UE Conception Orientée Objet

Devoir Surveillé - Corrigé

3h

Copie des diapositives de cours annotée autorisée.
Fiches « *design pattern* » du cours autorisées.
Dictionnaires de langue autorisés.
Autres documents interdits.

Sauf mention expresse, la javadoc et les tests des classes à écrire ne sont pas demandés.

Exercice 1 :

Q 1. Donnez un **ou plusieurs** diagrammes UML de classes détaillés qui présentent clairement votre proposition de conception pour gérer les différents types d'unités.

Dans votre diagramme, vous ferez apparaître clairement les attributs, constructeurs et méthodes (et leurs signatures), ainsi que les surcharges.

Faites des diagramme clairs et lisibles. N'hésitez pas à prendre votre copie au format paysage, ou mieux à faire votre diagramme sur les deux pages intérieures de votre copie.

<<Java Class>> *Unit*
#speed: int
+Unit(int)
+*cost():int*
+*workOn(Land):int*
+getSpeed():int

<<Java Class>> *SingleUnit*
#strength: int
#ability: int
#tool: Tool
+SingleUnit(int,int,int)
+cost():int
+getStrength():int
+getAbility():int
+getTool():Tool
+changeTool():void
+workOn(Land):int
+*createTool():Tool*

<<Java Class>> GroupUnit
+GroupUnit()
+addUnit(SingleUnit):void
+allUnits():List<SingleUnit>
+getSpeed():int
+cost():int
+workOn(Land):int

#units 0..*

#decorated 0..1

<<Java Class>> Farmer
+Farmer(int,int,int)
+createTool():Tool

<<Java Class>> Miner
+Miner(int,int,int)
+createTool():Tool

<<Java Class>> LumberJack
+LumberJack(int,int,int)
+createTool():Tool

<<Java Class>> *AbstractUnitDecorator*
+AbstractUnitDecorator(SingleUnit)
+cost():int
+getTool():Tool
+changeTool():void
+createTool():Tool
+workOn(Land):int

<<Java Class>> Horse
-SPEED: int
+Horse()
+getSpeed():int

#horse 0..1

<<Java Class>> StrongUnit
+StrongUnit(SingleUnit)
+getStrength():int

<<Java Class>> SkilledUnit
+SkilledUnit(SingleUnit)
+getAbility():int
+cost():int

<<Java Class>> MountedUnit
+MountedUnit(SingleUnit)
+getSpeed():int
+cost():int
+getHorse():Horse

Q 2. Quel(s) *design pattern(s)* avez-vous mis en œuvre dans votre solution ?

Justifiez clairement dans chaque cas en quelques phrases claires pourquoi l'utilisation de ce *design pattern* est pertinent dans votre solution.

Composite pour décrire des groupes d'unités `GroupUnit`,

Factory method pour créer les outils associés aux unités via `SingleUnit`,

Décorator pour définir des unités avancées via `AbstractUnitDecorator`,

Q 3. Donnez le code du constructeur de la classe de base des unités simples.

```java
public abstract class SingleUnit extends Unit {
  // ...

  public SingleUnit(int speed, int strength, int ability) {
    super(speed);
    this.strength = strength;
    this.ability = ability;
    this.tool = this.createTool();
  }

  // ...
}
```

Q 4. Donnez les lignes de code qui, à partir de votre proposition, permettent de créer :

- une unité *bûcheron*,
- une unité *fermier plus habile*,
- une unité *mineur à cheval*,
- une unité *mineur plus fort à cheval*,
- une unité groupe regroupant les quatre unités précédentes.

```java
SingleUnit unit[] = new SingleUnit[0];
unit[0] = new LumberJack(10, 10, 10);
unit[1] = new SkilledUnit(new Farmer(10, 10, 10));
unit[2] = new MountedUnit(new Miner(10, 10, 10));
unit[3] = new StrongUnit(new MountedUnit(new Miner(10, 10, 10)));

GroupUnit group = new GroupUnit();
for (SingleUnit u : unit)
  group.addUnit(u);
```

Q 5. Donnez le code de la ou des méthodes de test pour vérifier le comportement de la méthode `collectWood` de la classe `Land`.

```java
@RunWith(Parameterized.class)
public class LandTest {
  @Parameters(name = "{index}: Land({0}).collectWood({1})")
  public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] { { 10, 5 }, { 10, 10 }, { 10, 20 } });
  }

  @Parameter(0)
  public int wood;
  @Parameter(1)
  public int collect;

  private Land land;

  @Before
  public void setUp() throws Exception {
    this.land = new Land(this.wood, 0, 0);
  }

  @Test
  public void collectWoodShouldReturn() {
    int expect = Math.min(this.wood, this.collect);
    assertEquals(expect, this.land.collectWood(this.collect));

    expect = Math.max(this.wood - this.collect, 0);
    assertEquals(expect, this.land.getWood());
  }
}
```

Q 6. Donnez le code de la ou des méthodes de test pour vérifier que la méthode `workOn` déclenche une exception si l'outil de l'unité est cassé.

Indiquez comment il faut faire pour que ce test soit vérifié pour chacune des classes d'unités simples en donnant le code nécessaire pour la classe des *bûcherons* et en précisant comment cela se généralise aux autres classes d'unités simples.

```java
public abstract class SingleUnitTest {
  protected SingleUnit unit;
  protected Land land;

  @Before
  public void setUp() {
    this.unit = createUnit();
    this.unit.changeTool();
    this.land = new Land(0, 0, 0);
  }

  protected abstract SingleUnit createUnit();

  @Test(expected = BrokenToolException.class)
  public void workOnShouldReturnAnException() throws BrokenToolException {
    breakTool();
    this.unit.workOn(this.land);
    fail("BrokenToolException should be raised when tool is broken");
  }

  private final void breakTool() throws BrokenToolException {
    do {
      this.unit.workOn(this.land);
    } while (!this.unit.getTool().isBroken());
  }
}

public class LumberJackTest extends SingleUnitTest {
  @Override
  protected SingleUnit createUnit() {
    return new LumberJack(0, 0, 0);
  }
}
```

Q 7. Donnez le code de la ou des méthodes de test qui permettent de vérifier que la méthode `collect()` d'un outil est appelée exactement une fois par la méthode `workOn()` de l'unité associée, qu'une exception soit déclenchée ou non.

```java
public class MockUnitTest extends SingleUnitTest {
  @Test
  public void workOnShouldInvokeCollectOnce() {
    try {
      this.unit.workOn(this.land);
    } catch (BrokenToolException e) {
      // Silent the exception if raised
    } finally {
      assertEquals(1, this.mockUnit.getMockTool().getCount());
    }
  }

  private MockUnit mockUnit;

  @Override
  protected SingleUnit createUnit() {
    this.mockUnit = new MockUnit();
    return this.mockUnit;
  }
}

public class MockUnit extends SingleUnit {

  public MockUnit() {
```

```java
      super(0, 0, 0);
    }

    private MockTool mockTool;

    public MockTool getMockTool() {
      return this.mockTool;
    }

    @Override
    protected Tool createTool() {
      this.mockTool = new MockTool();
      return this.mockTool;
    }
  }

  public class MockTool implements Tool {
    private int count = 0;

    public int getCount() {
      return count;
    }

    @Override
    public int collect(Land land, SingleUnit collector) throws BrokenToolException {
      this.count++;
      return 0;
    }

    @Override
    public boolean isBroken() {
      return false;
    }
  }
```

Q 8. Donnez le code de la ou des méthodes de test qui permet de vérifier que le coût d'une *unité à cheval* est augmenté de 2 par rapport à la même unité qui n'est pas à cheval.

```java
public abstract class SingleUnitTest {
  // ...

  @Test
  public void horseUnitCostMoreThanSingleUnit() {
    MountedUnit mountedUnit = new MountedUnit(this.unit);
    try {
      this.unit.workOn(this.land);
    } catch (BrokenToolException e) {
      // Silent the exception if raised
    } finally {
      assertEquals(this.unit.cost() + MountedUnit.COST_OVERHEAD, mountedUnit.cost());
    }
  }
}
```

Q 9. Pour tous les types d'unités où elles sont **définies ou redéfinies (surchargées)** donnez le code java des méthodes :

Q 9.1. `workOn()`

Q 9.2. `getSpeed()`

Précisez clairement à chaque fois la classe où se situe le code fourni.

```java
public abstract class Unit {

  protected int speed;

  public Unit(int speed) {
    this.speed = speed;
```

```java
  }

  public int getSpeed() {
    return this.speed;
  }
}

public abstract class SingleUnit extends Unit {
  // ...

  @Override
  public int workOn(Land land) throws BrokenToolException {
    return this.tool.collect(land, this);
  }
}

public abstract class AbstractUnitDecorator extends SingleUnit {
  // ...

  @Override
  public int workOn(Land land) throws BrokenToolException {
    return this.decorated.workOn(land);
  }
}

public class GroupUnit extends Unit {
  // ...

  @Override
  public int getSpeed() {
    List<Integer> speeds = new ArrayList<>();
    for (SingleUnit u : units) {
      speeds.add(u.getSpeed());
    }
    return Collections.min(speeds);
  }

  @Override
  public int workOn(Land land) {
    int result = 0;
    for (SingleUnit u : units) {
      try {
        result += u.workOn(land);
      } catch (BrokenToolException e) {
        u.changeTool();
      }
    }
    return result;
  }
}

public class MountedUnit extends AbstractUnitDecorator {
  // ...

  @Override
  public int getSpeed() {
    return this.horse.getSpeed();
  }
}
```