Práctica 4

Tecnologías para el Desarrollo de Software
Grado en Ingeniería Informática
Mención Ingeniería del Software

Y. Crespo

Curso 2019-2020

La realización de esta práctica se hace en sustitución del examen de convocatoria ordinaria. Equivale al 25 % de la asignatura. La práctica tiene carácter **individual**.

Tarea a realizar

El objetivo de la práctica es realizar el ciclo TDD (Rojo-Verde-Refactor) completo para el supuesto que se describe y de acuerdo a las instrucciones que se dan a continuación.

Desiderata

En una ciudad se va a implantar un servicio de metro sencillo.

En el primer sprint del desarrollo del software de información sobre el la red de metro se ha decidido incluir las siguientes necesidades:

La red de metro está formada por varias líneas. Una red de metro para serlo deberá tener al menos dos líneas.

Se desea poder obtener, a partir de la red de metro, una línea dado su número o color.

Se desea poder añadir líneas a la red de metro y retirar líneas del servicio.

Una línea está unívocamente identificada en la red de metro por un número consecutivo y un color que no puede coincidir con colores de otras líneas esa red de metro.

Se desea obtener un array con todas las líneas que forman la red.

En la red de metro dado el nombre de una estación deberá proporcionarse información sobre las líneas que pasan por dicha estación.

En la red de metro dado el nombre de una estación deberá proporcionarse todas las líneas que pasan por dicha estación.

Se desea que se pueda consultar si una línea tiene correspondencia con otra línea y conocer en qué estación o estaciones (si hay más de una) se produce dicha correspondencia.

Se desea saber si hay conexión entre dos estaciones dadas en una misma línea (sin trasbordo). En caso de haber conexión sin trasbordo se desea saber la línea que contiene ambas estaciones.

Se desea saber si hay conexión entre dos estaciones dadas (con trasbordo). En caso de haber conexión con trasbordo se desea saber las líneas que forman parte del trayecto en un array, con las líneas ordenadas. La primera línea en el array contiene la estación de partida, la última línea en el array contiene la estación de destino.

Se desea que se pueda consultar si una línea tiene correspondencia con otra línea y conocer en qué estación o estaciones (si hay más de una) se produce dicha correspondencia.

Dada una coordenada GPS se desea saber si hay una estación cercana, indicando la distancia máxima para establecer la cercanía.

Se ha decidido obtener toda la información sobre la red de metro (todas sus líneas en formato json) y recíprocamente, dado un json que contiene información sobre las líneas de una red, construir el objeto que representa dicha red.

Como punto de partida, en

- `https://gitlab.inf.uva.es/yania/lineasyestacionesmetro/raw/master/es.uva.inf.maps/CoordenadasGPS.java`,
- `https://gitlab.inf.uva.es/yania/lineasyestacionesmetro/raw/master/es.uva.inf.tds.redmetro/Estacion.java`
- `https://gitlab.inf.uva.es/yania/lineasyestacionesmetro/raw/master/es.uva.inf.tds.redmetro/Linea.java`

se encuentran los stubs de las clases Línea, Estación y CoordenadaGPS que se han obtenido como resultado de un proceso TDD realizado por otros miembros de la organización. En dicho diseño se tiene que una línea tiene al menos una estación inicial y una estación final (se trata de una línea simple, no es circular y no tiene bifurcaciones en su trayectoria) y estaciones intermedias.

Las estaciones están unívocamente identificadas por un nombre en la línea. Sin embargo, en la red de metro esta unicidad tendrá alcance no solamente en la línea sino también en toda la red, si bien en dos redes de metro diferentes puede haber estaciones con igual nombre. Cada estación tiene asociada varias coordenadas GPS correspondientes a las entradas y salidas, tienen la particularidad de estar todas en un radio de 200m.

Control de versiones

Se llevará un control de versiones utilizando `git` y se utilizará `gitLab` (`gitlab.inf.uva.es`) como repositorio centralizado y `Atlassian Bitbucket` como repositorio de respaldo. El repositorio en `gitLab` deberá nombrarse con el login de quien realiza el proyecto en un **grupo** creado para ello llamado practica4-tds-1920.

Al finalizar, para realizar la entrega, se añadirá a la profesora al proyecto (cuando ya no se vayan a realizar más *commits* y *pushes* al repositorio centralizado en `gitLab` y al respaldo en `Atlassian Bitbucket`) tal y como se indica en las **normas de entrega** (ver página 7). En el repositorio no se tendrá ningún `.class` o `.jar` ni tampoco la documentación de las clases que puede generarse con *javadoc* en cualquier momento.

En esta práctica **SÍ** se valorará el uso de `git` en cuanto a ramificación (*branchs*) y fusión (*merge*). Las ramas del proyecto incluirán: la rama *master*, una rama *develop* y se tendrá una **rama por tarea (*issue*)**.

Las integraciones en *master* deberán hacerse solamente de partes funcionales testadas, con los tests en verde.

Se utilizarán pruebas en aislamiento para poder probar la implementación realizada en la fase green, tenieno en cuenta que se aportan los stubs de las clases que ha diseñado otro "equipo" y que aún no están implementadas.

Integración Continua

El proyecto estará gestionado automáticamente mediante ant y maven.

En el repositorio gitLab no se tendrá ningún .class o .jar pero deberá bastar hacer un *pull* o *clone* del repositorio y utilizar el script ant (build.xml) para compilar, ejecutar, ejecutar los tests, analizar la cobertura, etc. Por ello, se gestionarán las dependencias externas del proyecto mediante maven. El script ant básicamente se utilizará para delegar en maven y así no tener que memorizar las tareas y parámetros maven necesarios para realizar el objetivo.

El proyecto deberá responder a un arquetipo maven descrito con el pom.xml correspondiente. Estará basado en el arquetipo maven-archetype-quickstart. La versión básica del pom.xml obtenida a partir del arquetipo indicado se debe revisar y modificar para añadir la gestión de dependencias, actualizar versiones así como la configuración de plugins necesarios para el análisis de cobertura, el análisis de calidad y su generación de informes y cualquier otro que los autores necesiten.

Para comenzar el trabajo, se recomienda realizar un proceso en el que se crea el proyecto maven en un repositorio git iniciado y se importa en Eclipse como "Eclipse Maven Project". Se recuerda la recomendación de que el repositorio git no debe residir dentro del Eclipse workspace.

El script ant (build.xml) deberá contar al menos con los siguientes objetivos que tendrán que llamarse exactamente así:

compilar garantiza la obtención de todas las dependencias y genera los .class a partir de los fuentes (depende de limpiar). Este objetivo es el default.

ejecutarTestsTDDyCajaNegra se ejecutan los tests categorizados como TDD y caja negra sin incluir secuencia.

ejecutarPruebasSecuencia se ejecutan solamente las pruebas de secuencia pruebas de secuencia.

ejecutaCajaBlanca se ejecutan solamente las pruebas categorizadas de caja blanca que se añadieron para aumentar la cobertura

ejecutarTodoSinAislamiento incluirá la ejecución de los tres objetivos anteriores.

ejecutarTodosEnAislamiento ejecuta todos los tests de boletín que se tengan basados en mocks.

obtenerInformeCobertura obtener los informes de cobertura de sentencia, de rama, decisión/condición obtenidos con el total de todos los tests realizados (sin incluir los tests en aislamiento).

documentar genera la documentación del proyecto, incluida la carpeta doc con la documentación autocontenida en los fuentes mediante javadoc. La carpeta doc no debe alojarse nunca en el control de versiones.

limpiar deja el proyecto limpio eliminando todo lo generado con las ejecuciones de los objetivos.

calidad realiza un análisis de la calidad del proyecto utilizando sonarqube en modo *publish*.

Como se ha mencionado antes, para la consecución de estos objetivos será recomendable descansar en llamadas a maven.

En gitLab se utilizará el mecanismo de integración continua basado en sintaxis yaml para lo que se podrá utilizar como base el archivo .gitlab-ci.yml que se muestra en el snippet `https://gitlab.inf.uva.es/snippets/25`.

Para usar sonarqube se podrá usar el lint como plugin en Eclipse. En Eclipse ir al menú Help − > Eclipse Marketplace. Buscamos Sonarlint y damos a instalar. Una vez instalado, si dais click derecho en el proyecto en Eclipse veréis una opción de Menú llamada SonarLint. Hay que hacer un bind del proyecto. La configuración del servidor sonar que tenéis que usar es `https://sonarqube.inf.uva.es/`. En dicho servidor os valen las credenciales que tenéis para los labs de la Escuela, mismas que en gitlab.inf.uva.es. Sin embargo no es seguro introducir login y password en Eclipse. Es mejor usar la opción token. Se deberá generar un token en el apartado Security de la configuración de usuario en el servidor `https://sonarqube.inf.uva.es/`.

Se debe hacer un bind al proyecto con key tds-general-for-sonarlint preparado especialmente para ello. A partir de ese momento, podéis usar el lint, la primera vez dais la opción Analyze, las siguientes veces podéis dar la opción Analyze changed files. Esta información la da `git`, proporciona los cambios y entonces se realizan los análisis sólo sobre los cambios que se han realizado.

El uso de SonarLint no hace análisis en publish en el servidor SonarQube. Los análisis en modo publish se lanzan con la órden maven correspondiente desde el pipeline de gitlab o bien en local usando maven o su envoltorio ant.

En el archivo yml que os doy como referencia para el pipeline de `gitLab` tenéis explicado cómo lanzar un análisis que quede registrado en el servidor. También en los comentarios del snippet 25 se indican las órdenes maven y sus parámetros para ejecutar análisis de cobertura y análisis de calidad con sonarqube en modo publish. Haced esto de vez en cuando para ir registrando la evolución de la calidad de vuestro proyecto, dado que es importante como opción de desempate en el concurso.

1. Fase 1: Rojo

A partir de esta desiderata inicial, la práctica consistirá en aplicar los pasos 1 y 2 del ciclo TDD (Rojo-Verde-Refactor) correspondientes a la fase Rojo:

1. Escribe los tests que ejercitan el código que deseas tener,
2. Comprueba que los tests fallan (fallo, no error).

Siguiendo el proceso de diseño dirigido por pruebas (TDD) :

(a) definiremos qué clase o clases vamos a crear

(b) definiremos las clases de tests para realizar el diseño dirigido por las pruebas

(c) definiremos en las clases de tests cómo se usan los objetos de las clases que hemos creado

(d) especificaremos su funcionalidad en los tests

(e) describiremos esta funcionalidad en el javadoc de las clases creadas

El historial de *commits* debe permitir apreciar el proceso TDD.

Una vez establecida la primera versión de los tests que nos ha permitido crear los *stubs* de las clases, especificar su funcionalidad en dichos tests y describir ésta en el javadoc:

Aplique la filosofía *Test First*.

Añada nuevos tests (en clases de test separadas) teniendo en cuenta las técnicas de caja negra (pruebas unitarias (utilizando particiones basadas en los datos (fronteras del invariante), de secuencia (aleatorias o pruebas de particiones basadas en el estado según se necesite).

Aclaración: No se usará programación por contrato basada en asertos Java. Se diseñará aplicando Programación Defensiva.

Tenga en cuenta que los stubs de la clases `Linea`, `Estacion` y `CoordenadasGPS` ya han sido creado.

Categorice los tests tal y como se especifica en el apartado 2.

2. Fase 2: Verde

En esta fase se implementará el diseño basado en TDD realizado en la Fase 1.

Se utilizarán mocks para conseguir que los tests de la clase diseñada pasen en verde.

Tests

Los tests serán implementados con JUnit. Se utilizará jUnit 4 o jUnit 5 como opción personal. Aplique criterios de modularidad para que las clases de test no crezcan demasiado. Considere la posibilidad de definir una o varias *fixtures*.

Categorice los tests según los criterios especificados en las prácticas 2 y 3.

Tests en la Fase 1

Como resultado de la Fase 1 se espera que el resultado de la ejecución de los tests sea **Rojo**. Por lo tanto, de haberse realizado algo de implementación de los stubs, ésta deberá ser una implementación falseada (*fake implementation*) con el propósito de que **los tests no produzcan errores sino fallos**. El objetivo es que todos los tests implementados fallen, salvo casos excepcionales claramente señalados y comentados. En dichos casos se escribirá `fail` como última instrucción del test para conseguir el fallo. Se indicará claramente con un TODO para saber que se necesita modificar en la fase Verde

2.1. Tests en la Fase 2

Los tests de la Fase 2 serán de pruebas de caja blanca para aumentar la cobertura.

Se probará en aislamiento la clase que implementa una red de metro (se aislará del resto de clases) utilizando mocks objects (basados en EasyMock o Mockito a elección).

Se espera que el resultado de la ejecución de los tests en esta fase sea **Verde** en todos los casos.

En esta práctica se medirá el grado de cobertura alcanzado y se intentará maximizar dicho grado de cobertura con la menor cantidad de test posibles. También se medirá la ratio estática *code to test* y la complejidad ciclomática por método y por clase de la clase implementada.

Fase 3: Refactor

Complete el ciclo TDD con la fase Refactor. Elimine duplicaciones. Mejore el código mediante Refactoring. Siga las recomendaciones para mejorar la calidad que encontrará ejecutando análisis con `https://sonarqube.inf.uva.es`.

Realice las refactorizaciones con Eclipse. En el menú Refactor de Eclipse se podrá ver con la opción History las operaciones de refactoring que se han aplicado.

Debe recordarse que el comportamiento de Eclipse por defecto es tener una única carpeta .refactoring para todo el workspace y no por proyecto. Esta configuración debe cambiarse para que Eclipse almacene el historial de refactoring para el proyecto. Para más información consultar en el aula virtual.

Compruebe que esto se almacena correctamente en el repositorio central y que cuando se hace un *pull* se puede consultar el historial de refactoring realizado. Para ello añada al versionado y al repositorio remoto la carpeta oculta '.refactoring' en la que se hace persistente el historial de refactoring.

Recuerde que al inicio de un proceso de refactoring, los tests deben pasar en verde. Al finalizar esta fase el resultado de la ejecución de tests debe seguir siendo VERDE.

Issues, estimación y tiempo empleado

El proyecto en `gitlab.inf.uva.es` tendrá un listado de *issues* que se irán creando bien al comenzar el proyecto, bien durante el desarrollo del mismo. Al crear cada *issue*, se asignará a un miembro del equipo, se estimará (comando corto `/estimate`) el tiempo que se cree que será necesario para completar el *issue* y al finalizar, se indicará el tiempo que realmente ha sido necesario (comando corto `/spend`). En esta práctica, como en la anterior, no se valorará cómo se ha dividido el proyecto en *issues*, ni tampoco tendrá ningún efecto en la nota la diferencia entre el tiempo estimado y el empleado. Lo que se valora es que se haya estimado y se haya registrado lo real.

Gamificando TDS. Compitiendo en calidad

En esta Práctica 4 introduciremos un pequeño juego, una competición con ganadores y premios.

La competición tiene las siguientes reglas.

Se elaborará un ranking entre los participantes.

Para entrar en el ranking se requiere obtener en el servidor sonarqube:

- Test Success del 100 %
- Condition Coverage superior al 95 %
- Density of Duplications inferior al 1 %
- 0 Bugs
- 0 Vulnerabilities
- Deuda Técnica inferior a 5 horas

Una vez dentro del ranking se establecerá un orden asignando una puntuación a cada participante: $3 * C_d + (60 * 5 - DT) + (100 - DD)$

Donde C_d es la Cobertura de Decisión, DT es la deuda técnica acumulada y DD es la densidad de duplicados.

Los empates se resolverán en este orden

1. el que tenga mayor ratio code to test,
2. el que tenga menor ratio de deuda técnica,
3. el que tenga menor severidad de los code smells, es decir, el que tenga menor (blocking+major)/violations,
4. el que ha conseguido alcanzar antes el estado que ha conducido al empate.

El premio a repartir son puntos en la asignatura. Se otorgará 0,9, 0,6 y 0,3 como premio al primer lugar, segundo, y tercer lugar en el concurso, respectivamente. Como se ha indicando previamente, no habrá empates en ninguna de las tres posiciones del concurso. Al sumar los premios obtenidos en la nota de los ganadores, no podrá sobrepasarse los 10 puntos como nota global de la asignatura.

Características del proyecto Eclipse y Maven

- El proyecto Eclipse deberá nombrarse tds-login-1920 con el login del alumno en los laboratorios de la Escuela (ejemplo: tds-migarci-1920).
- Deberá ser un proyecto válido para Eclipse IDE (release instalada en los labs de la Escuela) y jdk 9 u 8.
- El proyecto será un proyecto maven por lo que será necesario un archivo pom.xml correspondiente al arquetipo maven-archetype-quickstart.
- La estructura de carpetas deberá corresponderse con la definida por el arquetipo maven.
- En el archivo pom.xml los tags `groupId`, `artifactId` y `name` tendrán el siguiente contenido:

```
<groupId>2019-2020-tds-login</groupId>
<artifactId>p4</artifactId>
<name>2019-2020 práctica 4 de TDS del alumno login</name>
```

- Las clases y sus tests estarán en el mismo paquete aunque en carpetas de fuentes diferentes (las carpetas que genera para ello el arquetipo maven).
- El proyecto deberá tener sus propios *settings* indicando el conjunto de caracteres utilizado para evitar problemas de importación en entornos diferentes (indicado en el pom file).

Normas de entrega

- El repositorio `git` y el centralizado en `gitLab`, así como el respaldo en `Atlassian Bitbucket`, deberá llamarse de la misma forma que el proyecto Eclipse.
- Cualquier *push* al repositorio una vez realizada la entrega será penalizado con 0 en la Práctica.
- La entrega se realizará añadiendo a la profesora (usuario `yania` en `gitLab` y `yania@infor.uva.es` en `Atlassian Bitbucket`) con rol Reporter al repositorio que contiene el proyecto en `gitLab` (y con permisos de sólo lectura al respaldo en `Atlassian Bitbucket`) cuando no se vaya a realizar ningún otro *commit & push*.
- El script ant (build.xml) debe tener los objetivos descritos en el enunciado.
- Al entregar la práctica todo deberá quedar integrado en la rama *master*.
- En el tracking de versiones y por tanto en el repositorio remoto solamente residirán los archivos de la configuración del proyecto Eclipse, los fuentes de las clases de la solución, los fuentes de los tests y los archivos pom.xml (maven), build.xml (ant), gitlab-ci.yml (pipeline de `gitLab`) y la carpeta .refactoring para poder consultar el historial de refactoring.

- En los fuentes entregados se hará referencia al autor de la práctica en cada archivo fuente con el tag `@author` de *javadoc* y solamente en ese punto.
- El proyecto tendrá un archivo `README.md` que indicará toda la información que quiera aportar el autor sobre su proyecto además de la siguiente información:
  - Tiempo total en horas-hombre empleado en la realización de la práctica.
  - Clases que forman parte de la solución y por cada clase: SLOC (Source Lines of Code) y LLOC (Logic Lines of Code, es decir, sin contar líneas de comentarios. Estos datos pueden obtenerse con Eclipse Metrics Plugin o mediante Sonarqube.
  - Clases de tests de las clases diseñadas (separadas por clases) y por cada clase de test correspondiente a una clase diseñada indicar SLOC y LLOC, y la suma de estos valores para todas las clases de test de una clase diseñada. Se indicará la ratio *code to test* calculada.
- Fecha límite de entrega: **13 de enero de 2020**. No se admitirán entregas fuera de plazo o por otra vía distinta a la indicada en estas normas.

Criterios de evaluación y recopilación de problemas frecuentes

En la Figura 1 se muestran los criterios que se aplicarán al evaluar esta práctica. En los enunciados de las prácticas 2 y 3 se encuentran enumerados problemas a evitar que se encuentran frecuentemente.

Red	40%	
javadoc	0,5	Clase y métodos con javadoc. Se deben reflejar las excepciones que se lanzan y por qué motivo
Stubs	2,5	Los stubs están bien fabricados, se obtiene un buen diseño orientado a objetos
r.Tests	7	Se desglosa en 5 items
	10	
Desglose de Tests		
r.test1	3	Los tests especifican la clase y sirven para crear los stubs de cada método y su javadoc
r.test2	1	Los tests no tienen errores, fallan todos. Se añaden fake implementations si se necesitan para no tener errores en los tests.
r.test3	2	Se aplican bien las técnicas de caja negra. Se hacen pruebas de secuencia.
r.test4	0,5	Se crea al menos una fixture fundamentalmente usando mocks
r.test5	0,5	Se cumplen los principios FIRST (Fundamentalmente en este caso se mide: Independent, Repeatable, Self-Checking). No hay malos usos de los asserts. Calidad en los tests. Modularidad, legibilidad, comprensibilidad
	7	
Green	40%	
IC	5,5	Integración Continua: Control de versiones, issues maven, ant, gitlab pipeline. Se desglosa en 3 items
G.Tests	4,5	Tests:Green, cobertura, mocks. Se desglosa en 3 items
	10	
Desglose de Tests		
g.test1	1,5	Green. Todos los tests en verde.
g.test2	1,5	Cobertura. Se consigue subir el nivel de cobertura aportando nuevos tests de caja blanca. Se busca cobertura de condición-decisión
g.test3	1,5	Aislamiento. Los tests aportados hasta esta fase se convierten en test en aislamiento basados en mocks en caso necesario
	4,5	
Desglose de Integración Continua		
IC1	1,5	Control de versiones. Frecuencia de commits+push, ramificaciones. Issues, crea, estima tiempo, anota tiempo real.
IC2	1,5	Gestión del proyecto y de dependencias con maven. Es inmediato hacer funcionar el proyecto en otro entorno
IC3	1,5	Objetivos ant, se consiguen los objetivos ant basados en maven de forma que se sigue consiguiendo la gestión de dependencias automatizada y la independencia del entorno
IC4	1	gitlab pipeline
	5,5	
Refactor	20%	
Refactor1	2	Se hace refactor solamente con los tests en verde.
Refactor2	5	Se aplican varias operaciones de refactor no solamente renames
Refactor3	3	Se aprecian buenos indicadores de calidad a partir de los análisis SonarQube
	10	

Figura 1: Criterios de evaluación de la Práctica 4