Test unitaire, couverture de code, un peu de test d'intégration

Premiers pas avec JUnit

On donne deux classes Monome et Polynome.

La classe Monome

- Un constructeur paramétré et un constructeur par copie
- méthode toString : retourne "+3x^5" pour le monôme $3x^5$, "+4" pour le monôme $+4x^0$ et "-5x" pour le monôme $-5x^1$.
- la méthode ajoutAuCoeff permet d'ajouter au coefficient du monôme la valeur prise en paramètre.
- la méthode eval permet d'évaluer le monôme pour une valeur de x prise en paramètre. Par exemple, le monôme $3x^5$ évalué pour la valeur 2 donnera $3 \times 2^5 = 96$.
- la méthode derivee calcule et retourne le monome dérivé du monome receveur. On rappelle que la dérivée du monôme cx^d est cdx^{d-1}.

La classe Polynome

Les polynomes sont représentés sous forme de liste de monômes non nuls, en les triant par degré croissant.

- constructeur pour créer le polynome nul
- la méthode toString retourne une représentation sous forme de chaîne du polynôme, dans laquelle les monômes nuls n'apparaissent pas.
- la méthode ajoutMonome permet d'ajouter un monôme à un polynôme. On notera au passage que suite à un ajout, un monôme peut devenir nul (si on ajoute $-3x^2$ à $3x^2$ par exemple). Il est alors retiré de la liste des monomes.

Travail à réaliser

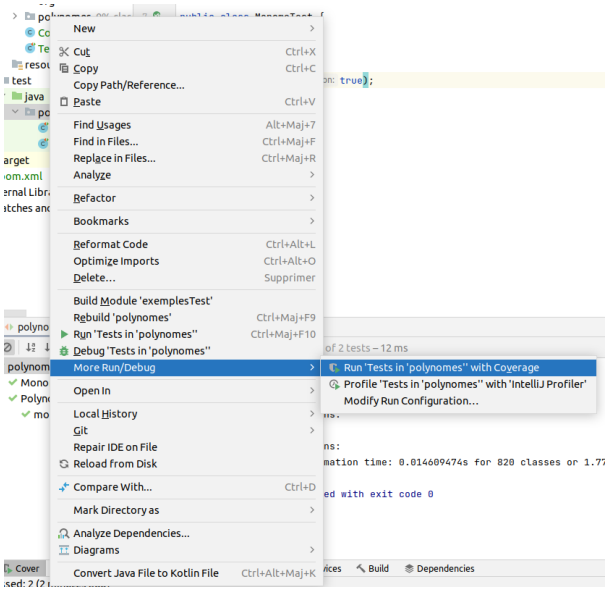
1. Créez un nouveau projet maven, et placez-y les sources des 2 classes, ainsi que de l'exception, fournies sur moodle.
2. Créez une nouvelle classe de test (dans le dossier test, pas mélangé avec les sources) `TestMonome` dans laquelle vous ajouterez des méthodes de test pour tester que :
    1. lors de l'appel à la méthode toString avec le monome $4x^0$ le résultat est "+4".
    2. lors de l'appel à la méthode toString avec le monome $4x^1$ le résultat est "+4x".
    3. lors de l'appel au constructeur avec un degré négatif, on reçoit une DegreNegatifException.
3. Exécutez vos tests, et corrigez les bugs si vous en trouvez.
4. Créez une nouvelle classe de test `TestPolynome` dans laquelle vous ajouterez des méthodes de test pour tester les points qui suivent. Notez que la classe Polynome est difficilement testable en l'état sans tout ramener à la représentation sous forme de chaîne de caractères. Vous introduirez les méthodes qui vous semblent pertinentes pour rendre la classe plus testable, au fur et à mesure que vous en aurez besoin. Exécutez vos tests au fur et à mesure, et corrigez les bugs si vous en trouvez. Points à tester :

1. l'ajout de monome, en réfléchissant aux données de test intéressantes, par exemple ajout d'un monome de degré non existant dans le polynome, ajout de monome de degré existant dans le polynome, etc.
2. le produit de polynomes.

Couverture de code

Couverture de code sous IntelliJ

1. Au lieu d'exécuter vos tests "simplement" avec JUnit, exécutez-les avec Coverage, qui est un outil de couverture de code, en cliquant sur le package contenant les tests puis en suivant l'illustration ci-dessous.



2. Vous obtenez la couverture de votre code par vos tests. Etudiez cette couverture. 3. Ouvrez votre code source, il apparait maintenant avec des éléments de couleur sur la gauche, pour indiquer les parties couvertes et les parties non couvertes. 4. Ajoutez quelques tests pour augmenter la couverture, et relancez l'exécution des tests.

Couverture de code avec Maven et Jacoco

A l'étape précédente, vous avez généré la couverture de code avec votre IDE. La génération de couverture se lance donc à votre initiative. Nous allons regarder comment intégrer cette génération au lifecycle de build de Maven, en bloquant les phases suivantes si un certain taux de couverture n'est pas atteint.

1. Ouvrez le fichier pom.xml et rajoutez-y les lignes suivantes, qui :
   - configurent la version du plug-in qui lance les tests unitaires (surefire), de manière à avoir une version suffisante pour gérer les tests JUnit 5.
   - configurent l'utilisation du plugin de récolte de couverture de test, basé sur Jacoco et Coverage.

```
<build>
    <plugins>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
```

```
                <version>3.0.0-M7</version>
            </plugin>
            <plugin>
                <groupId>org.jacoco</groupId>
                <artifactId>jacoco-maven-plugin</artifactId>
                <version>0.8.3</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>prepare-agent</goal>
                        </goals>
                    </execution>
                    <!-- attached to Maven test phase -->
                    <execution>
                        <id>report</id>
                        <phase>test</phase>
                        <goals>
                            <goal>report</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

2. Ré-exécutez vos tests avec maven :

```
mvn clean test
```

ou en cliquant sur les éléments correspondants dans l'interface IntelliJ pour Maven. Les rapports de couverture sont générés dans target/site/jacoco

3. Rajoutez à votre configuration ce qui suit pour que Jacoco vérifie lors de la phase verify que la couverture est bien d'au moins 50%.

```
<execution>
    <id>jacoco-check</id>
    <goals>
        <goal>check</goal>
    </goals>
    <configuration>
        <rules>
            <rule>
                <limits>
                    <limit>
                        <counter>LINE</counter>
                        <value>COVEREDRATIO</value>
                        <minimum>0.50</minimum>
                    </limit>
                </limits>
```

```
            </rule>
        </rules>
    </configuration>
</execution>
```

puis relancez maven clean verify. Si vos tests couvrent plus de 50% du code, tout va bien, sinon, un message de violation de règle est émis.

Test unitaire et d'intégration dans le projet TER

Test unitaire (ou pseudo unitaire ici comme nous allons le voir)

Il y a très peu de code à tester unitairement dans ce que nous avons fait. Nous allons juste ici vérifier que nos règles de contrôles d'accès des repositories sont correctes.

Nous allons vérifier que le gestionnaire de TER peut ajouter un nouvel enseignant quand il en est le gestionnaire, mais qu'en enseignant ne peut pas ajouter un enseignant.

On est ici à la frontière entre test unitaire et test d'intégration, puisque nous allons devoir lancer tout l'environnement Spring pour réaliser notre test.

1. Créez une classe de test pour TeacherRepository : pour aller plus vite et la placer où il faut directement, placez-vous dans la classe, puis clic droit, generate ... puis Test.
2. Annotez cette classe de test avec les annotations suivantes afin que l'environnement Spring se lance lors du test.

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
```

3. Placez dans la classe de test les attributs suivants de manière à pouvoir accéder au repository des teachers et des managers.

```
@Autowired
private TeacherRepository teachers;
@Autowired
private TERManagerRepository managers;
```

4. Placez dans la classe de test la méthode suivante :

```
 @Test
void saveIsPossibleForManager() {
    SecurityContextHolder.getContext().setAuthentication(
            new UsernamePasswordAuthenticationToken("lechef", "peu
importe", AuthorityUtils.createAuthorityLis("ROLE_MANAGER")));
    TERManager terM1Manager = new TERManager("Mathieu", "lechef",
"mdp", "ROLE_MANAGER");
```

```
        this.managers.save(terM1Manager);
        this.teachers.save(new Teacher("Margaret", "Hamilton", "margaret",
    terM1Manager, "ROLE_TEACHER"));
        assertThat(teachers.findByLastName("Hamilton"),
    is(notNullValue()));
    }
```

Dans cette méthode, on simule une authentification, on ajoute un nouveau manager de TER, et on sauve un nouvel enseignant ayant pour gestionnaire le gestionnaire nouvellement créé. Puis on vérifie qu'il existe avec le assertThat de hamcrest (il faudra faire des imports, IntelliJ vous les suggèrera). Notez au passage que vous utilisez hamcrest sans l'avoir ajouté comme dépendance dans le pom. C'est en effet une dépendance transitive de org.springframework.boot:spring-boot-starter-test, qui est une dépendance présente dans notre pom.

5. Exécutez votre test, qui devrait être un succès.
6. Créez un autre test vérifiant qu'au contraire si un enseignant est connecté, lors de l'ajout d'un enseignant, une exception `org.springframework.security.access.AccessDeniedException` est levée, et l'ajout ne fonctionne pas.

Test des contrôleurs (test d'intégration)

Pour tester les contrôleurs, il va falloir émettre des requêtes http, et vérifier les réponses obtenues. Il s'agit bien ici de test d'intégration, puisqu'on va interroger le système dans son environnement, derrière un serveur web.

Simuler le MVC

Spring dispose de mécanismes pour simuler le MVC, avec notamment `MockMVC`. Cela va nous permettre de construire facilement dans un test (en Java) des requêtes http et d'en étudier les réponses reçues.

Pour cela, on écrit dans une nouvelle classe de test quelque chose du genre :

```
@ExtendWith(SpringExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
class TeacherControllerTest {
    @Autowired
    private MockMvc mvc;

    @Test
    void addTeacherGet() throws Exception {

        MvcResult result = mvc.perform(get("/addTeacher"))
                .andExpect(status().isOk())
                .andExpect(content().contentType("text/html;charset=UTF-
8"))
                .andExpect(view().name("addTeacher"))
                .andReturn();
```

```
    }

}
```

- Avec l'annotation `SpringBootTest` on met en route l'environnement Spring
- Avec l'annotation `AutoConfigureMockMvc` on autoconfigure notre simulateur de MVC
- On déclare comme attribut du test auto-injecté un simulateur de MVC, nommé mvc
- Puis dans la méthode de test on demande à mvc de réaliser :
    - un get sur la route "/addTeacher"
    - une vérification sur le statut de la réponse obtenue qui doit être "OK"
    - une vérification sur le type de réponse obtenu : le contenu doit être de l'html encodé en utf8
    - une vérification sur le nom de la vue (de la réponse)
- Ici on aurait pu éviter de stocker la réponse dans la variable result puisqu'on ne s'en sert plus.

Se faire passer pour un utilisateur ...

Problème : notre SUT utilise Spring Security pour contrôler les droits d'accès. Il faut donc que le test se fasse passer pour un utilisateur autorisé. C'est prévu dans Spring. Ajouter au pom.xml :

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

Puis annotez ainsi la méthode de test :

```
@Test
@WithMockUser(username = "Chef", roles = "MANAGER")
void addTeacherGet() throws Exception {

    MvcResult result = mvc.perform(get("/addTeacher"))
            .andExpect(status().isOk())
            .andExpect(content().contentType("text/html;charset=UTF-8"))
            .andExpect(view().name("addTeacher"))
            .andReturn();

}
```

On se fait ici passer pour le manager Chef.

Envoi de paramètres

Nous allons maintenant simuler un post pour ajouter un nouvel enseignant, ici Anne-Marie Kermarrec, d'identifiant non existant 10.

```
@Test
@WithMockUser(username = "Chef", roles = "MANAGER")
void addTeacherPostNonExistingTeacher() throws Exception {
        assertTrue(teacherService.getTeacher(10l).isEmpty());
        MvcResult result = mvc.perform(post("/addTeacher")
                        .param("firstName", "Anne-Marie")
                        .param("lastName", "Kermarrec")
                        .param("id", "10")
                )
                .andExpect(status().is3xxRedirection())
                .andReturn();
    // il faut ici vérifier que le nouvel enseignant a bien été ajouté
}
```

- Les paramètres sont ajoutés à la requête, cela simule l'envoi de données du formulaire.

Travail à réaliser

1. Reprendre la méthode de test précédente. Sa première ligne vérifie qu'il n'y a pas d'utilisateur d'identifiant 10. S'il y en avait un, cela ne devrait pas déclencher d'erreur, juste conditionner la suite du test. Utiliser pour cela un assumingThat (voir cours).
2. Ecrire une autre méthode de test addTeacherPostExistingTeacher() qui vérifie que si on ajoute un enseignant qui est en fait existant (identifiant déjà connu) alors l'enseignant est mis à jour avec les nouvelles valeurs.

Discussion

Notre façon de tester n'est pas très satisfaisante.

- On travaille directement sur la base de données de l'application, et on a besoin pour tester d'avoir beaucoup d'a priori sur son contenu.
- Nos tests ne sont pas vraiment indépendants, puisqu'ils alimentent la même base de donnée.
- Nos tests modifient la base de donnée de l'application. Ici ça n'a pas de réelle conséquence car notre base de données ne persiste pas au delà de l'exécution de l'application, mais sur des cas réels, c'est très embétant.
- Nous verrons plus loin comment procéder autrement.