PRÁCTICA 6

Calidad del Software

The Video Game Box

INTEGRANTES DEL GRUPO:

MARIO CAMPOS SOBRINO
CARLOS CARNERO MÉRIDA
JOSÉ DÍAZ REVIEJO
DAVID ELÍAS PIÑEIRO
CARLOS GÓMEZ LÓPEZ
ÁLVARO GÓMEZ SITTIMA
JULIÁN MOFFATT
JUAN ROMO IRIBARREN
JAVIER DE VICENTE VÁZQUEZ
GONZALO VÍLCHEZ RODRÍGUEZ

0. Índice

1. Plan de pruebas

En cuanto a las pruebas para testear y validar el software que vamos a realizar las vamos a dividir en diferentes módulos en función a que están enfocadas. Todas las pruebas que se van a explicar a continuación una vez realizadas serán validadas por un miembro del equipo de desarrollo independiente a la funcionalidad testada.

1.1. Pruebas realizadas

Empezando por las pruebas de *unidad* que tras probar diferentes herramientas nos hemos decantado por realizarlas usando *JUnit* en su versión 4. Estas pruebas las realizará la persona perteneciente al equipo de desarrollo que se ha encargado de codificar la funcionalidad que será testada. Estas pruebas a su vez se dividirán en dos capas, una que testeará la capa de *datos* que se encarga de la conexión con la base de datos (BD) y se comprobará el acceso a la misma. Por otro lado, en la capa de *lógica* valga la redundancia se testeará la lógica de la aplicación.

Una vez se hayan realizado las pruebas unitarias se harán las de *integración* las cuales se centran en validar el funcionamiento de los distintos módulos de la aplicación como grupo y que tratan de identificar los problemas que surgen cuando los distintos componentes se vinculan y combinan entre sí. Estas pruebas las realizarán uno o varios miembros del equipo de desarrollo cuando la codificación y los test unitarios de todos los módulos que van a ser testeados hayan sido completados, el o los encargados de hacerlos serán asignados en función a la carga de trabajo que tengan en ese momento. El método elegido para realizar estas pruebas es el centrado en funciones ya que se centra en testear los componentes necesarios por la función correspondiente de la aplicación y usaremos *JUnit 4* para automatizarlas.

Una vez todos los módulos de la aplicación estén completados y testeados se harán las pruebas de *sistema* que se encargan de verificar si la aplicación cumple con los requisitos marcados, tanto en cuanto a requisitos funcionales como no funcionales. Estas pruebas las realizará el equipo de desarrollo al completo una vez la codificación y testeo de toda la aplicación este completado. Las pruebas de *funcionalidad* se harán de forma manual usando la propia *UI*, las de carga no se harán ya que usamos una base de datos gratuita con un tamaño limitado por lo que sabemos que sobrecargar el sistema es bastante fácil. Las de *estrés* tampoco se realizarán ya que al igual que en la anterior nuestra *BD* no es capaz de soportar un gran número de usuarios concurrentes y por último las de usabilidad las realizará un miembro independiente al equipo de desarrollo de forma manual ya que al no pertenecer al equipo tendrá una concepción distinta de la aplicación ya que la desconoce.

Por último las pruebas de *aceptación* validarán la aplicación desde el punto de vista de un usuario final, las realizará el equipo de desarrollo una vez las pruebas unitarias y de integración estén completas.

Además, este tipo de pruebas son de *caja negra*, por lo que el sistema será probado con un nivel de abstracción alto, sin tener que conocer ningún detalle de la estructura, el diseño o la implementación interna.

Se harán de forma automática y manual, de forma automática usando *JBehave*, consideramos también el uso de *Cucumber* pero al final nos decantamos por la anterior ya que nos parece que se adapta más a nuestra aplicación, una vez estas pruebas estén completas el *Product Owner* validará la aplicación de forma manual probando todas las funcionalidades y verificando si cumple con sus requisitos.

Para entender el funcionamiento de estas pruebas, se debe contemplar que prueban las funcionalidades de la aplicación desde el punto de vista del usuario final, validando que el software hace lo que debe.

Aunque las pruebas son automáticas, los casos de prueba están creados manualmente. Para los casos de prueba, el formato utilizado por JBehave es el llamado *BDD-format* (Dado/Cuando/Para), con la sintaxis indicada en la *Tabla 1.1.1*.

Dado	Precondición
Y	Otra precondición
Cuando	Comportamiento que se desea testear
Y	Más comportamientos por testear
Entonces	Resultado de los comportamientos testeados

Tabla 1.1.1 BDD-format

Este formato ayudará a que todo el equipo sea capaz de entender muy fácilmente en que consiste cada prueba de aceptación.

En *JBehave*, los casos de prueba se denominan *scenarios*, por lo que nos referimos a ellos así a partir de ahora. El entorno donde se escriben estos scenarios es en los llamados *story-files* que son una serie de ficheros, en el que cada uno recoge todos los tests para una sola prueba. Los *story files* estarán localizados en la carpeta *resources* del repositorio.

En el caso de que las pruebas sean manuales, el *Product Owner* comprobará para las *Historias de Usuario* que dependan de alguna entrada de datos del usuario, se explorarán las restricciones que se reflejan en los *criterios de aceptación*: introducir más caracteres de los aceptados, crear un usuario que ya existe o adjuntar una imagen de un formato distinto al soportado, etc. También se probarán los caminos que llevan a una ejecución correcta del programa según los criterios de aceptación, para asegurarnos que la necesidad que motivó a crear la historia de usuario está completamente aprobada.

Por otro lado, se comprobará a lo largo de las pruebas de aceptación, mientras se ejecutan las distintas tareas de la aplicación, si el rendimiento del sistema está a la altura de un software de calidad aceptable. Sobre todo las tareas que manejan grandes cantidades de datos o realizan operaciones complejas para mostrarlos: accesos a base de datos, algoritmos de ordenación de datos, etc.

1.2. Prueba piloto

En cuanto a las pruebas unitarias, realizadas con *JUnit 4*, se han realizado de manera automatizada, verificando la funcionalidad de una unidad de software. En este caso, esa unidad de software se trata del método *add* de la clase *PersonFunctions* (*Imagen 1.2.1*), que añade una nueva persona a la aplicación.

```
public class PersonFunctions {

    public ObjectId add(TPerson tPerson) {
        if (tPerson.getNif().length() != 9)
            return null;
        else if (!tPerson.getNif().substring(0, 8).matches("^[0-9]+$"))
            return null;
        else if (!Character.isAlphabetic(tPerson.getNif().charAt(8)))
            return null;

        PersonData personData = new PersonData();
        ObjectId id = personData.add(tPerson);
        return id;
    }
```

Imagen 1.2.1 Método add de PersonFunctions.

Las pruebas unitarias están orientadas a la estructura, el diseño, el código y la implementación interna, es decir, son de caja blanca. Por ello, el objetivo principal es tomar la pieza más pequeña posible de software (en este caso el método *add*) y determinar si se comporta exactamente como esperamos. Además, cada caso de prueba es independiente del resto. En las imágenes se muestran distintos casos de prueba, que han sido obtenido a través de las condiciones dadas por el método add. En la *Imagen 1.2.2*, se muestra la prueba de añadir una persona de manera correcta. En la *Imagen 1.2.3*, es el código de la prueba al intentar añadir una persona que ya ha sido añadida previamente, por lo que para que resulte correcto, esta persona no debería de ser añadida.

```
public class UnitTest {
    private static PersonFunctions persons;
    private static ObjectId idP;

    @BeforeClass
    public static void BeforeClass() {
        persons = new PersonFunctions();
    }

    // Este caso de prueba añade a una persona, con todos los datos correctos.
    @Test
    public void addNewPerson() {
        TPerson tperson = new TPerson();
        tperson.setNif("12345678P");
        tperson.setNombre("Prueba");
        tperson.setApellidos("junit");
        tperson.setActivo(true);

        idP = persons.add(tperson);
        assertNotNull(idP);
    }
```

Imagen 1.2.2 Caso de prueba añadir una persona.

```
// Este caso de prueba intenta añádir a una persona que ya ha sido añadida previamente, por lo que para que sea correcto, debería no añadirlo.
@Test
public void addExistingPerson() {
    TPerson tperson = new TPerson();
    tperson.setNif("12345678P");
    tperson.setNombre("Prueba2");
    tperson.setApellidos("junit");
    tperson.setActivo(true);

    idP = persons.add(tperson);
    ObjectId idExisting;
    idExisting = persons.add(tperson);
    assertNull(idExisting);
}
```

Imagen 1.2.3 Caso de prueba añadir una persona existente.

En las siguientes pruebas se mostrarán las pruebas que tendrían que salir error relacionadas con el DNI. La primera es la *Imagen 1.2.4*, está tratando de añadir una persona con un DNI en el que el número de caracteres es superior a 10, por lo que no debería de ser añadida.

```
// El siguiente caso de prueba tratará de añadir a una persona con un dni en el que el número de caracteres supera 10, por lo que no debería añadirse.
@Test
public void addWrongDNIPerson() {
    TPerson tperson = new TPerson();
    tperson.setNif("123456789P");
    tperson.setNombre("DNIMayor");
    tperson.setApellidos("junit");
    tperson.setActivo(true);

    idP = persons.add(tperson);
    assertNull(idP);
}
```

Imagen 1.2.4 Caso de prueba añadir una persona con el DNI incorrecto 1.

La siguiente prueba es la *Imagen 1.2.5*, en la cual está intentando añadir una persona con un DNI en el que el número de caracteres es menor a 9, por lo que tampoco debería añadirse.

```
// El siguiente caso de prueba tratará de añadir a una persona con un dni en el que el número de caracteres es menor a 9, por lo que no debería añadirse.
@Test
public void addWrongDNI2Person() {
    TPerson tperson = new TPerson();
    tperson.setNif("12345P");
    tperson.setNombre("DNIMenor");
    tperson.setApellidos("junit");
    tperson.setActivo(true);

    idP = persons.add(tperson);
    assertNull(idP);
}
```

Imagen 1.2.5 Caso de prueba añadir una persona con el DNI incorrecto 2.

En las imágenes *Imagen 1.2.6*, *Imagen 1.2.7* e *Imagen 1.2.8* se muestran los casos en los que el DNI sus primeros 8 caracteres no son números, en el segundo que justo el último carácter no es una letra alfabética, y en el último, juntando los anteriores casos.

```
// El siguiente caso de prueba tratará de añadir a una persona con un dni en el que alguno de los 8 primeros caracteres no es un número, por lo que no debería añadirse.
@Test
public void addWrongDNI3Person() {
    TPerson tperson = new TPerson();
    tperson.setNif("123RT678P");
    tperson.setNombre("DNINum");
    tperson.setApellidos("junit");
    tperson.setActivo(true);

    idP = persons.add(tperson);
    assertNull(idP);
}
```

Imagen 1.2.6 Caso de prueba añadir una persona con el DNI incorrecto 3.

```java
// El siguiente caso de prueba tratará de añadir a una persona con un dni en el que el último carácter no es un alfabético (una letra), por lo que no debería añadirse.
@Test
public void addWrongDNI4Person() {
    TPerson tperson = new TPerson();
    tperson.setNif("123456789");
    tperson.setNombre("DNIAlpha");
    tperson.setApellidos("junit");
    tperson.setActivo(true);

    idP = persons.add(tperson);
    assertNull(idP);
}
```

Imagen 1.2.7 Caso de prueba añadir una persona con el DNI incorrecto 4.

```java
// El siguiente caso de prueba tratará de añadir a una persona con un dni en el que el último carácter no es un alfabético (una letra) y además tendrá caracteres entre
// los 8 primeros que no es un número., por lo que no debería añadirse.
    @Test
    public void addWrongDNI5Person() {
        TPerson tperson = new TPerson();
        tperson.setNif("12rt56789");
        tperson.setNombre("DNIWrong");
        tperson.setApellidos("junit");
        tperson.setActivo(true);

        idP = persons.add(tperson);
        assertNull(idP);
    }
```

Imagen 1.2.8 Caso de prueba añadir una persona con el DNI incorrecto 5.

Una vez establecidos los casos de prueba, es importante indicar que, tras la ejecución de un test unitario, es necesario que tanto el entorno como la capa de persistencia queden exactamente igual a como estaban antes de realizar la prueba. Para ello está el método *deletePerson* (*Imagen 1.2.9*), con la etiqueta *@After*, lo que indica que se ejecutará después de la ejecución de los tests. Este método simplemente llama a otro localizado en *PersonData* (*Imagen 1.2.10*), que pasando el id de la persona, elimina físicamente el objeto de la base de datos.

```java
// El siguiente método sirve para situar al entorno exactamente igual a como estaba antes de ejecutar los casos de prueba.
@After
public void deletePerson() {
    persons.deleteFromDataBase(idP);
}
```

Imagen 1.2.9 Método deletePerson en los tests unitarios.

```java
public void deleteFromDataBase(ObjectId id){
    try {
        MongoDatabase db = Connection.getDataBase();

        MongoCollection<Person> collection = db.getCollection("personas", Person.class);

        collection.deleteOne(eq("_id", id));
    } catch (MongoException e) {
    }
}
```

Imagen 1.2.10 Método deleteFromDataBase en PersonData

Para ejecutar los tests, hay que hacer click derecho en la clase *UnitTest* y seleccionar la opción de *Run As 1JUnit Test* (en el caso de *Eclipse*). Ya explicados todos los casos de prueba y documentados, así como explicado su funcionamiento, lo siguiente es mostrar el resultado de estos tests, así como su tiempo de ejecución (*Imagen 1.2.11*).

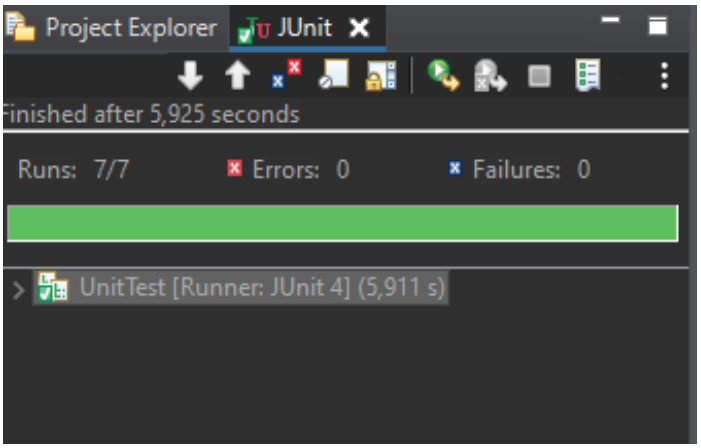


Imagen 1.2.11 Resultado de los tests

Como podemos observar, los 7 casos de prueba han sido ejecutados de manera limpia, sin ningún tipo de error o fallo. Además el tiempo de ejecución es bastante eficiente, pues es de solamente *5,911 segundos*.

En la *Imagen 1.2.11* podemos observar los diversos casos desplegados, en el que cada uno tiene asignado su tiempo de ejecución. Analizando esta muestra, el caso que más tarda es el *addExistingPerson*, lo cual tiene sentido, pues se está llamando al método add 2 veces.

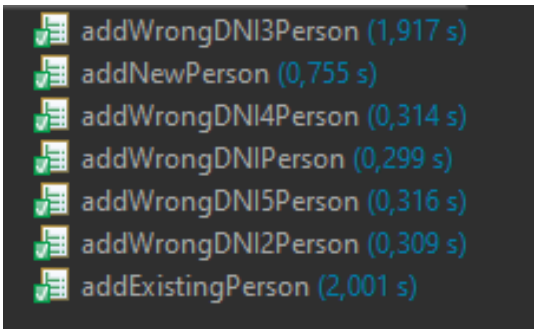


Imagen 1.2.11 Tiempo de ejecución de cada test

2. Integración continua

La *Integración continua* es la práctica de desarrollo software donde los miembros del equipo integran su trabajo frecuentemente, al menos una vez al día. Entendemos por integración la compilación y ejecución de pruebas de todo un proyecto. Cada integración se verifica con una build automática (que incluye la Ejecución de pruebas) para detectar errores de integración tan pronto como sea posible.

Los objetivos clave de la integración continua consisten en encontrar y arreglar errores con mayor rapidez, mejorar la calidad del software y reducir el tiempo que se tarda en validar y publicar nuevas actualizaciones de software.

2.1. Descripción de la herramienta

Para la *Integración continua* hemos contemplado diversas opciones, entre ellas *Jenkins* y *GitHub Actions con Maven*.

Jenkins presenta varias ventajas con respecto a la opción de *GitHub Actions*, como que es una herramienta de software libre desarrollada en *Java*, además de que está diseñada específicamente para gestionar la *Integración continua*. Esta herramienta sin embargo, presenta algunos inconvenientes como que debe ser alojada en un servidor. Esta es la razón principal de porque hemos decidido usar *GitHub Actions*, la cual explicamos a continuación.

GitHub Actions permite establecer un flujo de acciones a realizar cuando ocurre un determinado evento, como un *pull* o un *push*, y filtrar según las ramas. Al utilizarla con *Maven*, la cual es una herramienta que permite administrar la compilación del proyecto, es fácil el establecer que cuando se realice un *push* sobre una rama, se ejecute en una máquina *Ubuntu* las instrucciones de compilación de *Maven* sobre el trabajo de esa rama y así comprobar el resultado de los test de unidad e integración.

Además, al tener nuestro proyecto almacenado en un repositorio de *GitHub* (https://github.com/TheVideoGameBox/TheVideoGameBox.git), resulta mucho más cómodo utilizar éste directamente para realizar las tareas de Integración continua sin depender de herramientas externas.

2.2. Pipeline

Cuando se quiere implantar integración continua, es necesario definir un *Pipeline*. Un *Pipeline* consiste en una serie de fases automatizadas que deben ejecutarse de forma secuencial para distribuir la versión nueva de un sistema software. Si queremos mejorar la velocidad de nuestros despliegues y la calidad de nuestros desarrollos, utilizar *Pipelines* para la integración y entrega continua nos será de gran utilidad.

En nuestro proyecto, el *Pipeline* definido que se seguirá en la integración continua consta de las fases detalladas a continuación.

2.2.1. Fase de compilación

En primer lugar y tras detectar un cambio en el código del repositorio de control de versiones, se activa la herramienta de integración continua que ejecuta su correspondiente *Pipeline*. En esta fase, se combina el código fuente y sus dependencias para construir un ejecutable de nuestro producto.

No pasar la fase de compilación es un indicador de un problema fundamental en la configuración del proyecto, y es mejor abordarlo de inmediato.

2.2.2. Fase de tests

En esta fase, ejecutamos las pruebas automatizadas para validar la corrección de nuestro código y el comportamiento de nuestro producto. La etapa de tests actúa como una red de seguridad que evita que los errores fácilmente reproducibles lleguen a los usuarios finales.

2.2.3. Fase de lanzamiento o despliegue

Tras haber superado la fase de tests, estamos listos para trasladar la aplicación al repositorio.

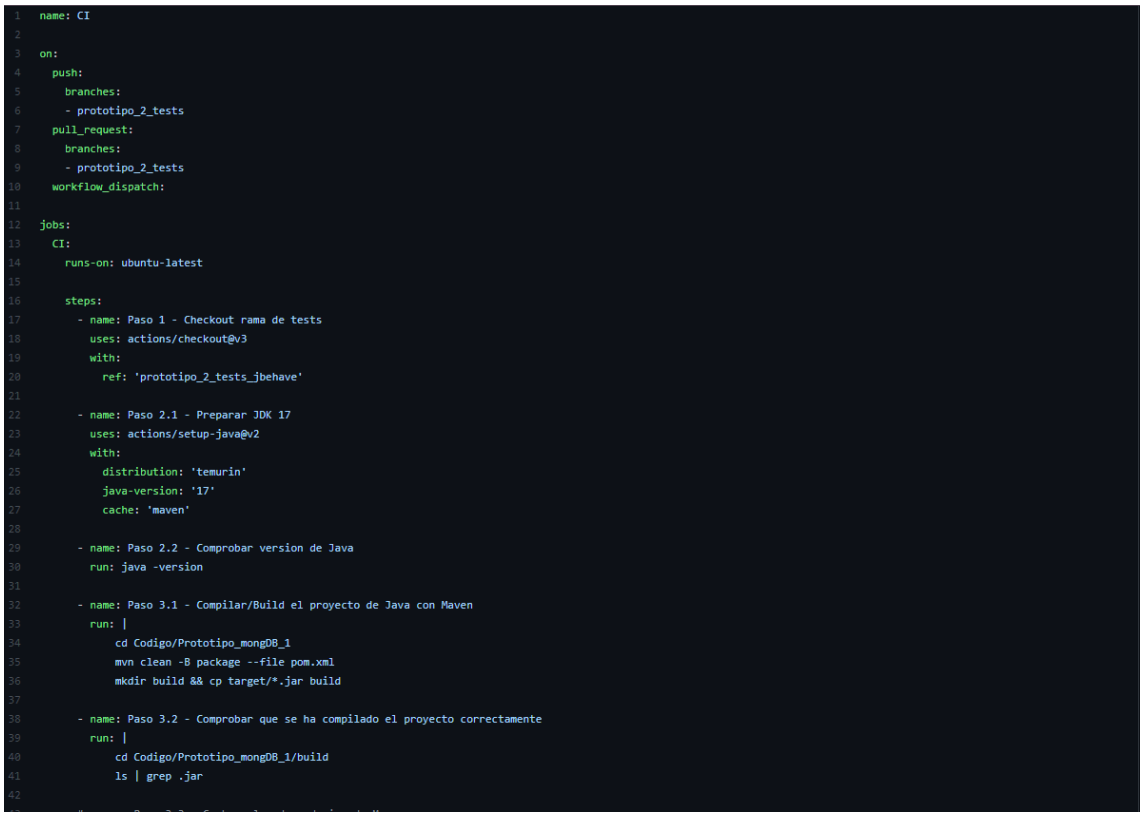
2.3. Prueba piloto

Utilizando la herramienta *Actions* integrada en *GitHub* hemos llevado a cabo una prueba piloto para comprobar que el *Pipeline* de *Integración continua*, que hemos definido en el punto anterior, se puede ejecutar correctamente de forma automática en nuestro sistema de control de versiones (GitHub) siempre que se sube un cambio a la rama de las pruebas.

Para ello hemos desarrollado un código que se encuentra en las imágenes *Imagen 2.3.1* e *Imagen 2.3.2*, el cual realiza un trabajo dividido en distintos pasos haciendo referencia a las fases del *Pipeline*. Empezamos inicializando la imagen del sistema operativo donde se van a ejecutar los distintos pasos del *Pipeline*. En esta inicialización hacemos *Checkout* de la rama de pruebas, preparamos el entorno de desarrollo de *Java* con la versión 17, y preparamos el caché para un proyecto *Maven*, de esta manera las dependencias generadas al compilar el proyecto en el siguiente paso se tendrán en cuenta en las próximas veces que se ejecute este código.

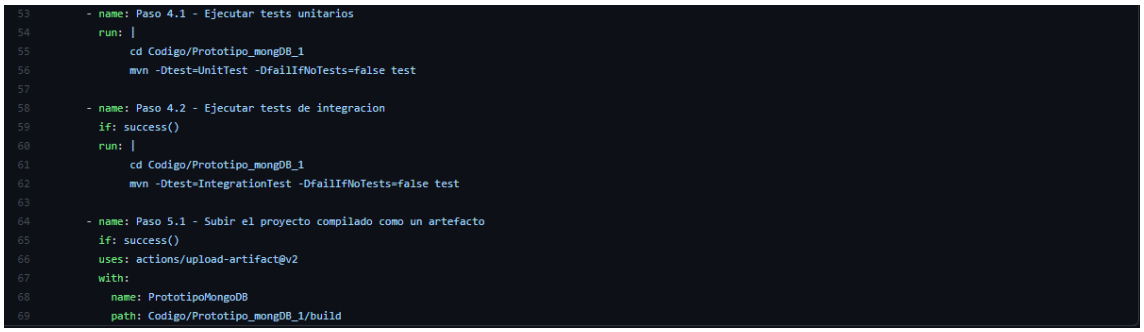
El siguiente paso realizado, es compilar el proyecto de *Maven* y generar los *archivos JAR*. A continuación, se ejecutan los *JUnit* y los *test de integración*, para asegurarnos que los nuevos cambios no afecten al estado del proyecto y los test siguen siendo válidos. Mostrándose el resultado de los JUnit en la *Imagen 2.3.3* y de los test de integración en la *Imagen 2.3.4* (el prototipo no tiene test de integración).

Por último, se crea un *artefacto* a partir de los ejecutables generados en la fase de compilación y se da acceso a éste a través de la herramienta de *Actions*. En la *Imagen 2.3.5* se muestra el resultado de todo el proceso realizado.



```yaml
name: CI

on:
  push:
    branches:
    - prototipo_2_tests
  pull_request:
    branches:
    - prototipo_2_tests
  workflow_dispatch:

jobs:
  CI:
    runs-on: ubuntu-latest

    steps:
      - name: Paso 1 - Checkout rama de tests
        uses: actions/checkout@v3
        with:
          ref: 'prototipo_2_tests_jbehave'

      - name: Paso 2.1 - Preparar JDK 17
        uses: actions/setup-java@v2
        with:
          distribution: 'temurin'
          java-version: '17'
          cache: 'maven'

      - name: Paso 2.2 - Comprobar version de Java
        run: java -version

      - name: Paso 3.1 - Compilar/Build el proyecto de Java con Maven
        run: |
            cd Codigo/Prototipo_mongDB_1
            mvn clean -B package --file pom.xml
            mkdir build && cp target/*.jar build

      - name: Paso 3.2 - Comprobar que se ha compilado el proyecto correctamente
        run: |
            cd Codigo/Prototipo_mongDB_1/build
            ls | grep .jar
```

Imagen 2.3.1 Primera parte del código



```yaml
      - name: Paso 4.1 - Ejecutar tests unitarios
        run: |
            cd Codigo/Prototipo_mongDB_1
            mvn -Dtest=UnitTest -DfailIfNoTests=false test

      - name: Paso 4.2 - Ejecutar tests de integracion
        if: success()
        run: |
            cd Codigo/Prototipo_mongDB_1
            mvn -Dtest=IntegrationTest -DfailIfNoTests=false test

      - name: Paso 5.1 - Subir el proyecto compilado como un artefacto
        if: success()
        uses: actions/upload-artifact@v2
        with:
          name: PrototipoMongoDB
          path: Codigo/Prototipo_mongDB_1/build
```

Imagen 2.3.2 Segunda parte del código

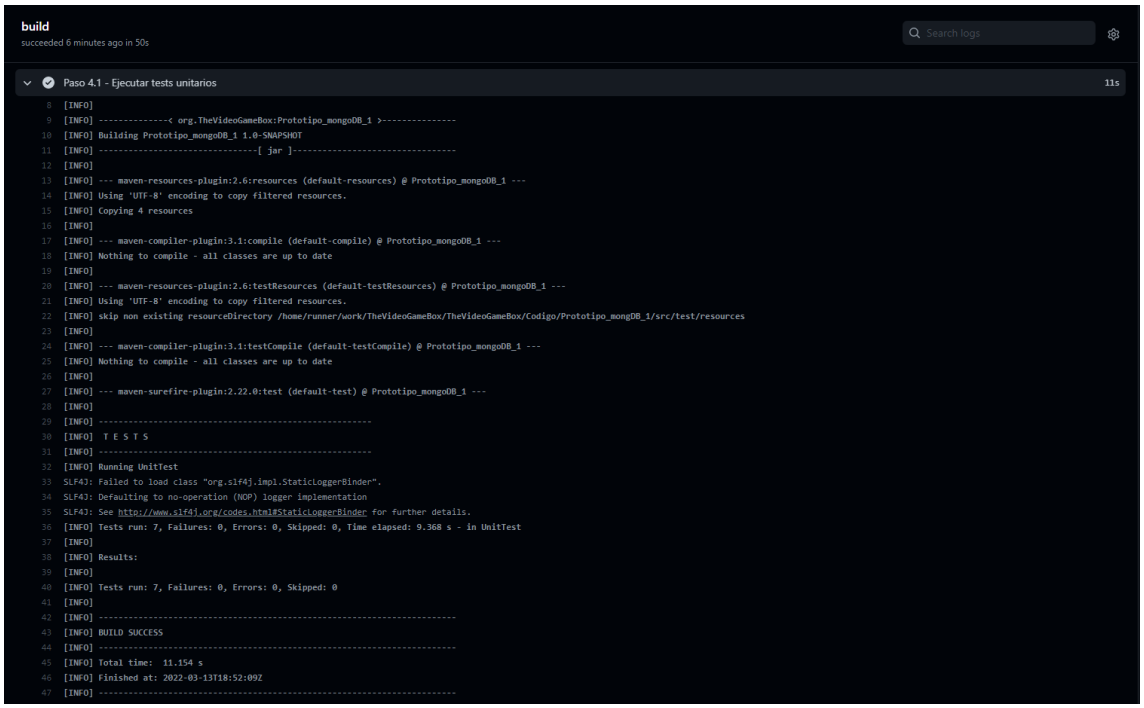


Imagen 2.3.3 Resultados de los JUnit

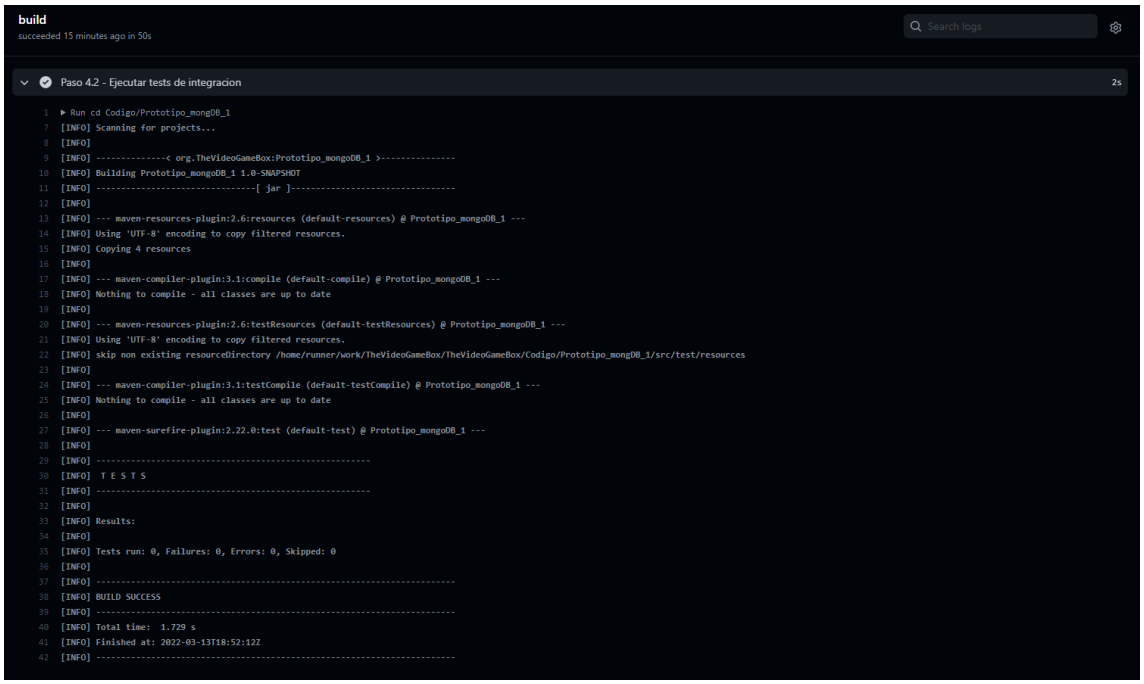


Imagen 2.3.4 Resultados de los test de integración

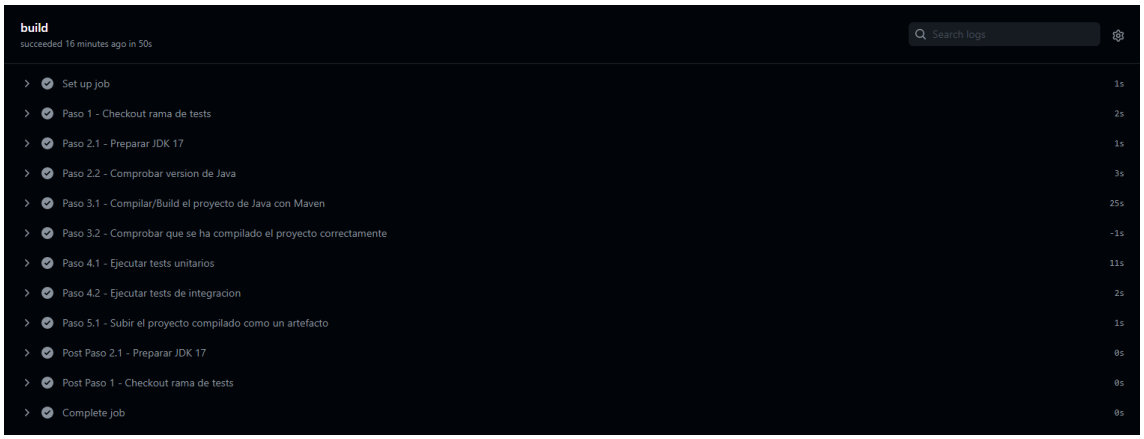


Imagen 2.3.5 Resultado final

3. Definición de Hecho

La *Definition of Done* (*DoD*) es un conjunto de reglas que determinan cuándo un elemento está terminado. Se entiende por elemento terminado cuando éste está listo para poner en producción a disposición del usuario. Eso sí, la decisión de *subir* la toma el *Product Owner*. Por tanto, se puede aplicar a nivel de ítem, de categoría, o release o de *Sprint*.

En nuestro caso, el *DoD* incluye una serie de aspectos explicado cada uno de ellos a continuación.

3.1. Todas las tareas asociadas están hechas

Consideramos que este es el primer punto para tener en cuenta, puesto que si esto falla el proyecto no podrá entregarse o se entregará incompleto, lo cual nos perjudicaría gravemente. Por ello, prestamos especial atención a la hora de dividir las tareas y nos aseguramos de que cada miembro completa todo lo que se le ha asignado.

3.2. Incidencias detectadas en fase de desarrollo resueltas

Durante la fase de desarrollo iremos anotando los diferentes problemas que nos han surgido a la hora de realizar las distintas tareas. La idea principal es evitar que estos vuelvan a convertirse en un problema en el futuro, a la vez que nos aseguramos un guion a revisar para que no se nos olvide solucionar ninguno. Consideramos imprescindible revisar dicho guion para considerar que un elemento está terminado.

3.3. Código documentado y/o comentado

Como en todo proyecto que implique escribir código (programas), consideramos que es importante documentarlo añadiendo suficiente información como para explicar lo que hace, punto por punto, de forma que no sólo los ordenadores y los propios programadores sepan qué hace, sino que además cualquier persona entienda qué está haciendo y por qué.

En nuestro caso consideramos importante prestar especial atención a que todo el código esté comentado y documentado para que en caso de que el *Product Owner* o cualquier persona, que no pertenezca al *Equipo de desarrollo*, pueda entender el código de manera sencilla a pesar de ser la primera vez que lo vea. De esta forma nos aseguramos también de que, si algún miembro del equipo de desarrollo abandona el proyecto, tanto el resto del equipo como su posible sustituto tendrá todo lo que éste hizo bien detallado para su mayor comprensión.

3.4. Documentación hecha

A la hora de presentar un proyecto pensamos que no solo es importante cumplir con todas las tareas que se marcan, si no también ir documentando todo para que a la hora de ser visto por el usuario entienda perfectamente todo el proceso. Por ello, consideramos importante el llevar al día la documentación de cada entrega por cada paso realizado en el proyecto, y también revisamos cuidadosamente que no quede nada sin explicar.

3.5. El trabajo de cada miembro del equipo ha sido revisado por al menos otro miembro del equipo

Todo el trabajo realizado individualmente por cada miembro del grupo es siempre revisado por otro distinto. De esta manera, se ayuda a mantener la calidad del trabajo a lo largo del proyecto. La manera de corregir es simple y respetuosa, un miembro del equipo revisa el trabajo de otro y siempre con la intención de mejorar el proyecto, informará algunas posibles mejoras al otro miembro.

3.6. Pruebas unitarias y de integración superadas

Para avanzar con esta práctica, sometemos a nuestro a unas pruebas de unidad y de integración. Las primeras, prueban cada unidad del software independientemente, y las segundas, agrupaciones de unidades software. Nuestro proyecto ha de pasar todas estas pruebas para considerarse hecho.

El *Equipo de desarrollo* será el encargado de diseñar y pasar estas pruebas.

3.7. Superar los criterios de aceptación o pasar todas las pruebas de aceptación

Para considerar el trabajo como hecho, nuestro proyecto deberá pasar unas pruebas de aceptación, en las cuales se prueba el sistema completo en el entorno real de trabajo. Estas pruebas las realizarán y diseñarán el *Equipo de desarrollo*.

3.8. Pasar revisión de implementación del diseño

Como en todo, deberemos revisar la implementación del diseño del proyecto. Con esta revisión conseguiremos un avance correcto y evitaremos errores e inconvenientes más adelante. Ayudando así a mantener la calidad de trabajo en nuestro proyecto.

3.9. Pasar pruebas automatizadas

Una historia de usuario se considerará hecha cuando haya pasado todas las pruebas automatizadas diseñadas para esa historia, además de superar las pruebas automatizadas de las historias relacionadas con la nueva historia de usuario.

El *Equipo de desarrollo* será el encargado de diseñar y pasar estas pruebas.

3.10. Hacer pull-request a la rama dev y que éste sea aprobado

Al hacer pull-request a la *rama dev*, el resto del equipo de desarrollo que no ha realizado la *HU* va a revisar el código que se quiere llevar a esa rama, si el resto del equipo aprueba dicho código entonces se subirá a la *rama dev* para seguir con las pruebas y finalmente presentarlo al *Product Owner*.

3.11. Aprobada por el Product Owner

Como *Product Owner* apruebo una historia de usuario cuando ésta satisface las necesidades funcionales del consumidor que motivaron su creación. La historia de usuario deberá esta implementada correctamente siguiendo los criterios de aceptación de tal manera que el usuario no pueda con un mal uso hacer cosas que no están contempladas. La interfaz deberá ser clara y sencilla para facilitar al usuario la experiencia de uso.

3.12. El producto sigue los estándares de calidad interna y ha sido refactorizado para conseguir mantenibilidad

El equipo de desarrollo ha creado unos estándares de calidad basados en su experiencia en asignaturas como *Tecnologías de la programación*, *Ingeniería del software* y *Modelado del software*. Estos conocimientos previos ayudados de los profesores en las horas de laboratorio son los que determinan los estándares de calidad, los cuales tenemos previsto que vayan evolucionando a medida que se desarrolla el proyecto. A medida que se terminan las historias de usuario el código será revisado por el *Equipo de desarrollo* para asegurarse de que cumple con la calidad y mantenibilidad esperada.

4. Product Backlog

El *Product Backlog* es un documento vivo y cambiante que hemos adaptado con respecto a la última versión según las indicaciones de la corrección y nuevos detalles que han surgido en el diálogo del equipo.

Para ello hemos empezado realizando cambios en los *criterios de aceptación* asegurándonos de que todos siguen el esquema *cuando, si, entonces*. Así, todos los criterios de aceptación tienen el mismo formato, haciendo las historias de usuario más sencillas de visualizar y entender. También hemos agregado en algunas historias de usuario limitaciones en cuanto al número de caracteres que se pueden introducir para realizar una búsqueda o crear un nombre de usuario, y se ha especificado cómo deben rellenarse ciertos campos, describiendo cuando será correcto y cuando dará error según la forma en la que haya sido rellenada.

Por último, hemos modificado el *Product Backlog* traspasando historias de usuario desde el *MVP* hasta la *segunda release*. Siguiendo las indicaciones de la corrección, hemos decidido quitar del MVP las historias de usuario *Búsqueda de usuarios por nombre* y *Ver atributos básicos de un usuario*. Estas dos historias de usuario sobrecargaron el *MVP* y podían provocar un exceso de trabajo para el primer *sprint*, no siendo esenciales para un producto mínimo viable.

5. Actividad grupal

En esta práctica hemos decidido llevar a cabo una dinámica grupal, la cual consiste en un *Kahoot!* de *JUnit*, en el que se realizaron 18 preguntas generales sobre la herramienta con dos objetivos en mente, averiguar cuanto sabía cada integrante de la herramienta y obtener conocimientos nuevos sobre ella si se llegaban a desconocer.

Para desarrollar esta dinámica nos reunimos todos en la plataforma de *Discord*, para comentar cada una de las preguntas y justificar porque eran algunas las respuestas. Antes de realizar el Kahoot! tanto Carlos G. como Carlos C. se encargaron de crearlo, introduciendo preguntas con sus respectivas respuestas habiendo investigado antes para poder desarrollarlas. Una vez el Kahoot! estaba creado, Carlos G. le dio comienzo, habiendo preguntas variadas tanto generales sobre la herramienta, como funcionalidades de esta. En general se mostraron los conocimientos ya adquiridos en otras asignaturas como *Ingeniería del Software* o *Modelado del Software* en las cuales ya se llegaron a utilizar *JUnit*. Todo el desarrollo de la actividad está registrado en el *Vídeo 5.1*.

Vídeo 5.1 Kahoot! sobre Junit

Como resultado obtenido se puede decir que se ha logrado el objetivo esperado, ya que como equipo nos hemos ayudado unos a otros a entender algunas respuestas y se ha mostrado que la gran mayoría tienen amplios conocimientos sobre la herramienta.

Esta actividad grupal fue realizada antes de empezar las pruebas lo cual proporcionó al equipo no solo refrescar la información básica de *JUnit* sino que también ayudo a comprender nuevos conocimientos.