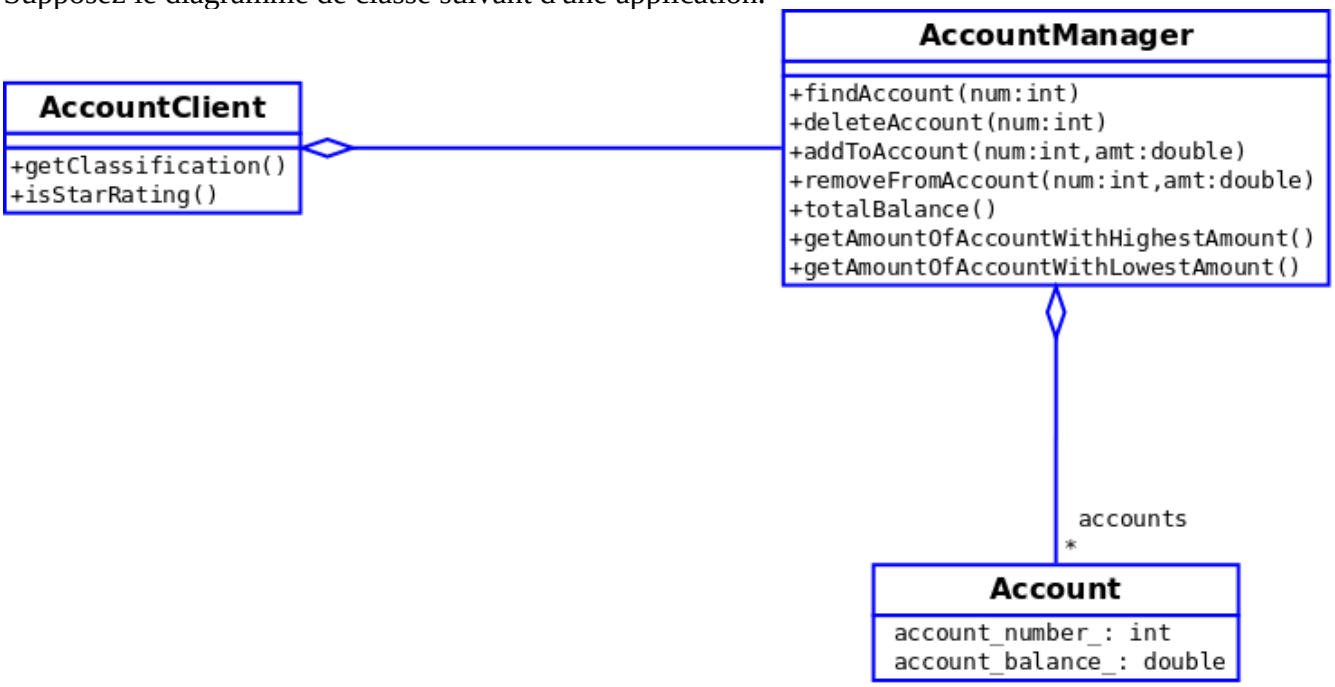
Test d'objects en isolation avec des Mock Objects

Mock Object

1. Un objet mock est un objet créé pour se substituer à un objet avec lequel votre code collabore. Votre code peut appeler des méthodes sur l'objet mock, qui donneront des résultats selon vos objectifs de tests.
2. Les mocks remplacent les objets avec lesquels vos méthodes testées collaborent, offrant ainsi une couche d'isolation.
3. In order to utilize mocks, sometimes you have to refactor your code. However, such a refactoring is welcomed anyway as it makes the code more flexible for future extensions.
4. Vous devez parfois remanier votre code (refactoring) pour introduire des mocks. Cependant, un tel refactoring est la bienvenue de toute façon car il rend le code plus flexible pour les extensions futures.
5. Un gros avantage des objets mocks par rapport aux stubs traditionnelles est le plus grand contrôle offert.

Exemple

Supposez le diagramme de classe suivant d'une application.



Le code original de l'exemple se trouve dans ***account.zip***. Supposez que nous voulions tester la classe ***AccountClient***, en particulier les méthodes ***getClassification*** et ***isStarRating*** sans impliquer l'implémentation réelle de ***AccountManager***. Diverses raisons peuvent motiver, par exemple,

- assurer l'exactitude de la logique de l'implémentation des méthodes isolément,
- ***AccountManager*** n'est peut-être pas prêt,
- pour exercer correctement les interfaces entre ***AccountClient*** et ***AccountManager*** avant de les intégrer,
- etc

Pour tester ***AccountClient*** sans impliquer le ***AccountManager*** fourni, il faut remplacer (stubbing) la

variable `accManager_` variable et les appels de méthodes tels que `accManager_.totalBalance()`.

```
public class AccountClient {
    private AccountManager accManager_;

    /** Creates a new instance of AccountClient */
    public AccountClient() {
        accManager_ = new AccountManager();
    }

    /** Determine a client classification. Returns
     * 1 - if total balance < 1000
     * 2 - if 1000 <= total balance < 10000
     * 3 - if 10000 <= total balance < 100000
     * 4 - if total balance >= 100000
     */
    public int getClassification() {
        double balance = accManager_.totalBalance();
        if (balance < 1000) {
            return 1;
        } else if (balance < 10000) {
            return 2;
        } else if (balance < 100000) {
            return 3;
        } else return 4;
    }

    /** Determine a client's star rating. Returns
     * STAR - if at least one account > 100000 and no account < 0
     * REGULAR - otherwise
     */
    public boolean isStarRating() {
        double highest = accManager_.getAmountOfAccountWithHighestAmount();
        double lowest = accManager_.getAmountOfAccountWithLowestAmount();
        if (highest > 100000 && lowest >= 0) {
            return true;
        } else return false;
    }
}
```

Usage de stubs traditionnels

Un stub est un petit morceau de code utilisé pour remplacer du code dont dépend une unité testée. L'approche stub traditionnelle est utilisée dans ***accountStub.zip***.

- Créez un projet eclipse avec les contenu décompressé de l'archive.
- Examinez l'implémentation du stub de ***AccountManager***
- Exécutez les tests dans ***AccountClientTest***

Question 1: Qu'est-ce qui est nécessaire pour que le cas de test `testStarRating` passe?

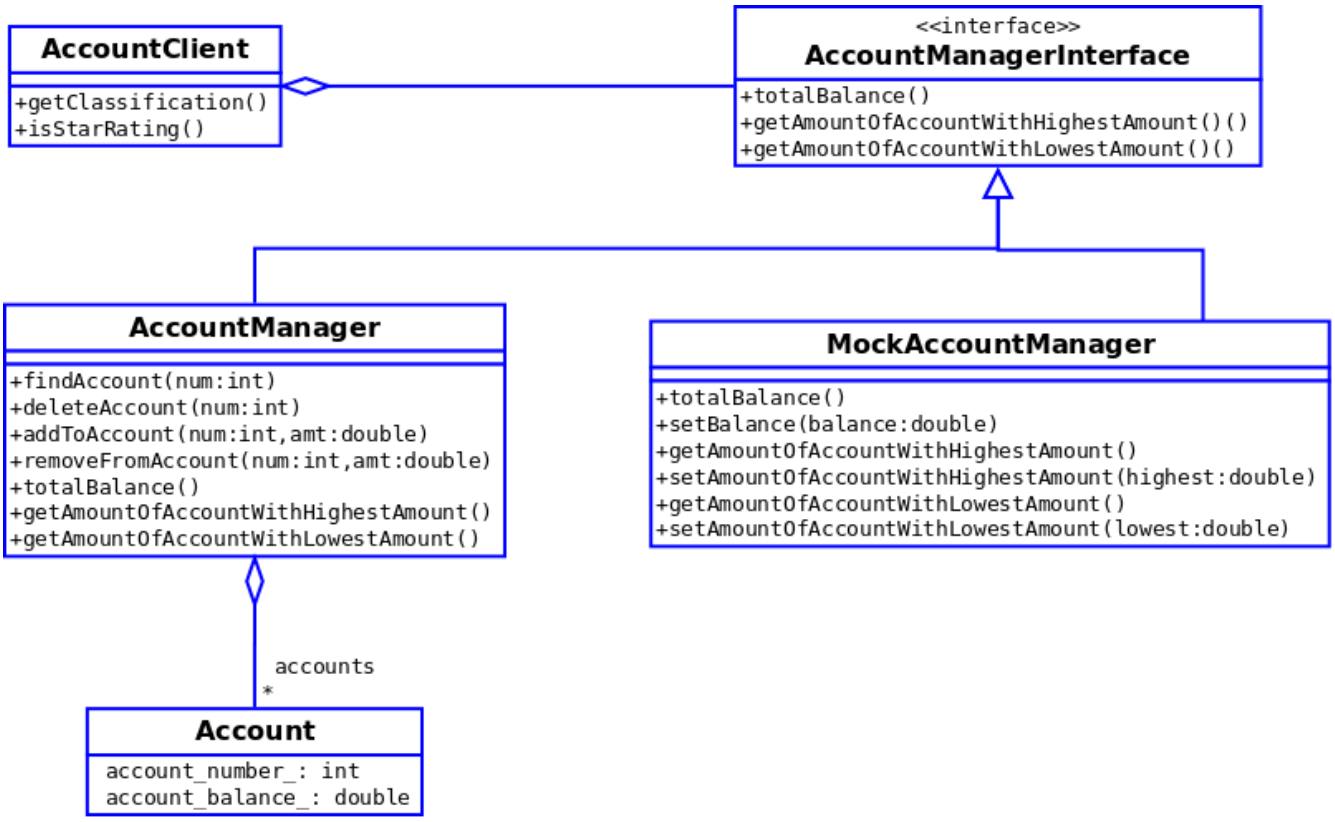
Maintenant, pensez à ajouter plus de tests par exemple, pour les autres niveaux de classification. Il va falloir remplacer l'implémentation de ***AccountManager*** pour chaque classification. Le test en isolation est difficile avec des stubs traditionels car,

- plusieurs stubs doivent être créés et mis en place conformément aux'objectifs de tests
- l'environnement de test doit être configuré en utilisant les stubs appropriées.

Usage d'objets mocks

Nous allons utiliser l'approche des objets mocks pour rendre le test de classes en isolation plus flexible. Dans un premier temps, nous allons créer manuellement notre classe mock. Dans les sections suivantes, nous considérerons la création dynamique de mocks en utilisant des frameworks de mocks.

- Refactoring pour supporter les mocks. L'application originale doit être remaniée pour supporter les objets mocks (en plus d'améliorer la conception). De façon plus précise, nous allons
  - introduire une interface ***AccountManagerInterface*** pour ***AccountManager***
  - avoir la classe ***AccountClient*** dépendre de ***AccountManagerInterface*** plutôt que d'une classe concrète, et
  - avoir l'objet dépendant ***AccountManagerInterface*** injecté dans ***AccountClient*** au lieu d'être créé par celui-ci.
- Une classe mock est créée comme implémentrice de la nouvelle interface.



Nous pouvons maintenant créer des cas de test utilisant un gestionnaire de compte factice plutôt que le vrai. En outre, nous pouvons spécifier à partir des cas de test quelle valeur de *totalBalance*, *highestAmount* ou *lowestAmount* nous voulons que l'objet mock retourne. Cela permet plusieurs tests sans avoir à mettre en place une implémentation différente de la classe ***AccountManager***.

```
@Test
public void testGetClassification1() {
	AccountClient instance = setAccountInstance(0.0);

	int expResult = 1;
	int result = 0;
	try {
		result = instance.getClassification();
	} catch (Exception e) {
		e.printStackTrace();
		Assert.fail();
	}
	Assert.assertEquals(expResult, result);
}
```

Le code complet pour cet exemple est dans ***accountManualMock.zip***.

- Créez un projet eclipse avec les contenus décompressés de l'archive.
- Examinez le refactoring qui a été fait et comparer à la version précédente.
- Exécutez les tests dans ***AccountClientTest***

Question 2: ajoutez un cas de test pour vérifier que la classification 4 est déterminée pour un total de 100000.

Création dynamique d'objets mocks avec EasyMock

- La création manuelle de classes mocks peut devenir fastidieuse lorsque une interface comprend plusieurs méthodes.
- La création de mocks peut être automatisé à l'aide de cadres tels que EasyMock
- EasyMock crée dynamiquement des objets mocks en trois étapes
    1. création d'un Objet Mock pour l'interface à simuler,
    2. enregistrement du comportement attendu, et
    3. switch à l'Objet Mock pour “rejouer” le comportement enregistré.

EasyMock fournit différents moyens pour la création et la configuration d'objets Mock. Nous avons utilisé l'approche par injection où le Framework détermine quand instancier le Mock et ensuite l'injecte dans l'unité testée. Lors de l'utilisation de cette approche, un *test runner* spécial doit être utilisé. Nous pouvons spécifier ce *test runner* en utilisant une annotation @Rule.

```
@Rule
public EasyMockRule mocks = new EasyMockRule(this); // set the test runner with
EasyMock

@TestSubject
private AccountClient instance = new AccountClient(); // specifies unit under test

@Mock
private AccountManagerInterface mmanag; // injects mocked resource
```

Nous avons créé une méthode *helper* pour enregistrer qu'un appel à ***mmanag.totalBalance ()*** est attendu et que cet appel doit retourner une valeur donnée. Nous rejouons ensuite, le mock.

```
private void setAccountInstance(double balance) {
   expect(mmanag.totalBalance()).andReturn(balance); // records mock  expectation
   replay(mmanag);                                   // replays mock
}
```

Plusieurs cas de test peuvent alors utiliser la méthode helper en fonction de leur objectif de test. Par exemple pour tester la classification 1.

```
    @Test
    public void testGetClassification1() {
      setAccountInstance(0.0);

        int expResult = 1;
        int result = 0;
            try {
                  result = instance.getClassification();
            } catch (Exception e) {
                  e.printStackTrace();
                  Assert.fail();
            }
        Assert.assertEquals(expResult, result);
        verify(mmanag);  // check that mocked expected calls were made
  }
```

Nous vérifions à la fin du test que tous les appels attendus ont bien eu lieu.

Des exemples de cas de tests de Account Manager utilisant EasyMock sont dans ***accountEasyMock.zip***. Cette archive contient un projet *Maven* exporté.

- Décompressez l'archive
- Importez le projet **File -> Import..** puis étendre **Maven** et sélectionnez **Existing Maven Projects**
- Cliquez sur **Next**, puis sur **Browse** pour sélectionner le répertoire décompressé et **Finish**.
- Examinez l'implémentation et exécutez les tests dans ***AccountClientTest.***

Question 3: ajouter un cas de test pour vérifier que la classification 4 est déterminée pour un total de 100000.

Notez que nous avons utilisé *Maven* (https://maven.apache.org/) pour faciliter la gestion des dépendances JAR. Les frameworks tels que EasyMock ont tendance à nécessiter plusieurs JARs interdépendants qui doivent tous être définis sur le Classpath du projet. Maven (parmi plusieurs autres fonctionnalités) automatise la résolution de ces dépendances et peut automatiquement télécharger et configurer les fichiers JAR manquants.

Vous n'avez cependant pas besoin d'utiliser Maven pour utiliser EasyMock. Vous pouvez simplement télécharger et configurer manuellement les JARs du framework manuellement dans le classpath de votre projet. Une documentation complète sur EasyMock est disponible sur http://easymock.org/

Création dynamique d'objets mocks avec Mockito

Une implémentation avec Mockito (http://site.mockito.org/) se trouve dans le projet Maven exporté a***ccountMockito.zip***.

- Décompressez l'archive

- Importez le projet **File -> Import..** puis étendre **Maven** et sélectionnez **Existing Maven Projects**
- Cliquez sur **Next**, puis sur **Browse** pour sélectionner le répertoire décompressé et **Finish**.
- Examinez l'implémentation et exécutez les tests dans ***AccountClientTest.***

Mockito utilise un cycle d'exécution *stub-verify* sans relecture (*replay*). Un objet mock peut être créé et configuré avec une annotation @Mock, mais cela implique l'utilisation d'un test runner spécial.

```
@RunWith(MockitoJUnitRunner.StrictStubs.class)
public class AccountClientTest {
	private AccountClient instance;

	@Mock
	private AccountManagerInterface mmanag;
}
```

Nous passons l'objet mock à chaque cas de test avant son exécution en utilisant une méthode setup.

```
	@Before
	public void setUp() {
		instance = new AccountClient();
		instance.setAccountManager(mmanag);
	}
```

Nous avons introduit des méthodes helpers tels que suivant pour enregistrer les comportements attendus

```
	private void setAccountInstance(double balance) {
	  when(mmanag.totalBalance()).thenReturn(balance);
	}
```

Plusieurs cas de test peuvent alors utiliser la méthode helper en fonction de leur objectif de test. Par exemple pour tester la classification 1.

```
    @Test
    public void testGetClassification1() {
      setAccountInstance(0.0);

        int expResult = 1;
        int result = 0;
            try {
                  result = instance.getClassification();
            } catch (Exception e) {
                  e.printStackTrace();
                  Assert.fail();
            }
        Assert.assertEquals(expResult, result);
        verify(mmanag).totalBalance();
    }
```

Question 4: ajouter un cas de test pour vérifier que la classification 4 est déterminée pour un total de 100000.