TecnoCampus Escola Superior Politècnica

Centre adscrit a la Universitat Pompeu Fabra Barcelona

Grau en Enginyeria Informàtica de Gestió i Sistemes d'Informació

CREADOR AUTOMÀTIC DE TESTS

Memòria

GERARD TORRENT CASTELL
TUTOR: DAVID RÓDENAS PICÓ

2022-2023

TecnoCampus Mataró-Maresme

Resum

La generació de cobertura de codi és essencial per avaluar l'eficàcia de les proves en el desenvolupament de software. Automatitzar aquest procés millora la detecció d'errors i la qualitat del software. L'objectiu d'aquest TFG és determinar si assolir un determinat percentatge de cobertura de codi garanteix la seva qualitat. Es realitzarà un experiment per comparar els resultats de la generació de cobertura automatitzada mitjançant un procés iteratiu per augmentar la cobertura en cada iteració.

Resumen

La generación de cobertura de código es esencial para evaluar la eficacia de las pruebas en el desarrollo de software. Automatizar este proceso mejora la detección de errores y la calidad del software. El objetivo de este TFG es determinar si alcanzar un determinado porcentaje de cobertura de código garantiza su calidad. Se realizará un experimento para comparar los resultados de la generación de cobertura automatizada mediante un proceso iterativo para aumentar la cobertura en cada iteración.

Abstract

Code coverage generation is essential to assess the effectiveness of testing in software development. Automating this process improves error detection and software quality. The objective of this TFG (Final Degree Project) is to determine whether achieving a specific code coverage percentage guarantees its quality. An experiment will be conducted to compare the results of automated coverage generation using an iterative process to increase coverage in each iteration.

Índex.

Índex de figures. III

Índex de taules. VII

Glossari de termes. IX

1. Objecte del projecte. 1

2. Marc teòric 3

2.1 Antecedents 3

2.2 Necessitats d'informació 4

2.2.1 Cobertura de codi 4

2.2.2 Qui cobreix la cobertura de codi? 5

2.2.3 Regles de negoci 6

2.2.4 Creació de la cobertura de regles de negoci 7

2.2.5 Llenguatge de programació Java 8

2.2.6 Java "Reflection" 8

2.2.7 IntellIJ 9

2.2.8 IntelliTest 10

2.2.9 Anàlisis de codi i "Code Coverage" 11

2.2.10 Large Language Models (LLM): ChatGPT & GitHubCopilot 12

3. Abast del projecte. 15

4. Metodologia 17

5. Requeriments funcionals i tecnològics. 19

6. Desenvolupament 21

6.1 Primera iteració 21

6.2 Segona iteració 25

6.3 Tercera iteració 28

6.4 Quarta iteració 31

6.5 Cinquena iteració 33

6.6 Sisena iteració 35

6.7 Setena iteració 37

6.8 Vuitena iteració 40

6.9 Novena iteració 42

6.10 Desena iteració 44

6.11 Resultats iteracions 46

7. Conclusions 49

8. Referències. 51

Índex de figures.

Fig 2.1. Twit Allan holub 3

Fig. 2.2.1. Cobertura de codi 5

Fig. 2.2.3. Cobertura de regles de negoci 6

Fig. 2.2.4. Cobertura de regles de negoci 7

Fig. 2.2.7. Cobertura de codi IntellIJ 10

Fig 2.2.8 IntelliTest 11

Fig 2.2.9 Cobertura "if" 12

Fig 2.2.9 Cobertura "while" 12

Fig 2.2.9 Cobertura "switch" 12

Fig 4. Iteracions 17

Fig 6.1. Resultat test 22

Fig 6.1. Variables projecte 22

Fig 6.1. "main" 23

Fig 6.1. "getPackagesFromPackage" 23

Fig 6.1. "getClassesFromPackage" 24

Fig 6.1. "generateJUnitForClass" 24

Fig 6.1. "writeFile" 25

Fig 6.2. Resultat test 25

Fig 6.2. "main" 26

Fig 6.2. "generateJUnitForClass" 27

Fig 6.2. "constructorParametrType" 28

Fig. 6.3 Resultat Test 29

Fig. 6.3 “methodsTests" 29

Fig. 6.3 “methodCallTests” 30

Fig. 6.3 “methodParameters” 31

Fig. 6.4 Resultat Tests 32

Fig. 6.4 “methodCallTests” 33

Fig. 6.5 Resultat Tests 34

Fig. 6.5 “privateMethodCallTests” 35

Fig. 6.6 Resultat Tests 36

Fig. 6.6 “constructorParametrType case” 36

Fig. 6.6 “constructorParametrType default” 37

Fig. 6.7 Resultat Tests 38

Fig. 6.7 “methodParameters” 39

Fig. 6.7 “methodCallTests” 40

Fig. 6.8 Resultat Tests 41

Fig. 6.8 “constructorParametrType” 41

Fig. 6.8 “switch” per a mètodes i constructors 42

Fig. 6.9 Test 43

Fig. 6.9 “constructorsTests” 43

Fig. 6.9 Resultat Tests 44

Fig. 6.10 Resultat Tests 45

Fig. 6.10 “generateJUnitForClass” part 1 45

Fig. 6.10 “generateJUnitForClass” part 2 46

Fig. 6.10 “constructorsTests” 46

Fig. 6.11 Resultats iteracions 47

Fig. 6.11 Resultats iteracions solucionades 48

Índex de taules.

Taula 6.11 Resultats iteracions 47

Taula 6.11 Resultats iteracions solucionades 48

Glossari de termes.

Code coverage	Cobertura de codi
Busines rules coverage	Cobertura de les regles de negoci
BDD	Behavior Driven Development
C++	Llenguatge de programació C++
If	Condicional de llenguatge Java
TFG	Treball Final de Grau
IDE	Entorn de Desenvolupament Integrat

1. Objecte del projecte.

La idea darrera del propi projecte es facilitar la obtenció de la cobertura de codi requerida, i intentar eliminar la creença que un bon "code coverage" és el mateix que una bona qualitat de codi. Si el projecte pot arribar a alts nombres de "code coverage", sense escriure cap codi, demostra que més "coverage" no vol dir més qualitat.

L'objecte del projecte es crear un creador de testos automàtic. Aquest creador llegirà el nom de classes, variables i mètodes, un cop llegides, intentarà executar el màxim possible provant entrades aleatòries.

El projecte pretén ser un procés iteratiu. Es vol anar creant el software de mica en mica, aprenent com es pot millorar, començant analitzant projectes petits i anar escalant poc a poc amb la complexitat d'aquests.

Així doncs, l'objectiu del projecte no es pot quantificar amb números, sinó que es demostrar el que es diu al primer paràgraf, que més "code coverage" no implica més qualitat. Si bé és cert que es difícil definir un objectiu en concret, es sap quan es compleix l'objectiu de les diferents iteracions quan els software dissenyat arribi a un "coverage" el més alt possible.

2. Marc teòric

En aquest tema s'analitzen els antecedents que han portat a la creació del propi projecte i alguns conceptes i informació bàsica que ajudarà a entendre amb més detall el propi projecte.

2.1 Antecedents

Allen Holub *twitejava* aquest estiu de 2022 transmetent els seus pensaments sobre crear un programa que generes de forma automàtica un "code coverage" del 80%. Aquest faria crides a les diferents funcions i mètodes del programa amb arguments aleatoris i sempre passarien.

Allen Holub @allenholub@mstdn.social
@allenholub

I've thought about writing an automated coverage generator that just created tests that called every function/method in a program with random arguments and always passed. Poof! 80% coverage! Coverage is not a useful metric.

Tradueix el tuit

1:41 a. m. · 20 d'ag. de 2022

Fig. 2.1. Twit Allen holub [5]

Aquest projecte té per objectiu posar a prova aquesta hipòtesi, veure la viabilitat d'aquesta idea i porta-la el més lluny possible.

Es planteja que tan útil pot arribar a ser la cobertura de codi. Ja que la cobertura de codi només fa referencia al codi en sí mateix, no a la totalitat del projecte, no es tenen en compte aspectes com la interfície d'usuari, el rendiment, les regles de

negoci, la integració d'aquesta amb altres sistemes, etc. Així doncs es pot dir que la cobertura de codi no proporciona una visó completa en termes de funcionalitat i usabilitat. I si la cobertura de codi es pogués fer de forma automàtica, es pot demostrar que obligar a un mínim de cobertura de codi no és efectiu.

2.2 Necessitats d'informació

En aquest apartat es dona a conèixer la informació bàsica necessari a per a poder entendre tot el projecte. El coneixement que s'explica a continuació, es una part necessària per a poder entendre la resta del projecte.

David Ródenas i Picó ens explica en el seu article a Mèdium [2] molts aspectes importants que necessitem entendre abans d'aprofundir en aquest projecte.

La cobertura de codi indica quines parts s'executen en els testos, però no garanteix la qualitat ni el funcionament de l'aplicació.

Les proves de codi es centren en el mateix codi i no en les regles de negoci, deixant de cobrir els objectius comercials.

Cal crear una mètrica per avaluar la cobertura de les regles de negoci en els tests, però és un repte a causa de la seva naturalesa no estandarditzada i difícil d'automatitzar.

El BDD permet generar tests basats en criteris d'acceptació i calcular la cobertura de les regles de negoci. No obstant això, no tots els tests es poden automatitzar.

Als següents apartats, aquesta informació es veu més detallada.

2.2.1 Cobertura de codi

Un dels estàndards més bàsics i extensos a l'hora de parlar sobre la qualitat i que tant ben dissenyat són els testos del codi, és la cobertura de codi, o "code coverage". Aquests paràmetre es calcula de forma automàtica un cop s'executen

els testos, i es te en compte quines parts del codi s'executen mitjançant els testos i quines no. El **percentatge de cobertura és** resultat de dividir les línies executades entre les totals del codi, aquest resultat es la cobertura que fan els tests sobre el codi.

$$code\ coverage\,\% = \frac{executed\ code\ by\ tests}{size\ of\ the\ code}$$

Fig. 2.2.1. Cobertura de codi [2]

Encara que la cobertura de codi es fa servir com a mètrica per avaluar la qualitat dels testos, també ofereix altre tipus d'informació sobre el codi que pot ser molt important. Primerament, pot donar informació sobre les parts del codi que no s'utilitzen. Ja que normalment es tendeix a sobre analitzar els possibles casos que es poden produir, moltes parts del codi acaben no sent funcionals. Això pot no semblar evident a primera vista, però amb el "code coverage" podem veure on els testos no arriben mai a executar línies de codi.

Però buscar sempre la màxima cobertura de codi no es bona idea. La raó es que aconseguir un percentatge alt de cobertura es relativament fàcil, però això no és garantia que els testos creats siguin bons o vàlids. A vegades es fan testos per arribar al mínim "coverage" que es necessita, sense tenir en compte si l'aplicació funciona no. I encara que sembli impossible, aquesta classe de males practiques es duen a terme.

2.2.2 Qui cobreix la cobertura de codi?

Els propis desenvolupadors del codi són els encarregats de realitzar les proves, les quals es basen principalment en el propi codi i no en l'aplicació en si mateixa. Com a resultat, no es tenen en compte els objectius comercials

Quan els desenvolupadors posen més èmfasi en el codi que en les regles de negoci,

hi ha un desequilibri en el plantejament dels tests. En lloc de centrar-se únicament en el codi, s'han d'enfocar els tests cap a les implementacions que afecten directament les regles de negoci. Aquesta transició en la forma de testejar dona lloc a tests que tenen un significat i una rellevància més contundents en relació a les regles de negoci en sí.

La cobertura de codi permet al desenvolupadors saber que es el que realment fa faltar provar. Ja que podem tenir una cobertura molt elevada, però que aquesta faci referencia al codi en si, i que la part que no està coberta representi les regles de negoci. Llavors veuríem que encara que el codi funciona com es esperat, la pròpia aplicació podria arribar a fallar.

Això no es gens estrany, de fet, es el més comú. Normalment, les parts relacionades al propi negoci, no son gens fàcils de provar, i així porta a que els propis desenvolupadors decideixin no fer-ho.

Per tant es pot dir, que la cobertura de codi, no es res més que una prova per al codi.

2.2.3 Regles de negoci

Se sap que la cobertura de codi és quin percentatge del codi es executat per els testos. Però, i si es crea un paràmetre que ens indiqui quin es el percentatge de regles de negoci que s'estan executant?

Seria una mètrica que mesura fins a quin punt les regles de negoci estan cobertes pels tests d'un programa. Les regles de negoci són les condicions, restriccions i comportaments que defineixen com ha de funcionar l'aplicació per satisfer els objectius i les necessitats de l'empresa o dels usuaris.

$$business\,rules\,coverage\,\% = \frac{automated\,acceptance\,criteria}{total\,acceptance\,criteria}$$

Fig. 2.2.3. Cobertura regles de negoci [2]

Pot semblar una idea fàcil, però realment no es fàcil de medi, com a mínim no tant com la cobertura de codi. Primerament, les regles de negoci no sempre son clares, no sempre estan escrites en documents normalitzats, i això deriva en que són difícils d'automatitzar. Es comú que estiguin escrites en llenguatges col·loquials o imprecisos, i que això doni lloc a lliures interpretacions. I aquestes s'han de traduir abans de que pugin ser automatitzats.

2.2.4 Creació de la cobertura de regles de negoci

Molts d'aquests problemes ja han estat resolts, i l'exemple més clar es en "Behaviour Driven-Development" (BDD). Amb aquest procés es poden escriure les criteris d'acceptació de manera natural, i la pròpia màquina s'encarrega d'executar-los.

Si s'escriuen tots els criteris d'acceptació en BDD, llavors es pot començar a comptabilitzar. El càlcul és realment semblant al que es fa servir per a la cobertura de codi. Primer es conten quants criteris s'han escrit, i la quantitat d'aquests que han estat automatitzats. El resultat de dividir ambos números, es la cobertura que es té de les regles de negoci en percentatge.

$$business\, rules\, coverage\, \% = \frac{automated\, scenarios}{total\, scenarios}$$

Fig. 2.2.4. Cobertura regles de negoci[2]

El numero resultat indica que tan fiable es la aplicació que es posa a prova, tenint en compte les regles de negoci que pretén complir.

Tot i això, cal tenir en compte que encara que BDD proporciona una estructura per escriure tots els tests, és important destacar que no tots els tests es poden automatitzar. Les proves d'usabilitat o proves de validació visual necessiten d'interacció humana per dur-se a terme de forma eficient. Per tant es important combinar aquest mètode amb altres tècniques de test per tal d'aconseguir una cobertura completa i una validació adequada.

2.2.5 Llenguatge de programació Java

Java es un llenguatge orientat a objectes, que utilitza una sintaxis semblant a la de C++ però més reduïda. Es un llenguatge fàcil d'aprendre i que disposa d'un gran ventall de funcionalitats. Aquest llenguatge de programació ofereix un maneig automàtic de la memòria, el que porta a una disminució dels errors.

Java té una comunitat de programadors molt extensa, arribant a xifres que ronden els nou milions[6]. A més també es una comunitat molt activa, el que permet tenir a mà molta informació actualitzada.

Avui dia existeixen molt llenguatges de programació, i Java és un dels més estesos. I no només això, sinó que també es un dels millor remunerats en la industria de la programació[7]. En moltes pagines webs importants com ve a ser Amazon s'utilitza aquest llenguatge.

2.2.6 Java "Reflection"

La introspecció a Java, o Java "Reflection", és la capacitat que te aquest llenguatge que el permet examinar i obtenir informació sobre classes, mètodes, constructors i camps en temps d'execució. Amb aquest recurs, es possible accedir a tota aquesta informació de manera dinàmica, sense necessitat de conèixer el codi.

La introspecció es basa en l'ús de la classe "Class" i les classes relacionades, fent-les servir com a punt d'entrada per rebre la informació d'una classe especifica. Gracies a la classe "Class", es poden obtenir mètodes, constructors, i variables de la classe desitjada. Es pot fins hi tot inspeccionar i modificar atributs i invocar mètodes en temps d'execució.

Es important tenir en compte que la introspecció pot tenir un impacte en el rendiment de l'aplicació en la que es fa servir, ja que implica una avaluació dinàmica de les classes i l'execució de crides reflectides. És important utilitzar-la de manera apropiada i eficient per tal d'evitar que l'aplicació baixi el seu

rendiment.

2.2.7 IntellIJ

IntellIJ es un editor de JetBrains, que va veure la llum el gener de 2001. Usable en Windows, macOS i Linux.

IntellIJ compta amb diverses funcionalitats d'assistència que durant la codificació ajuda a l'usuari a veure els errors i ofereix possibles solucions de millora mentre aquest escriu, a l'hora que mostra altres practiques de la comunitat respecte a la codificació, noves funcionalitats del llenguatge i molt més.

IntellIJ coneix tot sobre el codi que s'està escrivint, i utilitza aquest coneixement per oferir a l'usuari opcions rellevants en qualsevol context que poden facilitar la seva experiència.

Permet treballar amb un equip en temps real, creant sessions ja sigui per depurar codi, revisar-lo, etc. Permet també traslladar els projectes a maquines remotes per a poder utilitzar tots els seus recursos.

Ofereix a l'usuari una sèrie d'eines des del primer moment, aquestes, entre d'altres, són una sèrie de llenguatges compatibles i macros de treball. I cap d'aquestes eines té cap necessitat de complements massa complicats. Una d'aquestes eines, i que pot ser molt útil de cara a aquest projecte, es un analitzador de "code coverage".

Coverage: com.drpicox.game in game22

Element	Class, %	Method, %	Line, %
com.drpicox.game	100% (78/78)	95% (347/362)	84% (848/999)
ClassroomCardsGame2022	100% (1/1)	0% (0/1)	50% (1/2)
moon	100% (2/2)	100% (2/2)	100% (4/4)
EndMoonStep	100% (0/0)	100% (0/0)	100% (0/0)
MoonService	100% (1/1)	100% (2/2)	100% (3/3)
EndMoonSettings	100% (1/1)	100% (0/0)	100% (1/1)
constants	100% (3/3)	100% (21/21)	97% (82/84)
Constants	100% (1/1)	100% (12/12)	100% (47/47)
ConstantsLoader	100% (1/1)	100% (6/6)	100% (22/22)
ConstantsCollection	100% (1/1)	100% (3/3)	86% (13/15)
tag	100% (7/7)	94% (18/19)	70% (46/65)
TagRepository	100% (0/0)	100% (0/0)	100% (0/0)
api	100% (1/1)	100% (2/2)	100% (3/3)
food	100% (1/1)	66% (2/3)	34% (10/29)
Tag	100% (1/1)	100% (5/5)	100% (8/8)
TagId	100% (1/1)	100% (1/1)	100% (1/1)
TagFactory	100% (1/1)	100% (2/2)	100% (14/14)
TagService	100% (1/1)	100% (2/2)	100% (4/4)
TagFactorySettings	100% (1/1)	100% (4/4)	100% (6/6)
game	100% (8/8)	100% (21/21)	98% (49/50)
GameDeleteStep	100% (0/0)	100% (0/0)	100% (0/0)
GameRepository	100% (0/0)	100% (0/0)	100% (0/0)
GameFactoryStep	100% (0/0)	100% (0/0)	100% (0/0)
Game	100% (1/1)	100% (2/2)	100% (3/3)
GameFactory	100% (1/1)	100% (3/3)	100% (13/13)
GameService	100% (1/1)	100% (2/2)	100% (3/3)
GameFactorySettings	100% (1/1)	100% (5/5)	100% (7/7)
api	100% (4/4)	100% (9/9)	95% (23/24)
blog	100% (9/9)	100% (52/52)	91% (143/156)
util	100% (10/10)	86% (32/37)	66% (55/83)
card	100% (18/18)	97% (119/122)	87% (286/326)
idea	100% (20/20)	94% (82/87)	79% (182/229)

Fig. 2.2.7. Cobertura de codi IntellIJ. Font: Elaboració pròpia

2.2.8 IntelliTest

IntelliTest, creat per Microsoft Reserch el 2010, analitza el codi per a generar tests i un conjunt de proves unitàries.

Per a cada declaració en el codi, es genera una entrada de prova que executarà aquesta declaració. Es realitza un anàlisi de cas per a cada branca condicional en el codi. Per exemple, "if" i totes les operacions que poden generar excepcions s'analitzen. Aquesta anàlisi s'usa per a generar proves unitàries parametritzades per a cadascun dels mètodes, creant proves unitàries amb una alta cobertura de codi.

Quan s'executa IntelliTest, es poden veure fàcilment els tests que estan fallant i qualsevol part del codi que necessiti ser arreglada. Es pot seleccionar quin dels

tests generats guardar en un projecte de prova per a proporcionar un conjunt de regressió. A mesura que es canvia el codi, es pot tornar a executar IntelliTest per a seguir general tests que estiguin sincronitzats amb els canvis que han modificat el codi.

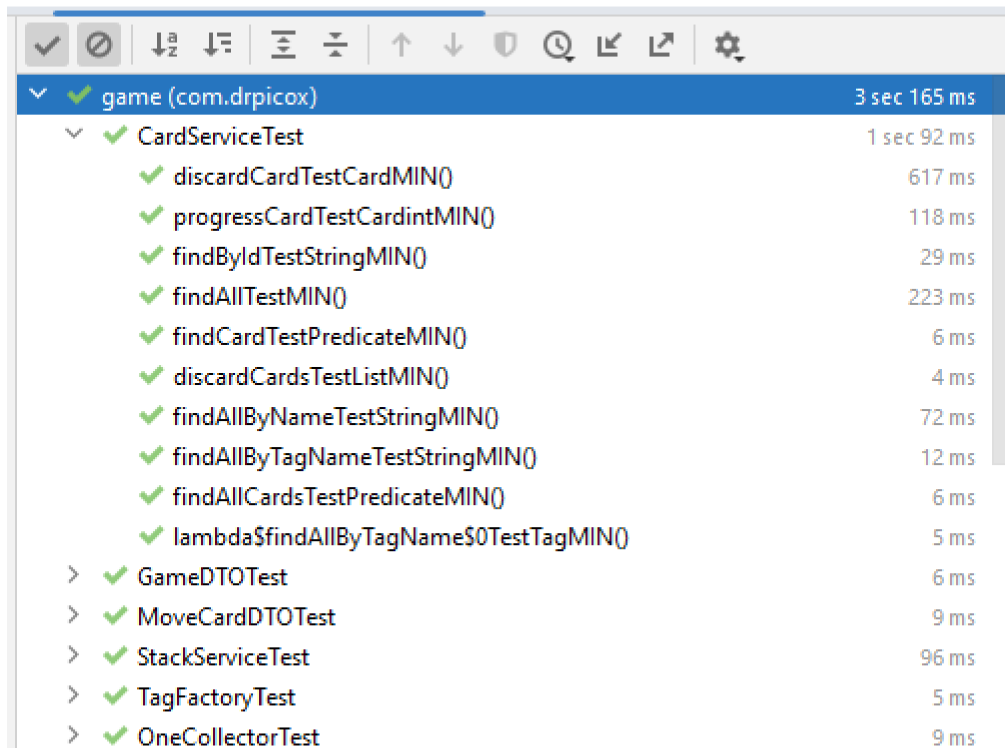

Fig 2.2.8 IntelliTest. Font: Elaboració pròpia

2.2.9 Anàlisi de codi i "Code Coverage"

Per tal de fer un anàlisi de codi i així augmentar la coberta de codi no cal que es satisfacin els requisits de l'aplicació, sinó simplement passar per totes les branques.

En el cas d'un "if", només s'han de fer 2 tests com a molt, un que compleixi la condició del propi condicional, per tal de poder entrar en la primera part del codi, i un altre test que no compleixi aquesta condició per tal de cobrir l' "else" en cas que existeixi.

```
if (condició) {
    //...
} else {
    //...
}
```

Fig 2.2.9 Cobertura "if". Elaboració pròpia

En cas dels "while", només cal crear un test que compleixi la condició i entre dins del bucle. Un cop dins ja s'executen les línies i es genera la cobertura de codi.

```
while (condició) {
    //...
}
```

Fig 2.2.9 Cobertura "while". Font: Elaboració pròpia

Un altre exemple serien els "switch", en aquest cas haurem de crear tants tests com "case" tingui, més un extra per el cas "default". En cada test es modifica el valor per tal de que l'entrada al "switch" vagi canviant i es generi la màxima cobertura possible.

```
switch (valor){
    case 1: //...
        break;
    case 2: //...
        break;
    default:
}
```

Fig 2.2.9 Cobertura "switch". Elaboració pròpia

2.2.10 Large Language Models (LLM): ChatGPT & GitHubCopilot

Els LLM, inclosos els desenvolupats per OpenAI, són una tecnologia que té com

a objectiu generar textos en diferents àmbits i estils, donant suport en la redacció de continguts, generació de codi, traducció, respostes a preguntes i molts altres casos d'ús. És una eina molt versàtil que utilitza models de llenguatge avançats per a entendre i produir text de manera coherent i precisa.

Dins de l'àmbit de la generació de codi, existeixen altres eines com el ChatGPT i GitHub Copilot, també desenvolupades per OpenAI. ChatGPT és una implementació específica del LLM que permet interactuar amb l'usuari en temps real i respondre preguntes o proporcionar informació en forma de text. Per part seva, GitHub Copilot és una eina de programació assistida per IA que ofereix suggeriments i autocompletat de codi basant-se en la comprensió de milions de línies de codi de programari obert.

A l'hora de generar codi, el ChatGPT i el GitHub Copilot són eines molt útils per als desenvolupadors, ja que ajuden a millorar la productivitat en el procés de desenvolupament al resoldre preguntes i proporcionar exemples de codi. Però és important recordar que sempre s'ha de revisar el codi que aquestes eines ofereixen, per tal de garantir-ne la qualitat i coherència amb els requisits del projecte.

3. Abast del projecte.

L'abast del projecte compren el disseny d'un nou software programat en Java com a mètode experimental per a comprovar que tan viable es la idea de que els testos relacionats a la cobertura de codi es facin de manera automàtica.

Es farà el disseny de manera iterativa, partint de la creació de testos per a un programa senzill, amb poques classes i mètodes, i intentant automatitzar-los. Un cop aquests siguin acceptats, es passarà a fer les proves en un altre programa i es milloraran els testos per a fer-los compatibles amb projectes cada vegada més grans

Caldrà estudiar tant els unit test així com les capacitats que te el propi llenguatge de programació, en aquest cas Java, per a poder extreure el màxim d'informació de les classes creades.

La fita concreta que a la que es vol arribar en cada iteració es la d'obtenir una cobertura de codi del vuitanta per cent, o en el seu defecte, el més alt possible.

Per tal de provar si la iteració actual es considera un èxit o cal continuar modificant-la, es posarà en marxa el projecte sobre un altre codi que es consideri més complicat que l'anterior. Si la cobertura de codi es superior o igual a la esmentada en el paràgraf anterior. S'obtindran les conclusions respecte la solució proposada, i es decidirà si s'avança o no cap a la següent iteració.

Fora de l'abast del projecte queda usar les LLM com a eina per a generar els tests i així tenir nivells més elevats de cobertura. Ja que el valor a la cobertura obtinguda ve donat de la desconeixença total del codi. Usar les LLM com a generadors de tests es considera un pas per a un altre projecte.

4. Metodologia

Aquest TFG es divideix en dues parts rellevants, la primera sent la cerca d'informació, i la segona sent l'aplicació d'aquesta i el desenvolupament del software relatiu al projecte.

Per tal de buscar informació, s'ha considerat adient fer un anàlisis de que pot ser el més interesant de conèixer per al lector d'aquest document per tal d'entendre la totalitat del projecte. I també s'ha buscat la informació necessària per a poder-lo dur a terme.

Per a la creació del software que aquest projecte vol crear, es fan diverses iteracions, en cada qual es busca de quina manera augmentar la cobertura de codi que aquest es capaç de proporcionar. Així mateix, es comença amb un projecte que intenta replicar totes les classes existents dins del paquet que l'usuari desitja, però sent aquestes noves testos en blanc per a cada una de les anterior.

Un cop assolit aquest primer pas, es generen testos de forma generalitzada per a totes les classes, sempre intentat millorar la cobertura assolida en la iteració anterior.

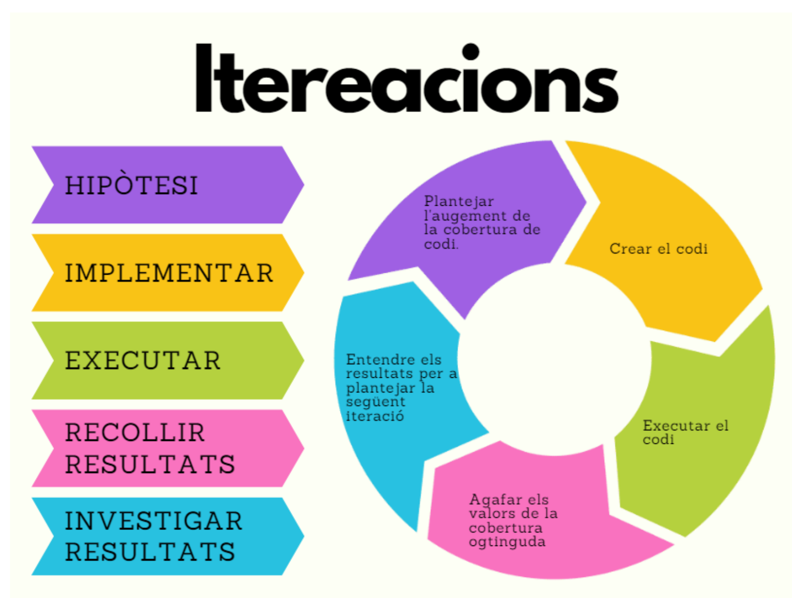

Fig 4. Iteracions. Elaboració pròpia

5. Requeriments funcionals i tecnològics.

Per a dur a terme aquest projecte es requereix de les següents funcionalitats:

- Un entorn que permeti l'ús de test
- Un entorn que permeti l'obtenció de cobertura de codi
- Un llenguatge que permeti l'escriptura de tests.
- Un llenguatge que permeti llegir les característiques de codi extern.
- Una llibreria que permeti la introspecció de codi
- Una llibreria que permeti la creació de tests

6. Desenvolupament

En aquest apartat es pot veure l’evolució del projecte, passant per les diferents iteracions. Amb l’objectiu de cada una, les millores respecte les següents iteracions, i una explicació del codi si es considera necessària.

Aquestes iteracions consten de:

- Primera iteració: crear testos de classes
- Segona iteració: constructors
- Tercera iteració: mètodes “void” públics
- Quarta iteració: mètodes públics
- Cinquena iteració: crea testos per a mètodes privats
- Sisena iteració: Augmentar el nombre variables per a les crides de constructors
- Setena iteració: Augmentar el nombre variables per a les crides de mètodes
- Vuitena iteració: Crear testos per a casos extrems
- Novena iteració: Correcció del problema amb els primers tests
- Desena iteració: Injecció automàtica de dependències

6.1 Primera iteració

La primera iteració té per objectiu crear tantes classes test com classes hi ha dins del paquet que definim. No només es mira les classes que hi ha dins del paquet, sinó que també es mira si hi ha més paquets dins i quines son les seves classes.

```java
package com.drpicox.tests;
import org.junit.Test;

public class AuthorsServiceTest {

    @Test
    public void test() {
        // Add test code here
    }

}
```

Fig. 6.1 Resultat test. Font: Elaboració pròpia

Primerament es demana que es defineixin tant el paquet que es vol llegir com el paquet on es crearan. En el cas del destí, se separa en dos "string" diferents, ja que més endavant necessitem aquesta divisió per als "import" de cada test.

```java
1 usage
static String packageName = "com.drpicox.game";
2 usages
static String outputDirShort = "com.drpicox.tests";
1 usage
static String outputDir = "src/test/java/" + outputDirShort;
```

Fig. 6.1 Variables projecte. Font: Elaboració pròpia

El "main" del projecte fa una crida a diferents mètodes per tal de fer una cerca de tot el que hi ha en el paquet dins la variable de "packageName", la segona crida permet trobar totes les classes que hi ha dins dels paquets trobats prèviament.

Es fa una ultima crida per tal de crear el test per la classe amb la que s'està treballant i per últim es crea la nova classe test.

```java
public static void main(String[] args) {
    try {
        List<String> packageNames = getPackagesFromPackage(packageName);
        for (String pkg : packageNames) {
            List<Class<?>> classes = getClassesFromPackage(pkg);
            for (Class<?> clazz : classes) {
                String junitCode = generateJUnitForClass(clazz);
                String fileName = outputDir + "/" + clazz.getSimpleName() + "Test.java";
                writeFile(fileName, junitCode);
            }
        }
    } catch (ClassNotFoundException | IOException e) {
        e.printStackTrace();
    }

    System.out.println("Done");
}
```

Fig. 6.1 “main”. Font: Elaboració pròpia

La primera de les crides a main, al mètode getPackagesFromPackage, és un mètode recursiu, que a través del nom d’un paquet busca els paquets que s’hi troben dins. I en cas de trobar un altre paquet, es crida a ell mateix per tal de treure els paquets del segon paquet. Aquest procés es repeteix fins que troba tots els paquets dins del paquet que se li envia com a paràmetre i que no contenen cap paquet més dins d’ells mateixos. Finalment retorna una llista de “strings” amb el nom dels paquets que ha trobat.

```java
private static List<String> getPackagesFromPackage(String packageName)
throws ClassNotFoundException, IOException {
    List<String> packages = new ArrayList<>();
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    String path = packageName.replace( oldChar: '.',  newChar: '/');
    for (File file : new File(classLoader.getResource(path).getFile()).listFiles()) {
        if (file.isDirectory()) {
            String subPackage = packageName + "." + file.getName();
            packages.add(subPackage);
            packages.addAll(getPackagesFromPackage(subPackage));
        }
    }
    return packages;
}
```

Fig.6.1 “getPackagesFromPackage”. Font: Elaboració pròpia

La segona crida de main, al mètodes getClassesFromPackage, fa una troba totes les classes que hi ha dins del paquet enviat com a paràmetre, i retorna una llista de classes

que ha trobat.

```
private static List<Class<?>> getClassesFromPackage(String packageName) throws ClassNotFoundException {
    List<Class<?>> classes = new ArrayList<>();
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    String path = packageName.replace( oldChar: '.',  newChar: '/');
    for (File file : new File(classLoader.getResource(path).getFile()).listFiles()) {
        if (file.isFile() && file.getName().endsWith(".class")) {
            String className = packageName + '.' + file.getName().substring(0, file.getName().length() - 6);
            classes.add(Class.forName(className));
        }
    }
    return classes;
}
```

Fig. 6.1 “getClassesFromPackage”. Font: Elaboració pròpia

Un cop trobades totes les classes que conté el paquet inicial, es moment de enviar les classes una a una per a generar la informació d’una nova classe test per a cada una d’elles. En aquesta primera iteració, la classe test que es crea es buida, només conte els “import” necessaris i la informació del paquet on es troba. Crea un “string” amb tota la informació necessària per a la creació de la classe i el retorna.

```
private static String generateJUnitForClass(Class<?> clazz) {
    String className = clazz.getSimpleName();
    String junitCode = "package " + outputDirShort + ";\n";
    junitCode += "import org.junit.Test;\n\n";
    junitCode += "public class " + className + "Test {\n\n";
    junitCode += "    @Test\n";
    junitCode += "    public void test() {\n";
    junitCode += "        // Add test code here\n";
    junitCode += "    }\n\n";
    junitCode += "}\n";
    return junitCode;
}
```

Fig. 6.1 “generateJUnitForClass”. Font: Elaboració pròpia

Un cop retornat el “string” amb la informació de la classe test, es crea un nou fitxer en la direcció ingressada com a “output” amb la informació generada per el mètode anterior, “generateJUnitForClass”.

```java
    private static void writeFile(String fileName, String content) throws IOException {
        File file = new File(fileName);
        file.getParentFile().mkdirs();
        Files.write(Paths.get(fileName), content.getBytes(), StandardOpenOption.CREATE);
    }
}
```

Fig. 5.1 “writeFile”. Font: Elaboració pròpia

A l’apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s’obtenen. En el cas d’aquesta primera iteració, la cobertura de codi es del 0% tant el classes, com en mètodes i línies recorregudes.

6.2 Segona iteració

Aquesta iteració, al igual que les seves posteriors, té objectiu d’ampliar els test creats en la iteració anterior per tal d’assolir una cobertura de codi més elevada. Per tal d’aconseguir-ho, es generen els test per a cada constructor de la classe.

```java
package com.drpicox.tests;
import com.drpicox.game.blog.AuthorsService;
import org.junit.Test;

public class AuthorsServiceTest {

    AuthorsService instance;

    @Test
    public void AuthorsServiceTestConstantsLoader() {
        try{
            instance = new AuthorsService( constantsLoader: null);
        } catch (Exception e) {}
    }

}
```

Fig. 6.2 Resultat test. Font: Elaboració pròpia

En aquest cas es fa un petit canvi al “main” del projecte per a impedir que aquest intenti crear test si la classe que esta mirant es una interfície. Ja que no es poden realitzar proves unitàries directament sobre una.

```
public static void main(String[] args) {
    try {
        List<String> packageNames = getPackagesFromPackage(packageName);
        for (String pkg : packageNames) {
            List<Class<?>> classes = getClassesFromPackage(pkg);
            for (Class<?> clazz : classes) {
                if (!clazz.isInterface()) {
                    String junitCode = generateJUnitForClass(clazz);
                    junitCode += "}\n";
                    String fileName = outputDir + "/" + clazz.getSimpleName() + "Test.java";
                    writeFile(fileName, junitCode);
                }
            }
        }
    } catch (ClassNotFoundException | IOException e) {
        e.printStackTrace();
    }

    System.out.println("Done");
}
```

Fig. 6.2 “main”. Font: Elaboració pròpia

El canvi important d’aquesta iteració ve donat al mètode “generateJUnitForClass”, ja que en aquest punt del desenvolupament es fan testos per als constructors de cada classe, si es que en tenen. La primera part del mètode segueix igual, es crea el principi de la classe test, però a l’hora de fer els test d’aquesta es on es troba el primer canvi.

Es busquen els constructors de la classe i en cas de tindre es creen tantes instancies de la classe com constructors hi ha.

Un cop fetes les instancies de la classe es fan testos per als constructors que la classe tingui. Els testos son generalitzats, es fa un “try” “catch” per prevenir errors i excepcions que es puguin produir.

Per a fer la crida al constructor es fan les crides a les diferents instancies que s’han creat anteriorment.

I la s’envien al constructor els paràmetres necessaris amb valors per defecte.

```java
private static String generateJUnitForClass(Class<?> clazz) {
    String className = clazz.getSimpleName();
    String junitCode = "package " + outputDirShort + ";\n";
    junitCode += "import " + clazz.getPackageName() + "." + className + ";\n";
    junitCode += "import org.junit.Test;\n\n";
    junitCode += "public class " + className + "Test {\n\n";
    Constructor[] constructors = clazz.getConstructors();
    int i = 1;
    for (Constructor constructor : constructors){
        if (i == 1)
            junitCode += "\t" +className + " instance";
        else
            junitCode += "\t" +className + " instance" + i;
        junitCode += ";\n\n";
        i++;
    }
    i = 1;
    for (Constructor constructor : constructors){
        junitCode += "\t@Test\n";
        junitCode += "\tpublic void " + className + "Test";
        for (Class<?> paramType : constructor.getParameterTypes()){
            String param = paramType.getSimpleName();
            junitCode += param.replace( target: "[]",  replacement: "$$");
        }
        junitCode += "() {\n";
        junitCode += "\t\ttry{\n";
        if (i == 1)
            junitCode += "\t\t\tinstance = new " + className + "(";
        else
            junitCode += "\t\t\tinstance" + i + " = new " + className + "(";
        junitCode += constructorParametrType(constructor);
        junitCode += ");\n";
        junitCode += "\t\t} catch (Exception e) {}\n";
        junitCode += "\t}\n\n";
        i++;
    }
    return junitCode;
}
```

Fig. 6.2 “generateJUnitForClass”. Font: Elaboració pròpia

En la imatge següent es pot veure com es busquen els diferents paràmetres que hi ha al constructor i es crea un “string” amb els seus valors per defecte. Un cop acabats els paràmetres es retorna al mètode anterior per acabar de crear el test per al constructor.

```java
private static String constructorParametrType(Constructor constructor) {
    String junitCode = "";
    Class[] parameters = constructor.getParameterTypes();
    int i = 0;
    for (Class parameter : parameters) {
        switch(parameter.getSimpleName()) {
            case "byte": junitCode += "0";
                break;
            case "short": junitCode += "0";
                break;
            case "int": junitCode += "0";
                break;
            case "long": junitCode += "0.0L";
                break;
            case "float": junitCode += "0.0f";
                break;
            case "double": junitCode += "0.0d";
                break;
            case "char": junitCode += "\' \'";
                break;
            case "String": junitCode += "\" \"";
                break;
            case "boolean": junitCode += "false";
                break;
            default: junitCode += "null";
        }
        if (i < constructor.getParameterCount() - 1) {
            junitCode += ", ";
        }
        i++;
    }
    return junitCode;
}
```

Fig. 6.2 “constructorParametrType”. Font: Elaboració pròpia

A l’apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s’obtenen. En el cas d’aquesta segona iteració, la cobertura de codi es del 91% per a les classes, 19% en mètodes i 20% en les línies recorregudes.

6.3 Tercera iteració

Aquesta iteració, al igual que les seves posteriors, té objectiu d’ampliar els test creats en la iteració anterior per tal d’assolir una cobertura de codi més elevada.

Aquesta però, es centra en ampliar la cobertura a traves de crear testos per als mètodes que són “void”, és a dir, que no tenen cap retorn, de les diferents classes que s’han trobat en les iteracions anteriors.

En aquest apartat els testos es veuen de la següent manera. No es fa cap canvi al resultat dels apartats anteriors, però a s'afegeixen testos per a cada un dels mètodes de les classes. I els mètodes "void" tenen el codi del seu test creat. En el cas dels mètodes amb retorn, tenen el test creat però no omplert.

```
@Test
public void AuthorsServiceTestConstantsLoader() {
    try{
        instance = new AuthorsService( constantsLoader: null);
    } catch (Exception e) {}
}

@Test
public void containsGitHubUserTestString() {
    try{
        //TODO: Add code here
    } catch (Exception e) {}
}
```

Fig. 6.3 Resultat Test. Font: Elaboració pròpia

Primerament es crea un nou mètode, "methodsTests" que rep com a paràmetre la classe de la qual es volen crear els testos i el nom de la mateixa classe. Aquest obtindrà els mètodes de la classe creats per l'usuari un a un y comprovarà quins són públics. Si el mètode trobat ho és, es generà un nou test per al mètode en concret. I es fa una crida a "methodCallTests" per generar el contingut d'aquests.

```
public static String methodsTests(Class<?> clazz, String className){
    String junitCode = "";
    for (Method method : clazz.getDeclaredMethods()){
        int modifiers = method.getModifiers();
        if(Modifier.isPublic(modifiers)) {
            junitCode += "\t@Test\n";
            junitCode += "\tpublic void " + method.getName() + "Test";
            for (Class<?> paramType : method.getParameterTypes()) {
                String param = paramType.getSimpleName();
                junitCode += param.replace( target: "[]",  replacement: "$$");
            }
            junitCode += "() {\n";
            junitCode += "\t\ttry{\n";
            junitCode += methodCallTests(clazz, method);
            junitCode += "\t\t} catch (Exception e) {}\n";
            junitCode += "\t}\n\n";
        }
    }
    return junitCode;
}
```

Fig. 6.3 "methodsTests". Font: Elaboració pròpia

En el mètode “methodCallTests” es rep com a paràmetre la classe i el mètode amb el que s’està treballant. Primerament es comprova si el mètode que s’envia es “void” o no, en cas de ser-ho es comença a escriure el codi per al test. Aquest utilitza la instancia que s’ha creat anteriorment per fer una crida al mètode del que es vol obtenir cobertura. I es fa una crida al mètode “methodParameters” per tal d’escriure la crida al mètode amb els paràmetres adients.

```
private static String methodCallTests(Class<?> clazz, Method method) {
    String junitCode = "\t\t\t";
    switch (method.getReturnType().getSimpleName()){
        case "void":
            if (!Modifier.isStatic(method.getModifiers())){
                junitCode += "instance." + method.getName() + "(";
                int i = 0;
                for (Class<?> param : method.getParameterTypes()){
                    i++;
                    junitCode += methodParameters(param);
                    if(method.getParameterTypes().length != i)
                        junitCode += ", ";
                }
                junitCode += ");\n";
            } else {
                junitCode += clazz.getSimpleName() + "." + method.getName() + "(";
                int i = 0;
                for (Class<?> param : method.getParameterTypes()){
                    i++;
                    junitCode += methodParameters(param);
                    if(method.getParameterTypes().length != i)
                        junitCode += ", ";
                }
                junitCode += ");\n";
            }
            break;
        default:
            junitCode += "//TODO: Add code here\n";

    }
    return junitCode;
}
```

Fig. 6.3 “methodCallTests”. Font: Elaboració pròpia

“methodParameters” té un funcionament molt similar al mètode “constructorParametrType”. Per a cada tipus de valor es dona com a resultat un valor per defecte, per tal de que els paràmetres que el mètode demana es pugin enviar.

```
private static String methodParameters(Class param){
    String junitCode = "";
    switch(param.getSimpleName()) {
        case "byte": junitCode += "0";
            break;
        case "short": junitCode += "0";
            break;
        case "int": junitCode += "0";
            break;
        case "long": junitCode += "0.0L";
            break;
        case "float": junitCode += "0.0f";
            break;
        case "double": junitCode += "0.0d";
            break;
        case "char": junitCode += "\u0000'";
            break;
        case "String": junitCode += "\" \"";
            break;
        case "boolean": junitCode += "false";
            break;
        default:
            junitCode += "null";
    }
    return junitCode;
}
```

Fig. 6.3 “methodParameters”. Font: Elaboració pròpia

A l’apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s’obtenen. En el cas d’aquesta tercera iteració, la cobertura de codi es del 91% per a les classes, 19% en mètodes i 20% en les línies recorregudes.

6.4 Quarta iteració

Aquesta iteració, al igual que les seves posteriors, té objectiu d’ampliar els test creats en la iteració anterior per tal d’assolir una cobertura de codi més elevada.

Per tal d’aconseguir-ho, en aquesta iteració es fan els testos per a aquells mètodes que si tenen algun tipus de retorn, és a dir, tots aquells mètodes que no són “void”. Per tant en canvia el mètode “methodCallTests” i s’afegeix codi que permet escriure un test generalitzat per a aquest casos.

Un cop posat en marxa, els testos dels mètodes que tenen algun tipus de retorn es veuen de la següent manera. Els testos anteriors no es veuen afectats, només

s'omplen les test on abans hi havia un comentaria de "//TODO: Add code here".

```
@Test
public void getGitHubUsersTest() {
    try{
        java.util.Collection retorn = instance.getGitHubUsers();
    } catch (Exception e) {}
}
```

Fig. 6.4 Resultat Tests. Font: Elaboració pròpia

S'aconsegueix el tipus de retorn que te el mètode que es crida per tal de crear una variable en el test d'aquest tipus. Un cop fet es mira si el mètodes es o no "static", ja que en cas de ser-ho no cal utilitzar la instancia que es crea en apartats anteriors per a fer la crida del mètode, i en cas que no ho sigui si fa faltà fer-la servir.

Un cop decidida com es fa la crida, es torna a crida al mètode "methodParameters" per tal d'enviar els paràmetres que necessiti el mètode del que s'està fent el test per a funcionar.

```java
private static String methodCallTests(Class<?> clazz, Method method) {
    String junitCode = "\t\t\t";
    switch (method.getReturnType().getSimpleName()){
        case "void":
            if (!Modifier.isStatic(method.getModifiers()))
                junitCode += "instance." + method.getName() + "(";
            else
                junitCode += clazz.getSimpleName() + "." + method.getName() + "(";
            int i = 0;
            for (Class<?> param : method.getParameterTypes()){
                i++;
                junitCode += methodParameters(param);
                if(method.getParameterTypes().length != i)
                    junitCode += ", ";
            }
            junitCode += ");\n";

            break;
        default:
            junitCode += method.getReturnType().getCanonicalName() + " retorn = ";
            if (!Modifier.isStatic(method.getModifiers()))
                junitCode += "instance." + method.getName() + "(";
            else
                junitCode += clazz.getSimpleName() + "." + method.getName() + "(";
            int j = 0;
            for (Class<?> param : method.getParameterTypes()){
                j++;
                junitCode += methodParameters(param);
                if(method.getParameterTypes().length != j)
                    junitCode += ", ";
            }
            junitCode += ");\n";
            break;

    }
    return junitCode;
}
```

Fig. 6.4 “methodCallTests”. Font: Elaboració pròpia

A l’apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s’obtenen. En el cas d’aquesta quarta iteració, la cobertura de codi es del 97% per a les classes, 25% en mètodes i 23% en les línies recorregudes.

6.5 Cinquena iteració

Aquesta iteració del projecte és una mica diferents a les anteriors. Bo i que també busca augmentar la cobertura de codi, en aquesta iteració s’experimenta amb els mètodes privats de les classes. En un principi, no s’hauria de poder augmentar la cobertura invocant aquest tipus de mètodes, però ja que tot el projecte tracta sobre

experimentar en l'augment de cobertura de codi, es considera important portar aquesta prova endavant.

En la següent imatge es pot veure un exemple de com queden els testos creats, enfocats a invocar mètodes privats, ja que fer una crida normal com s'ha fet per als mètodes públics no es possible.

```
@Test
public void readAuthorsFileTest() {
    try{
        java.lang.reflect.Method method = instance.getClass().getDeclaredMethod( name: "readAuthorsFile");
        method.setAccessible(true);
        com.drpicox.game.constants.Constants retorn = (com.drpicox.game.constants.Constants)method.invoke(instance);
    } catch (Exception e) {}
}
```

Fig. 6.5 Resultat Tests. Font: Elaboració pròpia

Per tal de generar testos per als mètodes privats, es crea un nou mètode anomenat "privateMethodCallTests". Aquest fa una crida utilitzant la instancia creada anteriorment, i obté el mètode que vol invocar. Un cop guardat, es canvia la accessibilitat d'aquest, i es fa una tria a traves del mètode "invoke".

```java
private static String privateMethodCallTests(Class<?> clazz, Method method) {
    String junitCode = "\t\t\t";
    junitCode += "java.lang.reflect.Method method = ";
    junitCode += "instance.getClass().getDeclaredMethod(\"" + method.getName() + "\");\n";
    junitCode += "\t\t\tmethod.setAccessible(true);\n";
    junitCode += "\t\t\t";
    if(method.getReturnType().getSimpleName() != "void") {
        junitCode += method.getReturnType().getCanonicalName() + " ";
        junitCode += "retorn = (" + method.getReturnType().getCanonicalName() + ")";
    }
    junitCode += "method.invoke(instance";

    if (method.getParameterTypes() != null) {
        int i = 0;
        for (Class<?> param : method.getParameterTypes()) {
            i++;
            if (i == 1)
                junitCode += ", ";
            junitCode += methodParameters(param);
            if (method.getParameterTypes().length != i)
                junitCode += ", ";
        }
    }
    junitCode += ");\n";

    return junitCode;
}
```

Fig. 6.5 “privateMethodCallTests”. Font: Elaboració pròpia

A l’apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s’obtenen. En el cas d’aquesta cinquena iteració, la cobertura de codi es del 97% per a les classes, 25% en mètodes i 23% en les línies recorregudes. Això demostra que a través del “invoke” no es pot generar cobertura de codi, i si es vol augmentar utilitzant mètodes privats, ha de ser a través de l’ús d’aquests en mètodes públics.

6.6 Sisena iteració

Aquesta iteració, al igual que les seves posteriors, té objectiu d’ampliar els test creats en la iteració anterior per tal d’assolir una cobertura de codi més elevada.

Per tal d’aconseguir-ho, en aquesta iteració es canvia la cas per defecte del “switch” creat en l’apartat 6.2. Es pretén que els constructors del testos també facin “new” als paràmetres que les classes necessiten. Fins a aquest punt, en comptes de fer-se la crida per la creació d'una classe, només s'hi afegia “null”.

En aquest punt els testos creats per a cada classe es veuen de la següent manera.

Es segueix amb la mateixa base que es tenia en l'apartat 6.2, però aquesta vegada no s'afegeix un "null" en cas de que el constructor requereixi d'una altre classe, sinó que es fa el "new" d'aquesta.

```
@Test
public void AuthorsServiceTestConstantsLoader() {
    try{
        instance = new AuthorsService(new com.drpicox.game.constants.ConstantsLoader(
            ));
    } catch (Exception e) {}
}
```

Fig. 6.6 Resultat Tests. Font: Elaboració pròpia

Per tal de generar els nous testos, es fan canvis en el mètode "constructorParametrType". Primerament s'afegeixen algunes variables més per defecte en els "case", com poden ser el "Map" i "List".

```
case "byte":
case "short":
case "int":
    junitCode += "0";
    break;
case "long": junitCode += "0.0L";
    break;
case "float": junitCode += "0.0f";
    break;
case "double": junitCode += "0.0d";
    break;
case "char": junitCode += "\u0000'";
    break;
case "String": junitCode += "\" \"";
    break;
case "boolean": junitCode += "false";
    break;
case "Map": junitCode += "new java.util.HashMap<>()";
    break;
case "List": junitCode += "new java.util.ArrayList()";
    break;
```

Fig. 6.6 "constructorParametrType case". Font: Elaboració pròpia

Un cop canviats els "case", es procedeix a fer els canvis pertinents al "default" del switch. Primerament es comprova si el paràmetre requerit es una interfície, en aquest cas es tornarà "null" com fins ara. En cas de no ser-ho, es descobreix quin paràmetre necessita la classe i es fa una cerca de possibles constructors que tingui aquest. En cas de trobar-ne, es tria aquell que és públic i es pot usar per a crear el paràmetre requerit. Llavors es fa una crida recursiva al mètode "constructorParametrType" per

tal de proporcionar els paràmetres al constructor que s'ha triat.

```
default:
    if (parameter.isInterface()) junitCode += "null";
    else {
        junitCode += "new " + parameter.getCanonicalName();
        if (parameter.isArray()) junitCode += "{";
        else junitCode += "(";

        Constructor<?> paramConstructor = null;
        Constructor<?>[] constructors = parameter.getDeclaredConstructors();

        for (Constructor constr : constructors) {
            int modifiers = constr.getModifiers();
            if (Modifier.isPublic(modifiers)) {
                paramConstructor = constr;
                break;
            }
        }

        if (paramConstructor != null) {
            junitCode += "\n\t\t\t";
            int r = 0;
            while (r < run) {
                junitCode += "\t";
                r++;
            }
            junitCode += constructorParametrType(paramConstructor, run: run+1);
        }

        if (parameter.isArray()) junitCode += "}";
        else junitCode += ")";
    }
```

Fig. 6.6 "constructorParametrType default". Font: Elaboració pròpia

A l'apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s'obtenen. En el cas d'aquesta sisena iteració, la cobertura de codi es del 97% per a les classes, 33% en mètodes i 31% en les línies recorregudes.

6.7 Setena iteració

Aquesta iteració, al igual que les seves posteriors, té objectiu d'ampliar els test creats en la iteració anterior per tal d'assolir una cobertura de codi més elevada.

Per tal d'aconseguir-ho, en aquesta iteració es canvia la cas per defecte del "switch" creat en l'apartat 6.3. Es pretén que els mètodes del testos també facin "new" als paràmetres que les classes necessiten. Fins a aquest punt, en comptes de fer-se la crida per la creació d'una classe, només s'hi afegia "null".

En aquest punt els testos creats per a cada classe es veuen de la següent manera.

Es segueix amb la mateixa base que es tenia en l'apartat 6.3, però aquesta vegada no s'afegeix un "null" en cas de que el constructor requereixi d'una altre classe, sinó que es fa el "new" d'aquesta.

```
@Test
public void lambda$findPost$0TestStringPost() {
    try{
        java.lang.reflect.Method method = instance.getClass().getDeclaredMethod( name: "lambda$findPost$0");
        method.setAccessible(true);
        boolean retorn = (boolean)method.invoke(instance,  ...args: " ", new com.drpicox.game.blog.Post
            ( postId: " ",new java.util.HashMap<>(), title: " ", body: " ", md5: " "));
    } catch (Exception e) {}
}
```

Fig. 6.7 Resultat Tests. Font: Elaboració pròpia

En el mètode "methodParameters" s'afegeixen nous paràmetres, com son el "Map", "List" i "File". L'últim que s'afegeix, "Settings", es degut a que el programa per al qual s'estan fent els testos utilitza varies classes que estenen de la classe "Settings", el problema ve donat quan s'utilitza "reflect" per saber la classe que usen els mètodes, ja que no se sap quina classe necessita, nomes la classe a la que estén, en aquest cas "Settings".

Per la resta de casos que no es contemplen en el "switch", primer es mira que no siguin ni interfícies ni que vinguin de "java.lang.", ja que els classes que es poden iniciar ja estan contemplades en els casos, i les que no, com "Exception", "Math", "System", etc. No es poden inicialitzar. Per tant en aquests dos casos, s'envia un "null" com a paràmetre.

Per la resta, el mètode per fer els "new" necessaris es molt similar al de l'apartat 5.6. Es busca un constructor públic per poder inicialitzar el paràmetre, i després s'envia al mètode "constructorParametrType" per a que aquest afegeixi els paràmetres necessaris al constructor.

```java
            case "Map": junitCode += "new java.util.HashMap<>()";
                break;
            case "List": junitCode += "new java.util.ArrayList()";
                break;
            case "File": junitCode += "new java.io.File(\" \")";
                break;
            case "Settings": junitCode += "null";
                break;
            default:
                if (param.isInterface()) junitCode += "null";
                else {
                    if (param.getCanonicalName().contains("java.lang."))
                        junitCode += "null";
                    else {
                        junitCode += "new " + param.getCanonicalName();

                        if (param.isArray()) junitCode += "{";
                        else junitCode += "(";

                        Constructor<?> paramConstructor = null;

                        Constructor<?>[] constructors = param.getDeclaredConstructors();
                        for (Constructor constr : constructors) {
                            int modifiers = constr.getModifiers();
                            if (Modifier.isPublic(modifiers)) {
                                paramConstructor = constr;
                                break;
                            }
                        }
                        if (paramConstructor != null)
                            junitCode += constructorParametrType(paramConstructor,  run: 1);

                        if (param.isArray()) junitCode += "}";
                        else junitCode += ")";
                    }
                }
        }
        return junitCode;
    }
```

Fig. 6.7 “methodParameters”. Font: Elaboració pròpia

Aquesta iteració porta problemes amb aquells mètodes “compareTo”, ja que es un “Override”, i no es detecta amb quin objecte vol comparar la classe, sinó que es detecta un objecte de la classe “Object”. Per tal de solucionar-ho, s’afegeix una condició al mètode “methodCallTests”, i sempre que aquest troba un mètode que es diu compareTo, en comptes d’enviar els paràmetres que demana el mètode a “methodParameters”, envia la mateixa classe que conte el mètode.

```
private static String methodCallTests(Class<?> clazz, Method method) {
    String junitCode = "\t\t\t";
    if (method.getReturnType().getSimpleName() != "void") {
        junitCode += method.getReturnType().getCanonicalName() + " retorn = ";
    }
    if (!Modifier.isStatic(method.getModifiers()))
        junitCode += "instance." + method.getName() + "(";
    else
        junitCode += clazz.getSimpleName() + "." + method.getName() + "(";
    int i = 0;
    for (Class<?> param : method.getParameterTypes()) {
        i++;
        if (method.getName().equals("compareTo"))
            junitCode += methodParameters(clazz);
        else
            junitCode += methodParameters(param);
        if (method.getParameterTypes().length != i)
            junitCode += ", ";
    }
    junitCode += ");\n";
    return junitCode;
}
```

Fig. 6.7 “methodCallTests”. Font: Elaboració pròpia

A l’apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s’obtenen. En el cas d’aquesta setena iteració, la cobertura de codi es del 97% per a les classes, 33% en mètodes i 32% en les línies recorregudes.

6.8 Vuitena iteració

Aquesta iteració, al igual que les seves posteriors, té objectiu d’ampliar els test creats en la iteració anterior per tal d’assolir una cobertura de codi més elevada.

Per tal d’aconseguir-ho, es pretén crear més d’un test per a cada mètode i constructor. Canviant el valors dels paràmetres que s’envien a cadascun d’ells.

D’aquesta manera es proven casos extrems per veure si s’entra en algun bucle que no es pot entrar amb valors per defecte.

```java
@Test
public void getByCardNameTestStringMIN() {
    try{
        com.drpicox.game.constants.Constants retorn = instance.getByCardName("");
    } catch (Exception e) {}
}

@Test
public void getByCardNameTestStringNEUTRAL() {
    try{
        com.drpicox.game.constants.Constants retorn = instance.getByCardName("Hello");
    } catch (Exception e) {}
}

@Test
public void getByCardNameTestStringMAX() {
    try{
        com.drpicox.game.constants.Constants retorn = instance.getByCardName(String.valueOf(Character.MAX_VALUE));
    } catch (Exception e) {}
}
```

Fig. 6.8 Resultat Tests. Font: Elaboració pròpia

Es canvien els casos dins dels "switch" dels mètodes "constructorParametrType" i "methodParameters", i també s'envia un nou paràmetre que fa la funció de controlar si es necessita enviar un valor mínim, màxim o neutre.

```java
3 usages
private static String methodParameters(Class param, int value){
```

Fig. 6.8 "constructorParametrType". Font: Elaboració pròpia

```
case "byte":
    junitCode += (value == 0) ? "Byte.MIN_VALUE" :
            (value == 1) ? "0" :  (value == 2) ? "Byte.MAX_VALUE" : "";
    break;
case "short":
    junitCode += (value == 0) ? "Short.MIN_VALUE" :
            (value == 1) ? "0" : (value == 2) ? "Short.MAX_VALUE" : "";
    break;
case "int":
    junitCode += (value == 0) ? "Integer.MIN_VALUE" :
            (value == 1) ? "0" : (value == 2) ? "Integer.MAX_VALUE" : "";
    break;
case "long":
    junitCode += (value == 0) ? "Long.MIN_VALUE" :
            (value == 1) ? "0L" :  (value == 2) ? "Long.MAX_VALUE" : "";
    break;
case "float":
    junitCode += (value == 0) ? "Float.MIN_VALUE" :
            (value == 1) ? "0.0f" : (value == 2) ? "Float.MAX_VALUE" : "";
    break;
case "double":
    junitCode += (value == 0) ? "Double.MIN_VALUE" :
            (value == 1) ? "0.0d" : (value == 2) ? "Double.MAX_VALUE" : "";
    break;
case "char":
    junitCode += (value == 0) ? "Character.MIN_VALUE" :
            (value == 1) ? "'\u0000'" : (value == 2) ? "Character.MAX_VALUE" : "";
    break;
case "String":
    junitCode += (value == 0) ? "\"\"" :
            (value == 1) ? "\"Hello\"" : (value == 2) ? "String.valueOf(Character.MAX_VALUE)" : "";
    break;
case "boolean":
    junitCode += (value == 0) ? "false" :
            (value == 1) ? "false" : (value == 2) ? "true" : "";
    break;
```

Fig. 6.8 “switch” per a mètodes i constructors. Font: Elaboració pròpia

A l’apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s’obtenen. En el cas d’aquesta vuitena iteració, la cobertura de codi es del 97% per a les classes, 33% en mètodes i 32% en les línies recorregudes.

6.9 Novena iteració

Arribats a aquest punt, veient que la cobertura de codi no augmenta, es plantegen dues hipòtesis, la primera, que l’experiment no pot arribar més enllà d’una cobertura rondant al 30%, i la segona, que hi ha algun error que s’ha passat per alt durant les iteracions.

Una de les proves que es dur a terme per trobar algun error dins dels testos, abans de donar l'experiment per finalitzat, es treure els "try" i "catch" que s'utilitzen per a prevenir els errors dels propis mètodes de les classes.

```
@Test
public void getAuthorsTestMIN() {
        java.lang.reflect.Method method = instance.getClass().getDeclaredMethod( name: "getAuthors");
        method.setAccessible(true);
        com.drpicox.game.constants.Constants retorn = (com.drpicox.game.constants.Constants)method.invoke(instance);
}
```

Fig. 6.9 Test. Font: Elaboració pròpia

Aquest cop, es el propi test que avisa de que "instance" encara no ha sigut inicialitzada, i salta l'error de "NullPointerException". Això porta a fer una reviso dels testos que es duen a terme per als constructor de la classe.

I finalment es troba l'error de que cap d'ells es "@Before". Això implica que "instance" només sigui valida per als tests dels constructors, i que per als test dels mètodes sempre salti l'error de "NullPointerException". Al haver els "try" i "catch" per prevenir les excepcions dels propis mètodes, aquest problema no s'ha detectat abans.

Això porta a fer un petit canvi al mètode "constructorsTests", Es comprova que si el test que s'està creant els el primer de tots, i en cas de ser-ho, en comptes d' "@Test", s'afegeix "@Before".

```
if (i == 1 && value == 0)
    junitCode += "\t@Before\n";
else
    junitCode += "\t@Test\n";
```

Fig. 6.9 "constructorsTests". Font: Elaboració pròpia

El principi dels test en aquest apartat es veuen així.

```java
    AuthorsService instance;

    @Before
    public void AuthorsServiceTestConstantsLoaderMIN() {
        try{
            instance = new AuthorsService(new com.drpicox.game.constants.ConstantsLoader(
                ));
        } catch (Exception e) {}
    }

    @Test
    public void AuthorsServiceTestConstantsLoaderNEUTRAL() {
        try{
            instance = new AuthorsService(new com.drpicox.game.constants.ConstantsLoader(
                ));
        } catch (Exception e) {}
    }
```

Fig. 6.9 Resultat Tests. Font: Elaboració pròpia

Això implica que tots els test tenen "instance" inicialitzada abans de treballar, el que es tradueix en que ara els test si que actuen com s'espera, i es veu en l'augment de cobertura de codi que es genera a partir del canvi.

A l'apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s'obtenen. En el cas d'aquesta novena iteració, la cobertura de codi es del 97% per a les classes, 93% en mètodes i 69% en les línies recorregudes.

6.10 Desena iteració

Aquesta iteració, al igual que les seves posteriors, té objectiu d'ampliar els test creats en la iteració anterior per tal d'assolir una cobertura de codi més elevada.

Per tal d'aconseguir-ho, es busca fer la injecció automàtica de dependències, i obtenir una instancia del component que correspongui a la classe. A més, s'usen també les següents anotacions, @ActiveProfiles("test"), que es una anotació per activar perfils específics durant les proves, permetent configuracions adaptades sense afectar altres entorns; @SpringBootTest, aquesta anotació es per carregar el context de "Spring" per a proves d'integració, inicialitzant tots els components necessaris per a l'execució de les proves; i @AutoConfigureMockMvc, que serveix per configurar l'entorn de proves per a proves d'integració en controladors de "Spring MVC", permetent

simular sol·licituds HTTP i verificar respostes sense desplegar en un servidor real.

A part, es canvien els imports per als test i per el "@Before", ja que aquests no serveixen per "spring". Ara s'usen des de "org.junit.jupiter.api".

```
package com.drpicox.game;
import com.drpicox.game.blog.AuthorsService;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ActiveProfiles;

@ActiveProfiles("test")
@SpringBootTest
@AutoConfigureMockMvc
public class AuthorsServiceTest {

    AuthorsService instance;

    @Autowired
    ApplicationContext applicationContext;

    @BeforeEach
    public void AuthorsServiceTestConstantsLoaderMIN() {
        try{
            instance = applicationContext.getBean(AuthorsService.class);
            return;
        } catch (Exception e) {}
        try{
            instance = new AuthorsService(new com.drpicox.game.constants.ConstantsLoader(
                ));
        } catch (Exception e) {}
    }
```

Fig. 6.10 Resultat Tests. Font: Elaboració pròpia

Es fan canvis al mètode "generateJUnitForClass" per tal de que tots els testos incorporin els nous import i substituir els antics.

```
1 usage
private static String generateJUnitForClass(Class<?> clazz) {
    String className = clazz.getSimpleName();
    String junitCode = "package " + outputDirShort + ";\n";
    junitCode += "import " + clazz.getPackageName() + "." + className + ";\n";
    junitCode += "import org.junit.jupiter.api.Test;\n" +
        "import org.junit.jupiter.api.BeforeEach;\n" +
        "import org.springframework.beans.factory.annotation.Autowired;\n" +
        "import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;\n" +
        "import org.springframework.boot.test.context.SpringBootTest;\n" +
        "import org.springframework.context.ApplicationContext;\n" +
        "import org.springframework.test.context.ActiveProfiles;\n\n";

    junitCode += "@ActiveProfiles(\"test\")\n" +
        "@SpringBootTest\n" +
        "@AutoConfigureMockMvc\n";
```

Fig. 6.10 "generateJUnitForClass" part 1. Font: Elaboració pròpia

Als testos també s'afegeix un "@Autowired" "ApplicationContext", ja que utilitzant-lo, s'injecta automàticament una instància de l'objecte "ApplicationContext" en la classe o component, i aquest proporciona accés als "beans" i recursos de l'aplicació.

```
junitCode += "\t@Autowired\n" +
        "\tApplicationContext applicationContext;\n\n";
```

Fig. 6.10 "generateJUnitForClass" part 2. Font: Elaboració pròpia

En el mètode "constructorsTests" es canvia la primera creació de constructors, es a dir, aquell que porta "@BeforeEach". Per tal de que aquesta faci primer un intent d'obtenir una instància del bean "AuthorsService" de l' "ApplicationContext". Si l'obtenció és exitosa, s'assigna a la variable "instance", en cas contrari s'inicialitza "instance" com fins ara es feia.

```
if (i == 1) {
    if (value == 0) {
        junitCode += "\t\t\tinstance = applicationContext.getBean(" + className + ".class);\n" +
                "\t\t\treturn;\n" +
                "\t\t} catch (Exception e) {}\n";

        junitCode += "\t\ttry{\n";
    }
    junitCode += "\t\t\tinstance = new " + className + "(";
}
```

Fig. 6.10 "constructorsTests". Font: Elaboració pròpia

A l'apartat 6.11 es pot veure un recull de les cobertures de codi que en cada iteració s'obtenen. En el cas d'aquesta desena iteració, la cobertura de codi es del 100% per a les classes, 95% en mètodes i 85% en les línies recorregudes.

6.11 Resultats iteracions

En aquest apartat podem veure un recull de l'evolució de les diferents iteracions, i com ha crescut la cobertura de codi en cada pas.

En la primera part es veu un recull de les dades que cada iteració a aportat al projecte. Com es pot veure, degut al error de no posar el "@Before" per al primer tests, les iteracions més rellevants han estat aquelles que han treballat amb els constructors.

Taula 6.11 Resultats iteracions

Iteració	Classes	Mètodes	Línies
1	-	-	-
2	91	19	20
3	91	19	20
4	91	25	23
5	91	25	23
6	97	33	31
7	97	33	32
8	97	33	32
9	97	93	69
10	100	95	85

Font: Elaboració pròpia

Tant a la taula com a la gràfica, es pot veure que el més fàcil ha estat augmentar la cobertura de les classes. Ja que amb només cobrir una línia, fos la que fos, ja assegura cobertura per a la classe. La cobertura més complicada d'augmentar ha estat la de els línies de codi, ja que al no saber res del programa que s'analitza és difícil passar per tots els bucles.

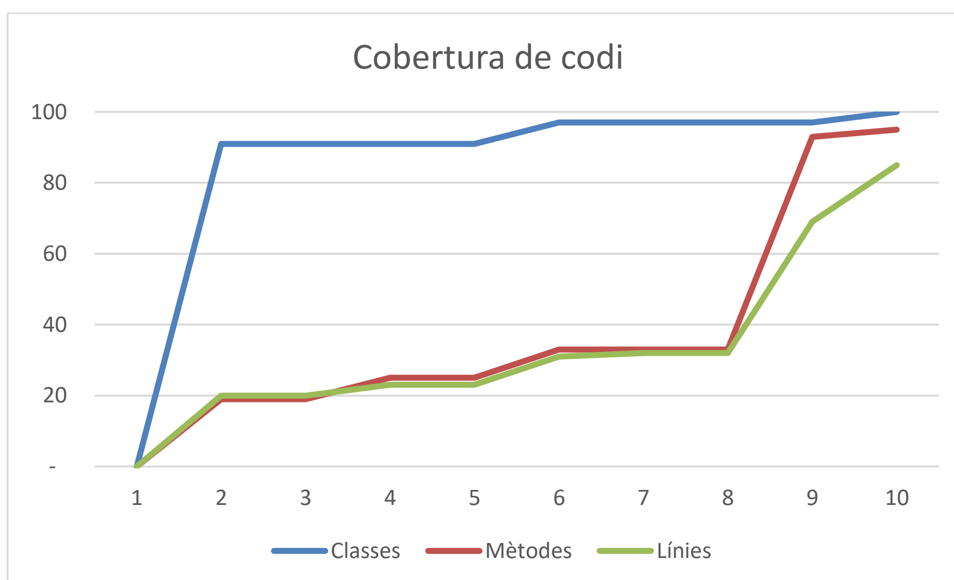

Fig. 6.11 Resultats iteracions. Font: Elaboració pròpia

En aquesta segona part es pot veure un recull de dades en el que el primer test de constructor sempre ha estat amb l'anotació "@Before". En aquest cas, la iteració més rellevant ha estat aquella que ha treballat amb els mètodes, a diferencia del cas anterior.

Taula 6.11 Resultats iteracions solucionades

Iteració	Classes	Mètodes	Línies
1	-	-	-
2	91	19	20
3	88	27	23
4	94	77	50
5	96	77	50
6	96	91	65
7	96	92	69
8	97	93	69
9	97	93	69
10	100	95	85

Font: Elaboració pròpia

Tant a la taula com a la gràfica, es pot veure que el més fàcil ha estat augmentar la cobertura de les classes. Però en aquest segon cas, també es veu que no es tan complicat augmentar la cobertura de mètodes, tot i que en la gràfica anterior ho semblava.

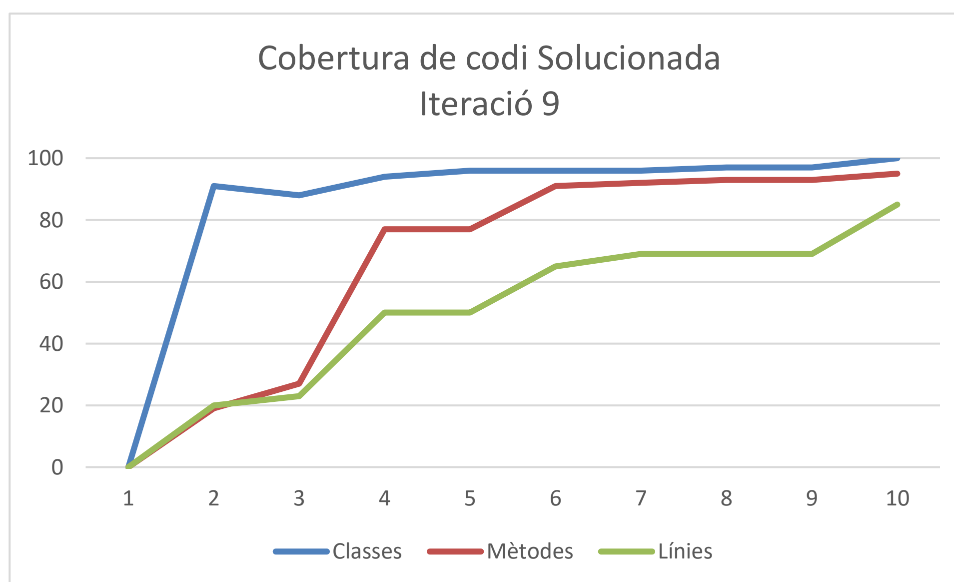

Fig. 6.11 Resultats iteracions solucionades. Font: Elaboració pròpia

7. Conclusions

L'objectiu principal del projecte era demostrar que una alta cobertura de codi no implica necessàriament una millora de la qualitat. Aconseguir un 85% de cobertura amb el creador de testos automàtic representa un èxit rellevant, ja que s'ha comprovat que es pot obtenir una alta cobertura sense necessitat d'escriure tots els casos de prova manualment.

No obstant això, és essencial tenir en compte que la cobertura de codi és només una part de l'avaluació de la qualitat del software. Malgrat arribar a un alt nivell de cobertura, no es garanteix automàticament que el codi sigui exempt d'errors o que funcioni de manera eficient. Altres aspectes com la correcció, l'optimització, l'ús adequat dels recursos o la gestió d'errors són igualment importants per assegurar una qualitat global del software.

Per tant, és crucial complementar la cobertura de codi amb altres tècniques i pràctiques de prova i revisió. Això pot incloure la cobertura de requisits, de casos d'us, de rutes i proves basades en riscos. Aquestes mesures complementàries ajuden a identificar i solucionar problemes que poden passar desapercebuts només amb la cobertura de codi.

Cobertura de requisits: En lloc de centrar-se exclusivament en el codi font, aquesta alternativa es dedica a avaluar fins a quin punt s'han acomplert els requisits del programari durant les proves. Aquests requisits poden ser tant funcionals (les accions específiques que el programari ha de realitzar) com no funcionals (els atributs de qualitat, com ara rendiment o seguretat). La cobertura de requisits és una forma d'assegurar que s'han realitzat proves adequades per a totes les funcionalitats i els criteris de qualitat establerts.

Cobertura de casos d'ús: Aquesta tècnica se centra en avaluar quins casos d'ús del sistema s'han tingut en compte durant les proves. Cada cas d'ús descriu una interacció específica entre un actor i el sistema per aconseguir un objectiu determinat. La cobertura de casos d'ús és essencial per garantir que s'han realitzat proves exhaustives i que s'han tingut en compte totes les interaccions importants.

Cobertura de rutes: En lloc de centrar-se en la quantitat de codi executat, aquesta alternativa se centra en les diferents rutes que pot prendre un programa durant l'execució. Cada ruta representa una seqüència única de decisions i bifurcacions en el codi. La cobertura de rutes és útil per identificar les àrees crítiques del programari que requereixen una prova exhaustiva.

Proves basades en riscos: Aquesta alternativa es centra a identificar i provar els components del programari que presenten un major risc. Aquests riscos es valoren tenint en compte factors com la complexitat del codi, la importància del component per al funcionament global del sistema o la freqüència i l'impacte dels errors anteriors. Les proves basades en riscos permeten prioritzar les àrees de prova crítiques per garantir una millor qualitat del programari.

Aquestes alternatives poden complementar o substituir la cobertura de codi tradicional, segons les necessitats i objectius del projecte de desenvolupament de programari. És important tenir en compte el context específic i les limitacions del projecte en seleccionar les tècniques de cobertura més adequades.

8. Referències.

[1] Microsoft (2022) *Generate unit tests for fuzz testing by using IntelliTest*, learn.microsoft.com

[2] Rodenas D. (2022, setembre 17*) What is business rules coverage?* Medium. https://medium.com/codex/what-is-business-rules-coverage-a7ec9fe5ebbd

[3] Mesa J. (2020) *Ser Test Automation es más que solo automatizar pruebas*, youtube.com

[4] Microsoft (2022) *Empleo de cobertura de codigo para pruebas unitarias*, learn.microsoft.com

[5] Hollub, A. [@allenholub@mstdn.social]. (2022, agost 20). *I've thought about writing an automated coverage generator that just created tests that called every function/method in a program with random arguments and always passed. Poof! 80% coverage! Coverage is not a useful metric.* [Twitter] https://twitter.com/allenholub/status/1560774031078465539?s=20&t=CNm1aBfLJj3j VCUcNM7_7w

[6] Korakitis K. (2021, novembre 12) *Size of Programming Language Communities in Q3 2021*. Developer Nation. https://www.developernation.net/blog/size-of-programming-language- communities-in-q3-2021

[7] Nativapps (2022, setembre 9) *Los lenguajes de programación mejor pagados en 2022*. Nativapps. https://nativapps.com/es/los-lenguajes-de-programacion-mejor-pagados-en-2022/