

Report Software Testing

Progetto “1+”: FastJSON

L'obiettivo del progetto 1+ è quello di sperimentare le tecniche di Software Testing e gli strumenti a supporto di esse, rendendo parametrici dei test cases di due classi di test preesistenti nel progetto *FastJSON*. Inoltre, si vuole valutare ed analizzare l'andamento della copertura strutturale al variare dei casi di test eseguiti e dei parametri passati in input agli stessi.

Le due classi di test in esame sono *JSONPatchTest_0.java* e *MapTest2.java*.

Per quanto riguarda la configurazione del progetto, è stato inserito in una pipeline usando il framework di CI delle GitHub Actions. Per stimare la coverage ottenuta con l'esecuzione dei test e generarne dei report, viene utilizzato JaCoCo, che è stato agganciato al processo di build della pipeline. Tuttavia, non essendo possibile leggerne i risultati su SonarCloud, in quanto i sorgenti testati sono instrumentati on-the-fly a partire da un jar, i report sulla coverage vengono analizzati esclusivamente in seguito ad una build in locale.

Come possibile dedurre dal Codice originale della classe *JSONPatchTest_0.java*, lo scopo di questa classe è quello di testare il metodo statico *apply* della classe *JSONPatch*. Tale metodo prende in input due rappresentazioni di *JSONObject* in formato *Stringa*, di cui la seconda descrive un pool di operazioni da effettuare sul primo *JSONObject*. Ognuna di queste operazioni può essere una *add*, *remove*, *replace*, *move*, *copy* o *test*.

La classe di test originale prevedeva un metodo di test separato per ogni passaggio di parametri differente al metodo da testare; inoltre, il risultato atteso è passato in maniera *hardcoded*. Dunque, in sostanza il test case risulta essere soltanto uno, ma invocato con parametri differenti. Quindi, la parametrizzazione dei test cases è avvenuta separando la logica di configurazione del test dall'effettiva esecuzione, operando nel seguente modo: il costruttore della classe di test chiama internamente un metodo privato *configureTestClass*, passandogli in input i parametri da utilizzare nel metodo di test; inoltre, viene passato anche il risultato atteso come output dell'invocazione del metodo sotto test.

Tale scelta è stata fatta in quanto FastJSON è basato su un concetto di oggetto JSON differente dallo standard ed implementare un oracolo per il calcolo dell'output atteso avrebbe richiesto una completa re-implementazione del metodo under test e della logica sottostante a livello di dominio. Tutto ciò è stato ritenuto andare oltre gli obiettivi del progetto 1+.

Il metodo *configureTestClass* non fa altro che configurare l'ambiente di test, inizializzando gli attributi privati della classe di test con i due parametri di input da usare nell'invocazione del metodo sotto test e con il valore dell'output atteso. Nel costruttore è lasciata de-commentata solo una invocazione per volta al metodo di *configure*; in questo modo, si seleziona ad ogni build una chiamata al *configure* con parametri diversi, così da valutare il cambiamento ottenuto in termini di copertura.

In [Codice 2](#) è riportata una visione complessiva del codice della classe *JSONPatchTest_0* re-implementata per avere dei test parametrici.

Come già anticipato, la coverage viene stimata utilizzando il framework JaCoCo, il quale produce principalmente due metriche di copertura: una *statement coverage*, basata sul numero di istruzioni in bytecode coperte dall'esecuzione dei casi di test, ed una *branch coverage*, la quale prende in considerazione i costrutti di *switch* e gli *if-statement*.

La prima analisi viene effettuata invocando il metodo under test con una patch comprendente un pool di operazioni; in particolare, si tratta di una operazione di *replace*, una di *add* ed una di *remove*.

Com'è possibile vedere in [Figura 1](#), si ottiene una *statement coverage* complessiva dei sorgenti del 14% e, in particolare, del package *com.alibaba.fastjson_* (in cui è presente la classe *JSONPatch*) del 7%. La singola classe *JSONPatch* risulta avere una copertura strutturale del 51% ed una *branch coverage* del 38 % ([Figura 2](#)). La copertura dei singoli metodi della classe in questione è visibile in [Figura 3](#).

Una seconda analisi è effettuata lanciando il test con una patch composta da una singola operazione di *move*. Per quanto riguarda la copertura strutturale del codice sorgente, la [Figura 4](#) ci dimostra che si hanno davvero pochi cambiamenti rispetto al caso precedente; in particolare, la percentuale di *statement coverage* del package *com.alibaba.fastjson_* è esattamente uguale. Invece, la copertura della classe *JSONPatch* risulta essere aumentata in *statement coverage* (dal 51% al 58%), ma non in *branch coverage* ([Figura 5](#)). Infatti, guardando la copertura strutturale dei metodi della classe ([Figura 6](#)), essa risulta essere aumentata sia per il metodo *apply(Object, String)* che per *isObject(String)*. Per comprendere tali risultati, è stato analizzato il decompilato ottenuto a partire dal file *JSONPatch.class*. Il blocco di codice che si occupa di gestire l'operazione di *move* richiede molteplici istruzioni, più di quante ne richiedano un'operazione di *add*, di *replace* e di *remove* insieme.

L'altra classe di test è *MapTest2.java*. Essa include originariamente un solo metodo di test in cui viene testato il metodo statico *parseObject(String text, TypeReference<T> type)* della classe *JSON*. Tale metodo ha il compito di fare il parsing della stringa in input (rappresentante un JSON Object) in un oggetto il cui tipo è indicato dalla classe *TypeReference*. In particolare, per un JSON Object ci si aspetta che venga trasformato in una *Map<Object, Object>*. Il test passa se e solo se ad ogni chiave è associato il valore atteso.

Il codice originale della classe è riportato in [Codice 3](#). Anche in questo caso, per la parametrizzazione si è fatto uso di un metodo *configureTestClass* invocato dal costruttore della classe di test. L'unico parametro passato al metodo di configure è la stringa rappresentante il *JSONObject*. La configurazione dell'ambiente di test consiste nel settare un attributo privato della classe di test con la stringa di cui sopra e nell'allocare un'istanza di *TypeReference<Map<Object, Object>>*, associando anch'essa ad un attributo privato.

Questa volta, è stato possibile fare un semplice parsing della stringa rappresentante il *JSONObject* per ottenere un array di chiavi di tipo *Object* da utilizzare nel metodo di test. Inoltre, è stata utilizzata la libreria **JSON-JAVA (org.json)** per ottenere un oggetto JSON da cui estrarre gli output attesi per il test. Tali responsabilità sono state attribuite ad una classe *Oracle* interna alla classe di test, la quale agisce appunto da oracolo di test. La nuova implementazione parametrica di *MapTest2* è consultabile in [Codice 4](#).

Essendo il test originario eseguito con una sola rappresentazione in stringa di un *JSONObject*, non è possibile valutare la variazione della copertura al cambio dei valori dei parametri di test (a meno di progettare ed inserire nuovi parametri appropriati). La coverage che si ottiene a seguito dell'esecuzione del solo test in esame è consultabile in [Figura 7](#) e [Figura 8](#): la statement coverage su tutti i package ammonta al 5%, quella del package *org.alibaba.fastjson* (contenente la classe *JSON*) appena all'1%, mentre la statement coverage della classe *JSON* è del 9%. Sono valori molto bassi, ma giustificati dall'esecuzione di un solo test case su un metodo di una singola classe, la quale ha una size molto elevata.

Sono state in seguito valutate le metriche di copertura dell'intera test-suite, composta dai due metodi di test delle classi parametrizzate in precedenza. Ci si aspetta naturalmente un incremento dei valori di coverage rispetto a quelli analizzati precedentemente.

Relativamente ai risultati ottenuti dal primo run del test di *JSONPatchTest_0.java*, la branch coverage è aumentata dal 6% al 7% su tutti i package ([Figura 9](#)), mentre, dal punto di vista percentuale, la statement coverage è rimasta assestata sul 14%. Tuttavia, è possibile notare che il numero di missed instructions è leggermente diminuito. Quindi, come ci si aspettava, si ha un miglioramento generale della coverage.

In particolare, è molto interessante notare come la statement coverage della classe *JSON* sia passata dal 9%, con l'esecuzione del test di *MapTest2*, al 14%, con l'esecuzione dell'intera test-suite ([Figura 10](#)). Questo perché ovviamente la classe viene coinvolta anche nel flusso d'esecuzione del test di *JSONPatchTest_0*, molto probabilmente per eseguire il parsing delle stringhe in oggetti JSON.

Repository GitHub: <https://github.com/AndreaPepe/fastjsonTesting>.

Progetto 2: Apache Syncope

Dominio

Apache *Syncope* è un sistema open-source per la gestione di identità digitali in ambienti enterprise. Il suo obiettivo primario è quello di gestire i dati e le informazioni degli utenti e dei loro account su sistemi e applicazioni. Inoltre, svolge il ruolo di access management system, dando la possibilità di definire delle politiche di accesso a risorse, applicabili a singoli utenti, a gruppi o persino ad altri oggetti definibili in modo personalizzato (e.g. stampanti, dispositivi IoT, etc.). Per ulteriori informazioni, consultare la documentazione ufficiale del progetto al seguente [link](#).

Prima classe: DefaultPasswordGenerator

Una delle due classi scelte per l'attività di Software Testing è stata *DefaultPasswordGenerator*. Tale scelta è stata dettata sia dall'importanza della classe, in quanto la gestione e la generazione delle passwords è un punto focale dell'identity management, sia dalla discreta documentazione trovata online. Infatti, una delle

grandi mancanze del progetto in questione è di certo una documentazione dal livello di dettaglio assolutamente insoddisfacente, sia ad alto livello che a basso livello (Javadoc nel codice).

Come desumibile, dal nome della classe e dal seguente commento di documentazione, la classe fornisce la funzionalità di generare automaticamente delle password conformi a determinate policies:

Generate random passwords according to given policies.

When no minimum and / or maximum length are specified, default values are set.

WARNING: This class only takes DefaultPasswordRuleConf into account.

Al seguente [link](#) è possibile trovare ciò che la documentazione ufficiale riporta in merito alle password e alle policies che le riguardano. In particolare, viene specificato che:

When defining a password policy, the following information must be provided:

- allow null password - whether a password is mandatory for Users or not
- history length - how many values shall be considered in the history
- rules - set of password rules to evaluate with the current policy

Inoltre, per quanto riguarda le *rules* possibili, abbiamo quanto segue:

The default password rule (enforced by *DefaultPasswordRule* and configurable via *DefaultPasswordRuleConf*) contains the following controls:

- maximum length - the maximum length to allow; 0 means no limit set;
- minimum length - the minimum length to allow; 0 means no limit set;
- non-alphanumeric required
- alphanumeric required
- digit required
- lowercase required
- uppercase required
- must start with digit
- must not start with digit
- must end with digit
- must not end with digit
- must start with alphanumeric
- must start with non-alphanumeric
- must not start with alphanumeric
- must not start with non-alphanumeric
- must end with alphanumeric
- must end with non-alphanumeric
- must not end with alphanumeric
- must not end with non-alphanumeric
- username allowed - whether a username value can be used
- words not permitted - list of words that cannot be present, even as a substring;
- schemas not permitted - list of schemas whose values cannot be present, even as a substring;
- prefixes not permitted - list of strings that cannot be present as a prefix;
- suffixes not permitted - list of strings that cannot be present as a suffix.

Come prima cosa, si decide di testare il metodo principale della classe, che genera una password in accordo alla lista di policies indicata. Il codice di tale metodo è consultabile nella tabella [Codice 5](#).

L'unico parametro di input del metodo è una lista di *PasswordPolicy*, la quale è un'interfaccia che è implementata dalla classe *DefaultPasswordPolicy*. Il tipo di ritorno è una stringa, ma può essere lanciata un'eccezione di tipo *InvalidPasswordRuleConf* se la configurazione delle policies è non valida.

Alla luce di queste informazioni, si procede nell'effettuare **domain partitioning** dei parametri di input e dei possibili output dell'operazione. Per quanto riguarda l'unico parametro di input, si tratta di una lista, quindi si considerano i seguenti sottoinsiemi:

- *policies*: {empty list}, {non empty list}, {null}

Tuttavia, si tratta di una lista di *PasswordPolicy*, dunque, un tipo di dato complesso che, per quanto descritto in precedenza può avere una configurazione più o meno valida all'interno del dominio di riferimento. Si nota che lo stato della classe *DefaultPasswordRule* è determinato dai parametri *allowNullPassword*, *historyLength* e *rules*; è ovvio che il fatto che la password sia o meno obbligatoria per l'utente o la history non sono rilevanti

dal punto di vista della generazione automatica della password stessa. L'unico fattore che realmente impatta il risultato ottenuto è l'insieme di regole che la password dovrà rispettare.

Dunque, si distingue subito che tra le *non empty list* è lecito distinguere tra liste con policies valide e non valide, quindi le classi di equivalenza individuate diventano:

- *policies*: {*empty list*}, {*vaild policies*}, {*invalid policies*}, {*null*}

Per quanto riguarda i possibili output, si hanno tre possibilità rappresentate dal verificarsi di un'eccezione per regole non valide, da password conforme alle policies e da password non conforme alle policies.

Effettuando la **boundary analysis**, si ottengono i seguenti boundary values:

- *policies*: *empty list*, *list with 1 valid policy*, *list with 1 invalid policy*, *null*

Essendo il parametro di input unico, è possibile soltanto avere un approccio unidimensionale. Si procede quindi alla definizione ad alto livello dei casi di test da implementare e del risultato atteso, come riportato in [Tabella 1](#). In occasione di una lista di policies vuota, ci si aspetta che vengano rispettate le regole di default, come confermato anche da una successiva analisi del codice sorgente. Tali regole hanno unicamente constraints sulla size della password: lunghezza minima di 8 caratteri e massima di 64.

La policy non valida è stata ottenuta imponendo due regole in contrasto: la password deve terminare con una lettera e la password deve terminare con un numero.

Per l'esecuzione dei test, è stato necessario effettuare un mock statico della classe *ImplementationManager*. Tutti i test eseguiti in questa prima iterazione vanno a buon fine e si ottengono risultati per la statement coverage e la branch coverage visibili in [Figura 11](#) e in [Figura 12](#). Le percentuali di coverage sono già molto elevate, questo perché il metodo testato invoca internamente quasi tutti gli altri metodi, perlopiù protected, della classe.

Si vuole cercare di migliorare la coverage ottenuta, sia di classe che di metodi, andando in particolare a stimolare con inputs differenti il metodo *check(DefaultPasswordRuleConf)*, il quale controlla che le policies siano valide. Infatti, è questo il punto che con più probabilità può essere buggato, ad esempio ritenendo valida una policy non valida o viceversa.

Per fare ciò, è necessario operare sullo stato interno della classe *DefaultPasswordRuleConf*, la quale contiene i settaggi di tutte le *rules* elencate nella documentazione, che viene utilizzata per caratterizzare le varie policies.

Tale stato è caratterizzato da ben 24 parametri (uno per ognuna delle *rules* indicate sulla documentazione), di cui due di tipo intero (massima e minima size della password), 16 di tipo booleano e 4 di tipo List<String> per le blacklist di parole da non contenere come prefisso, suffisso o all'interno della password.

Si procede con l'individuazione delle classi di equivalenza:

- *maxLength*: {< 0}, {0}, {> 0}
- *minLength*: {≤ *maxLength*}, {> *maxLength*}
- Per i parametri booleani si individuano le classi {true} e {false}; per completezza si elencano i parametri: *nonAlphanumericRequired*, *alphanumericRequired*, *digitRequired*, *lowercaseRequired*, *uppercaseRequired*, *mustStartWithDigit*, *mustntStartWithDigit*, *mustEndWithDigit*, *mustntEndWithDigit*, *mustStartWithNonAlpha*, *mustntStartWithNonAlpha*, *mustStartWithAlpha*, *mustntStartWithAlpha*, *mustEndWithNonAlpha*, *mustntEndWithNonAlpha*, *mustEndWithAlpha*, *mustntEndWithAlpha*, *usernameAllowed*;
- *wordsNotPermitted*, *schemasNotPermitted*, *prefixesNotPermitted*, *suffixesNotPermitted*: {*emptyList*}, {*nonEmptyList*}

Per questi ultimi parametri, non è stata considerata la classe di equivalenza {null} perché ritenuta non significativa all'interno dello scenario.

Effettuando la boundary-value analysis sui primi due parametri, si identificano i seguenti valori:

- *maxLength*: - 1, 0, 10
- *minLength*: = *maxLength*, *maxLength* + 1

Per essi si userà un approccio multidimensionale, essendo strettamente correlati. Come valore >0 per *maxLength* si è scelto un valore sufficientemente grande così che la stringa possa contenere abbastanza caratteri per poter verificare la validità dei requisiti espressi dai parametri booleani.

Allegati

```
public class JSONPatchTest_0 extends TestCase {

    public void test_for_multi_0() throws Exception {
        String original = "{\n" +
            "  \"baz\": \"qux\",\n" +
            "  \"foo\": \"bar\"\n" +
            "}";

        String patch = "[\n" +
            "  { \"op\": \"replace\", \"path\": \"/baz\", \"value\": \"boo\" },\n" +
            "  { \"op\": \"add\", \"path\": \"/hello\", \"value\": [\"world\"] },\n" +
            "  { \"op\": \"remove\", \"path\": \"/foo\" }\n" +
            "];";

        String result = JSONPatch.apply(original, patch);
        assertEquals("{\"baz\":\"boo\",\"hello\":[\"world\"]}", result);
    }

    public void test_for_add_1() throws Exception {
        String original = "{}";

        String patch = "{ \"op\": \"add\", \"path\": \"/a/b/c\", \"value\": [ \"foo\", \"bar\" ] }";

        String result = JSONPatch.apply(original, patch);
        assertEquals("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}", result);
    }

    public void test_for_remove_0() throws Exception {
        String original = "{}";

        String patch = "{ \"op\": \"remove\", \"path\": \"/a/b/c\" }";

        String result = JSONPatch.apply(original, patch);
        assertEquals "{}", result);
    }

    public void test_for_remove_1() throws Exception {
        String original = "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}";

        String patch = "{ \"op\": \"remove\", \"path\": \"/a/b/c\" }";

        String result = JSONPatch.apply(original, patch);
        assertEquals("{\"a\":{\"b\":{\"c\":[]}}}", result);
    }

    public void test_for_replace_1() throws Exception {
        String original = "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}";

        String patch = "{ \"op\": \"replace\", \"path\": \"/a/b/c\", \"value\": 42 }";

        String result = JSONPatch.apply(original, patch);
        assertEquals("{\"a\":{\"b\":{\"c\":42}}}", result);
    }

    public void test_for_move_0() throws Exception {
        String original = "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}";

        String patch = "{ \"op\": \"move\", \"from\": \"/a/b/c\", \"path\": \"/a/b/d\" }";

        String result = JSONPatch.apply(original, patch);
        assertEquals("{\"a\":{\"b\":{\"d\":[\"foo\",\"bar\"]}}}"; result);
    }

    public void test_for_copy_0() throws Exception {
        String original = "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}";

        String patch = "{ \"op\": \"copy\", \"from\": \"/a/b/c\", \"path\": \"/a/b/e\" }";

        String result = JSONPatch.apply(original, patch);
        assertEquals("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"],\"e\":[\"foo\",\"bar\"]}}}"; result);
    }

    public void test_for_test_0() throws Exception {
```

```

String original = "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}";

String patch = "{ \"op\": \"test\", \"path\": \"/a/b/c\", \"value\": \"foo\" }";

String result = JSONPatch.apply(original, patch);
assertEquals("false", result);
}
}

```

Codice 1 Codice originale della classe JSONPatchTest_0.java

```

public class JSONPatchTest_0 {

    /* Instance under test is not needed, because the tested method is static*/
    private String original;
    private String patch;
    private String expected;

    public JSONPatchTest_0() {
        configureTestClass("{\"\n" + "  \"baz\": \"qux\",\n" + "  \"foo\": \"bar\"\n" + "}",
            "[\n" +
                "    { \"op\": \"replace\", \"path\": \"\/baz\", \"value\": \"boo\" },\n" +
                "    { \"op\": \"add\", \"path\": \"\/hello\", \"value\": [\"world\"] },\n" +
                "    { \"op\": \"remove\", \"path\": \"\/foo\" }\n" + "]" ,
            "{\"baz\": \"boo\", \"hello\": [\"world\"]}");

        // configureTestClass("{", "{ \"op\": \"add\", \"path\": \"\/a/b/c\", \"value\": [ \"foo\", \"bar\" ] }",
        //     "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}}");
        //
        // configureTestClass("{", "{ \"op\": \"remove\", \"path\": \"\/a/b/c\" }\n",
        //     "{}");
        //
        // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}} ", "{ \"op\": \"remove\", \"path\": \"\/a/b/c\" }\n",
        //     "{\"a\":{\"b\":{\"c\":[]}}");
        //
        // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}} ", "{ \"op\": \"replace\", \"path\": \"\/a/b/c\", \"value\": 42 }",
        //     "{\"a\":{\"b\":{\"c\":42}}}");
        //
        // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}} ", "{ \"op\": \"move\", \"from\": \"\/a/b/c\", \"path\": \"\/a/b/d\" }",
        //     "{\"a\":{\"b\":{\"d\":[\"foo\",\"bar\"]}}}");
        //
        // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}} ", "{ \"op\": \"copy\", \"from\": \"\/a/b/c\", \"path\": \"\/a/b/e\" }",
        //     "{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"],\"e\":[\"foo\",\"bar\"]}}}");
        //
        // configureTestClass("{\"a\":{\"b\":{\"c\":[\"foo\",\"bar\"]}}} ", "{ \"op\": \"test\", \"path\": \"\/a/b/c\", \"value\": \"foo\" }",
        //     "false");
        //
    }

    private void configureTestClass(String original, String patch, String expected) {
        this.original = original;
        this.patch = patch;
        this.expected = expected;
    }

    @Test
    public void testJSONPatch() {
        String result = JSONPatch.apply(original, patch);
        Assert.assertEquals(expected, result);
    }
}

```

Codice 2 Classe JSONPatchTest_0.java con test parametrico

JaCoCo Coverage Report				
Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.alibaba.fastjson.parser		12%		6%
com.alibaba.fastjson.serializer		7%		3%
com.alibaba.fastjson.util		17%		6%
com.alibaba.fastjson		7%		4%
com.alibaba.fastjson.parser.deserializer		27%		13%
com.alibaba.fastjson.support.spring		0%		0%
com.alibaba.fastjson.asm		54%		35%
com.alibaba.fastjson.support.jaxrs		0%		0%
com.alibaba.fastjson.support.hsf		0%		0%
com.alibaba.fastjson.support.retrofit		0%		0%
com.alibaba.fastjson.support.geo		0%		0%
com.alibaba.fastjson.support.spring.messaging		0%		0%
com.alibaba.fastjson.support.config		0%		0%
com.alibaba.fastjson.support.moneta		0%		0%
com.alibaba.fastjson.support.springfox		0%		n/a
Total	108.577 of 126.842	14%	17.168 of 18.420	6%

Figura 1 Coverage eseguendo solo JSONPatchTest_0 con una patch da 3 operazioni

JSONPatch		51%		38%
---------------------------	--	-----	--	-----

Figura 2 Coverage della classe JSONPatch con una patch da 3 operazioni

Element	Missed Instructions	Cov.	Missed Branches	Cov.
apply(Object, String)		42%		33%
isObject(String)		68%		50%
JSONPatch()		0%		n/a
apply(String, String)		100%		n/a
Total	85 of 174	51%	16 of 26	38%

Figura 3 Coverage dei metodi della classe JSONPatch con una patch da 3 operazioni

JaCoCo Coverage Report				
Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.alibaba.fastjson.parser		11%		5%
com.alibaba.fastjson.serializer		6%		2%
com.alibaba.fastjson.util		17%		6%
com.alibaba.fastjson		7%		4%
com.alibaba.fastjson.parser.deserializer		26%		12%
com.alibaba.fastjson.support.spring		0%		0%
com.alibaba.fastjson.asm		54%		35%
com.alibaba.fastjson.support.jaxrs		0%		0%
com.alibaba.fastjson.support.hsf		0%		0%
com.alibaba.fastjson.support.retrofit		0%		0%
com.alibaba.fastjson.support.geo		0%		0%
com.alibaba.fastjson.support.spring.messaging		0%		0%
com.alibaba.fastjson.support.config		0%		0%
com.alibaba.fastjson.support.moneta		0%		0%
com.alibaba.fastjson.support.springfox		0%		n/a
Total	108.830 of 126.842	14%	17.217 of 18.420	6%

Figura 4 Coverage ottenuta eseguendo solo JSONPatchTest_0 con una patch da una singola operazione

 JSONPatch	 58%	 38%
---	---	---

Figura 5 Copertura della classe JSONPatch a seguito dell'esecuzione di un test con una patch da una singola operazione







Element	Missed Instructions	Cov.	Missed Branches	Cov.
● apply(Object, String)		51%		33%
● isObject(String)		72%		50%
● JSONPatch()		0%		n/a
● apply(String, String)		100%		n/a
Total	73 of 174	58%	16 of 26	38%

Figura 6 Coverage dei metodi della classe JSONPatch in seguito al test con patch con singola operazione

```
public class MapTest2 extends TestCase {

    public void test_map () throws Exception {
        Map<Object, Object> map = JSON.parseObject("{1:\\\"2\\\",\\\"3\\\":4,'5':6}", new TypeRefer-
ence<Map<Object, Object>>() {});
        Assert.assertEquals("2", map.get(1));
        Assert.assertEquals(4, map.get("3"));
        Assert.assertEquals(6, map.get("5"));
    }
}
```

Codice 3 Codice originale della classe MapTest2

```
public class MapTest2 {

    /* The instance to be tested is not needed because the tested method is static*/

    private String jsonObject;
    private TypeReference<Map<Object, Object>> typeReference;

    public MapTest2 () {
        configureTestClass("{1:\\\"2\\\",\\\"3\\\":4,'5':6}");
    }

    private void configureTestClass(String jsonObject) {
        this.jsonObject = jsonObject;
        this.typeReference = new TypeReference<>() {};
    }

    /**
     * This test method invokes the <i>parseObject</i> static method of the JSON class.
     * In particular, it checks that the conversion from the string representation
     * of the jsonObject to a Map between keys and values is well done.
     * The test is passed only if each value is associated to the correct key.
     */
    @Test
    public void test_map () {
        Map<Object, Object> map = JSON.parseObject(this.jsonObject, this.typeReference);

        // Use an oracle to compute the keys and the expected values bounded with those keys
        Oracle oracle = new Oracle();
        Object[] keys = oracle.getKeys(jsonObject);
        Object[] expected = oracle.getExpectedResults(jsonObject, keys);

        boolean passed = true;
        for (int i = 0; i < keys.length; i++) {
            if (map.get(keys[i]) == null) {
                if (! (map.get(keys[i]) == expected[i])) {
                    passed = false;
                    break;
                }
            }
            else if (!map.get(keys[i]).equals(expected[i])) {
                passed = false;
                break;
            }
        }
        Assert.assertTrue(passed);
    }
}
```



```

private static class Oracle{
    /**
     * This private method is used as a Software Testing oracle to know the
     * expected values of the JSON Object referred by the key in the input array of keys.
     * It's an oracle because it uses the library <b>org.json</b> as a trusted entity.
     *
     * @param jsonObject The string representing the JSONObject of the FastJSON project
     * @param keys The array of the objects representing the keys of the JSON Object
     * @return an array of Object, containing the values expected to be contained in the
jsonObject,
     * in the same order of their keys
     */
    private Object[] getExpectedResults(String jsonObject, Object[] keys){
        JSONObject obj = new JSONObject(jsonObject);
        List<Object> expected = new LinkedList<>();
        for (int i = 0; i < keys.length; i++){
            expected.add(i, obj.get(String.valueOf(keys[i])));
        }
        return expected.toArray();
    }

    /**
     * This private method is used to retrieve the array of keys
     * of the json object string representation in input.
     * @param jsonObject The string representing the JSONObject of the FastJSON project
     * @return an array of Objects, containing the keys of the JSONObject.
     */
    private Object[] getKeys(String jsonObject){
        String noBrackets = jsonObject.replace("{", "");
        noBrackets = noBrackets.replace("}", "");
        noBrackets = noBrackets.replace(" ", "");
        noBrackets = noBrackets.replace("\n", "");
        noBrackets = noBrackets.replace("\r", "");
        noBrackets = noBrackets.replace("\t", "");

        String[] pairs = noBrackets.split(",");

        List<Object> ret = new LinkedList<>();
        for (int i = 0; i < pairs.length; i++){
            String key = pairs[i].split(":")[0];
            if (key.contains("\"") || key.contains("'')){
                ret.add(i, key.replace("\"", "").replace("'", ""));
            }
            else if (key.contains("."))
                ret.add(i, Float.valueOf(key));
            else
                ret.add(i, Integer.valueOf(key));
        }
        return ret.toArray();
    }
}
}

```

Codice 4 Classe MapTest2.java con test parametrico e inner class Oracle

Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.alibaba.fastjson.serializer		1%		0%
com.alibaba.fastjson.parser		8%		2%
com.alibaba.fastjson.util		12%		2%
com.alibaba.fastjson.parser.deserializer		2%		1%
com.alibaba.fastjson		1%		0%
com.alibaba.fastjson.asm		0%		0%
com.alibaba.fastjson.support.spring		0%		0%
com.alibaba.fastjson.support.jaxrs		0%		0%
com.alibaba.fastjson.support.hsf		0%		0%
com.alibaba.fastjson.support.retrofit		0%		0%
com.alibaba.fastjson.support.geo		0%		0%
com.alibaba.fastjson.support.spring.messaging		0%		0%
com.alibaba.fastjson.support.config		0%		0%
com.alibaba.fastjson.support.moneta		0%		0%
com.alibaba.fastjson.support.springfox		0%		n/a
Total	119.924 of 126.842	5%	18.158 of 18.420	1%

Figura 7 Coverage ottenuta eseguendo il solo test della classe MapTest2.java

JSON		9%		4%
----------------------	--	----	--	----

Figura 8 Class coverage della classe JSON in seguito al test di MapTest2.java

Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.alibaba.fastjson.parser		13%		7%
com.alibaba.fastjson.serializer		7%		3%
com.alibaba.fastjson.util		17%		6%
com.alibaba.fastjson		7%		4%
com.alibaba.fastjson.parser.deserializer		28%		15%
com.alibaba.fastjson.support.spring		0%		0%
com.alibaba.fastjson.asm		54%		35%
com.alibaba.fastjson.support.jaxrs		0%		0%
com.alibaba.fastjson.support.hsf		0%		0%
com.alibaba.fastjson.support.retrofit		0%		0%
com.alibaba.fastjson.support.geo		0%		0%
com.alibaba.fastjson.support.spring.messaging		0%		0%
com.alibaba.fastjson.support.config		0%		0%
com.alibaba.fastjson.support.moneta		0%		0%
com.alibaba.fastjson.support.springfox		0%		n/a
Total	107.981 of 126.842	14%	17.081 of 18.420	7%

Figura 9 Coverage in seguito all'esecuzione dell'intera test-suite

Element	Missed Instructions	Cov.	Missed Branches	Cov.
JSONPath		7%		5%
JSONPath.JSONPathParser		7%		5%
JSON		14%		7%
JSONObject		6%		5%

Figura 10 Coverage della classe JSON aumentata con l'esecuzione di tutti i test della test-suite

```

public String generate(final List<PasswordPolicy> policies) throws InvalidPasswordRuleConf {
    List<DefaultPasswordRuleConf> defaultRuleConfs = new ArrayList<>();

    policies.stream().forEach(policy -> policy.getRules().forEach(impl -> {
        try {
            ImplementationManager.buildPasswordRule(impl).ifPresent(rule -> {
                if (rule.getConf() instanceof DefaultPasswordRuleConf) {
                    defaultRuleConfs.add((DefaultPasswordRuleConf) rule.getConf());
                }
            });
        } catch (Exception e) {
            LOG.error("Invalid {}, ignoring...", impl, e);
        }
    }));

    DefaultPasswordRuleConf ruleConf = merge(defaultRuleConfs);
    check(ruleConf);
    return generate(ruleConf);
}

```

Codice 5 metodo generate(List<PasswordPolicy>) di DefaultPasswordGenerator

policies	Expected output
Empty list	Password conforme alle policies di default
List with 1 valid policy	Password conforme alla policy indicata
List with 1 invalid policy	Eccezione
null	Eccezione

Tabella 1 Test cases iterazione 1 metodo generate(List<PasswordPolicy>) di DefaultPasswordGenerator

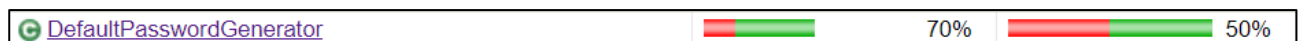


Figura 11 Class coverage DefaultPasswordGenerator (iterazione 1)

DefaultPasswordGenerator				
Element	Missed Instructions	Cov.	Missed Branches	Cov.
check(DefaultPasswordRuleConf)		42%		44%
checkRequired(String[], DefaultPasswordRuleConf)		51%		50%
checkEndChar(String[], DefaultPasswordRuleConf)		60%		64%
checkStartChar(String[], DefaultPasswordRuleConf)		59%		50%
generate(ExternalResource)		0%		0%
lambda\$checkPrefixAndSuffix\$5(String[], DefaultPasswordRuleConf, String)		0%		0%
lambda\$checkPrefixAndSuffix\$4(String[], DefaultPasswordRuleConf, String)		0%		0%
lambda\$generate\$1(List, Implementation)		53%		n/a
lambda\$merge\$3(DefaultPasswordRuleConf, DefaultPasswordRuleConf)		97%		50%
merge(List)		89%		75%
generate(DefaultPasswordRuleConf)		100%		100%
generate(List)		100%		n/a
checkPrefixAndSuffix(String[], DefaultPasswordRuleConf)		100%		n/a
firstEmptyChar(String[])		100%		100%
lambda\$generate\$0(List, PasswordRule)		100%		50%
lambda\$generate\$2(List, PasswordPolicy)		100%		n/a
static {...}		100%		n/a
DefaultPasswordGenerator()		100%		n/a
Total	183 of 629	70%	68 of 138	50%

Figura 12 Method coverage DefaultPasswordGenerator (iterazione 1)

```
protected static void check(final DefaultPasswordRuleConf defaultPasswordRuleConf)
    throws InvalidPasswordRuleConf {

    if (defaultPasswordRuleConf.isMustEndWithAlpha() && defaultPasswordRuleConf.isMustntEndWithAlpha()) {
        throw new InvalidPasswordRuleConf(
            "mustEndWithAlpha and mustntEndWithAlpha are both true");
    }

    if (defaultPasswordRuleConf.isMustEndWithAlpha() && defaultPasswordRuleConf.isMustEndWithDigit()) {
        throw new InvalidPasswordRuleConf(
            "mustEndWithAlpha and mustEndWithDigit are both true");
    }

    if (defaultPasswordRuleConf.isMustEndWithDigit() && defaultPasswordRuleConf.isMustntEndWithDigit()) {
        throw new InvalidPasswordRuleConf(
            "mustEndWithDigit and mustntEndWithDigit are both true");
    }

    if (defaultPasswordRuleConf.isMustEndWithNonAlpha() && defaultPasswordRuleConf.isMustntEndWithNonAlpha()) {
        throw new InvalidPasswordRuleConf(
            "mustEndWithNonAlpha and mustntEndWithNonAlpha are both true");
    }

    if (defaultPasswordRuleConf.isMustStartWithAlpha() && defaultPasswordRuleConf.isMustntStartWithAlpha()) {
        throw new InvalidPasswordRuleConf(
            "mustStartWithAlpha and mustntStartWithAlpha are both true");
    }

    if (defaultPasswordRuleConf.isMustStartWithAlpha() && defaultPasswordRuleConf.isMustStartWithDigit()) {
        throw new InvalidPasswordRuleConf(
            "mustStartWithAlpha and mustStartWithDigit are both true");
    }

    if (defaultPasswordRuleConf.isMustStartWithDigit() && defaultPasswordRuleConf.isMustntStartWithDigit()) {
        throw new InvalidPasswordRuleConf(
            "mustStartWithDigit and mustntStartWithDigit are both true");
    }

    if (defaultPasswordRuleConf.isMustStartWithNonAlpha() && defaultPasswordRuleConf.isMustntStartWithNonAlpha()) {
        throw new InvalidPasswordRuleConf(
            "mustStartWithNonAlpha and mustntStartWithNonAlpha are both true");
    }

    if (defaultPasswordRuleConf.getMinLength() > defaultPasswordRuleConf.getMaxLength()) {
        throw new InvalidPasswordRuleConf(
            "Minimun length (" + defaultPasswordRuleConf.getMinLength() + ") "
            + "is greater than maximum length (" + defaultPasswordRuleConf.getMaxLength() + ")");
    }
}
```

Figura 13 Coverage del metodo `check(DefaultPasswordRuleConf)` di `DefaultPasswordGenerator`

[illegible]

Tabella 2 Test cases per iterazione 2 di DefaultPasswordGenerator (fare doppio click sulla tabella per consultare il file Excel)