Centres universitaris adscrits a la

TecnoCampus

upf. Universitat Pompeu Fabra Barcelona

Grau en enginyeria informàtica de gestió i sistemes d'informació

Exploració de la metodologia BDD amb Cucumber

Memòria final

JORDI PAGES

Tutor: JOSEP ROURE

2023-2024

TecnoCampus

Abstract

The objective of this project is to thoroughly investigate and explore the BDD methodology using the Cucumber library and to develop a demonstration project using this methodology. In this project, the advantages and disadvantages of this development approach are evaluated, as well as the best practices to follow to ensure an efficient and maintainable implementation of the software.

Resum

L'objectiu d'aquest projecte és investigar i explorar detalladament la metodologia BDD mitjançant la llibreria Cucumber i elaborar un projecte de demostració usant la metodologia. En aquest projecte s'avaluen els avantatges i inconvenients que té aquesta aproximació de desenvolupament així com les millors pràctiques a seguir per garantir una implementació eficient i mantenible del software.

Resumen

El objetivo de este proyecto es investigar y explorar detalladamente la metodología BDD mediante la librería Cucumber y elaborar un proyecto de demostración usando la metodología. En este proyecto se evalúan las ventajas e inconvenientes que tiene esta aproximación de desarrollo, así como las mejores prácticas a seguir para garantizar una implementación eficiente y mantenible del software.

Index

Índex de figures

1. Objecte del projecte

Aquest projecte es basa en la importància de millorar l'eficiència i qualitat del procés de desenvolupament de software. El desenvolupament de software és un procés que requereix de planificació i de seguir metodologies eficients per minimitzar el risc de fracàs del projecte a llarg termini i contribuir a una millor comunicació i treball en equip.

Llavors, escollir una bona estratègia de desenvolupament i implementar-la correctament i de manera eficaç és molt important per aconseguir una bona qualitat final. L'estudi d'una estratègia de desenvolupament juntament amb una demostració avançada on s'aplica dita estratègia pot ser de gran utilitat per a la industria, ja que permet avaluar l'eficàcia de la metodologia veient la seva aplicació directe amb un exemple real. Concretament, en aquest projecte es vol profunditzar en el coneixement de la metodologia BDD (Behavior-Driven Development) i es desenvolupa una aplicació de demostració usant aquesta metodologia.

És un projecte que es basa, en part, en investigació en el camp de l'enginyeria del software, disciplina que busca oferir mètodes i tècniques per a desenvolupar i mantenir software de qualitat. La investigació en l'enginyeria del software és crucial per millorar els processos de desenvolupament de software i millorar l'educació en aquest camp. Amb aquest treball es contribueix també a l'elecció de la estratègia de desenvolupament més adequada per a un projecte.

Cal destacar que aquesta metodologia és àmpliament utilitzada en la indústria i ha guanyat molta popularitat en els últims anys. És per això que una de les motivacions del projecte és preparar a estudiants i gent que es vol formar en la creació d'aplicacions informàtiques per al seu futur professional, ja que aquests coneixements són importants en la indústria.

2. Estudi previ: context, antecedents i necessitats d'informació

Les estratègies de desenvolupament de software comencen a sorgir a la dècada del 1960 [0] amb l'aparició del cicle de vida clàssic del software: el desenvolupament en cascada. Aquest enfoc es basa en la planificació, anàlisi, disseny, codificació, prova i manteniment de software d'una manera seqüencial. Aquest procés té una sèrie de problemes que poden afectar a l'eficàcia i eficiència del procés, encara que hagi tingut èxit en el passat i pugui arribar a ser-ne útil en certs tipus de projectes.

Un dels problemes més importants d'aquest enfoc, és la falta de feedback i flexibilitat [1]. Degut a ser un procés seqüencial no es beneficia de cap retroalimentació contínua i tampoc permet ajustar el software en base als canvis en els requisits de l'usuari. Això genera solucions rígides i poc escalables al llarg del temps.

A més, la ineficiència en el model cascada és un problema causat també per la falta d'interacció i coordinació entre departaments. Cada departament treballa pel seu compte sense una adequada comunicació, la qual cosa pot portar a una duplicació d'esforços i retards en la consecució de les metes del projecte.

Als anys 90, van sorgir els primers llenguatges de modelatge gràfic, com UML, que van permetre als desenvolupadors comunicar-se de manera més clara i efectiva sobre el disseny i l'arquitectura d'un sistema. L'UML va ser una de les primeres metodologies a proporcionar un llenguatge comú que es podria utilitzar en tot el procés de desenvolupament de software, i que encara s'utilitza avui dia. L'UML no és una metodologia de desenvolupament en sí [2], sinó que va ser creat per ser compatible amb les metodologies que existien en aquell moment, com per exemple OMT, el mètode Booch o Objectory, fent especial èmfasi al procés de desenvolupament anomenat RUP.

RUP, a diferència del cicle de vida clàssic del software en cascada, es basa en un enfoc iteratiu i incremental. Va ser desenvolupat per la companyia IBM Rational [3] i s'ha convertit en un dels processos de desenvolupament de software més utilitzats a tot el món. RUP divideix el desenvolupament de software en quatre fases i permet una avaluació

continuada a mesura que es desenvolupa. Les fases són: Iniciació, Elaboració, Construcció i Transició. A cada fase es realitzen diferents activitats com la definició de requisits, el disseny, la implementació i el testing. Amb tot i això, RUP també es va conèixer per ser un procés complex i burocràtic, i requeria una gran quantitat de planificació i documentació abans de començar a codificar.

Als mateixos anys 90, també van començar a sorgir altres estratègies àgils que oferien flexibilitat i adaptabilitat als canvis en el projecte. Entre aquestes estratègies es troba Scrum, que es centra en la col·laboració i el lliurament incremental, i XP (Extreme Programming), que s'enfoca en la millora contínua i el lliurament ràpid de software.

L'Extreme Programming (XP) [4] és una metodologia de desenvolupament de software creada per Kent Beck. Es considera una de les primeres metodologies àgils i va començar a ser popular als anys 90 i principis dels 2000. Combina tant pràctiques tècniques com de gestió i s'enfoca en valors amplis, principis concrets i pràctiques específiques com: integració contínua, refactorització, TDD (Test Driven Development) i planificació àgil.

Una de les tècniques més importants i influents de l'Extreme Programming és el TDD (Test Driven Development). És una tècnica que està molt relacionada amb la metodologia BDD, ja que BDD sorgeix del TDD. És una tècnica molt estricte que es basa en seguir tres passes de manera cíclica [5]:

1. Escriure un test per la següent funcionalitat que es vulgui afegir.
2. Escriure codi funcional fins que el test passi.
3. Refactorizar tant el nou codi com el codi antic per a que tingui una estructura adequada.

Originalment, Kent Beck va dissenyar TDD com una pràctica de disseny de software, com una manera de dissenyar el software basat en escriure els test abans d'escriure codi funcional i fer refactor una vegada passin els test.

Va ser a principis dels 2000 quan es va començar a introduir la metodologia BDD (Behavior-Driven Development). El seu inventor, Daniel Terhorst-North, la va concebre

com una metodologia que es basa en pràctiques àgils i lean que inclouen sobretot TDD i DDD (Domain Driven Design) [6]. El DDD és un enfoc en el desenvolupament de software que va introduir Eric Evans i es centra en modelar el domini de l'aplicació utilitzant un llenguatge comú i compartit per experts del negoci i desenvolupadors.

La metodologia BDD s'enfoca en la idea de descriure el comportament del sistema desitjat en termes d'històries d'usuari i escriure els test abans d'implementar codi. De fet, North la va dissenyar originalment per fer més fàcil l'ensenyament de TDD. La diferència amb TDD és que BDD es basa en l'automatització de tests com una forma de verificar i validar el comportament del software: primer s'escriuen escenaris de prova en llenguatge natural que descriuen com hauria de funcionar el software i després aquests escenaris es converteixen en proves automatitzades que s'executen en cicles de forma regular per assegurar-se que el software segueixi funcionant com s'espera. Això ho aconsegueix fent ús d'un llenguatge comú a l'hora d'escriure els escenaris basat en oracions simples i estructurades en anglès, facilitant així la comunicació entre membres de l'equip i stakeholders. Aquest llenguatge es coneix generalment amb el nom de Gherkin.

En el blog de cucumber [7] s'explica que TDD és la tècnica de desenvolupament de software que s'ha convertit en estàndard en el desenvolupament Agile en els darrers anys, ja que preveu errors i garanteix un desplegament continu sense problemes. Però BDD suposa una evolució de TDD que permet una comunicació millor entre equips de negocis i tècnics, reduint la confusió sobre els criteris d'acceptació i garantint que l'aplicació compleixi les expectatives dels usuaris finals.

Ja que BDD té el seu origen en TDD, és convenient posar en relació ambdós enfocs per poder apreciar quines són les diferències entre ells o com aborden certs aspectes tant un com l'altre.

- BDD s'enfoca a provar el comportament des de la perspectiva de l'usuari final, mentre que el TDD s'enfoca a proves aïllades de petites peces de funcionalitat.

- BDD implica col·laboració entre gerents de productes, desenvolupadors i enginyers de proves per trobar exemples concrets de les funcionalitats desde una perspectiva d'usuari final que usa el producte, mentre que el TDD pot ser realitzat per un sol

desenvolupador, ja que no requereix aquesta perspectiva externa, sinó que es centra més en els components interns del sistema

- BDD i TDD no són mútuament exclusius, molts equips àgils utilitzen TDD sense BDD, però BDD assegura que la majoria dels casos d'ús de l'aplicació funcionin a un nivell més alt i ofereixin un major nivell de confiança.

- Hi ha moltes eines de TDD disponibles ja que ha sigut adoptat per moltes empreses que usen desenvolupaments àgils, però hi ha menys eines de BDD degut a la necessitat de comunicació entre equips de negocis i tècnics.

- BDD no requereix canviar pràctiques de TDD existents, requereix una inversió en comunicació addicional donant lloc a un resultat menys ambigu i amb major confiança.

En resum, BDD és una metodologia que fomenta la col·laboració entre els diferents departaments en expressar els requisits d'una manera més comprovable de forma que, tant l'equip de desenvolupament com els stakeholders puguin entendre fàcilment. D'aquesta manera, es poden identificar i construir característiques de valor per al negoci i usant BDD ens assegurem que aquestes característiques estiguin ben dissenyades i implementades.

Tanmateix, BDD funciona bé amb altres metodologies àgils com Scrum, però allò important a tenir en compte és que no reemplaça a qualsevol metodologia d'aquest estil, sinó que es construeix, incorpora i millora idees de moltes d'aquestes metodologies.

BDD es va començar a fer popular gràcies a la col·laboració de Dan North i Chris Matts en el desenvolupament d'una biblioteca d'automatització de test BDD que es deia JBehave. Més endavant, al 2005, va ser quan va aparèixer el projecte "RSpec" que donava suport BDD en el llenguatge de programació Ruby. "RSpec" estava dissenyat per programadors però també existia "RSpec Story Runner" que estava dissenyat per a tot l'equip incloent stakeholders no-tècnics. Aslak Hellesøy és un desenvolupador de software noruec que va contribuir a millorar RSpec i com que tenia moltes idees de millores va començar un projecte nou. Aquest projecte ha anat evolucionant i és el que es coneix actualment amb el nom de Cucumber [8], que no és més que un framework de desenvolupament que admet BDD i que fa servir la sintaxi Gherkin.

3. Objectius i abast del projecte

En el projecte es realitza una exploració detallada de la metodologia BDD utilitzant l'eina Cucumber, juntament amb una demostració pràctica de la seva aplicació. Els objectius del projecte inclouen:

1. Comprendre els conceptes i els principis de la metodologia BDD.
2. Analitzar els avantatges i els desavantatges de la metodologia BDD.
3. Avaluar l'aplicabilitat de la metodologia BDD amb l'eina Cucumber en projectes reals.
4. Realitzar una demostració avançada que mostri com s'aplica la metodologia BDD amb Cucumber en un entorn real.
5. Avaluar el procés de la creació de la demostració per proporcionar recomanacions de bones pràctiques a l'hora d'aplicar la metodologia.
6. Contribuir al coneixement i comprensió de la metodologia BDD i la seva aplicació.
7. Avaluar l'escalabilitat de la metodologia i la capacitat per adaptar-se a diferents tipus de projectes i equips de treball.

En relació a l'abast del treball, el projecte comprèn:

- **La investigació de la metodologia de desenvolupament BDD amb Cucumber.** Investigar i documentar la seva definició, característiques, avantatges i desavantatges, la seva aplicació amb Cucumber i valorar aspectes de la metodologia com la seva eficiència, l'escalabilitat i la flexibilitat.
- **Un projecte de demostració avançat.** Desenvolupar una aplicació utilitzant la metodologia estudiada. Concretament, es tracta d'un servei web backend (API REST) sense interfície gràfica.
- **Avaluació i documentació de resultats.** Avaluar i documentar el procés de desenvolupament de la demostració valorant-ne així la seva eficàcia.

L'objectiu principal del client del projecte, que en aquest cas és el propi tutor del TFG, és profunditzar en el coneixement i la comprensió de la metodologia BDD per poder avaluar la viabilitat de la seva implementació en projectes reals de software. El client vol conèixer les bases teòriques i pràctiques de la metodologia, la seva aplicació i els resultats obtinguts tant en la demostració que es desenvoluparà com en projectes similars on s'usi la metodologia.

Degut a que és un treball d'exploració i investigació d'una metodologia i la seva aplicació en una demostració, com a resultat del projecte no hi ha cap producte final que es llença al mercat i, per tant, no hi ha cap usuari potencial que consumeix el producte. De fet, els usuaris que hi hagin a l'aplicació no són usuaris reals sinó usuaris de prova, ja que la idea és simular com es comportaria l'aplicació amb usuaris reals i determinar si compleix amb els requisits i expectatives.

4. Metodologia

A continuació, s'aniran definint les metodologies que s'aplicaran en els diferents aspectes tant de la part d'investigació com de la part d'implementació del projecte.

Per a la planificació de les tasques del projecte:

- Ús de l'eina de gestió de projectes "Notion" per prioritzar tasques i poder-les organitzar en un calendari. L'eina permet també organitzar les tasques en diagrames de Gantt.

Com a estratègies de cerca d'informació:

- Consulta de fonts com: llibres, cursos online, articles, blogs, biblioteques en línia com Google Scholar, vídeos de YouTube, etc.
- Participació en fòrums i llocs web de preguntes i respostes per a programadors professionals i aficionats.

Per a la recopilació d'informació:

- Marcatge de llocs web d'interès.
- Ús de carpeta/s en Google Drive per afegir bibliografia i documents.

Per l'anàlisi de la informació:

- Comparació de diferents fonts per contrastar la informació.
- Anàlisi casos de projectes reals que hagin utilitzat BDD amb Cucumber.

Per la síntesi de la informació:

- Resumir la informació i documentar conclusions mitjançant aplicacions de notes o en documents word.
- Creació de diagrames i esquemes per a una millor comprensió.

Per la creació de la demostració avançada:

- Creació d'un exemple pràctic i funcional utilitzant la metodologia BDD amb Cucumber.
- Anar documentant el procés de desenvolupament.

5. Definició de requeriments funcionals i tecnològics

Els requeriments funcionals són aquells que defineixen el que el projecte ha de fer i com ho ha de fer. Són els següents:

- Realitzar una investigació exhaustiva d'informació existent sobre BDD i Cucumber.
- Definir la metodologia BDD i com s'aplica a través de Cucumber.
- Analitzar casos d'ús i projectes en els que s'hagi usat BDD amb Cucumber.
- Realitzar una demostració avançada de com s'aplica BDD amb Cucumber a un projecte de software.
- Presentar conclusions i recomanacions basades en la demostració desenvolupada.

El requeriments tecnològics són aquelles tecnologies, eines i sistemes necessaris per a dur a terme el projecte. Són els següents:

- Disposar d'ordinador amb sortida a internet.
- Tenir coneixements de programació web i base de dades.
- Tenir habilitats per implementar la metodologia de BDD amb Cucumber.
- Disposar d'eines de redacció de documents (Word).
- Disposar d'eines de gestió: eina de gestió de projectes (Notion).
- Disposar d'un entorn de desenvolupament local adequat per a la realització de la demostració avançada que inclou: editors de codi (IntellIJ), sistema de control de versions per el desenvolupament de la demostració (Github) i una base de dades open source (MySql/MariaDB).

6. Exploració de la metodologia BDD

En aquesta part d'investigació sobre BDD s'utilitzaran termes en anglès, ja que són àmpliament usats en la indústria del desenvolupament de software i es consideren estàndards en molts casos. També és important aclarir que es prendran com a referència les clean architectures com l'hexagonal a l'hora de parlar sobre l'arquitectura del software i les seves capes. L'arquitectura hexagonal és una arquitectura de software que consta de quatre capes: la capa de framework drivers, la capa d'infrastructura, la capa d'aplicació i la capa de domini.

La metodologia BDD està estretament relacionada amb diferents termes, com els tests d'acceptació, TDD, el continuous delivery, etc. Al llarg d'aquest apartat es defineix quina relació té BDD amb tots aquests termes i la seva importància.

Abans de res, és adient definir quins tipus principals de test existeixen i amb quins d'ells la metodologia BDD està més relacionat o s'idenifica més:

- **Unit tests.** Són proves que es centren en una sola unitat de codi, com ara una funció o mètode.
- **Integration tests.** Són proves que es centren en com diverses unitats de codi funcionen juntes com un sistema.
- **Acceptance tests.** Són proves que es centren en verificar que les diferents parts del sistema compleixin els requisits del negoci i les expectatives dels usuaris.
- **End-to-end tests (E2E).** És un tipus de prova que es centra en verificar que tot el sistema funcioni correctament des del punt de vista de l'usuari final, simulant una situació real passant per totes les capes de l'aplicació.

La metodologia BDD es relaciona més amb els tests d'acceptació, ja que validen el comportament del sistema des del punt de vista de l'usuari final i s'enfoquen en aspectes amplis com la funcionalitat i el comportament general. BDD també es pot relacionar amb els tests end-to-end, més tècnics i detallats, que executen tot el sistema des de la interfície de l'usuari fins a la base de dades i altres components.

Tot i que sovint es confonen, els tests d'acceptació no han de ser necessàriament end-to-end. Aquests tests es centren en verificar que el software compleixi els requisits i ofereixi el valor comercial que s'espera, sense necessitat d'avaluar tota l'aplicació de d'extrem a extrem. Poden centrar-se en casos d'ús específics o funcionalitats individuals, interactuant amb la interfície d'usuari, les API o els serveis interns.

D'aquesta manera, mentre els tests end-to-end avaluen el funcionament complet de l'aplicació, els tests d'acceptació poden ser més específics i centrats en escenaris particulars, fet que els fa més àgils i fàcils de mantenir en un marc de treball on s'utilitza BDD.

6.1. Acceptance testing automatitzat

Els tests d'acceptació són proves centrades en l'usuari, avaluades des de la perspectiva de l'usuari i orientades a les funcionalitats del negoci.

Dave Farley és un enginyer de software que ha parlat i creat molt contingut sobre els tests d'acceptació i manté una filosofia similar a la de Robert C. Martin (l'Uncle Bob), . Dave Farley proposa cinc propietats dels tests d'acceptació:

1. Han d'assegurar-se que el codi funciona com els usuaris volen, buscant aconseguir una especificació del comportament que capturi la intenció de l'usuari.
2. Representen una definició automatitzada de "fet" o "done", cosa que és una excel·lent manera de guiar el desenvolupament utilitzant TDD: partir d'una especificació i treballar amb TDD fins que es compleixi l'especificació.
3. Han d'assegurar que el codi funciona en un entorn similar al de producció, provant la implementació i la configuració del sistema.
4. Han de proporcionar retroalimentació oportuna, permetent que es detectin problemes en un moment adequat.
5. Els tests han de ser repetibles i fiables.

És important destacar que els següents termes: "tests d'acceptació", "BDD", "ATDD", "Specification by exemple" i "especificacions executables" es consideren sinònims. Tots

intenten capturar, des de la perspectiva de l'usuari, la intenció del sistema. BDD es pot aplicar a diferents nivells de granularitat i, encara que s'associa sovint amb els tests d'acceptació, en realitat es podria arribar a utilitzar per a qualsevol altre tipus de tests, com els tests unitaris, encara que no és gaire comú ni oportú. Els tests unitaris se centren en la verificació del comportament de petites parts del codi de manera ràpida i eficient. Usar la metodologia BDD en aquests tests pot afegir overhead i complexitat innecessaris, fent-los menys efectius per a la seva finalitat específica.

Un test d'acceptació és una especificació executable del comportament del sistema. S'ubica dins d'una pipeline de continuous delivery que mostra múltiples bucles de retroalimentació, que comencen amb la idea i acaben amb el desplegament. La majoria de les organitzacions fan proves manuals d'acceptació, però això no és repetible, és costós, fràgil, lent i ineficient. El pas següent a això és automatitzar aquests tests, però no de qualsevol manera, ja que sinó acabem amb un testing que és lent i de baixa qualitat, a més de ser costós, poc fiable, propens a errors i fràgil.

El problema principal es deu a l'acoblament entre els tests i el sistema sota prova. El que es vol es crear un test que estigui desacoblat perquè el sistema sota prova pugui canviar lliurement sense requerir canvis en els tests. Per aconseguir-ho, s'ha de separar el "què" fa el sistema del "com" ho fa. Els tests que escrivim no han de dir res sobre com el sistema aconsegueix els seus objectius. Per tant, és essencial expressar els casos de test en el llenguatge del domini del problema i capturar tots els requisits des de la perspectiva d'un usuari extern del sistema. La clau és enfocar-se a allò que l'usuari vol del sistema i enllaçar aquestes especificacions executables amb les històries d'usuari, utilitzant el llenguatge del domini del problema.

6.2. Continuous delivery i els test d'acceptació

L'automatització dels tests d'acceptació escrits amb BDD són un procés crucial a la pipeline de Continuous Delivery, ja que ajuden a verificar que les característiques del sistema compleixin els requisits del negoci i les expectatives de l'usuari. En automatitzar aquests tests, es poden detectar ràpidament els errors i problemes en el software, permetent als equips de desenvolupament corregir-los abans que arribin a producció. A més,

l'execució automatitzada de tests d'acceptació a la pipeline de Continuous Delivery és fonamental per garantir que cada nova versió de l'aplicació s'entregui amb la màxima qualitat possible i amb la menor quantitat d'errors possible. Això és essencial per permetre que les organitzacions entreguin software de manera contínua i amb confiança.

6.3. Diversos enfocs d'implementació de BDD

Es poden seguir diferents enfocaments i filosofies per implementar BDD. Es pot dir que hi han tres enfocs a l'hora de desenvolupar software amb testing: "Outside-In", "Middle-Out" i "Inside-Out".

Les diferències entre aquests enfocs es centren en la manera com s'aborda l'escriptura dels tests i la seva integració amb el codi.

- **Outside-In** es centra en escriure els tests d'acceptació abans d'escriure el codi real. Això vol dir que es comença per les proves d'acceptació que simulen la interacció de l'usuari amb el sistema i que després s'escriuen les proves que permeten construir la funcionalitat requerida. En aquest enfocament es fan servir sovint mocks o stubs per simular els serveis externs i el comportament de les dependències i és conegut amb el terme de "Chicago school", fent referencia a una de les escoles de TDD.
- **Middle-Out** es centra en l'escriptura de proves unitàries (unit tests) i després s'avança a proves més àmplies com les d'integració o acceptació. Això vol dir que es comença per les proves unitàries, on es desenvolupen les funcionalitats més bàsiques i es van construint les funcionalitats més complexes.
- **Inside-Out** comença amb el desenvolupament de la lògica de negoci i després s'escriuen les proves unitàries per validar aquesta funcionalitat. Un cop s'han construït les funcionalitats més bàsiques, s'avancen a proves més àmplies com les d'integració. Aquest enfocament s'enfoca a desenvolupar el codi de manera incremental, construint la funcionalitat pas a pas.

Un mock és un objecte simulat que es fa servir en els tests per imitar el comportament d'objectes reals d'una manera controlada. Aquests mocks estan pre-programats amb

expectatives, de manera que es configuren per esperar certes crides específiques i comportar-se d'una manera específica quan s'invoquen. Llavors, en quant a la utilització de mocks, tots els enfocaments poden fer-ne ús, encara que l'enfocament Outside-In sol utilitzar-los amb més freqüència per simular els serveis externs i el comportament de les dependències, ja que es comença amb un enfoc més general i abstracte.

A l'hora d'implementar BDD a l'aplicació de demostració, és important valorar quin enfoc s'usa per al desenvolupament, així com les diferents maneres d'aplicar-ho amb Cucumber i quines facilitats proporciona per escriure els tests en llenguatge natural.

6.4. Tests d'acceptació efectius amb BDD cucumber

D'entre els aspectes que s'han de tenir en compte a l'hora d'escriure els tests usant BDD amb cucumber tenim que:

- Cada requeriment ha de convertir-se en història d'usuari, definint exemples concrets.
- Cada exemple ha de ser un escenari d'un usuari al sistema.
- Ser conscient de la necessitat de definir "l'especificació del comportament d'un usuari" en comptes de "la prova unitària d'una classe".
- Comprendre la fórmula *'Given-When-Then'* o la de les històries d'usuari *'Role-Feature-Reason'*.

Per definir els casos BDD per a una història d'usuari es defineixen amb el patró *Given-When-Then*:

- **Given 'donat'.** S'especifica l'escenari, les precondicions.
- **When 'quan'.** Les condicions de les accions que s'executaran.
- **Then 'llavors'.** El resultat esperat, les validacions a realitzar.

Cadascun d'aquests punts s'anomenen "steps". Un exemple pràctic seria:

- **Given:** Donat que l'usuari no ha introduït cap dada en el formulari.

- **When:** Quan fa click en el botó 'Enviar'.
- **Then:** S'han de mostrar els missatges de validació apropiats.

Aquests steps estan escrits en el "llenguatge natural" Gherkin en un fitxer amb l'extensió *.feature,* però necessiten ser implementats posteriorment en codi Java en una classe a la carpeta de tests que sol tenir un nom similar a "StepDefinitions". Aquesta implementació en Java dels steps és el que s'anomena "glue code", fent referencia al codi que està lligat a cadascun dels steps. És en aquest "glue code" on s'inserten les dades a la base de dades, on s'executaran els casos d'ús i on s'especificaran totes les verificacions que comprovaran que el comportament del codi és l'esperat.

L'altre patró és el de Role-Feature-Reason. Aquest patró s'utilitza a BDD per definir la features de la que volem escriure els escenaris:

- **As a 'Com a'.** Se especifica el tipus d'usuari.
- **I want 'desitjo'.** Les necessitats que té.
- **So that 'per tal que'.** Les característiques per complir l'objectiu.

Un exemple pràctic seria:

- **Com a** client interessat
- **desitjo** posar-me en contacte mitjançant el formulari
- **per tal que** atenguin les meves necessitats.

L'eina més destacada basada en el patró *'Given-When-Then'* és **Cucumber**, un framework de test amb suport BDD. A Cucumber, les especificacions de BDD estan escrites en llenguatge *Gherkin*, basat en *'Given-When-Then'*. D'entre els avantatges de BDD es poden destacar:

- Ja no estem definint 'proves', sinó que estem definint 'comportaments'.
- Millora la comunicació entre desenvolupadors, testers, usuaris i direcció.
- Com que BDD s'especifica utilitzant un llenguatge simplificat i comú, la corba d'aprenentatge és molt més curta que TDD.
- Com que la seva naturalesa no és tècnica, pot arribar a un públic més ampli.

- L'enfocament de definició ajuda a una acceptació comuna de les funcionalitats prèviament al desenvolupament.
- Aquesta estratègia encaixa bé en les metodologies àgils, ja que s'hi especifiquen els requisits com històries d'usuari i d'acceptació.

Per escriure tests d'acceptació efectius amb BDD usant cucumber i en un entorn en el que s'usa java i spring, hi ha algunes bones pràctiques que es poden seguir.

En primer lloc, es pot especificar un "background" per a tots els escenaris per establir un context comú per a tots els casos de test. Per exemple, si un usuari necessita iniciar sessió al sistema, es pot definir aquest pas d'inici de sessió com a part del "background" en lloc de repetir-lo a cada escenari. Això ajuda a reduir la redundància en els casos de prova i a simplificar-ne l'escriptura i el manteniment. De mateixa manera, és recomanable reutilitzar la definició dels steps, no escriure passos des de zero cada cop.

També, és recomanable afegir variables per defecte al steps per simplificar els escenaris. Per exemple, en comptes d'escriure:

```
WHEN an admin session successfully logs in with username 'JohnSmith' and the password 'JohnSmith123'
```

Escriure només:

```
WHEN an admin user logs in
```

Especificant els valors d'inici de sessió en una variable de classe en la definició dels steps. D'aquesta manera queden els casos de test més simples i els valors dels paràmetres queden amagats en un variable interna de la definició dels steps, ja que a l'escenari no té importància quins valors són exactament els d'inici de sessió. Tant se val que el username de l'admin sigui 'admin_user' o 'JohnSmith' o 'usuariDeTest', i el mateix passa amb el valor de la contrasenya. Per això, aquests valors queden amagats al setup de la classe on es defineixen els steps per a què els escenaris es vegin més simples.

Un altre cosa a tenir en compte és quant de tècnic hauria de ser el llenguatge dels escenaris BDD. Escriure els acceptance tests d'un backend API sol involucrar fer testing sobre requests, responses, http status, etc. Hi ha desenvolupadors que prefereixen no utilitzar terminologia tècnica en els tests d'acceptació i prefereixen escriure'ls en un llenguatge no tècnic. Un exemple d'això podria ser que en un escenari hipotètic en què es vol actualitzar dades d'un compte, en comptes d'escriure un step que sigui:

```
THEN the api client receives empty response with status 202
```

Escriure:

```
THEN the account is succesfully updated
```

Hi ha altres desenvolupadors, com per exemple els enginyers de la plataforma de cursos de CodelyTV, que en escriure test d'acceptació per a una api HTTP consideren que és generalment és una bona idea fer servir la terminologia i els conceptes més rellevants per al públic objectiu de l'API, que en molts casos són altres desenvolupadors que consumiran l'API. Llavors, usar terminologia tècnica com a codis d'estat HTTP i discutir sol·licituds i respostes és apropiat en aquest context.

En tot cas, és important que les "executable especifications" mantinguin el focus en què és el que el sistema ha de fer i no com ho fa. S'ha de parlar dels requeriments (el què) i no de la solució (el com).

Més enfocat en l'entorn en el que es desenvolupa l'aplicació de demostració, hi han alguns consells per a java java spring boot que poden servir per assegurar-se que les teves proves siguin efectives i cobreixin adequadament totes les capes de la teva aplicació.

L'ús de MockMvc en tests d'acceptació no és gaire adient, ja que MockMvc és més adient per a les proves unitàries del controlador ja que ens permet limitar l'abast de la prova a les responsabilitats principals del controlador: rebre les sol·licituds HTTP i serialitzar/deserialitzar la petició i la resposta. En aquesta prova unitaria del controlador

qualsevol interacció amb la capa de serveis estaria simulada (usant mocks) per aïllar i assegurar la funcionalitat del controlador.

Però per a les proves d'acceptació/end-to-end és més adient l'ús de *@SpringBootTest* per carregar el context complet de l'aplicació i fer ús de RestTemplate (o altres llibreries similars) és una bona opció per fer sol·licituds HTTP que assegurin que s'estan cobrint totes les capes de l'aplicació fins a obtenir la resposta. Fent ús de Gherkin/Cucumber poder tenir un happy path i codis d'error del endpoint documentats i coberts amb llenguatge natural. D'aquesta manera, ens podem assegurar que totes les funcionalitats de l'aplicació estan cobertes i funcionant correctament.

6.5. Preparació de les dades de la base de dades

En el llibre de "BDD In Action" es recomanen utilitzar base de dades de prova i inserir dades en steps previs. Tot i això, hi ha varis enfocaments per omplir una base de dades a l'hora de fer tests d'acceptació.

Un enfocament és utilitzar una base de dades in-memory com ara H2, per executar els test d'acceptació. En aquest enfoc, es configuren els steps per a que utilitzin la base de dades en memòria i després es deineixen les consultes a la base de dades als steps de Cucumber per inserir les dades de test directament:

```java
private JdbcTemplate jdbcTemplate;

@Before
public void setup() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    builder.setType(EmbeddedDatabaseType.H2);
    builder.addScript("classpath:schema.sql");
    builder.addScript("classpath:test-data.sql");
    jdbcTemplate = new JdbcTemplate(builder.build());
}

@Given("a registered user with username {string} and password {string}")
public void a_registered_user_with_username_and_password(String username, String password) {
    String insertUserSql = "INSERT INTO users (username, password) VALUES (?, ?)";
    jdbcTemplate.update(insertUserSql, username, password);
}
```

Fig. 1. Exemple de preparació de base de dades amb H2.

Un dels principals problemes amb l'ús d'una base de dades a la memòria és que pot portar a tests que passen a l'entorn local però que fallin en l'entorn de producció. Això es deu al fet que la base de dades en memòria pot comportar-se de manera diferent a la base de dades de producció. Per exemple, es poden trobar problemes amb tipus de dades, índexs, restriccions de clau forana o dialectes SQL que no són compatibles amb H2.

Utilitzar una base de dades diferent en comparació amb la base de dades de producció té els seus inconvenients. Per tant, un millor enfocament és el de simular al màxim el que hi ha en un entorn de producció. Per tant, es podria utilitzar l'anotació @Sql per executar un conjunt de consultes en una base de dades local creada amb l'objectiu de fer tests:

```java
@Sql("/sql/users/LoginUsersUseCase.sql")
@Given("a registered user with username {string} and password {string}")
public void a_registered_user_with_username_and_password(String username, String password) {
    // nothing in here because @Sql already inserted the data
}
```

Fig. 2. Exemple de preparació de base de dades amb @Sql.

Un altre enfocament similar podria ser emplenar la base de dades mitjançant la interfícies "Repository" de la capa d'infrastructura. Per exemple:

```
@Autowired
private UserRepository userRepository;

@Given("a registered user with username {string} and password {string}")
public void a_registered_user_with_username_and_password(String username, String password) {
  User user = new User(username, password);
  userRepository.save(user);
}
```

Fig. 3. Exemple de preparació de base de dades amb les interfícies de Repository.

Independentment de l'enfocament que es triï, és important valorar el fet que usar una base de dades de test igual que a la de producció és més segur i confiable que usar una in-memory com la de H2. És important que les dades de prova siguin representatives de les dades del món real que es trobaran a l'aplicació en producció i que els test reflecteixin amb precisió el comportament de l'aplicació a la producció.

Sobre la qüestió de si s'ha d'omplir o no una base de dades a les proves d'acceptació depèn de cada equip de treball, de la naturalesa del software que s'està desenvolupant i en quina etapa del cicle de vida del projecte ens trobem. En general, és una bona pràctica fer tests de l'aplicació amb un conjunt de dades realista i representatiu que imiti l'entorn de producció. Això ajuda a garantir que les proves reflecteixen el comportament de l'aplicació al món real. Però, no sempre és necessari o factible omplir una base de dades a gran escala en les proves d'acceptació. Per exemple, si l'aplicació depèn molt de serveis externs o API, pot ser més apropiat usar mocks d'aquests serveis. En aquests cas, s'han de simular amb precisió el comportament i les respostes dels serveis reals, i assegurar-se de que els test comprovin tots els camins d'èxit i d'error.

7. Desenvolupament del projecte de demostració

7.1. Concepte

L'aplicació de demostració a realitzar serà el backend d'una aplicació web (API HTTP) que permet a l'usuari mantenir un seguiment dels videojocs que està jugant actualment, que tingui planejats jugar o que ja hagi jugat/abandonat. Això es tradueix en un sistema que permetrà a l'usuari tenir un perfil a l'aplicació on podrà veure una espècie d'històric de videojocs que va jugant i que pugui inclús, segons la seva voluntat, puntuar-los. A futur, aquesta aplicació podria inclús créixer per admetre també altres tipus de productes com llibres, sèries de televisió, pel·lícules, etc.

7.2. Objectiu

L'objectiu principal del projecte de demostració és mostrar la implementació de la metodologia BDD amb Cucumber en un entorn real i mostrar i avaluar el procés realitzat per a posteriorment detallar-ne les conclusions i recomanacions sobre la metodologia.

7.3. Abast

El projecte que es dur a terme té com a finalitat el desenvolupament d'una aplicació API HTTP per a la part del backend. És important destacar que aquesta aplicació no es contempla que inclogui el desenvolupament de cap capa del frontend. L'abast del desenvolupament es centra, per tant, en l'elaboració de la part del servidor que ha de donar resposta als diferents requeriments descrits.

7.4. Arquitectura i metodologia de software

El desenvolupament del projecte es realitza amb el framework de Java Spring. Es seguirà la metodologia BDD amb cucumber i una arquitectura hexagonal, formada per tres capes principals: la capa d'infraestructura, la capa d'aplicació i la capa de domini. Aquesta arquitectura proporciona una separació clara entre les capes internes i externes de

l'aplicació, cosa que facilita la mantenibilitat del codi a llarg termini. És per això que aquesta arquitectura és molt adient en projectes on el software està en constant evolució i sotmès a molts canvis de requeriments i adició de noves funcionalitats, sobretot en equips de desenvolupaments mitjans i grans. És una arquitectura que està molt relacionada amb el desacoblament entre components i, per tant, això ajuda molt a l'hora de plantejar una i executar una bona estratègia de testing.

En aquesta arquitectura s'utilitza un dels patrons més reconeguts, el patró repository, per gestionar l'accés a la base de dades i proporcionar una capa de persistència desacoblada de la lògica de negoci de les capes internes.

7.5. Llistat de requeriments inicial

- L'usuari s'ha de registrar indicant un nom d'usuari únic, una contrasenya i el correu electrònic.
- L'usuari ha d'identificar-se amb el seu usuari i contrasenya.
- L'usuari ha de poder veure els videojocs de la seva biblioteca.
- L'usuari ha d'escollir en quin estat es troba una videojoc en afegir-lo a la biblioteca.
- L'usuari ha de poder triar entre els estats següents: want to play, playing, played, abandoned.
- L'usuari ha de poder modificar l'estat de qualsevol videojoc de la biblioteca.
- L'usuari ha de poder eliminar un videojoc de la biblioteca.
- L'usuari ha de poder afegir una puntuació personal al videojoc.
- L'usuari ha de poder modificar la puntuació personal d'un videojoc.
- L'usuari ha de poder eliminar la puntuació personal d'un videojoc.
- L'usuari ha de poder veure quin és la puntuació mitjana d'un videojoc.

7.6. Tecnologies

S'ha sel·leccionat **Java Spring Boot** com a framework de java ja que inclou una integració eficient amb Cucumber, que és una eina fonamental a la metodologia BDD. Spring Boot és el framework de backend que s'utilitza al propi grau universitari i l'estudiant compta amb experiència. A més a més, és un framework que proporciona comoditat, modularitat i facilita la implementació de serveis web i la gestió de dependències gràcies a maven. S'ha utilitzat la darrera versió LTS de Spring Boot, incloent la nova versió de seguretat que va incloure un canvi important en la forma de configurar la seguretat del projecte.

Com a editor de codi (IDE) per desenvolupar l'aplicació s'ha escollit **Intellij**. IntelliJ ofereix una gran quantitat de característiques útils i eines que agiliten el procés de desenvolupament. A més, és àmpliament reconegut i utilitzat arreu del món a l'hora de desenvolupar aplicacions Java Spring.

Per provar l'API, s'ha utilitzat **Postman**, una eina externa per fer peticions i provar que quan el backend està en marxa es poden fer peticions i el backend respon adequadament, tal i com ho faria qualsevol altre servei que vulgui consumir la nostra API. Postman ofereix una interfície fàcil d'utilitzar per enviar peticions HTTP a l'API, així com la possibilitat de gestionar i organitzar col·leccions de peticions per a diferents escenaris de prova.

S'ha escollit **XAMPP** per crear un entorn de desenvolupament local amb una base de dades MariaDB. XAMPP proporciona una solució tot en un per executar un servidor web Apache, un servidor de base de dades que són necessàries per al desenvolupament d'aquesta demostració. La base de dades proporcionada amb XAMPP és la que s'utilitza en el projecte quan l'aplicació s'inicia. Per als tests, s'utilitza **Testcontainers**, una tecnologia que es descriu més endavant com està integrada en els tests per instanciar una base de dades abans de la seva execució.

7.7. Seguretat i autenticació

Un dels aspectes clau del backend desenvolupat és la seguretat i autenticació. S'ha implementat un sistema d'autenticació basat en tokens JWT (JSON Web Tokens). Cada petició al backend requereix autenticació, cosa que implica que els usuaris han d'existir a la base de dades i sol·licitar un token JWT a través d'un endpoint d'autenticació. Aquest token és el que es fa servir per autoritzar les sol·licituds a altres endpoints que sí requereixen d'autenticació prèvia. En cada petició, l'usuari es sotmet a un sistema d'autenticació en base al token JWT i les seves credencials.

És important esmentar que l'endpoint d'autenticació és l'únic que està obert a tothom, és a dir, no requereix autenticació en si mateix, ja que cal que qualsevol usuari pugui obtenir un token JWT vàlid per utilitzar els altres endpoints de l'aplicació.

7.8. Estructura del projecte

El projecte segueix una estructura de "vertical slicing", en la que el projecte es divideix per recursos. Per exemple, existeix una carpeta "library_game" que conté tot allò relacionat amb el recurs library game, que fa referencia a un joc que està a la biblioteca d'un usuari. Aquesta carpeta conté els casos d'us, la definició del model de domini, la implementació de l'accés a base de dades, etc. Cadascuna d'aquestes carpetes esta subdividida en les tres capes que conformen l'arquitectura hexagonal (domini, aplicació i infrastructura). D'igual manera existeix, per exemple, una altra carpeta anomenada "Users" que segueix la mateixa estructura.

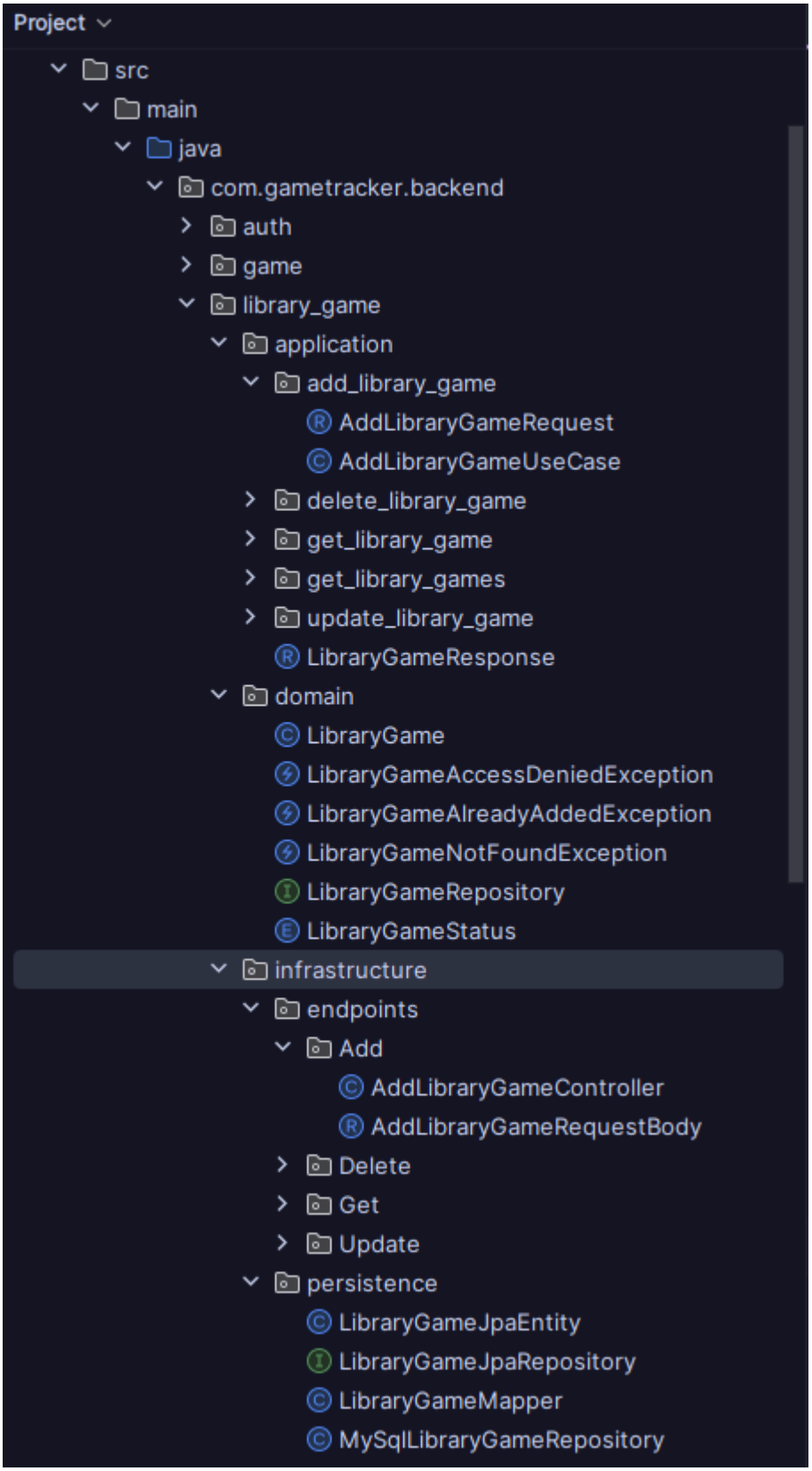


Fig. 4. Estructura del projecte

7.9. Implementació de BDD

En la implementació de la metodologia BDD (Behavior-Driven Development) s'han seguit una sèrie de passos i pràctiques amb l'objectiu de garantir la qualitat del sistema i posant en pràctica tot allò que prèviament s'ha investigat en la fase d'exploració. A continuació, es detallen els aspectes rellevants de la implementació:

- **Test d'acceptació.** Els tests realitzats pertanyen a la categoria de tests d'acceptació. Aquests tests tenen com a objectiu validar el comportament de l'aplicació des del punt de vista de l'usuari final. Tot i que s'utilitza un mock per simular un servei extern, la base de dades utilitzada en els tests d'acceptació és una rèplica de la que hi hauria en producció, assegurant-se que els tests es realitzin en un entorn similar al real.

  Per tant, per a executar aquests tests, és necessari aixecar l'aplicació sencera i fer peticions HTTP directament al servei per a provar que tot funciona correctament, utilitzant un mock per a la simulació dels serveis externs i assegurant-se que les interaccions amb la base de dades es comportin de manera esperada també.

- **Base de dades de test.** Per a la gestió de l'entorn de desenvolupament i de les proves d'acceptació, s'ha implementat **Testcontainers**, una llibreria que permet la creació automàtica de contenidors Docker per a entorns d'execució de tests. En el cas de la base de dades de prova, Testcontainers s'utilitza per aixecar automàticament un contenidor Docker de MariaDB configurat amb la base de dades de prova cada vegada que s'executen els tests d'acceptació. Aquesta aproximació automatitzada elimina la necessitat de crear manualment la base de dades de prova al nostre sistema, proporcionant un procés més eficient i reproduïble.

  Pel que fa a l'esquema de la base de dades, la gestió es realitza a través del codi del projecte. S'ha utilitzat la llibreria Jakarta Persistence (anteriorment coneguda com a JPA) en combinació amb Hibernate. Amb JPA, es defineix l'estructura de les taules de la base de dades a partir de classes Java anotades amb @Entity. El mapatge dels objectes Java a les taules de la base de dades es realitza automàticament en posar en marxa l'aplicació, simplificant el procés de desenvolupament i configuració de la base de dades.

Cal destacar que en un entorn de producció real, els valors de les bases de dades haurien d'estar configurats com a variables d'entorn del sistema, no escrits manualment al codi. Aquesta pràctica ofereix més seguretat al sistema i més flexibilitat, ja que permet adaptar la configuració de la base de dades segons l'entorn on es desplegui.

- **Classes StepDefinitions.** S'ha anat creant classes de Step Definitions per mòduls. Hi ha una classe de steps que s'encarrega de fer les crides a la api anomenada RestApiSteps.java i després hi ha una serie de classes step per a cada recurs de l'aplicació: GameSteps.java, LibraryGameSteps.java, RoleSteps.java i UserSteps.java. D'aquesta manera, es promou la modularitat i reutilització de codi en el desenvolupament de noves funcionalitats.

- **Ús d'interfícies repository al "glue code".** Finalment, s'ha optat per l'ús de les interfícies repository per a inserir les dades prèvies a l'execució dels escenaris de test. Cal dir que l'ús de les interfícies dels repository ha sigut una molt bona decisió i la més còmode para inserir dades a la base de dades als tests BDD. En utilitzar interfícies, s'estableix un contracte clar entre la capa de domini i la capa de persistència. Això facilita el desenvolupament dels tests, ja que es poden fer servir les implementacions dels repositoris usant objectes de domini sense arribar a conèixer el detall com es guarden en base de dades. Ajuda a la llegibilitat del codi i a mantenir una cohesió entre tota la lògica de domini i el comportament que s'està provant.

7.10. Integració d'aspectes del desenvolupament amb cucumber

En aquest apartat es detalla com diferents aspectes del desenvolupament es poden incloure als tests d'acceptació usant cucumber i spring boot:

- El seguiment de l'esquema JSON Draft-07 per a validar les respostes dels endpoints.

- L'ús de beans (dependencies) al "glue code".

- L'ús de mocks per a serveis externs.

7.10.1. Validació de les respostes http

En el context d'aquest treball, s'està utilitzant l'esquema JSON Draft-07 per verificar el format de les respostes generades pels endpoints. L'esquema JSON Draft-07 és el que l'especificació OpenAPI (anteriorment coneguda com a Swagger) utilitza. OpenAPI 3.1.0, que és l'última versió de l'especificació, fa referència a l'esquema JSON Schema Draft-07 com el format d'esquema predeterminat per descriure els payloads de la petició i de la resposta dels endpoints. Llavors, l'ús d'aquest esquema en aquest projecte ens proporciona un conjunt de regles i convencions per documentar i definir les respostes esperades dels endpoints. Aquest esquema només s'utilitza en els tests amb la finalitat de verificar el format de les respostes http.

En una carpeta anomenada "schemas" ubicada als recursos dels test, s'han inclòs uns fitxers en format json que contenen el esquema dels diferents tipus d'esquemes de respostes generades pels endpoints corresponents als diferents casos d'ús.

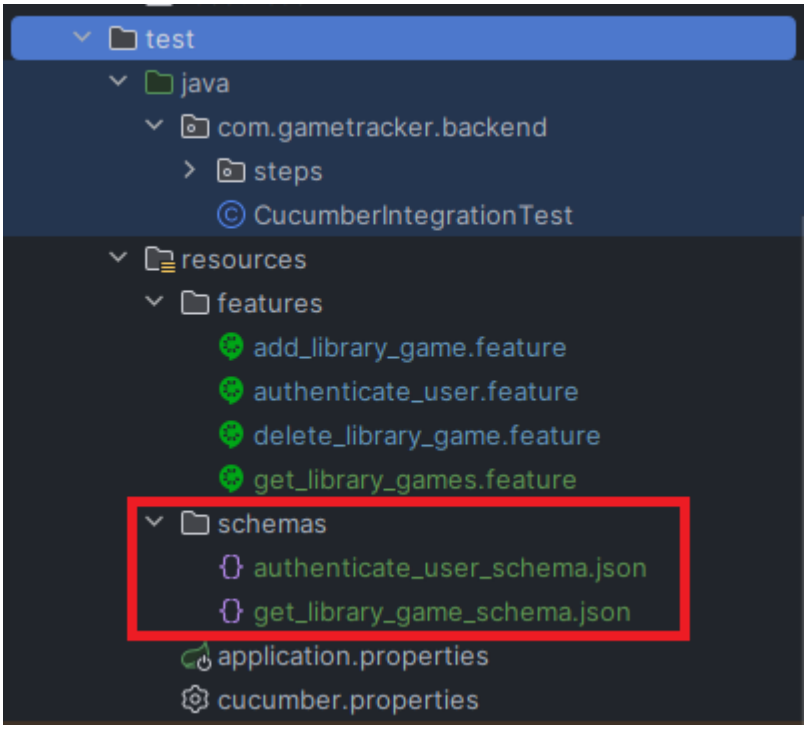


Fig. 5. Carpeta i arxius dels schemas

Durant l'execució dels tests d'acceptació, es verifica que les respostes dels endpoints s'ajustin adequadament a aquests esquemes definits. La carpeta schemas s'ha hagut d'afegir al classpath per a poder accedir i llegir els fitxers json d'una forma més còmode. Per tant, s'ha afegit al pom.xml el directori per a incloure'l al classpath:

```
<build>
    <resources>
        <resource>
            <directory>src/test/resources</directory>
            <includes>
                <include>schemas/**</include>
            </includes>
        </resource>
    </resources>
    <plugins>
```

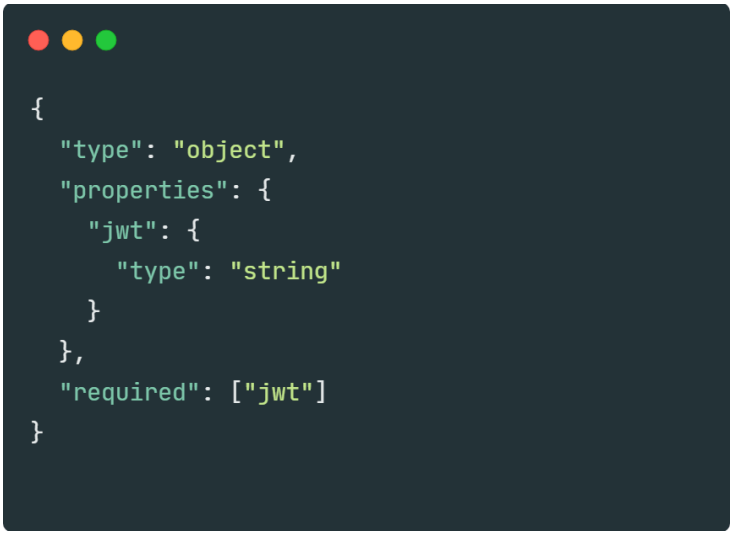
Fig. 6. Configuració per incloure els schemas al classpath

Per posar un exemple, la resposta a la petició d'autenticació de l'endpoint "/authenticate" és un json d'aquest estil:

```
{
  "jwt": "eyJhbGciOiJIUzI1NiJ9.eyJSb2xlIjoiQWRtaW4iLCJJc3N1ZXIiOiJJc3N1ZXIiLCJVc2VybmFtZS...."
}
```

Fig. 7. Resposta de l'endpoint d'autenticació

Per validar el format d'aquesta resposta s'ha de crear un nou fitxer json a la carpeta schemas que contindria l'esquema següent:



```
{
  "type": "object",
  "properties": {
    "jwt": {
      "type": "string"
    }
  },
  "required": ["jwt"]
}
```

Fig. 8. Schema de la resposta de l'endpoint d'autenticació

En aquest esquema s'especifica que és un objecte que ha de tenir una propietat amb el nom de "jwt", que ha de ser un string i que és un atribut obligatori.

Als tests d'acceptació, es duu a terme la validació del format de la resposta. És a dir, en base a aquest esquema definit, es comprova que la resposta contingui els atributs específics, els seus tipus respectius i que els atributs requerits siguin presents. Als tests d'acceptació, no es fa una validació dels valors dels atributs, ja que això forma part de la lògica del sistema i és millor que se n'encarreguin els tests unitaris i/o d'integració.

Com a exemple complet, considerem l'escenari de test on es verifica que un usuari pugui autenticar-se amb èxit:

ATÈS QUE existeix l'usuari al sistema
QUAN l'usuari fa una petició d'autenticació amb el seu nom d'usuari i la seva contrasenya
ALESHORES el servidor indica que la petició ha estat un èxit
i **ALESHORES** el servidor retorna un objecte JSON que conté el token d'autenticació

En aquest cas, no s'està validant el valor real de l'atribut "token" de la resposta, sinó que es verifica que l'endpoint retorni una resposta en format JSON que contingui un atribut amb el token. Si ens fixem, això és bastant similar a com una persona no-tècnica escriuria un test d'acceptació de la nostra api. Si examinem com s'ha especificat aquest escenari a l'arxiu "authenticate_user.feature" utilitzant el llenguatge Gherkin, podem observar que es

fa servir un llenguatge una mica més tècnic i concret i es defineixen paràmetres específics per fer reutilitzables els steps, però és entenedor per a qualsevol persona que vulgui consumir la nostra api:

```
Feature: User authentication

  Background:
    Given the following users exist:
      | uuid      | username | password    | email              | role |
      | random_id | johnsnow | johnsnow123 | johnsnow@email.com | USER |

  Scenario: User authenticates successfully
    When the unauthenticated user sends a "POST" request to "/api/authenticate" with the following JSON body:
      """
      {
        "username": "johnsnow",
        "password": "johnsnow123"
      }
      """
    Then the server responds with a 200 status code
    And the response body should have the following JSON format:
    """
    {
      "type": "object",
      "properties": {
        "jwt": {
          "type": "string"
        }
      },
      "required": ["jwt"]
    }
    """
```

Fig. 9. Exemple de l'arxiu gherkin de la funcionalitat d'autenticació

7.10.2. Mocking

En el desenvolupament d'aquest projecte, s'estableix comunicació amb un servei extern anomenat IGDB (Internet Game Database) que allotja una base de dades actualitzada de videojocs. Aquest servei extern correspon a una base de dades en línia sobre videojocs llançada el 2014, la qual compta amb més de 690.000 entrades. Des del 2019, IGDB és propietat de Twitch, la plataforma de streaming més popular en l'actualitat i subsidiària d'Amazon.com. Aquest servei és gratuït encara que conté limitacions d'ús.

La integració amb aquest servei extern es realitza mitjançant comunicació HTTP. El servei proporciona una API pública a la qual és necessari autenticar-se per a realitzar peticions HTTP i obtenir informació sobre els videojocs existents. Per poder efectuar aquestes

peticions, cal accedir al "Twitch developer portal" de twitch.com utilitzant un compte de Twitch. Posteriorment, cal registrar l'aplicació i obtenir un "client-id" i una "secret key", els quals serveixen per autenticar les peticions a l'API i obtenir un token. Aquest token s'ha d'incloure en cada sol·licitud a l'API. És important destacar que el token té una validesa de 60 dies, per la qual cosa serà necessari renovar-lo una vegada transcorregut aquest període. Tot i que és possible automatitzar aquest procés de renovació, per a l'execució d'aquest projecte s'ha optat per mantenir un enfocament simple i utilitzar un token amb una durada de 60 dies i, quan caduqui, renovar-lo de manera manual.

Amb l'objectiu de seguir els principis de l'arquitectura neta i permetre futurs canvis en el servei extern sense afectar els clients que utilitzen aquesta interfície, s'ha creat una classe encarregada de la comunicació amb el servei extern. Aquesta classe implementa una interfície definida, seguint les millors pràctiques. Aquesta classe es diu GameService.

No obstant això, per provar els casos d'ús en els tests d'acceptació, és necessari realitzar un "mock" (simulació) d'aquesta classe responsable de la comunicació, ja que el servei extern té un límit de peticions per segon. Concretament, aquest límit es troba en 4 peticions per segon. Si aquest límit es supera, la següent petició retornarà un error amb codi d'estat http 429 de "Too many requests".

En realitzar el "mock" del component GameService, és possible modificar el seu comportament i fer que es comporti segons les nostres necessitats. Per exemple, podem establir que en cercar un joc pel seu títol, el component retorni la resposta que aquest joc no existeix, retornant null:

```
@Given("the game does not exist")
public void theGameDoesNotExist() {
    when(gameService.searchGame(anyString())).thenReturn(null);
}
```

Fig. 10. Exemple d'implementació del mock del servei extern

D'aquesta manera, estem simulant que fem una petició a l'API externa i ens retorna un null en cercar un joc.

En el projecte, s'usa la biblioteca Mockito per dur a terme aquesta tasca. Amb aquesta biblioteca, és possible utilitzar l'anotació @MockBean per fer un "mock" d'un component del sistema. És important destacar que el "mock" ha de ser declarat a la classe que conté l'anotació @CucumberConfiguration perquè tingui efecte correctament. Un cop fet això, podem usar els mètodes when() i thenReturn() en conjunció per modificar el comportament del component simulat.

Sovint sorgiran ocasions en què sigui necessari realitzar un "mock" d'un component, com quan s'utilitzen els serveis d'Amazon Web Services com S3 per allotjar fitxers o l'activació de funcions Lambda de AWS o qualsevol altre proveïdor de cloud. En fer això, s'evita executar crides reals als serveis d'Amazon, com pujar fitxers a un bucket o activar funcions lambda, el que permet controlar i simular el seu comportament en proves i evitar impactes en entorns de producció.

7.11. Exemples d'implementació dels test

En aquesta secció es mostraran exemples de la implementació dels test bdd.

Com a primer exemple tenim el d'afegir un videojoc a la biblioteca de l'usuari. L'especificació del fitxer *add_library_game.feature* queda de la següent manera:

```
Feature: Add library game
  As a basic user
  I want to add a game to my library
  So i can keep track of its status and personal rating

  Background:
    Given the following roles exist:
      | ROLE_USER |
    And a user with username "johnsmith" and role "ROLE_USER" is logged in


  Scenario: User adds an existing game to their library successfully
    Given the game exists
    When the authenticated user sends a "POST" request to "/api/library-games" with the following JSON body:
      """
      {
        "id": "da1e846d-25e1-44e7-91f8-3cca9348d1b6",
        "title": "Random game title",
        "rating": 4.5,
        "status": "PLAYING"
      }
      """
    Then the server responds with a 201 status code
    And library-games with the following ids should be in the database:
      | da1e846d-25e1-44e7-91f8-3cca9348d1b6 |
```

Fig. 11. Exemple de l'escenari d'afegir un videojoc a la biblioteca exitós

```
  Scenario: User fails to add an already added game in their library
    Given the game exists
    And the following library-games exist:
      | id                                   | title            | rating | status | username  |
      | da1e846d-25e1-44e7-91f8-3cca9348d1b6 | Random game title | 4.5    | PLAYED | johnsmith |
    When the authenticated user sends a "POST" request to "/api/library-games" with the following JSON body:
      """
      {
        "id": "da1e846d-25e1-44e7-91f8-3cca9348d1b6",
        "title": "Random game title",
        "rating": 4.5,
        "status": "PLAYING"
      }
      """
    Then the server responds with a 409 status code
    And the response body should contain the message "Library game with title 'Random game title' is already added"
```

Fig. 12. Exemple de l'escenari d'afegir un videojoc a la biblioteca erroni

Com es pot veure, amb la paraula clau *Feature* s'indica sobre quina funcionalitat es volen escriure els escenaris bdd. Primer de tot s'ha creat un step de *background* en el que es prepara un usuari a la base de dades i s'autentica fent una petició al propi backend per a obtenir el token jwt necessari per a les peticions posteriors.

Després s'indica quin escenari volem provar amb la paraula clau *Scenario*. En el primer escenari es vol provar que un usuari autenticat pot afegir correctament un videojoc a la seva biblioteca. En el segon escenari es vol comprovar que un usuari no pot afegir un videojoc existent a la seva biblioteca.

A continuació s'especifiquen els steps que s'han de realitzar amb les paraules clau *Given, When, Then*. Malgrat seguir un format en concret (gherkin) és un tipus d'especificació que qualsevol humà pot arribar a entendre. Com s'ha comentat prèviament, els usuaris finals d'un servei backend com el d'aquesta demostració seran altres desenvolupadors que consumiran l'api d'aquest servei, per tant, és adient usar terminologia tècnica sobre peticions webs com els termes "request" o els codis d'estat com el 409, que indica que hi ha hagut un conflicte a l'hora de processar la petició, ja que ja existeix un videojoc amb el mateix identificador.

A continuació algunes porcions de codi del glue code:

```
40      @Before
41      public void beforeEachScenario() {
42          databaseTruncator.truncateAllTables();
43      }
```

Fig. 13. Codi del @Before

En aquesta classe hi ha una anotació @Before que s'executa abans de cada escenari i neteja totes les dades de la base de dades, de manera que cada escenari comença amb la base de dades buida. D'aquesta manera ens assegurem que cada escenari és independent dels altres i es reproduible.

A continuació es pot veure el codi del pas d'autenticació. Bàsicament es crea un usuari a la base de dades amb el nom i rol especificats al step i després s'executa una petició de login amb l'usuari i contrasenya generats, cosa que retorna un token d'autenticació que s'ha de guardar per a les consegüents peticions de l'escenari.

```java
@Given("a user with username {string} and role {string} is logged in")
public void theFollowingUserLogsIn(String username, String role) throws IOException {
    Faker faker = new Faker();
    User user = User.builder()
            .id(faker.internet().uuid())
            .username(username)
            .password(faker.internet().password())
            .email(faker.internet().emailAddress())
            .role(RoleName.valueOf(role))
            .build();
    userRepository.save(user);

    currentJwt = loginAndGetAccessToken(user.getUsername(), user.getPassword());
}

private String loginAndGetAccessToken(String username, String password) throws IOException {
    var requestBody = new JSONObject()
            .put("username", username)
            .put("password", password);

    var mediaType = okhttp3.MediaType.parse( $this$parse: "application/json");
    var body = RequestBody.create(requestBody.toString(), mediaType);
    var request = new Request.Builder()
            .url(BASE_URL + "/api/auth/login")
            .method( method: "POST", body)
            .addHeader( name: "Content-Type",  value: "application/json")
            .addHeader( name: "Accept",  value: "application/json")
            .build();

    var response = okHttpClient.newCall(request).execute();
    return new JSONObject(response.body().string()).getString( key: "accessToken");
}
```

Fig. 14. Codi del step d'autenticació

A continuació es pot veure el codi del step que realitza la petició http:

```java
@When("the authenticated user sends a {string} request to {string} with the following JSON body:")
public void authenticatedUserSendsRequestWithJsonBody(String method, String uri, String jsonBody)
        throws IOException {
    MediaType mediaType = MediaType.parse( $this$parse: "application/json");
    RequestBody body = RequestBody.create(jsonBody, mediaType);
    Request request = new Request.Builder()
            .url(BASE_URL + uri)
            .method(method, body)
            .addHeader( name: "Content-Type",  value: "application/json")
            .addHeader( name: "Accept",  value: "application/json")
            .addHeader( name: "Authorization",  value: "Bearer " + currentJwt)
            .build();
    latestResponse = okHttpClient.newCall(request).execute();
}
```

Fig. 15. Codi del step d'execució de la request

Aquest codi guarda en una variable privada de classe la petició actual que s'està gestionant per a posteriorment poder-ne verificar el seu resultat en un altre step.

A continuació el codi del step que verifica que el codi http és el correcte:

```
124     @Then("the server responds with a {int} status code")
125     public void serverRespondsWithStatusCode(int expectedStatusCode) {
126         assertEquals(expectedStatusCode, latestResponse.code());
127     }
```

Fig. 16. Codi del step de verificació de la petició

El següent exemple és el d'obtenir videojocs de la biblioteca d'un usuari. L'arxiu s'anomena *get_library_games.feature* i és el següent:

```
Feature: Get library games
  As a basic user
  I want to be able to retrieve games of my library
  So i can see all of its information

  Background:
    Given the following roles exist:
      | ROLE_USER |
    And a user with username "johnsmith" and role "ROLE_USER" is logged in

  Scenario: User gets an existing game of their library successfully
    Given the following library-games exist:
      | id                                   | title            | rating | status | username  |
      | da1e846d-25e1-44e7-91f8-3cca9348d1b6 | Random game title | 4.5    | PLAYED | johnsmith |
    When the authenticated user sends a "GET" request to "/api/library-games/da1e846d-25e1-44e7-91f8-3cca9348d1b6"
    Then the server responds with a 200 status code
    And the response body should have the following JSON format "/response_schemas/get_library_game_schema.json"

  Scenario: User fails to get a non existing game of their library
    When the authenticated user sends a "GET" request to "/api/library-games/random_nonexistent_id"
    Then the server responds with a 404 status code

  Scenario: User fails to get a game of another users library
    Given the following users exist:
      | id          | username     | password        | email                  | role |
      | random_id_2 | marcgonzalez | marcgonzalez123 | marcgonzalez@email.com | ROLE_USER |
    Given the following library-games exist:
      | id                                   | title            | rating | status | username     |
      | da1e846d-25e1-44e7-91f8-3cca9348d1b6 | Random game title | 4.5    | PLAYED | marcgonzalez |
    When the authenticated user sends a "GET" request to "/api/library-games/da1e846d-25e1-44e7-91f8-3cca9348d1b6"
    Then the server responds with a 403 status code

  Scenario: User gets all games of their library
    Given the following library-games exist:
      | id                                   | title            | rating | status    | username  |
      | da1e846d-25e1-44e7-91f8-3cca9348d1b6 | Random game title | 4.5    | PLAYED    | johnsmith |
      | ef227608-ec18-40e7-b93f-714c8af40fc2 | Another game title | 0      | ABANDONED | johnsmith |
    When the authenticated user sends a "GET" request to "/api/library-games"
    Then the server responds with a 200 status code
    And the response body should have the following JSON format "/response_schemas/get_library_games_schema.json"
```

Fig. 17. Exemple de l'escenari d'obtenció de videojocs de la biblioteca de l'usuari

En aquest cas es vol comprovar que un usuari autenticat definit en el pas previ background, pot obtenir correctament els videojocs de la seva biblioteca, i únicament de la seva biblioteca. Com es pot veure en el penúltim escenari, un usuari no pot accedir a un videojoc que no sigui de la seva biblioteca.

Segueix la mateixa estructura que la funcionalitat anterior, ja que conté un *Background* que autentica a un usuari i és després amb aquest usuari que es realitzen les peticions al servei. Com es pot veure, els steps que verifiquen el codi d'estat http són els mateixos que a la

funcionalitat anterior. És per això que és important dissenyar els steps per a que puguin ser reutilitzables i així millorar la qualitat del codi i la comoditat a l'hora de crear nous test bdd.

7.12. Diferencia de test unitari amb test d'acceptació

En el desenvolupament de software, és important fer ús de diferents tipus de test per assegurar la qualitat del codi. Dues de les proves més importants són el test unitari i el test d'acceptació. En aquesta secció es detallen les diferències entre els dos tipus de test.

Com a exemple, tenim el cas d'us d'afegir un videojoc a la biblioteca d'un usuari. El primer que podem destacar es que hem d'especificar que les dependències del components seran objectes simulats amb la llibreria Mockito, que et permet crear mocks d'objectes:

```
1  package com.gametracker.backend.unit.library_game.application.add_library_game;
2
3  import ...
23
24 class UpdateLibraryGameUseCaseTest {
25
26     @Mock
27     private LibraryGameRepository libraryGameRepository;
28
29     @Mock
30     private GameRepository gameRepository;
31
32     @InjectMocks
33     private AddLibraryGameUseCase addLibraryGameUseCase;
34
35     @BeforeEach
36     void setUp() {
37         MockitoAnnotations.openMocks( testClass: this);
38     }
39
```

Fig. 18. Codi de la inicialització de les dependències del test unitari

Després tenim com a primer test el d'afegir un videojoc a la biblioteca quan l'usuari encara no l'ha afegit. Es pot veure que abans d'executar el cas d'ús, s'han de preparar les dades de les dependències simulades especificant com s'han de comportant concretament i al final es fa una verificació de si el mètode de guardar a la base de dades s'ha cridat internament o no.

```
@Test
void execute_ShouldAddLibraryGame_WhenGameExistsAndNotAlreadyAdded() {
    Game game = Game.builder()
            .title("Test Game")
            .build();
    AddLibraryGameRequest request = new AddLibraryGameRequest(
            id: "1",
            title: "Test Game",
            rating: 4.5,
            LibraryGameStatus.PLAYED,
            username: "user123"
    );
    when(gameRepository.findGame( title: "Test Game"))
            .thenReturn(Optional.of(game));
    when(libraryGameRepository.findByTitleAndUsername( title: "Test Game",  username: "user123"))
            .thenReturn( t: null);

    addLibraryGameUseCase.execute(request);

    verify(libraryGameRepository, times( wantedNumberOfInvocations: 1)).save(any(LibraryGame.class));
}
```

Fig. 19. Codi del test unitari d'afegir un videojoc a la biblioteca

Com a segon test d'exemple tenim quan el cas d'ús falla degut a que el joc no existeix. En aquest cas abans d'executar el cas d'ús s'ha de configurar el comportament del repository de jocs especificant que quan es vulgui buscar el joc retorni un objecte null. Al final del test es verifica que el cas d'ús ha tirat una excepció i que el mètode de guardar a la base de dades no s'ha cridat.

```
@Test
void execute_ShouldThrowGameDoesNotExistException_WhenGameDoesNotExist() {
    AddLibraryGameRequest request = new AddLibraryGameRequest(
            id: "1",
            title: "Nonexistent Game",
            rating: 4.5,
            LibraryGameStatus.PLAYED,
            username: "user123"
    );
    when(gameRepository.findGame( title: "Nonexistent Game"))
            .thenReturn( t: Optional.empty());

    assertThrows(GameDoesNotExistException.class, () -> addLibraryGameUseCase.execute(request));
    verify(libraryGameRepository, never()).save(any(LibraryGame.class));
}
```

Fig. 20. Codi del test unitari d'afegir un videojoc a la biblioteca erroni

Una vegada vist el test untari, en podem destacar la diferència entre els dos tipus de test:

- **Test Unitàri:** El test unitari prova components individuals o unitats del programa de forma aïllada. El seu propòsit és verificar que un tros específic de codi, com ara un mètode o una classe funcioni correctament sota unes condicions concretes. Aquest tipus de test es centra en la lògica interna de l'aplicació, assegurant-se que cada part funcioni com s'espera de manera independent. Té un nivell d'abstracció baix, ja que interactua directament amb el codi utilitzant objectes simulats com a dependències de manera que aïlla el component de la resta del codi.

- **Test d'Acceptació:** El test d'acceptació prova el sistema en el seu conjunt per assegurar-se que compleix els requisits empresarials i les necessitats dels usuaris. El seu propòsit és validar que tota l'aplicació funcioni com s'espera des de la perspectiva de l'usuari final, centrant-se en el comportament de l'aplicació en un escenari d'ús realista. Té un nivell d'abstracció alt, ja que simula l'ús del món real interactuant amb l'aplicació a través de les seves interfícies públiques, com ara les APIs o la interfície d'usuari (UI), per garantir que tots els components del sistema funcionen correctament en conjunt.

8. Conclusions

En aquest treball, s'ha realitzat una investigació i exploració de la metodologia BDD utilitzant la llibreria Cucumber, amb l'objectiu de desenvolupar un projecte de demostració que mostres la seva aplicació. Al llarg de l'estudi, s'han analitzat els antecedents i l'origen de la metodologia, s'han establert uns objectius clars i un abast per al projecte.

La metodologia utilitzada s'ha descrit detalladament, destacant aspectes com les proves d'acceptació automatitzades, el lliurament continu i diferents enfocaments d'implementació de BDD. A més, s'ha aprofundit en la preparació de dades de la base de dades, un aspecte molt important per a les proves d'acceptació i per a simular un entorn el mes realista possible.

En el desenvolupament del projecte de demostració, s'ha establert un concepte clar, objectius i abast definits, i s'ha descrit l'arquitectura i la metodologia utilitzada. S'han identificat i llistat els requeriments inicials, s'han seleccionat les tecnologies adequades i s'han tingut en compte els aspectes de seguretat i autenticació.

La implementació de BDD s'ha dut a terme de manera efectiva, integrant aspectes del desenvolupament amb Cucumber, com ara la validació de respostes HTTP i el mocking de serveis externs que s'escapen del nostre control. S'han proporcionat exemples d'implementació de proves i s'ha destacat la diferència entre les proves unitàries i les proves d'acceptació.

Com a conclusió final, aquest projecte ha permès avaluar els beneficis i desafiaments de la metodologia BDD, així com identificar les millors pràctiques per garantir una implementació eficient i mantenible del programari. S'espera que aquest treball contribueixi al coneixement existent en el camp del desenvolupament de software i pugui servir com a guia.

Com a conclusió personal, m'agradaria afegir que durant els meus anys de pràctiques a l'empresa, hem incorporat la metodologia BDD en el nostre procés de desenvolupament de software, una iniciativa que va sorgir a partir de les meves propostes i esforços per millorar

el procés de desenvolupament i desplegament del nostre software. Fins ara, només realitzàvem proves unitàries dels components més crítics. Aquesta metodologia, que hem adoptat com a part del nostre enfocament de TDD en proves end-to-end i d'acceptació, ens ha resultat molt efectiva. Els escenaris creats són extremadament nets, llegibles i mantenibles, de manera que qualsevol desenvolupador que pugui entrar nou a l'equip els pugui entendre llegint les especificacions en format gherkin sense cap problema. A més, una de les majors avantatges és la capacitat de reutilitzar steps, la qual cosa permet crear suites de testing de manera ràpida un cop que el glue code dels steps principals està programat. Aquestes proves ens proporcionen una gran confiança, ja que solen fer un ús mínim d'objectes simulats, de manera que ens donen una visió realista del funcionament del sistema. No obstant això, una desavantatge notable és el feedback loop prolongat a causa del temps d'execució, que és significativament major en comparació amb les proves unitàries.

9. Annexe

Aquest annex proporciona informació addicional sobre els requisits i l'execució del codi i tests del projecte.

Requisits d'execució:

- **Java 17.0.9:** Aquesta és la versió de Java utilitzada en el projecte.
- **Apache Maven 3.9.6:** És la versió de Maven utilitzada en el projecte per a la gestió de dependències i la construcció de l'aplicació.
- **Una instància local de MySQL/MariaDB:** Per a executar l'aplicació en la màquina local, és necessària una base de dades MySQL o MariaDB executant-se localment a localhost:3306 amb el nom 'game_tracker'. Es pot configurar utilitzant una instància de MySql/MariaDB de Docker o usant XAMPP.
- **Docker:** És necessari tenir Docker instal·lat al sistema per a executar els tests d'acceptació. Aquests tests fan ús de Testcontainers, l'eina que facilita la creació d'entorns de test temporals amb imatges de Docker.

Entorn de desenvolupament integrat (IDE):

- Es recomana utilitzar un IDE que admeti l'execució de codi Java, com ara IntelliJ IDEA. Aquest IDE ofereix una àmplia gamma de característiques i eines que faciliten el desenvolupament, depuració i execució del codi Java amb Spring Boot i integració amb Cucumber.

Per a l'execució dels tests, abans de procedir amb la seva execució, s'han de complir tots els requisits esmentats prèviament. Una vegada complerts, es poden executar de diverses maneres:

1. Des de l'IDE IntelliJ IDEA. Es poden executar els tests directament des de l'entorn de desenvolupament integrat IntelliJ IDEA. Per a fer-ho, s'ha de navegar fins al directori on

es troben els tests de les funcionalitats de Gherkin, clicar amb el botó dret sobre el fitxer de la feature i finalment escollir l'opció per executar la feature (“Run Feature: user_login”).

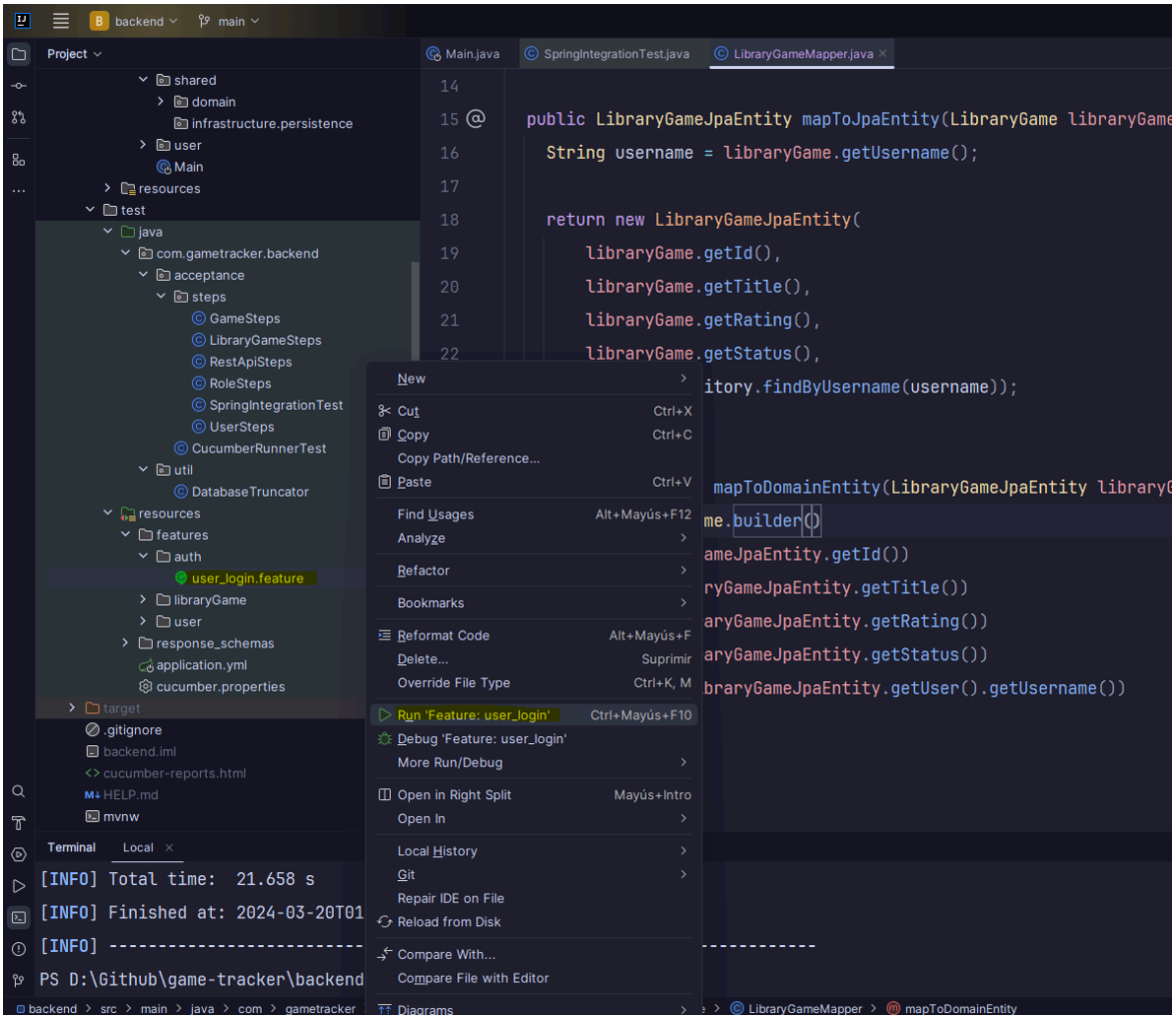


Fig. 21. Mostra d'execució dels test d’un arxiu gherkin

2. Utilitzant maven des de la terminal. També es pot utilitzar maven des de la terminal per a executar els tests obrint una terminal en el directori del projecte i executant la comanda: **mvn test**. Aquesta comanda executarà tots els tests definits al projecte utilitzant maven sempre que estigui instal·lat i reconegut com a comanda del sistema.

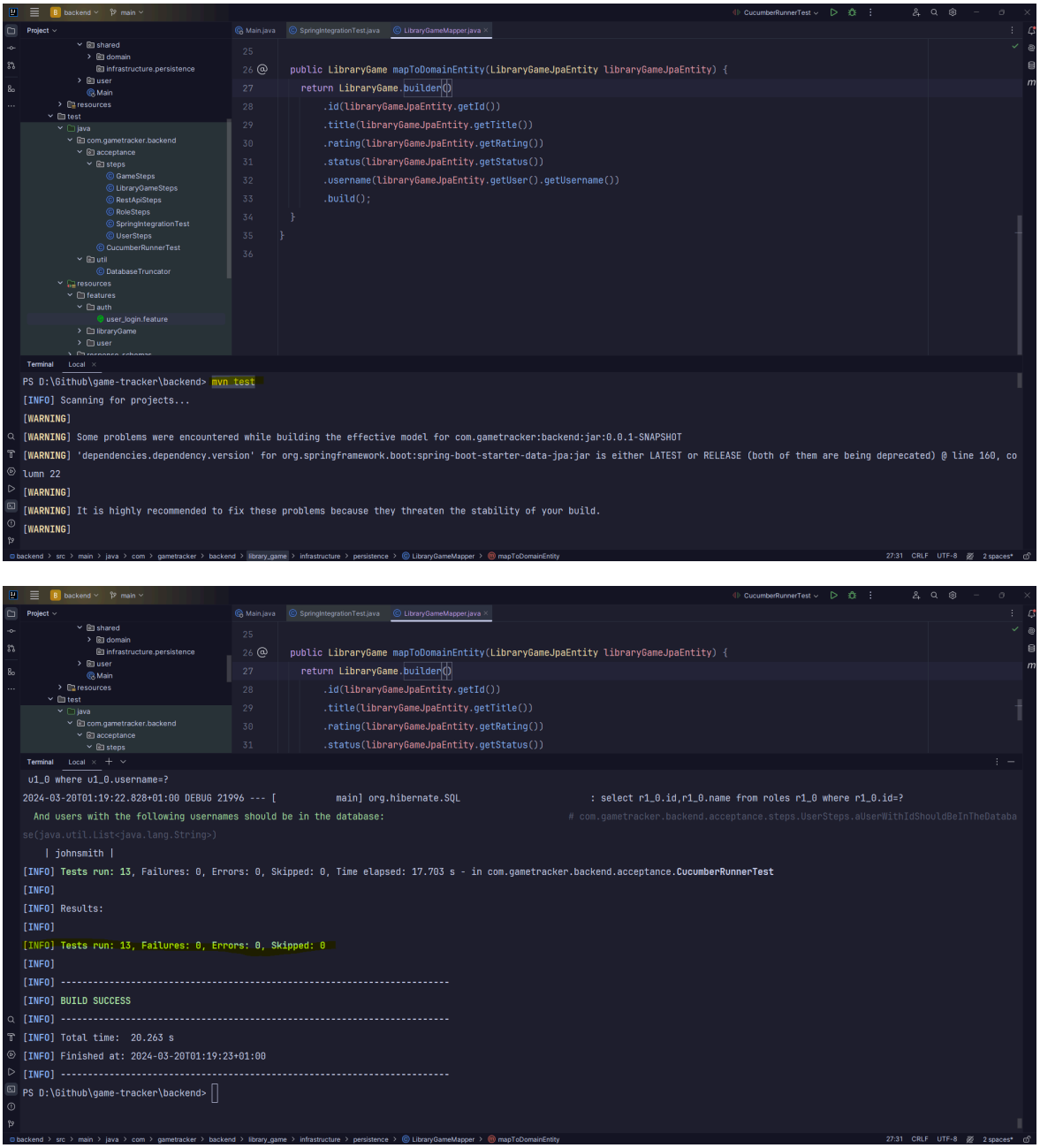


Fig. 22. Mostra d'execució dels test usant maven

3. Executant la classe CucumberRunnerTest. També es poden executar tots a l'hora utilitzant la classe CucumberRunnerTest navegant fins a ella i executant-la.

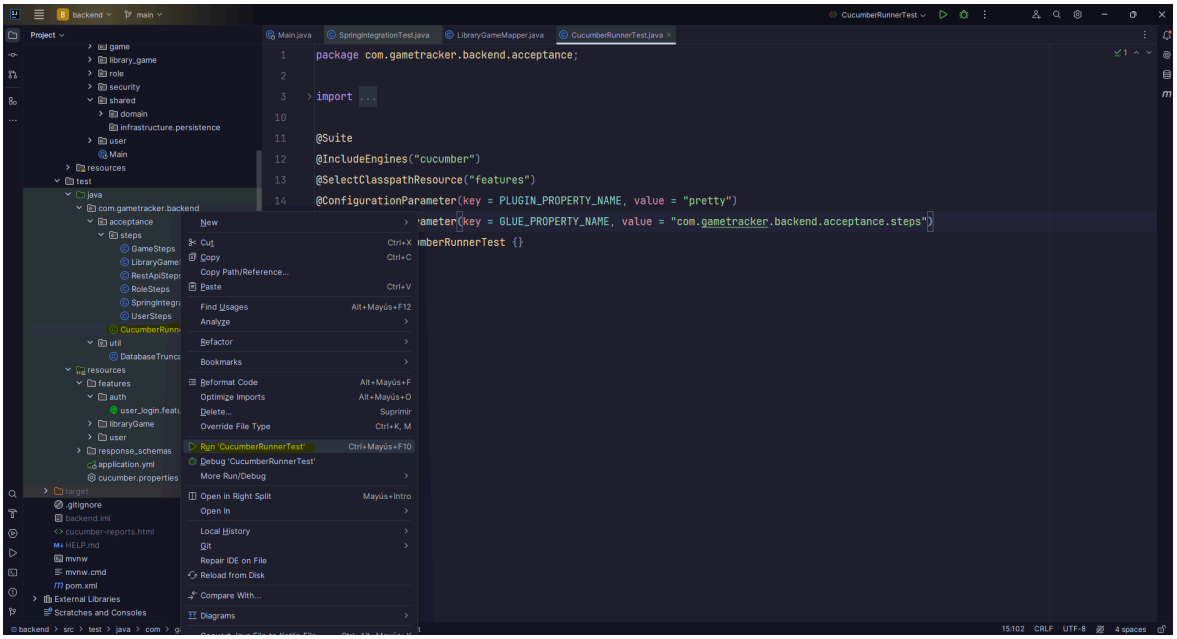


Fig. 23. Mostra d'execució dels test usant la classe CucumberRunnerTest

10. Bibliografia

[0] “Software Development process - Wikipedia” https://en.wikipedia.org/wiki/Software_development_process (accessed Jan. 25, 2023)

[1] “Software Engineering | Failure of Waterfall model - GeeksforGeeks” https://www.geeksforgeeks.org/software-engineering-failure-of-waterfall-model/ (accessed Feb. 7, 2023)

[2] “Unified Modeling Language - Wikipedia” https://en.wikipedia.org/wiki/Unified_Modeling_Language (accessed Feb. 7, 2023)

[3] “Rational Unified Process - Wikipedia” https://en.wikipedia.org/wiki/Rational_Unified_Process (accessed Feb. 8, 2023)

[4] “ExtremeProgramming” https://martinfowler.com/bliki/ExtremeProgramming.html (accessed Jan. 28, 2023)

[5] “TestDrivenDevelopment” https://martinfowler.com/bliki/TestDrivenDevelopment.html (accessed Jan. 28, 2023)

[6] J. F. Smart and J. Molak “Introducing Behavior Driven Development” in *BDD in action*, second edition, Manning Publications, 2023, ch. 2.1

[7] “BDD versus TDD – Understand the difference | Cucumber Blog” https://cucumber.io/blog/bdd/bdd-vs-tdd/ (accessed Feb. 7, 2023)

[8] “History of BDD - Cucumber Documentation” https://cucumber.io/docs/bdd/history/ (accessed Jan. 25, 2023)

[9] “Sueldo: Junior Software Developer (Febrero, 2023) | Glassdoor” https://www.glassdoor.es/Sueldos/junior-software-developer-sueldo-SRCH_KO0,25.htm (accessed Feb. 8, 2023)

[10] “Seguridad Social: Cotización / Recaudación de Trabajadores” https://www.seg-social.es/wps/portal/wss/internet/Trabajadores/CotizacionRecaudacionTrabajadores/36537 (accessed Feb. 8, 2023)

[11] “Horas laborales al año pactadas en convenios 2006-2021 | Statista” https://es.statista.com/estadisticas/478441/promedio-de-horas-de-trabajo-al-ano-segun-convenios-colectivos-de-espana/#:~:text=El%20promedio%20de%20horas%20laborales,fue%20de%20aproximadamente%201.725%20horas. (accessed Feb. 9, 2023)

[12] “Environmental Sustainability | Microsoft CSR” https://www.microsoft.com/en-us/corporate-responsibility/sustainability (accessed Feb. 10, 2023)

[13] “¿Qué es la huella de carbono digital? - Negocios Sostenibles” https://negociosostenible.camaravalencia.com/social/tendencias/que-es-la-huella-de-carbono-digital/#:~:text=La%20huella%20de%20carbono%20digital%20es%20producida%20por%20los%20gases,%2C%20empresas%2C%20mensajes%2C%20plataformas%E2%80%A6 (accessed Feb. 10, 2023)