

FindBugs es un programa que utiliza el análisis estático para buscar errores en el código de Java.
Informe Find bugs

Errores encontrados inicialmente: IMAGEN FINDBUGS1
Después de identificar los errores, resolvedos los problemas mencionados y comprobamos el resultado:
IMAGEN FINDBUGS2

6. Estatísticas

Deben incluírse como mínimo:

- *Número de erros encontrados diariamente e semanalmente.*
- *Nivel de progreso na execución das probas.*
- *Análise do perfil de detección de erros (lugares, compoñentes, tipoloxía).*
- *Informe de erros abertos e pechados por nivel de criticidade.*
- *Avaliación global do estado de calidade e estabilidade actuais.*

7. Outros aspectos de interese

Neste apartado se incluirán todos aqueles aspectos e detalles que non se mencionaran nos puntos anteriores, pero que o equipo do proxecto considere que poden aportar valor.