

---

# 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. Descripción 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. Componentes avaliados

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

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

## 3. Especificación de pruebas

### 3.1. Pruebas de unidad

#### Pruebas de unidad

En el documento de pruebas de unidad se especifican las pruebas definidas: Pruebas estáticas de unidad, dinámicas de de unidad, basadas en propiedades, pruebas no funcionales, estructurales y validación de las pruebas.

## 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 necesario 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 específicamente 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 múltiples mutaciones.

#### 4.7. Pruebas estructurales: CheckStyle

Checkstyle es una herramienta de desarrollo que ayuda 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: Aquí 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.

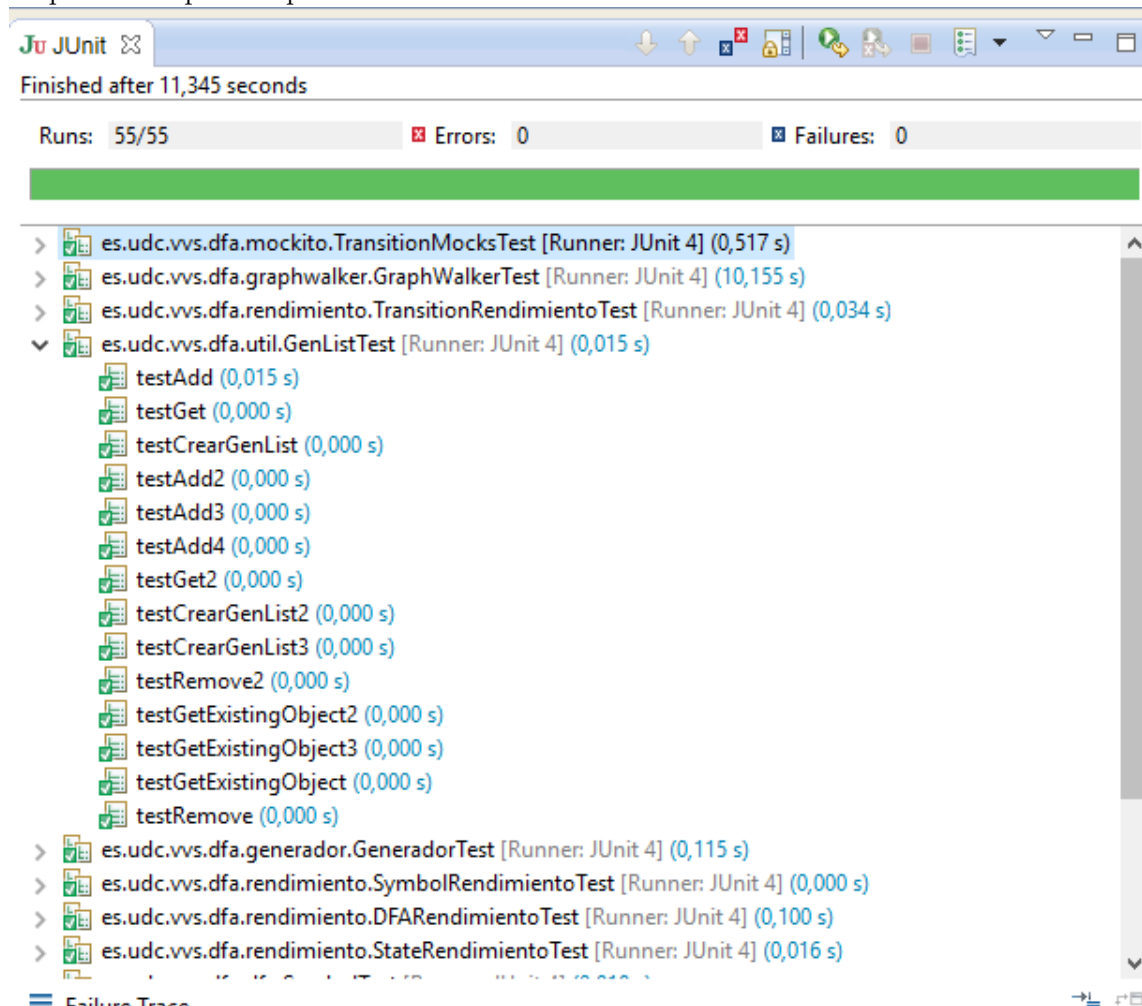
- Métricas: define máximos para métricas como complejidad ciclomática, complejidad de expresiones lógicas, npath, líneas de código seguidas sin comentar y dependencia de clases.
- Misceláneo: variables final, indentación, un buscador de expresiones regulares y varias cosas más.
- J2EE: reglas para EJBs.
- 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

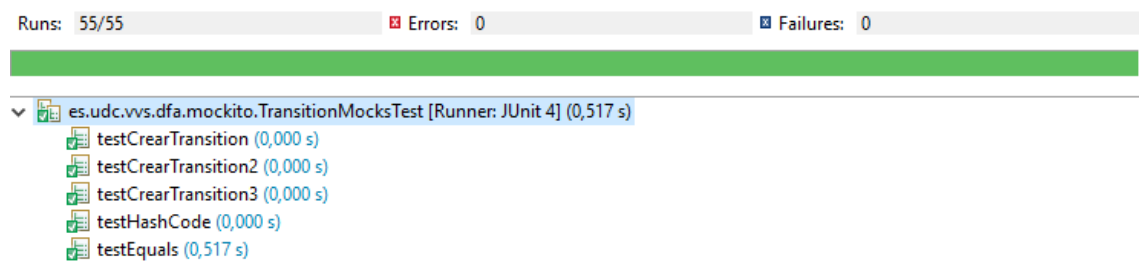
### 5.1. Pruebas unidad: JUnit

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:

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
vvs	57,2 %	1.732	1.295	3.027
src/main/java	24,9 %	341	1.028	1.369
es.udc.vvs.dfa	0,0 %	0	405	405
Main.java	0,0 %	0	405	405
es.udc.vvs.dfa.excepciones	0,0 %	0	4	4
MalformedDFAException.java	0,0 %	0	4	4
es.udc.vvs.dfa.dfa	30,3 %	209	480	689
DFA.java	4,4 %	18	395	413
Alphabet.java	48,1 %	26	28	54
Transition.java	68,2 %	75	35	110
Symbol.java	79,6 %	43	11	54
State.java	81,0 %	47	11	58
es.udc.vvs.dfa.util	48,7 %	132	139	271
GenList.java	48,7 %	132	139	271

Estado final:

Hemos realizado aumentando las pruebas en GenList para mejorar la cobertura de las pruebas, por definición habíamos decidido no realizar pruebas sobre DFA porque considerábamos que el funcionamiento de DFA estaba incluido en GenList.

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
vvs	64,8 %	2.250	1.223	3.473
src/main/java	28,7 %	393	976	1.369
es.udc.vvs.dfa	0,0 %	0	405	405
Main.java	0,0 %	0	405	405
es.udc.vvs.dfa.excepciones	0,0 %	0	4	4
MalformedDFAException.java	0,0 %	0	4	4
es.udc.vvs.dfa.dfa	31,6 %	218	471	689
DFA.java	4,4 %	18	395	413
Alphabet.java	64,8 %	35	19	54
Transition.java	68,2 %	75	35	110
Symbol.java	79,6 %	43	11	54
State.java	81,0 %	47	11	58
es.udc.vvs.dfa.util	64,6 %	175	96	271
GenList.java	64,6 %	175	96	271
src/test/java	88,3 %	1.857	247	2.104

## 5.4. Pruebas no funcionales: JETM

Resultado del rendimiento de JETM:

## JETM Timing Report

### Summary

This is a summary, by measurement name, of the measurements taken.

Name	Average (sec)	Measurements	Minimum (sec)	Maximum (sec)	Total
AlphabetRendimientoTest:addNewSymbol	0,00	5	0,00	0,01	0,02
AlphabetRendimientoTest:getAlphabet	0,00	5	0,00	0,00	0,01
AlphabetRendimientoTest:getExistingObject	0,00	4	0,00	0,00	0,01
DFArendimientoTest:getConnectedDFA	0,37	2	0,37	0,38	0,75

### File Breakdown

This is a list of, per XML file, the measurements taken.

#### Timing.xml

Name	Average (sec)	Measurements	Minimum (sec)	Maximum (sec)	Total
AlphabetRendimientoTest:getExistingObject	0,00	4	0,00	0,00	0,01
AlphabetRendimientoTest:getAlphabet	0,00	5	0,00	0,00	0,01
DFArendimientoTest:getConnectedDFA	0,37	2	0,37	0,38	0,75
AlphabetRendimientoTest:addNewSymbol	0,00	5	0,00	0,01	0,02

Copyright © 2016. All Rights Reserved.

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

Al pasar la herramienta de CheckStyle descubrimos que nuestra aplicación tiene muchos errores de estilo, es decir, que no cumple el estándar 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.

Files	Info	Warnings	Errors
9	0	0	46

### Files

File	I	W	E
es/udc/vvs/dfa/Main.java	0	0	2
es/udc/vvs/dfa/dfa/State.java	0	0	7
es/udc/vvs/dfa/dfa/Symbol.java	0	0	7
es/udc/vvs/dfa/dfa/Transition.java	0	0	16
es/udc/vvs/dfa/excepciones/MalformedDFAException.java	0	0	1
es/udc/vvs/dfa/util/GenList.java	0	0	7
testautomation/VVS.java	0	0	6



Category	Rule	Violations	Severity
blocks	EmptyBlock <a href="#">🔗</a>	1	✖ Error
	NeedBraces <a href="#">🔗</a>	6	✖ Error
	RightCurly <a href="#">🔗</a>	5	✖ Error
design	HideUtilityClassConstructor <a href="#">🔗</a>	1	✖ Error
javadoc	JavadocMethod <a href="#">🔗</a>	2	✖ Error
	JavadocType <a href="#">🔗</a>	1	✖ Error
misc	FinalParameters <a href="#">🔗</a>	14	✖ Error
	NewlineAtEndOfFile <a href="#">🔗</a>	1	✖ Error
naming	MethodName <a href="#">🔗</a>	2	✖ Error
whitespace	WhitespaceAfter <a href="#">🔗</a>	3	✖ Error
	WhitespaceAround <a href="#">🔗</a>	10	✖ Error

#### es/udc/vvs/dfa/Main.java

Severity	Category	Rule	Message	Line
✖ Error	design	HideUtilityClassConstructor	Utility classes should not have a public or default constructor.	19
✖ Error	misc	FinalParameters	Parameter filePath should be final.	28
✖ Error	misc	FinalParameters	Parameter dfa should be final.	47
✖ Error	misc	FinalParameters	Parameter args should be final.	113

#### es/udc/vvs/dfa/dfa/Alphabet.java

Severity	Category	Rule	Message	Line
✖ Error	misc	FinalParameters	Parameter n should be final.	26
✖ Error	misc	FinalParameters	Parameter symbol should be final.	34
✖ Error	misc	FinalParameters	Parameter symbol should be final.	45

#### es/udc/vvs/dfa/dfa/DFA.java

Severity	Category	Rule	Message	Line
✖ Error	misc	FinalParameters	Parameter states should be final.	44
✖ Error	misc	FinalParameters	Parameter alphabet should be final.	44
✖ Error	misc	FinalParameters	Parameter initialState should be final.	44
✖ Error	misc	FinalParameters	Parameter finalStates should be final.	45
✖ Error	misc	FinalParameters	Parameter transitions should be final.	45
✖ Error	coding	InnerAssignment	Inner assignments should be avoided.	77

#### es/udc/vvs/dfa/dfa/State.java

Severity	Category	Rule	Message	Line
✖ Error	misc	FinalParameters	Parameter state should be final.	20
✖ Error	misc	FinalParameters	Parameter obj should be final.	45

#### es/udc/vvs/dfa/dfa/Symbol.java

Severity	Category	Rule	Message	Line
✖ Error	misc	FinalParameters	Parameter symbol should be final.	20
✖ Error	misc	FinalParameters	Parameter obj should be final.	40

#### es/udc/vvs/dfa/dfa/Transition.java

Severity	Category	Rule	Message	Line
✖ Error	misc	FinalParameters	Parameter startState should be final.	30
✖ Error	misc	FinalParameters	Parameter endState should be final.	30
✖ Error	misc	FinalParameters	Parameter symbol should be final.	30
✖ Error	misc	FinalParameters	Parameter obj should be final.	67

#### es/udc/vvs/dfa/excepciones/MalformedDFAException.java

Severity	Category	Rule	Message	Line
✖ Error	misc	FinalParameters	Parameter message should be final.	15

#### es/udc/vvs/dfa/util/GenList.java

Severity	Category	Rule	Message	Line
✖ Error	misc	FinalParameters	Parameter buffer should be final.	38
✖ Error	blocks	NeedBraces	'if' construct must use '{}'.s.	39
✖ Error	misc	FinalParameters	Parameter obj should be final.	66
✖ Error	misc	FinalParameters	Parameter n should be final.	87
✖ Error	misc	FinalParameters	Parameter n should be final.	119
✖ Error	misc	FinalParameters	Parameter obj should be final.	148
✖ Error	blocks	EmptyBlock	Must have at least one statement.	163

Después de revisar los errores encontrados, resolvemos los errores de estilo encontrados y este es el resultado: [Informe CheckStyle](#)

## 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:

Summary

Classes	Bugs	Errors	Missing Classes		
8	4	0	0		

Files

Class	Bugs
es.udc.vvs.dfa.Main	1
es.udc.vvs.dfa.dfa.State	1
es.udc.vvs.dfa.dfa.Symbol	1
es.udc.vvs.dfa.util.GenList	1

es.udc.vvs.dfa.Main

Bug	Category	Details	Line	Priority
Found reliance on default encoding in es.udc.vvs.dfa.Main.getStringAutomataFromFile(String): new java.io.FileReader(String)	118N	DM_DEFAULT_ENCODING ☹	33	High

es.udc.vvs.dfa.dfa.State

Bug	Category	Details	Line	Priority
es.udc.vvs.dfa.dfa.State.toString() invokes toString() method on a String	PERFORMANCE	DM_STRING_TOSTRING ☹	34	Low

es.udc.vvs.dfa.dfa.Symbol

Bug	Category	Details	Line	Priority
es.udc.vvs.dfa.dfa.Symbol.toString() invokes toString() method on a String	PERFORMANCE	DM_STRING_TOSTRING ☹	58	Low

es.udc.vvs.dfa.util.GenList

Bug	Category	Details	Line	Priority
es.udc.vvs.dfa.util.GenList.toArray() may expose internal representation by returning GenList.list	MALICIOUS_CODE	EI_EXPOSE_REP ☹	130	Medium

Después de identificar los errores, resolvemos los problemas mencionados y comprobamos el resultado:

Summary

Classes	Bugs	Errors	Missing Classes
9	1	0	0

Files

Class	Bugs
es.udc.vvs.dfa.util.GenList	1

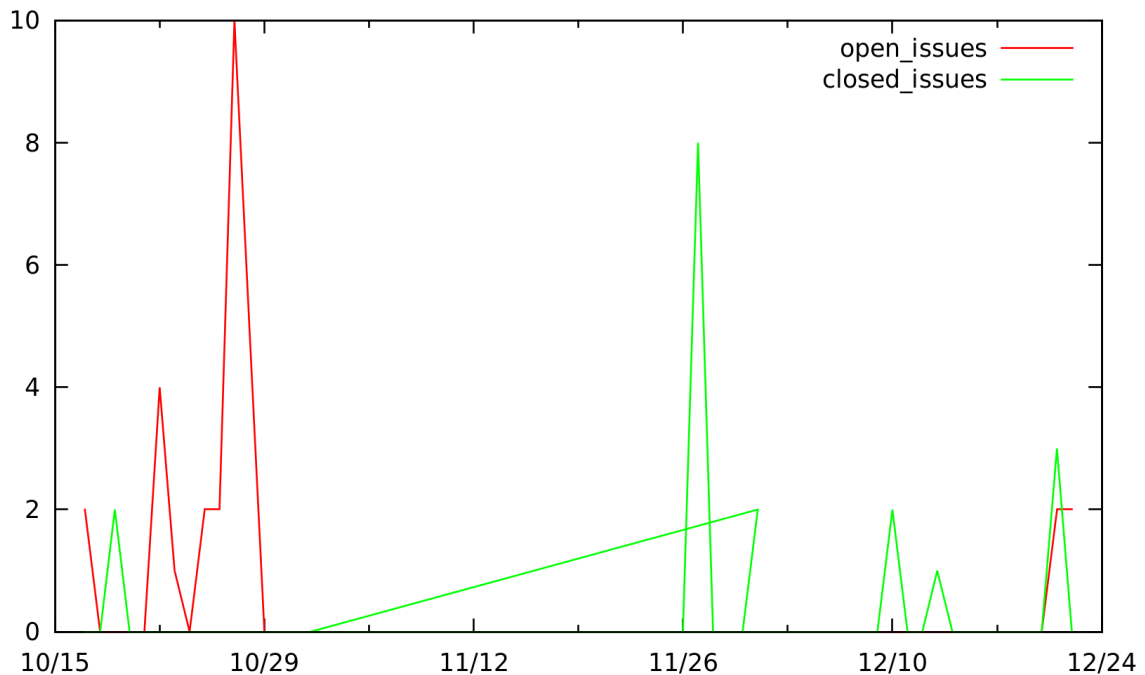
es.udc.vvs.dfa.util.GenList

Bug	Category	Details	Line	Priority
es.udc.vvs.dfa.util.GenList.toArray() may expose internal representation by returning GenList.list	MALICIOUS_CODE	EI_EXPOSE_REP	130	Medium

## 6. Estadísticas

### 6.1. Número de errores encontrados diariamente y semanalmente.

A partir de las pruebas unidad con la herramienta junit, el mayor número de errores encontrados son después de realizar las pruebas de análisis de caja blanca (checkstyle y findbugs) y mutación de código (PitTest). La tendencia después del punto álgido del 29/10/2016 (observar gráfica) a ir cerrando poco a poco las incidencias abiertas.



Sin embargo, se puede observar que en las últimas semanas se volvieron a abrir issues debido a que se detectó que las pruebas realizadas no eran válidas según la cobertura. Por lo que se abren dos issues de herramientas y test pendientes de implementar para dar por completo por finalizado de testar la aplicación.

## 6.2. Nivel de progreso de ejecución de las pruebas

Las pruebas diseñadas a priori fueron las pruebas iniciales de unidad, mockito y quickcheck. A posteriori se fueron añadiendo las pruebas obtenidas de ejecutar las herramientas de análisis estático de caja blanca (CheckStyle), Herramienta de mutación de código (PITest).

Podemos comprobar que con cobertura qué está al 68 por ciento y hemos diseñado pruebas para la clase DFA para mejorar la cobertura de nuestras pruebas. En una decisión inicial habíamos pensado en que estaban contenidas dentro de GenList pero por el resultado se comprueba qué es necesario realizar pruebas de las mismas. Faltarían por ejecutar.

## 6.3. Análisis del perfil de detección de errores

La mayoría de errores encontrados se encontraron sobre la clase GenList. Problemas con la lista de objetos y el buffer de objetos.

## 6.4. Informe de errores abiertos y cerrados

## 6.5. Evaluación global del estado y calidad y estabilidad actual

Actualmente se puede comprobar que el código es estable y las pruebas realizadas son correctas, sin embargo, eso no quiere decir de que no existan errores. En la definición de las pruebas decidimos no realizar pruebas sobre DFA porque considerábamos que estaban incluidas en GenList, sin embargo al validar con cobertura y con PIT(mutación testing) comprobamos que afecta negativamente a la validación de las pruebas. Se debería realizar más pruebas para una cobertura correcta del programa. Lo ideal sería obtener entre un 80 por ciento y 100 por cien de cobertura de las pruebas.

Faltaría realizar las pruebas basadas en modelos, nos da un error Grapwalker a la hora de generarlo:

```
[INFO] — graphwalker-maven-plugin:3.3.0:generate-sources (generate-sources) @ vvs  
— SLF4J: Failed to load class 'org.slf4j.impl.StaticLoggerBinder'. SLF4J: Defaulting to  
no-operation (NOP) logger implementation
```

Otro problema es que no hemos realizado pruebas de estrés a nuestro programa.

Una cobertura del 68 por ciento no nos garantiza que existan bugs en el programa. Se podría considerar que consiguiendo realizar las pruebas de estrés y basadas en modelos y aumentando un poco la cobertura de las pruebas podríamos considerar que el programa está completamente validado y listo para utilizarse en algún entorno de producción.

## **7. Otros aspectos de interés**