L3 SRI - COO

Cours 8 : Les tests unitaires

Les hypothèses et assertions
Les tests unitaires

Auteur : CHAUDET Christelle

Introduction

- Le test logiciel permet de maintenir le niveau de qualité logicielle.
- « *Le test d'un programme peut être un moyen très efficace pour montrer la présence de bugs, mais il est formellement insuffisant pour montrer leur absence.* » Edsger W. Dijkstra
- Dans un cycle de développement logiciel traditionnel, on distingue généralement 4 phases de test :
  - Phase de test unitaire -> test un module (ou une unité)
  - Phase de test d'intégration -> test l'interaction entre les modules
  - Phase de test système -> vérifie que les fonctionnalités sont conformes à leur spécifications fonctionnelles et techniques (performance, ressources consommées…)
  - Phase de test d'acceptation (recette) -> validation contractuelle par le client

Hypothèse et assertion

- Une hypothèse de test est une précondition, une propriété qui doit être satisfaite pour que le test soit significatif.
- Lorsqu'une hypothèse n'est pas satisfaite le test en cours est arrêté et son résultat est ignoré : le test n'a ni réussi ni échoué.
- Pour définir des hypothèses de test il faut utiliser méthodes statiques de :
  - la classe Assume du package *org.junit* (Junit 4) pour les programmes étant écrit avec des versions de java inférieur à 8
  - La classe Assumptions du package *org.junit.jupiter.api* (Junit 5) pour les programmes étant écrit avec des versions de Java 8 (et supérieur)

Hypothèse

Method Summary — JUnit 4.12 API — http://junit.org/junit4/javadoc/latest/index.html

static void	assumeFalse(boolean b) The inverse of assumeTrue(boolean).
static void	assumeFalse(String message, boolean b) The inverse of assumeTrue(String, boolean).
static void	assumeNoException(String message, Throwable e) Attempts to halt the test and ignore it if Throwable e is not null.
static void	assumeNoException(Throwable e) Use to assume that an operation completes normally.
static void	assumeNotNull(Object... objects) If called with one or more null elements in objects, the test will halt and be ignored.
static <T> void	assumeThat(String message, T actual, Matcher<T> matcher) Call to assume that actual satisfies the condition specified by matcher.
static <T> void	assumeThat(T actual, Matcher<T> matcher) Call to assume that actual satisfies the condition specified by matcher.
static void	assumeTrue(boolean b) If called with an expression evaluating to false, the test will halt and be ignored.
static void	assumeTrue(String message, boolean b) If called with an expression evaluating to false, the test will halt and be ignored.

Hypothèse

http://junit.org/junit5/docs/current/api/

JUnit 5.0.2 API

Modifier and Type	Method and Description
static void	**assumeFalse**(boolean assumption) Validate the given assumption.
static void	**assumeFalse**(boolean assumption, String message) Validate the given assumption.
static void	**assumeFalse**(BooleanSupplier assumptionSupplier) Validate the given assumption.
static void	**assumeFalse**(boolean assumption, Supplier<String> messageSupplier) Validate the given assumption.
static void	**assumeFalse**(BooleanSupplier assumptionSupplier, String message) Validate the given assumption.
static void	**assumeFalse**(BooleanSupplier assumptionSupplier, Supplier<String> messageSupplier) Validate the given assumption.

() optionnel
— ce qu'on apprend et ce qu'on va utiliser

Assertion

- Une assertion est une propriété qui doit être satisfaite pour que le test réussisse.
- Lorsqu'une assertion n'est pas satisfaite le test en cours est arrêté et le test a échoué.
- Pour définir des assertions il faut utiliser méthodes statiques de :
  - la classe Assert du package *org.junit* (Junit 4) pour les programmes étant écrit avec des versions de java inférieur à 8
  - La classe Assertions du package *org.junit.jupiter.api* (Junit 5) pour les programmes étant écrit avec des versions de Java 8 (et supérieur)

Hypothèse

JUnit 5.0.2 API

Modifier and Type	Method and Description
static void	**assumeTrue**(boolean assumption) Validate the given assumption.
static void	**assumeTrue**(boolean assumption, String message) Validate the given assumption.
static void	**assumeTrue**(BooleanSupplier assumptionSupplier) Validate the given assumption.
static void	**assumeTrue**(boolean assumption, Supplier<String> messageSupplier) Validate the given assumption.
static void	**assumeTrue**(BooleanSupplier assumptionSupplier, String message) Validate the given assumption.
static void	**assumeTrue**(BooleanSupplier assumptionSupplier, Supplier<String> messageSupplier) Validate the given assumption.
static void	**assumingThat**(boolean assumption, Executable executable) Execute the supplied Executable, but only if the supplied assumption is valid.
static void	**assumingThat**(BooleanSupplier assumptionSupplier, Executable executable) Execute the supplied Executable, but only if the supplied assumption is valid.

Assertion

- Les méthodes des classes Assert et Assertions sont toutes surchargées, par exemple :

static void	**assertNotNull**(Object actual) *Asserts* that actual is not null.
static void	**assertNotNull**(Object actual, String message) *Asserts* that actual is not null.
static void	**assertNotNull**(Object actual, Supplier<String> messageSupplier) *Asserts* that actual is not null.

- Ainsi il y a 30 surcharges de la méthode assertEquals dans la classe Assertions. Dans ce cours je présenterais chacune des méthodes avec en paramètre l'objet (ou les objets) et le message.

Method Summary	JUnit 4.12 API
static void	assertArrayEquals(String message, Object[] expecteds, Object[] actuals) Asserts that two object arrays are equal.
static void	assertEquals(String message, Object expected, Object actual) Asserts that two objects are equal.
static void	assertFalse(boolean condition) Asserts that a condition is false.
static void	assertNotEquals(Object unexpected, Object actual) Asserts that two objects are **not** equals.
static void	assertNotNull(String message, Object object) Asserts that an object isn't null.
static void	assertNotSame(String message, Object unexpected, Object actual) Asserts that two objects do not refer to the same object.
static void	assertNull(String message, Object object) Asserts that an object is null.
static void	assertSame(String message, Object expected, Object actual) Asserts that two objects refer to the same object.
static <T> void	assertThat(String reason, T actual, Matcher<? super T> matcher) Asserts that actual satisfies the condition specified by matcher.
static void	assertTrue(String message, boolean condition) Asserts that a condition is true.
static void	fail(String message) Fails a test with the given message.

Assertion

Modifier and Type	Method and Description	JUnit 5.0.2 API
static void	assertAll(String heading, Executable... executables) *Asserts* that *all* supplied executables do not throw exceptions.	
static void	assertAll(String heading, Stream<Executable> executables) *Asserts* that *all* supplied executables do not throw exceptions.	
static void	assertArrayEquals(long[] expected, long[] actual, String message) *Asserts* that expected and actual long arrays are equal.	
static void	assertEquals(Object expected, Object actual, String message) *Asserts* that expected and actual are equal.	
static void	assertFalse(boolean condition, String message) *Asserts* that the supplied condition is not true.	
static void	assertIterableEquals(Iterable<?> expected, Iterable<?> actual, String message) *Asserts* that expected and actual iterables are deeply equal.	
static void	assertLinesMatch(List<String> expectedLines, List<String> actualLines) *Asserts* that expected list of Strings matches actual list.	
static void	assertNotEquals(Object unexpected, Object actual, String message) *Asserts* that expected and actual are not equal.	
static void	assertNotNull(Object actual, String message) *Asserts* that actual is not null.	

Assertion

Modifier and Type	Method and Description	JUnit 5.0.2 API
static void	assertNotSame(Object unexpected, Object actual, String message) *Asserts* that expected and actual do not refer to the same object.	
static void	assertNull(Object actual, String message) *Asserts* that actual is null.	
static void	assertSame(Object expected, Object actual, String message) *Asserts* that expected and actual refer to the same object.	
static <T extends Throwable> T	assertThrows(Class<T> expectedType, Executable executable, String message) *Asserts* that execution of the supplied executable throws an exception of the expectedType and returns the exception.	
static void	assertTimeout(Duration timeout, Executable executable, String message) *Asserts* that execution of the supplied executable completes before the given timeout is exceeded.	
static void	assertTrue(boolean condition, String message) *Asserts* that the supplied condition is true.	
static <V> V	fail(String message, Throwable cause) *Fails* a test with the given failure message as well as the underlying cause.	

- Plus de AssertThat pour Junit 5

Hypothèse et Assertion

- Les fonctions *assumeTrue* et *assertTrue* sont suffisantes pour exprimer la plupart des hypothèses et assertions.
- Prenons par exemple une méthode qui transforme un entier en double.

```
public class TransformationType {
  public static double transformation(int entier) {
    return (double) entier;
  }
}
```

- Données du test

```
public void testTransformation() {
  String donneeTest = "2"; //Donnée du test en chaine
  String resultatTest = "2.0"; //Retour de la méthode attendu en chaine
  int entreeTransformation = 0; //Donnée du test convertie en int
  double resultatAttendu = 1; //Retour de la méthode attendu en double

  double resultatTransformation; //Retour de la méthode obtenu
```

```
public void testTransformation () {
  String donneeTest = "2";
  String resultatTest = "2.0";
  int entreeTransformation = 0;
  double resultatAttendu = 1;
  double resultatTransformation;
//Hypotheses de test
  Assumptions.assumeTrue(donneeTest != null);
  Assumptions.assumeTrue(resultatTest != null);
  try {
    entreeTransformation = Integer.parseInt(donneeTest);
    resultatAttendu = Double.parseDouble(resultatTest);
  } catch (NumberFormatException e){
    Assumptions.assumeTrue(e == null);//Test annulé si levée d'exception
  }
  //Test
  resultatTransformation
    = TransformationType.transformation(entreeTransformation);
  Assertions.assertTrue(resultatAttendu == resultatTransformation);
}
```

```
public class TransformationType {
  public static double transformation(int entier){
    return (double) entier;
  } }
```

Les tests Unitaires

- Les tests sont organisés en trois phases :
  - L'initialisation
  - Le test
  - La finalisation

L'initialisation

- Le rôle de l'initialisation est de mettre en place tous les éléments nécessaires au test. Elle est optionnelle.
- Le tag @Before (->4.12) et @BeforeEach (5 ->)
  - Les méthodes précédées de ce tag sont lancées avant chaque test.
- Le tag @BeforeClass(->4.12) et @BeforeAll (5 ->)
  - Les méthodes précédées de ce tag sont lancées une seule fois avant le test.
  - Elles sont déclarées static.

Le test

- Exécution de l'unité testée. Seule partie obligatoire.
- Le tag @Test (pour toutes les versions)
- La fonction doit être une méthode d'instance (non static), public et qui ne renvoie aucun résultat.

La finalisation

- Le rôle de la finalisation est de libérer les ressources et de réinitialiser les données. Elle est optionnelle.
- Le tag @After (->4.12) et @ AfterEach (5 ->)
  - Les méthodes précédées de ce tag sont lancées avant chaque test.
- Le tag @ AfterClass(->4.12) et @ AfterAll (5 ->)
  - Les méthodes précédées de ce tag sont lancées une seule fois avant le test.
  - Elles sont déclarées static.

Test Junit (-> 4.12)

```
public class StandardTests {
  @BeforeClass
  public static void setUpBeforeClass() { }
  @Before
  public void setUp() { }
  @Test
  public void testMethode1() { }
  @Test
  public void testMethode2() { }
  @After
  public void tearDown() { }
  @AfterClass
  public static void tearDownAfterClass() { }
}
```


Test Junit5

```
public class StandardTests {
  @BeforeAll
  public static void setUpBeforeClass() { }
  @BeforeEach
  public void setUp() { }
  @Test
  public void testMethode1() { }
  @Test
  public void testMethode2() { }
  @AfterEach
  public void tearDown() { }
  @AfterAll
  public static void tearDownAfterClass() { }
}
```


Illustration (JUnit5)

```
public class ExempleTest {
  @BeforeAll
  public static void
setUpBeforeClass(){
    System.out.println("beforeAll");
  }
  @BeforeEach
  public void setUp(){
    System.out.println
      ("beforeEach");
  }
  @Test
  public void testMethode1() {
    System.out.println("methode 1");
  }
  @Test
  public void testMethode2() {
    System.out.println("methode 2");
  }
  @AfterEach
  public void tearDown(){
    System.out.println("afterEach");
  }
  @AfterAll
  public static void
tearDownAfterClass (){
    System.out.println
      ("afterClass");
  }
}
```

```
beforeAll
beforeEach
methode 1
afterEach
beforeEach
methode 2
afterEach
afterClass
```

Exemple (1/2)

```
public class TransformationType {
  public static double transformation(int entier) {
    return (double) entier;
  }
}
```

Classe à tester

```
public class TransformationTypeTest {
int entreeTransformation = 0; //Donnée du test en int
double resultatAttendu = 1; //Retour de la méthode attendu en double
  @BeforeEach
  public void setUp(){
    try (BufferedReader input = new BufferedReader(
                                  new FileReader("./donneesTest.txt"))) {
      entreeTransformation = Integer.parseInt(input.readLine());
      resultatAttendu = Double.parseDouble(input.readLine());
    } catch (IOException e) {
      Assumptions.assumeTrue(e == null, "Pas d'exception sur les I/O");
    }
}
```

Exemple (2/2)

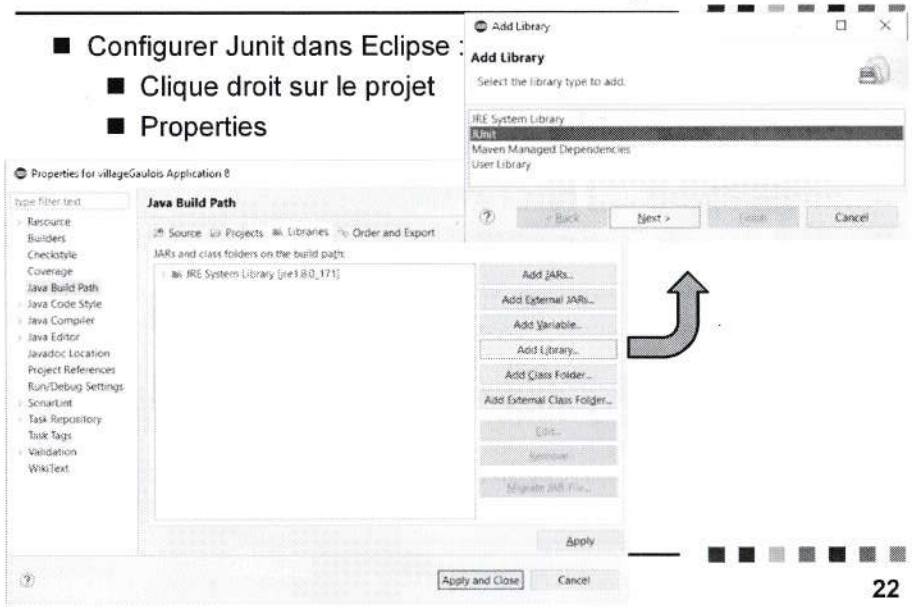
```
public class TransformationType {
  public static double transformation(int entier) {
    return (double) entier;
  }
}
```

Classe à tester

```
public class TransformationTypeTest {
int entreeTransformation = 0; //Donnée du test en int
double resultatAttendu = 1; //Retour de la méthode attendu en double
  @BeforeEach
  public void setUp(){...}
  @Test
  public void transformationTest(){
    double resultatTransformation =
              TransformationType.transformation(entreeTransformation);
    Assertions.assertTrue(resultatTransformation == resultatAttendu,
              "Transformation d'un int en double réussi");
  }
}
```

Test unitaire dans Eclipse

- Configurer Junit dans Eclipse :
  - Clique droit sur le projet
  - Properties



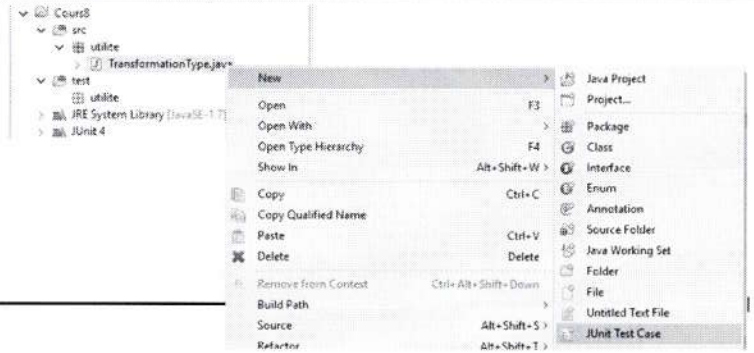
Test unitaire dans Eclipse

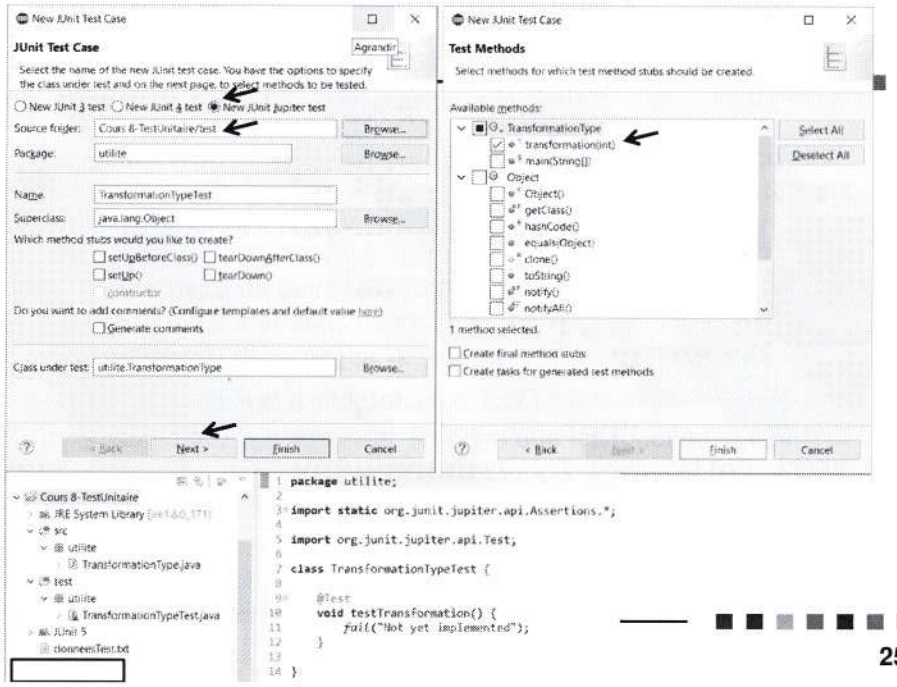
- Sélectionner la version

Test unitaire dans Eclipse

- Ci-dessous la classe à tester

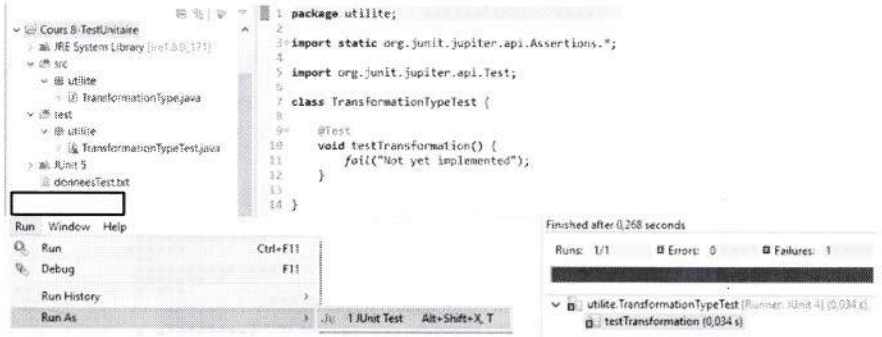
```
public class TransformationType {
  public static double transformation(int entier){
    return (double) entier;
  }
}
```





Test unitaire dans Eclipse

- Exécution du test



- Après écriture du test

Entity / Control / Boundary

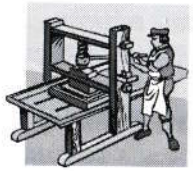
ECB

Auteur : CHAUDET Christelle

Auteur :
CHAUDET
Christelle

Notions de modélisation objets (1/3)

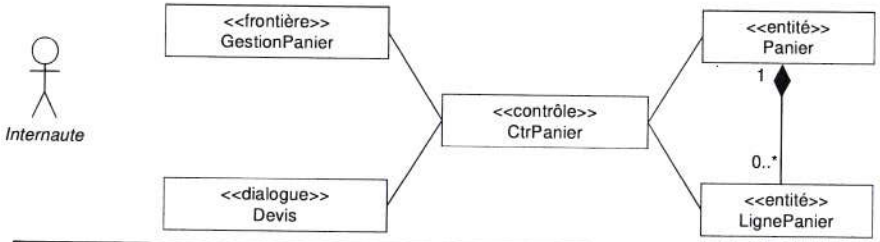
- **Les classes "métiers"** : Une classe métier ou entité métier est spécifique au domaine d'activité.
- Par exemple, si votre métier est l'édition, vos classes métiers seront
  - les livres,
  - les auteurs,
  - tout ce qui peut se rapporter à votre activité.



Auteur :
CHAUDET
Christelle

Notions de modélisation objets (2/3)

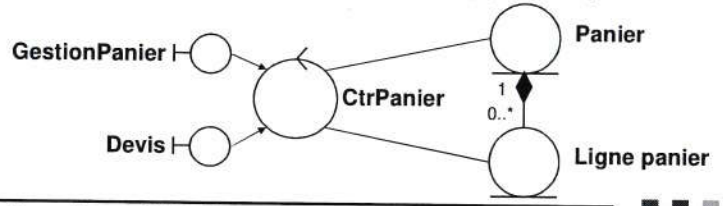
- Nous distinguerons ***trois types de classes d'analyse*** (comme proposé par I. Jacobson) :
  - les « ***frontières*** » qui représentent les moyens d'interaction avec le système,
  - les « ***contrôles*** » qui contiennent la logique applicative
  - les « ***entités*** » qui sont les objets métier manipulés.



Auteur :
CHAUDET
Christelle

Notions de modélisation objets (3/3)

- En ce qui concerne les attributs et les opérations dans les classes d'analyse pour :
  - Les « frontières » vont posséder des attributs et des opérations. (champs de saisie ou des résultats).
  - Les « contrôles » vont seulement posséder des opérations. (logique de l'application, règles métier, comportements du système informatique).
  - Les « entités » vont seulement posséder des attributs. (informations persistantes de l'application).



Auteur :
CHAUDET
Christelle

Différence MVC / ECB

- Nous avons deux approches qui se ressemblent sur la séparation de la vue et du domaine métier.
- Les deux approches :
  - parlent de la « vue » ou de « frontière » pour représenter la communication entre l'utilisateur et le système.
  - ont des entités afin de stocker les données,
  - ont un contrôleur qui sépare les deux parties précédentes
- Détaillons les deux approches.

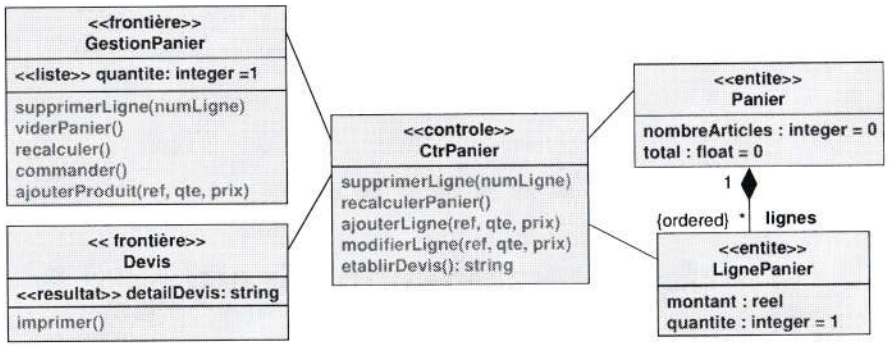
Auteur :
CHAUDET
Christelle

MVC

- **Modèle** : cette partie intègre les informations du système. Le rôle de cette partie est de récupérer les informations dans un Système de Gestion de Base de Données (SGBD) ou autres sources de données. Ces données seront traitées par la partie Contrôleur.
- **Vue** : cette partie se concentre sur la présentation. Elle ne contient pas de traitements sophistiqués et se contente de récupérer des variables et de les afficher.
- **Contrôleur** : cette partie contient la logique du code et contrôle l'exécution de l'application. Le contrôleur joue le rôle d'intermédiaire entre les parties modèles et vues. Il récupère des données de la partie modèle, il les analyse et déclenche les traitements selon celles-ci. Il envoie les informations à afficher à la partie vue.

Auteur :
CHAUDET
Christelle

MVC



Auteur :
CHAUDET
Christelle

ECB

- Les classes stéréotypées **Entity** correspondent aux classes du domaine. Elles sont généralement persistantes, survivent à l'exécution d'un cas d'utilisation. La durée de vie est plus longue que l'exécution d'un cas d'utilisation. Elles peuvent alors intervenir dans plusieurs cas d'utilisation. En plus de données métier, elles peuvent contenir des méthodes métier.
- Les classes stéréotypées **Boundary** représentent les objets frontières du système. Elle supporte les interactions entre le système et les acteurs. Elles ne contiennent ni traitement, sauf contrôle de 1er niveau, ni donnée. D'un point de vue méthodologique, ces classes sont issues de la maquette du système. Elles seront implémentées sous forme d'écrans web dans le cas de sites web, ou d'écrans, de manière plus générale.

Auteur :
CHAUDET
Christelle

ECB

- Les classes stéréotypées **Control** prennent en charge le traitement qui gère la dynamique du système lors de l'exécution d'une application. Elles contiennent les règles applicatives et les isolent des classes entités et les classes frontières. Intermédiaires entre les classes frontières et les classes entités, elles contrôlent les informations saisies, et redirigent les requêtes vers les classes métier. On crée dans un premier temps une classe control par cas d'utilisation qui ne contient que le traitement nécessaire à l'exécution de ce cas.

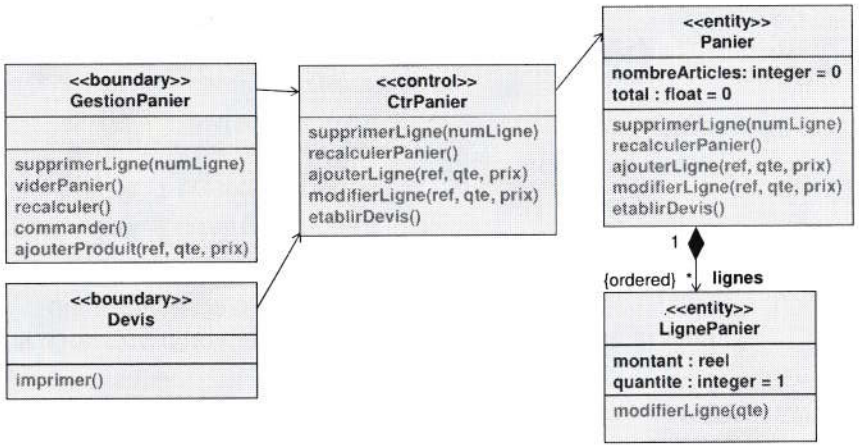
Auteur :
CHAUDET
Christelle

Différence MVC / ECB

- Il existe certaines similitudes entre ECB et le patron MVC Modèle Vue Controleur :
  - les entités appartiennent au modèle
  - les vues sont des éléments des frontières.
- Cependant, le rôle du contrôle ECB est très différent de celui du contrôleur MVC :
  - il englobe également la logique métier des cas d'utilisation, tandis que le contrôleur MVC traite les entrées utilisateur qui seraient de la responsabilité de la frontière dans le schéma ECB.

Auteur :
CHAUDET
Christelle

ECB



Auteur :
CHAUDET
Christelle

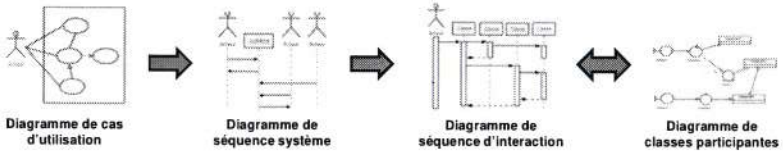
Démarche de modélisation

- Nous utiliserons l'approche ECB dans nos TP :
  - les « ***dialogues*** » ou « **boundary** » qui représentent les moyens d'interaction avec le système,
  - les « ***contrôles*** » ou « **control** » qui permette de découpler la vue du métier.
  - les « ***entités*** » ou « **entity** » qui sont les objets métiers manipulés contenant les méthodes permettant de gérer leur état.

Auteur :
CHAUDET
Christelle

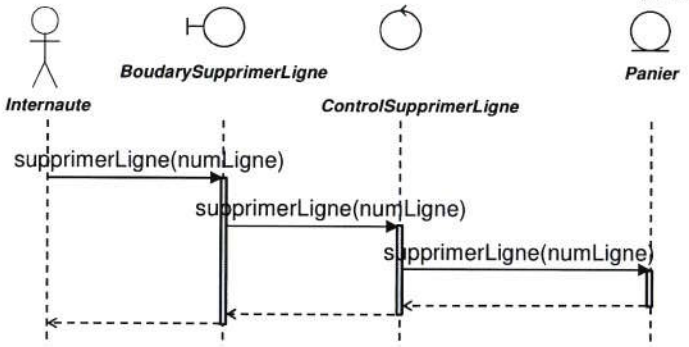
Démarche de modélisation

- Identification des principaux cas d'utilisation
- Création d'un diagramme de séquence système par cas d'utilisation
- Création d'un diagramme de séquence détaillé et d'un diagramme de classes participantes par cas
  - Particularités du diagramme :
    - 1 control par cas
    - 1 boundary par cas et par acteur
  - Adapté ce nombre par la suite



Auteur :
CHAUDET
Christelle

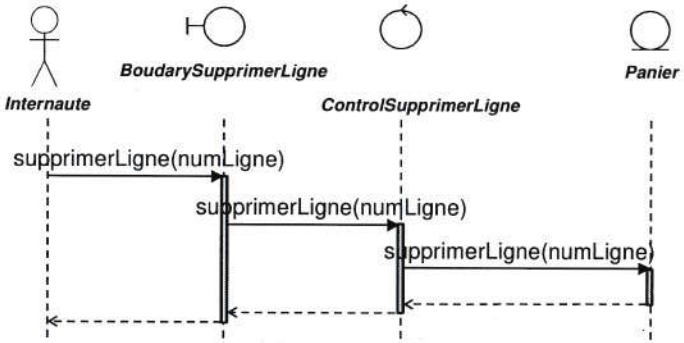
Démarche de modélisation



- Entre l'acteur et le boundary vous ne devez avoir que les interactions entre l'acteur et le système autrement dit vous devez retrouver les échanges décrits dans le diagramme de séquence système.

Auteur :
CHAUDET
Christelle

Démarche de modélisation



- Entre le boundary et le control vous ne devez échanger que des types primitifs d'UML (ex: string, boolean, date, double…) et des énumérés. Seul le control peut échanger avec les entités.

Auteur :
CHAUDET
Christelle

Démarche de modélisation

Représentation		Relation avec			
Rôle	symbole	Acteur	Frontière	Contrôle	Entité
Acteur		Oui	Oui	Non	Non
Frontière		Oui	Non	Oui	Non
Contrôle		Non	Oui	Oui	Oui
Entité		Non	Non	Oui	Oui