Les tests en orienté objet

J. Paul Gibson

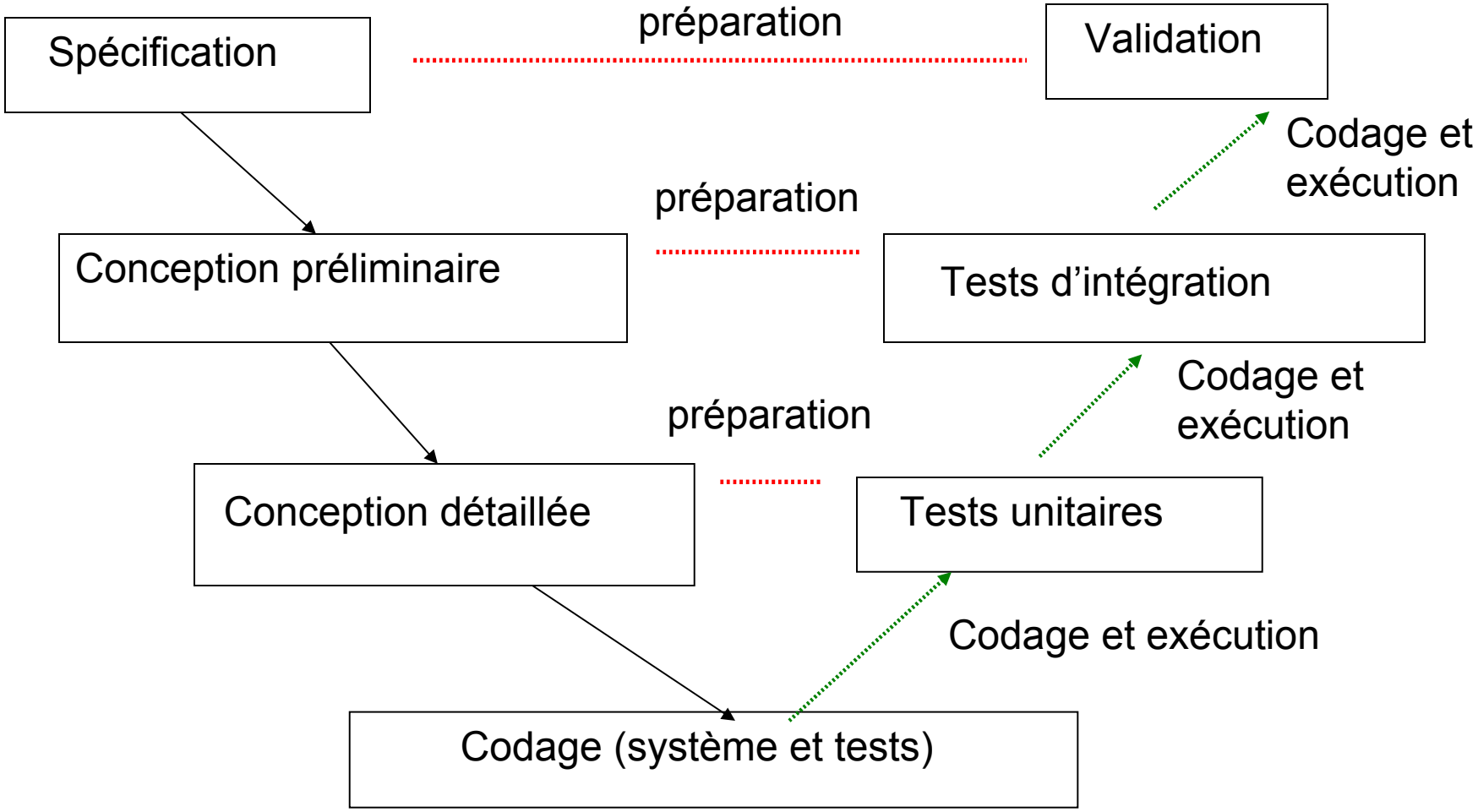
paul.gibson@telecom-sudparis.eu

http://www-public.telecom-sudparis.eu/~gibson/

Bureau A 207, Le département LOgiciels-Réseaux

Thanks to: Christian Bac, Denis Conan & Jean-Luc Raffy

Le cycle en V


Spécification
préparation
Validation
Codage et exécution
préparation
Conception préliminaire
Tests d'intégration
Codage et exécution
préparation
Conception détaillée
Tests unitaires
Codage et exécution
Codage (système et tests)

Les tests en orienté objet

I.Rappels sur les tests

II. Comment utiliser les modèles OO en UML?

III. Étude de cas – médiathèque

A) Préparation
- Tests de validation
- Tests d'intégration
- Tests unitaires

B) Codage et Exécution
- Tests unitaires avec Junit
- Tests d'intégration
- Tests de validation

Testing Consistency (not uniquely OO)

We need **consistency between documentation/models:**

- Natural Language Text (English/French)
- (UML) models
- (Java) Code
- Comments in (Java) Code

Develop tests in parallel with code to avoid inconsistency

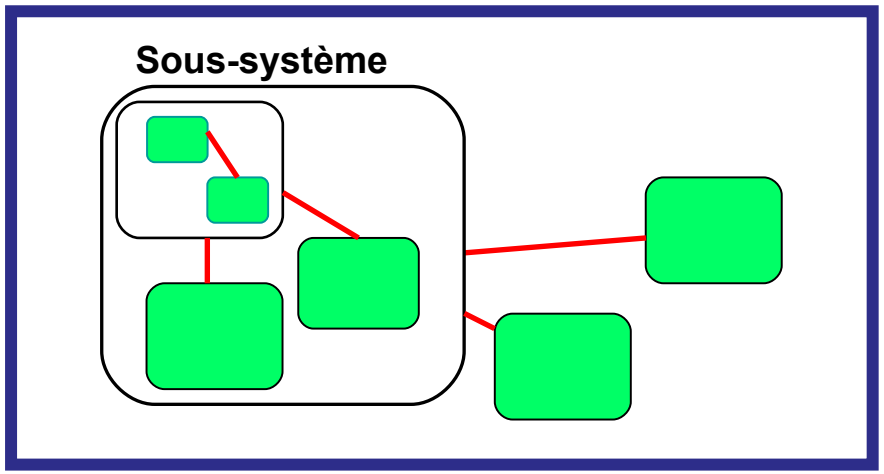
NOTE: testing after all the code is developed is usually a bad idea, but it is better than no testing at all!

Niveaux de test - pour chaque (sous)système

- Tests de validation
- Tests d'intégration
- Tests unitaires

Préparation

exécution



UML – les diagrammes

Spécification

- Diagramme des cas d'utilisation
- Niveau Sous-Système (Analyse et conception)
  - test des associations, des agrégations (diagramme de classes)
  - test de séquences (diagramme de séquence et de communications)
  - test des exceptions contrôlées
- Diagramme des classes (ébauche)
- Diagrammes de séquence/communications

Tests de validation

Tests d'intégration

Conception détaillée

- Classes détaillées
- Diagrammes de machine à états

Tests unitaires

How to Use UML

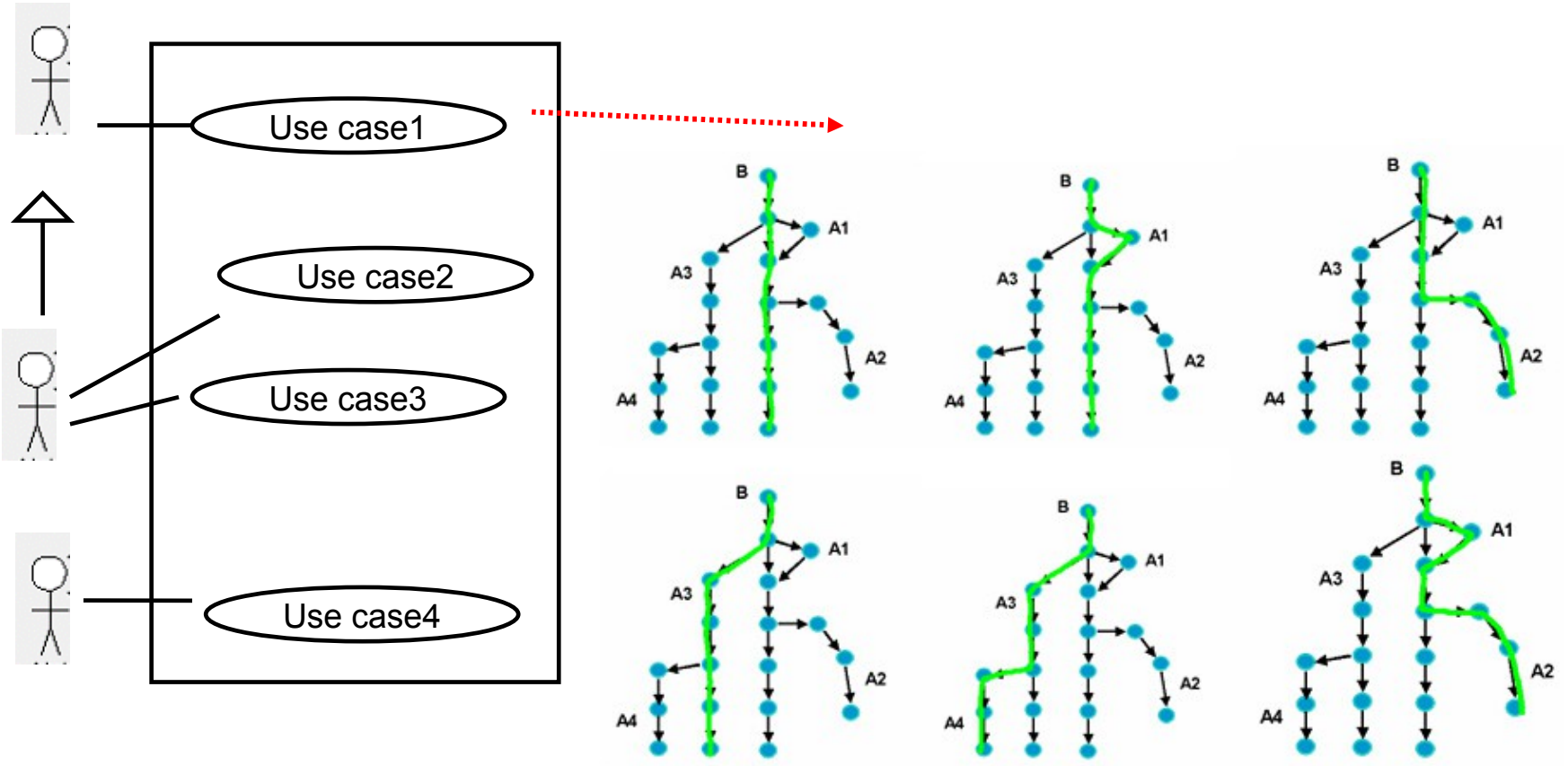
Tests should be derived from the requirements specification (in UML).

The UML diagrams should help us because:

- provided the UML is validated we have a good chance of testing the system against what is required
- the structure of the UML should correspond to the structure of the developed code, so we can re-use this design structure to structure our tests.

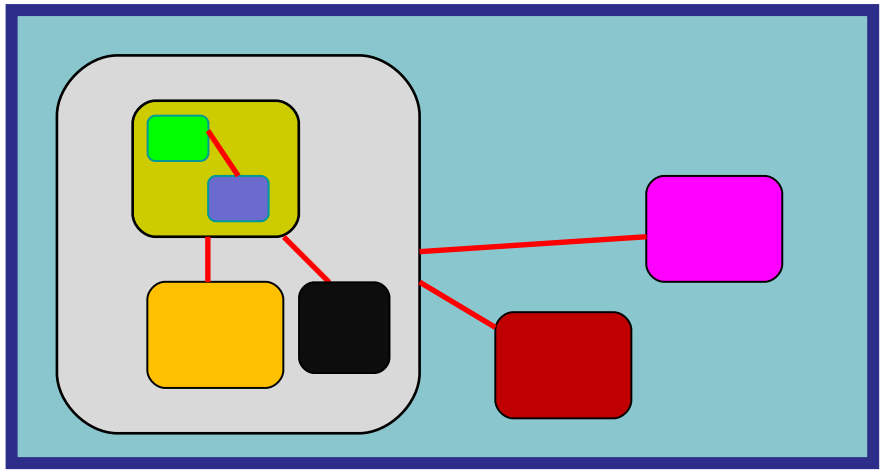
Validation

Use Case Diagrams – for each use case examine possible scenarios, and choose a subset of alternative paths for **testing**. For example:

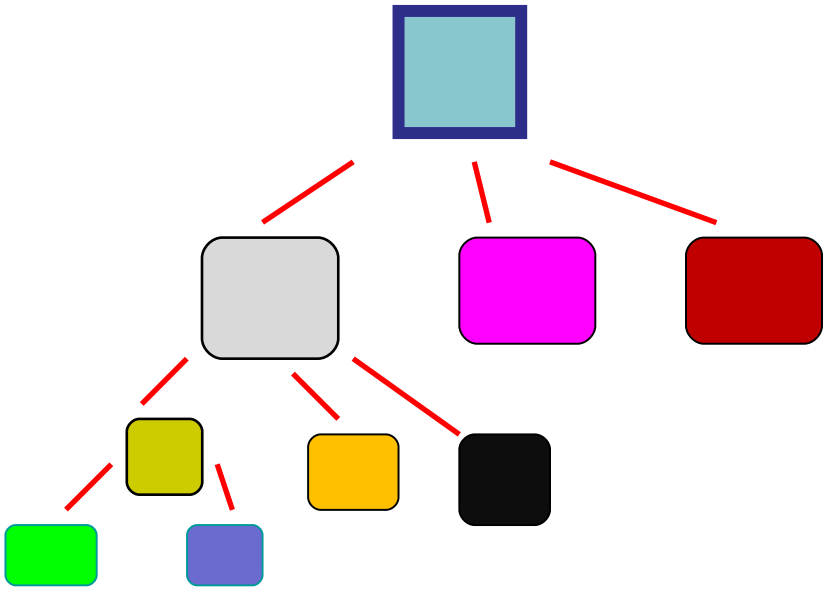


Integration : composition tree

The test sequence can be decided by looking at the *tree-like* structure of composition hierarchies. For example:



Système



Composition tree

Big Bang Testing : *all at once* at the system interface
Top-Down Testing : does not require all lower level components to be complete
Bottom-Up Testing : does not require all higher level components to be complete

Les tests unitaires

- Il faut tester les invariants de chaque classe
- Il faut tester la fonctionnalité de chaque méthode de chaque classe
- Des classes avec des contraintes séquencielles sur l'activation des méthodes et leurs clients peuvent avoir des erreurs de séquencement.

  => Le comportement requis doit être testé en utilisant un modèle de machine à états

Comment Tester la Médiathèque?

	Présentation Séquence
A) Préparation	
1. Tests de validation, e.g.: `Emprunter`	1st
2. Tests d'intégration, e.g.: `Emprunter`	3rd
3. Tests unitaires, e.g.: `Document`	5th
B) Codage et Exécution	
4. Tests unitaires avec JUnit	6th
5. Tests d'intégration	4th
6. Tests de validation	2nd

Comment Tester la Médiathèque?

Préparation

1. **Tests de validation, eg: `Emprunter`**
2. Tests d'intégration, eg: `Emprunter`
3. Tests unitaire, eg: `Document`

B) Codage et Exécution

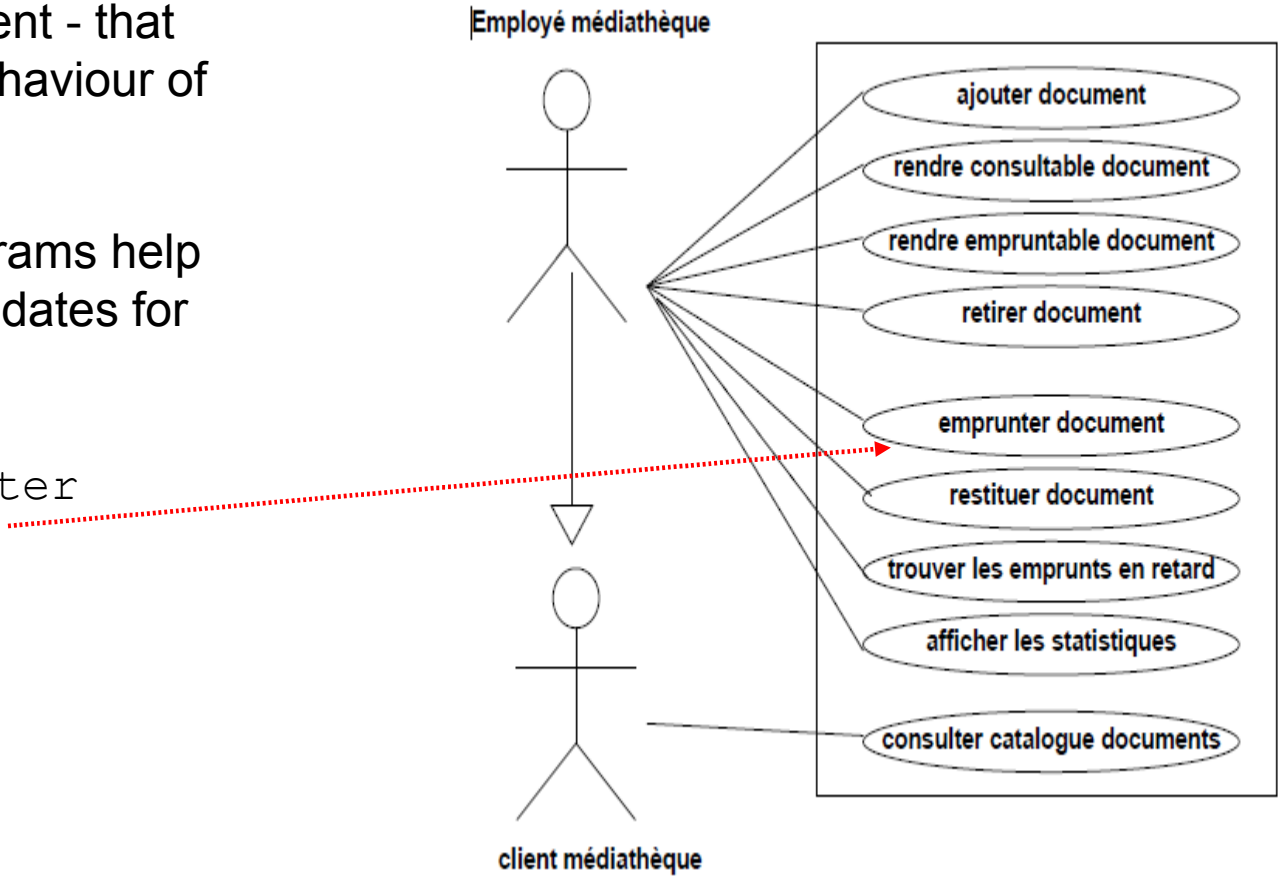
4. Tests unitaire avec JUnit
5. Tests d'intégration
6. Tests de validation

Emprunter

A validation test is a black box test - usually done by the client - that validates the (partial) behaviour of the whole system.

The UML use case diagrams help us to identify good candidates for validation tests.

We will test the `emprunter` functionality



Emprunter

Données d'entrée

client: peut être inscrit ou non;

emprunts: déja effectués par le client

- existe-t-il un emprunt en retard ?
- le nombre de documents empruntés correspond-il au nombre maximum de ce client ?

document:

- existe?
- empruntable ou consultable?
- déjà emprunté ou disponible?

Données de sortie

`Emprunt` accepté ou refusé.

Remarque: la définition des jeux de tests de validation pour le cas d'utilisation `emprunter document` permet de soulever au moins les questions suivantes (à poser au client):

- un abonné qui n'est pas à jour de sa cotisation peut-il tout de même emprunter un document?
- doit-il être considéré comme un client au tarif normal tant qu'il n'a pas renouvelé son abonnement?
- ou doit-il se réabonner avant de pouvoir emprunter un document?

D'une manière générale, la préparation des jeux de tests de validation permet de lever les ambiguïtés et les vides de la spécification.

NOTE : Plus les tests sont préparés tôt et moins les corrections coûtent chères

Emprunter

Table de décisions

1 : lorsque la condition exprimée sur la ligne est vraie,
0 : lorsque la valeur de la condition est fausse,
x : lorsque la valeur de la condition n'influence pas le résultat.

		1	2	3	4	5	6	7
Client	inscrit	0	1	1	x	x	x	1
Emprunts du client	sans retard	x	0	1	x	x	x	1
	< max	x	x	0	x	x	x	1
Document	existant	x	x	x	0	1	1	1
	empruntable	x	x	x	x	0	1	1
	disponible	x	x	x	x	x	0	1
Emprunt accepté		0	0	0	0	0	0	1

Emprunter

To illustrate the testing process, we will treat the first 2 test cases

Test 1 - Client n'est pas inscrit

Test Code Steps:

1. Intialise dummy Mediatheque
2. Check state of current Mediatheque (including statistics)
3. Attempt to « emprunter » a document for a client who does not exist
4. Check state of current Mediatheque (including statistics)

Emprunter

Test 2 - Client possède un emprunt qui est en retard

Test Code Steps:

1. Intialise dummy Mediatheque
2. Check state of current Mediatheque (including statistics)
3. Authorise « emprunter » of a document for a client
4. Advance the date so that the previous « emprunt » is now past its deadline
5. Attempt to « emprunter » by the same client before they return the document that is past its deadline
6. Check state of current Mediatheque (including statistics)

Comment Tester la Médiathèque ?

A) Préparation

1) Tests de validation, e.g.: `Emprunter`
2) Tests d'intégration, e.g.: `Emprunter`
3) Tests unitaires, e.g.: `Document`

B) Codage et Exécution

4) Tests unitaires avec JUnit
5) Tests d'intégration
6) **Tests de validation**

Validation Tests

Even if our system is not yet completely developed, we can write the code for the validation tests.

For this example, we will code the validation test as a **JUnit** test on the mediatheque class.

NOTE: A validation of the overall system is often known as an acceptance test; and can be thought of as a system unit test.

Test 1 - Client n'est pas inscrit

```
/**
 * Document TEST 1<br>
 * Client n'est pas inscrit
 */
@Test (expected= OperationImpossible.class) // we expect an exception
public void clientPasInscrit( ) throws OperationImpossible,
InvariantBroken{

 m1.emprunter("nom", "prenom", "Test_code1");

}
```

To see why we expect an exception we must look at the **`setup`** code

```
@Before
public void setUp() throws Exception {
// un test de validation est un test unitaire sur la classe Mediatheque
m1 = new Mediatheque("mediatheque test");
Genre g = new Genre("Test_nom1");
Localisation l = new Localisation("Test_salle1","Test_rayon1");
Document d1 = new Video("Test_code1",l, "Test_titre1", "Test_auteur1",
                        "Test_annee1" ,g, "Test_duree1", "Test_mentionLegale1");
Document d2 = new Video("Test_code2",l, "Test_titre2", "Test_auteur2",
                        "Test_annee2" ,g, "Test_duree2", "Test_mentionLegale2");

m1.ajouterDocument(d1); m1.metEmpruntable("Test_code1");
m1.ajouterDocument(d2); m1.metEmpruntable("Test_code2");

CategorieClient cat = new CategorieClient("Test_Cat1", 10, 1.5, 2.5, 3.5, true);
Client c1 = new Client("Test_Client_Nom1", "Test_Client_Prenom1",
                       "Test_Client_Address1",cat);
 m1.inscrire(c1);
Client c2 = new Client("Test_Client_Nom2", "Test_Client_Prenom2",
                       "Test_Client_Address2",cat);
}

@After
public void tearDown() throws Exception {m1 = null;}
```

Test 2 - Client n'est pas sans retard

```
/**
 * Document TEST 2<br>
 * Client n'est pas sans retard
 */
@Test (expected= OperationImpossible.class) //  we expect an exception
public void clientAvecRetard( ) throws OperationImpossible,
InvariantBroken{

m1.emprunter("nom1", "prenom1", "Test_code1");

Datutil.addAuJour(7);
Datutil.addAuJour(7);

m1.emprunter("nom1", "prenom1", "Test_code2");

}
```

Comment Tester la Médiathèque ?

A) Préparation

1. Tests de validation, e.g.: `Emprunter`
2. **Tests d'intégration, eg: `Emprunter`**
3. Tests unitaires, e.g.: `Document`

B) Codage et Exécution

4. Tests unitaires avec JUnit
5. Tests d'intégration
6. Tests de validation

Emprunter

The operation « emprunter » requires co-ordination between the

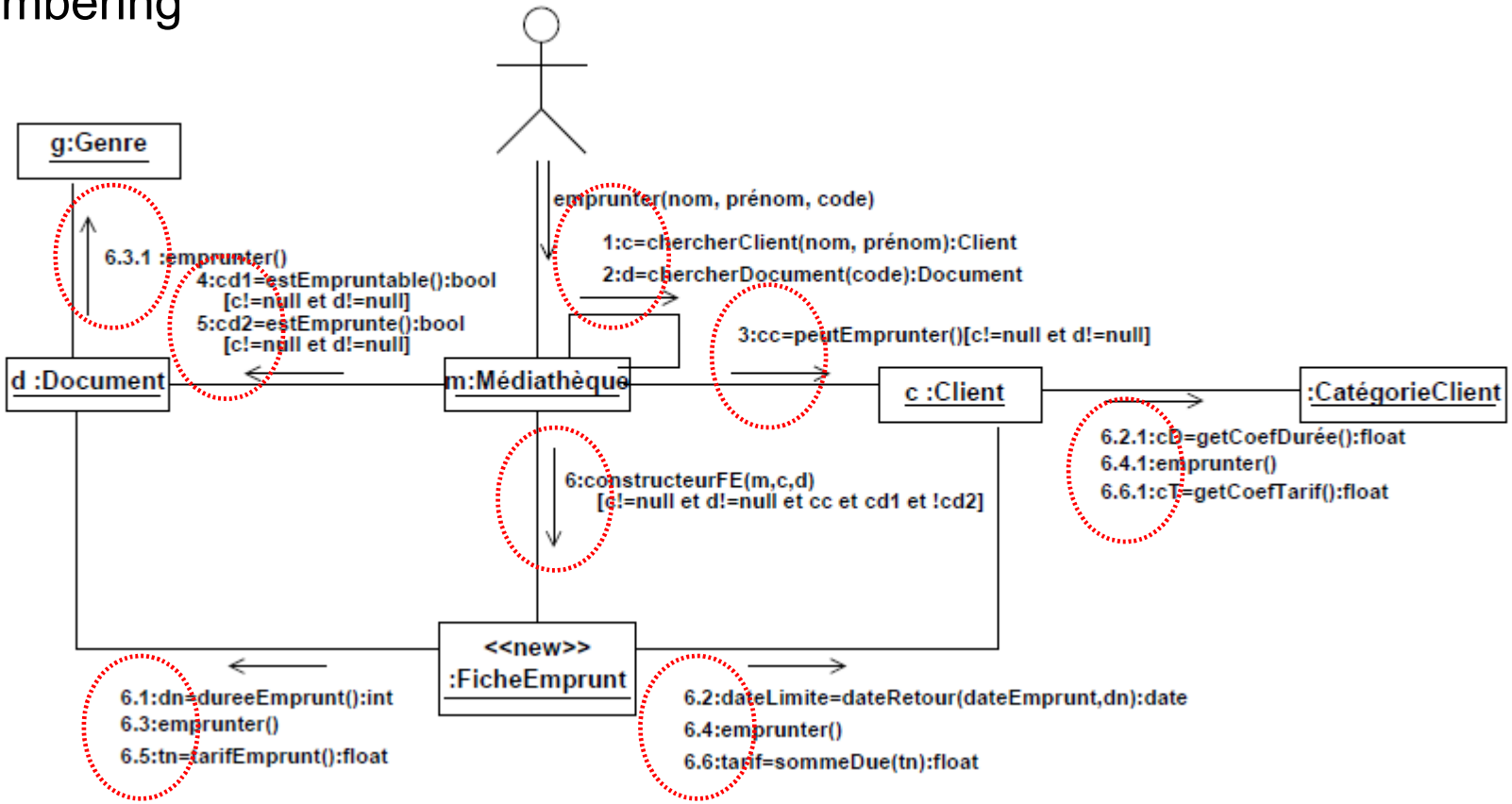
- `client`,
- `mediatheque`,
- `document`, and
- `ficheEmprunt` objects

We wish to verify that the traces (of communication) between the objects involved in the collaboration, as specified in UML, are executed by the implementation (in Java), following the specified *temporal ordering*.

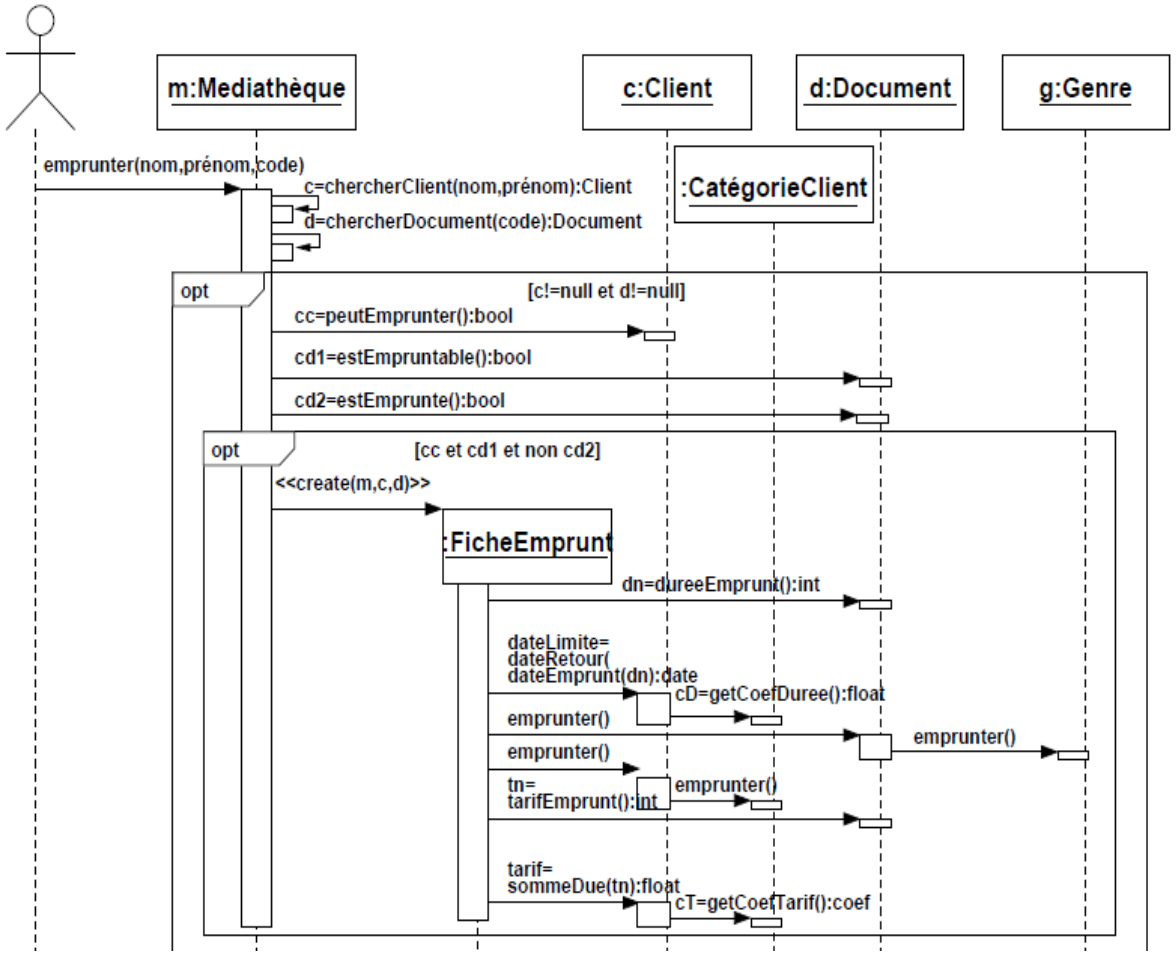
These tests can be derived from the communications and/or sequence diagrams …

Emprunter

In the communications diagram, the temporal order is specified by the numbering



`Emprunter`



1
2
3
4
5
6
6.1
6.2
6.2.1
6.3
6.3.1
6.4
6.4.1
6.5
6.6
6.6.1

The co-ordination between the `Client` and the `Document`:

Integration Test 1
An « *emprunt* » is not authorised because the document is not « empruntable »

Integration Test 2
An « *emprunt* » is not authorised because the document is « emprunté »

Integration Test 3
***Emprunt* is authorised**

`Emprunter`

Integration Test1 - An « emprunt » is not authorised because the document is not empruntable

- Construct a `document`, and make it `not Empruntable`
- Construct a `client`
- Construct a `FicheEmprunt` for the `client` and `document`
- Check that:
  1. the system handles the exceptional case in a meaningful way
  2. the `client` and `document` states/statistics have not been changed

`Emprunter`

Integration Test 2 Design: An « emprunt » is not authorised because the document is « *emprunté* »

- Construct a `document`, which is e`mpruntable` and `emprunté`
- Construct a `client`
- Construct a `FicheEmprunt` for the `client` and `document`
- Check that:
  1. the system handles the exceptional case in a meaningful way
  2. the `client` and `document` states/statistics have <u>not</u> been changed

`Emprunter`

Integration Test 3: Emprunt is authorised

- Construct a `document`, which is `empruntable` and `not emprunté`
- Construct a `client`
- Construct a `FicheEmprunt` for the `client` and `document`
- Check that the system handles the exceptional case in a meaningful way
- Check that:
  1. the `tarif` and `duree` des emprunts are as required
  2. the `client` and `document` states/statistics have been updated as required

Comment Tester la Médiathèque ?

A) Préparation

1. Tests de validation, e.g.: `Emprunter`
2. Tests d'intégration, e.g.: `Emprunter`
3. Tests unitaires, e.g.: `Document`

B) Codage et Exécution

4. Tests unitaires avec Junit
5. **Tests d'intégration**
6. Tests de validation

`Emprunter`

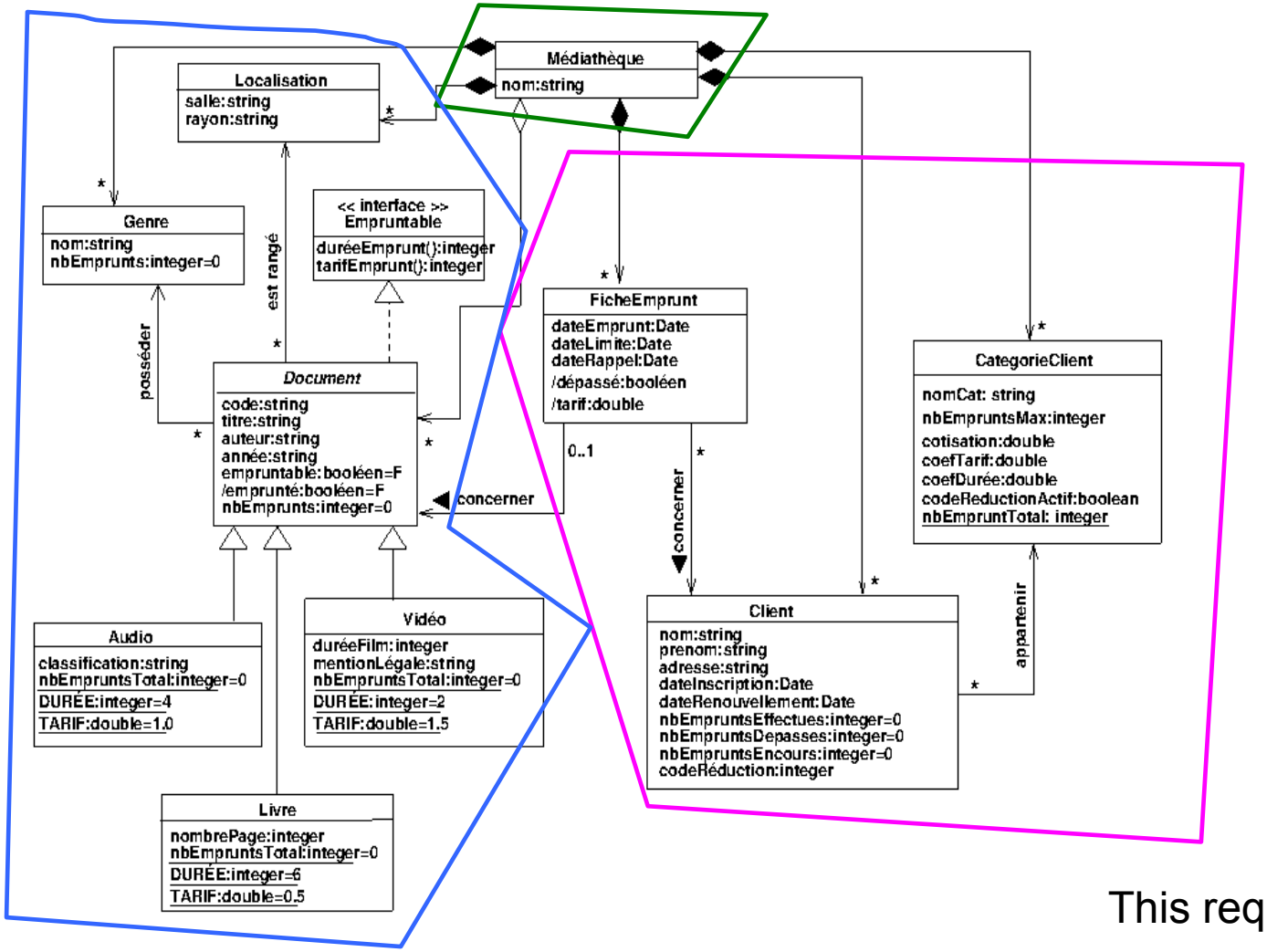
Integration Test1 Code: Verify correct co-ordination by FicheEmprunt

- Construct a `document`, and make it `Empruntable`
- Construct a `client`
- Construct a `FicheEmprunt` for the `client` and `document` using a dummy `mediatheque`
- Check that the `tarif` and "`duree des emprunts`" values for the `FicheEmprunt` are as required
- Check that the `client` and `document` states have been updated correctly

We should do the same for integration tests 2 and 3

Emprunter

Integration Tests: Typical/Example Development Status

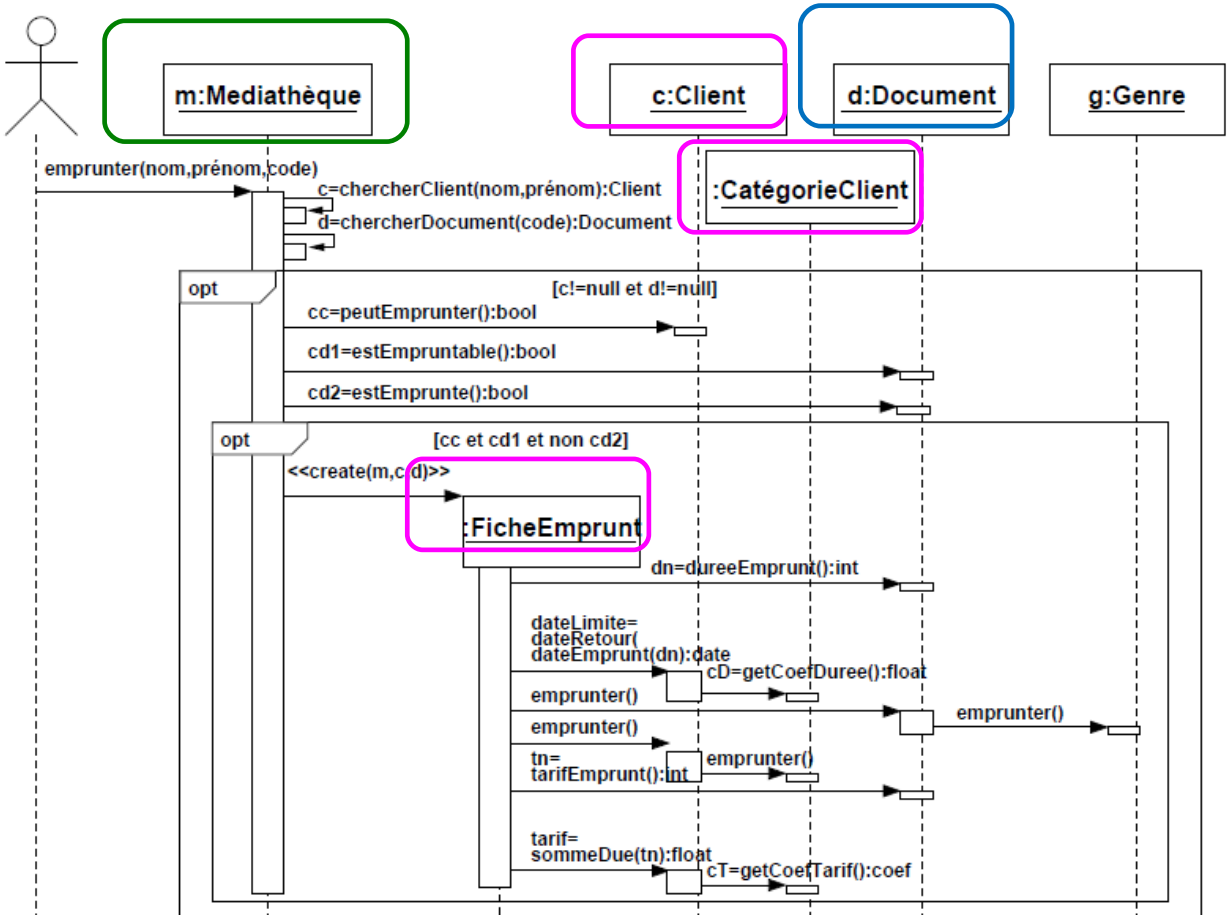


For example:
Completed
Partial
Not yet implemented

This requires further analysis

Emprunter

Integration Test 1



Analysis: it is too soon, in this example, to code the integration tests

Comment Tester la Médiathèque ?

A) Préparation

1. Tests de validation, e.g.: `Emprunter`
2. Tests d'intégration, e.g.: `Emprunter`
3. **Tests unitaires, e.g.: `Document`**

B) Codage et Exécution

4. Tests unitaires avec JUnit
5. Tests d'intégration
6. Tests de validation

The `Document` class

We will use the following UML diagrams to « derive » our unit tests for the `Document` class:

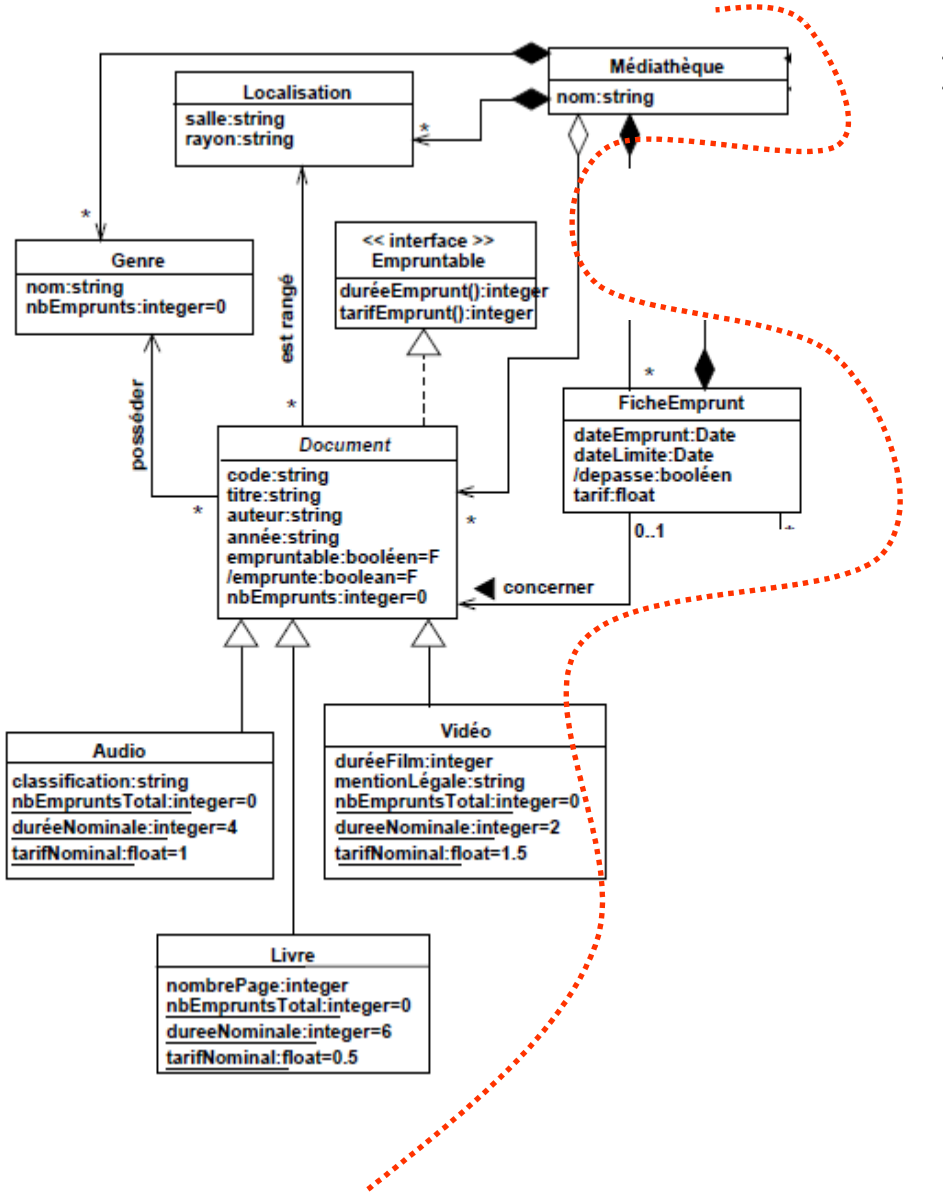
- Class Diagrams
- State machine diagrams

We may also need to use the original natural language text.

We should not have to examine the Java code `Document.java`, but can just call the code using the `Document.class` file – this is blackbox testing.

We may need access to the documentation for the code in order to understand code properties that are not specified in the UML models.

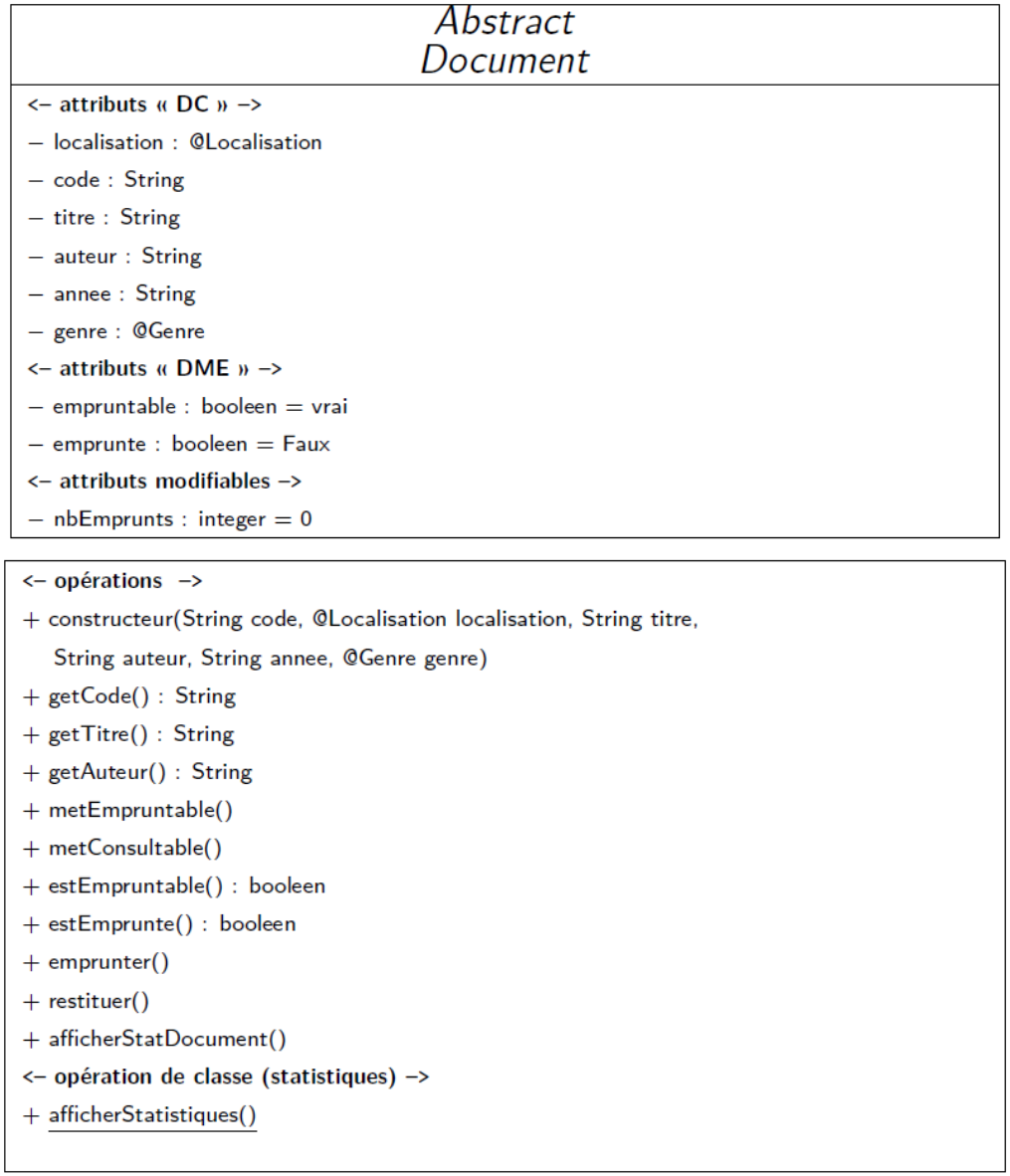
NOTE: If documentation is poor then we will have to examine the code in order to be able to guarantee that our tests compile and execute correctly.



The `Document` class

The high-level class diagram can be partitioned so that we abstract away from the classes that are not directly « connected » to the Document

The `Document` class



We test only the public attributes and operations/methods – following the blackbox approach

We should first examine the ***safety invariant*** … if it is not in the UML model then we need to add it.

The `Document` class

<– attributs « DME » –>

– empruntable : booleen = vrai

– emprunte : booleen = Faux

<– attributs modifiables –>

– nbEmprunts : integer = 0

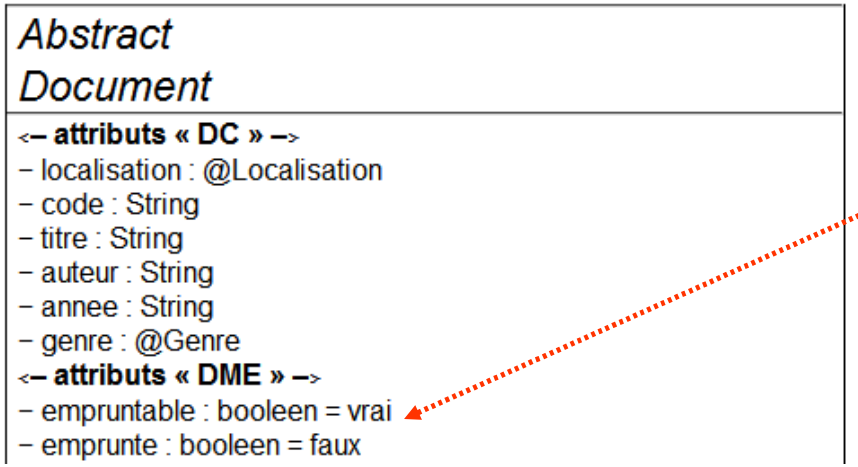
The ***invariant*** property is defined on the attributes of the class in order to say when a `Document` instance/object is in a SAFE state, i.e. a state which is meaningful/allowable.

Here, the invariant should « include »:
Emprunté => (empruntable
AND nbEmprunts >=0)

A formal interpretation that needs validation with the client

We should link this (invariant) requirement to the original text, where possible:
*« La médiathèque contient un certain nombre de documents disponibles à la consultation **ou** à l'emprunt »*

The `Document` class



BUT, these are <u>not</u> public

So we have a **test design choice** –

1) extend the `Document` class by adding a public `invariant` method:

- (a) creating a new subclass (and make attributes protected), or
- (b) editing/updating the `Document` class

OR

2) use public methods – directly in the testing code - that are equivalent to testing the invariant and are guaranteed not to change the state of the object (`Document`) being tested (e.g. `estEmpruntable` and `estEmprunte`.)

The `Document` class

All design decisions involve compromise.

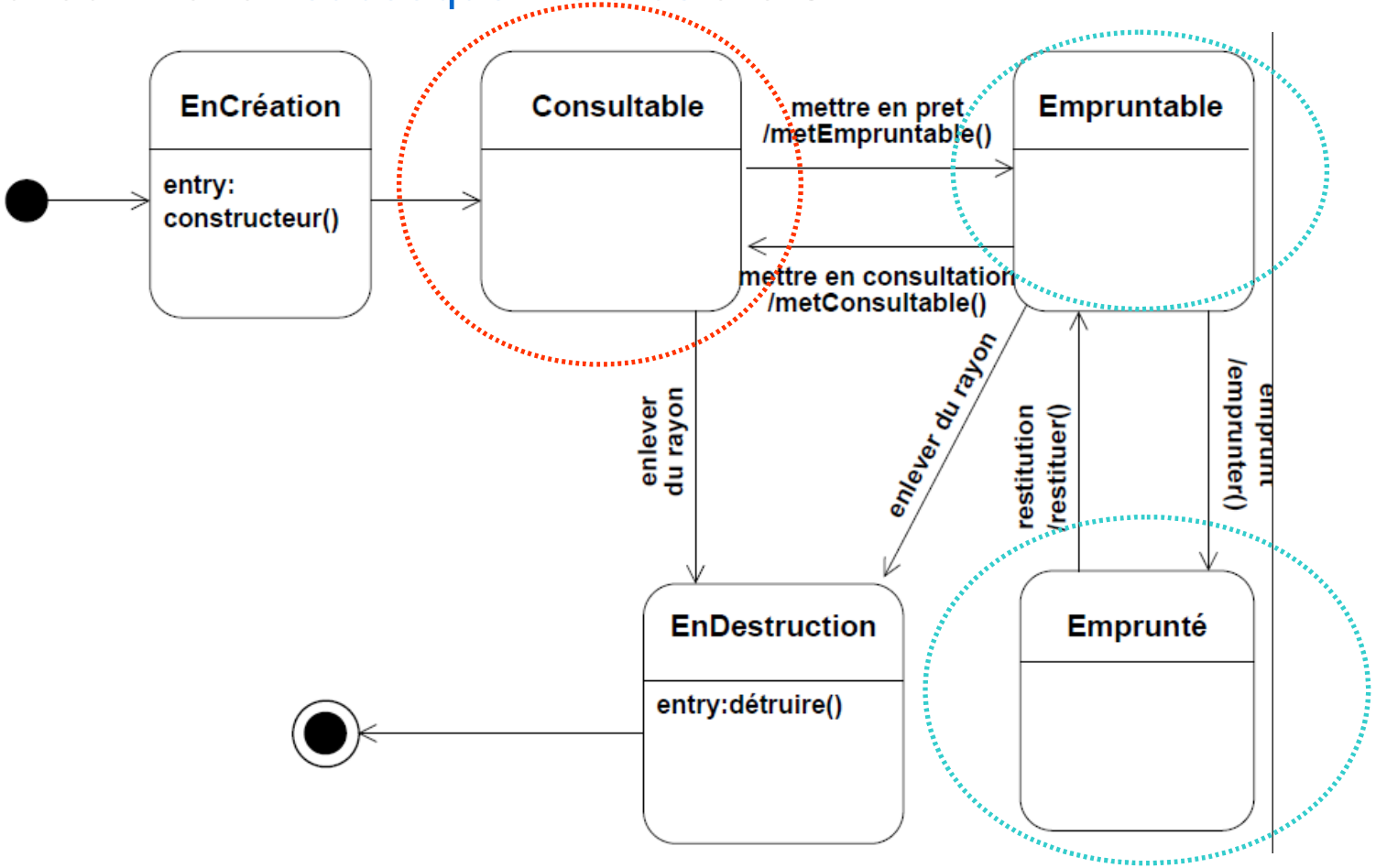
QUESTION: Can you see the advantages/disadvantages of each option for specifying the invariant property?

In this example, we choose to pursue option 1 (b) - Edit the `Document` class, because -

- It is good practice in OO development to have invariants specified for all classes, and
- It is the « simplest » coding option for « Java beginners »

Document

Now, let's consider the dynamic behaviour specified by the state machine. The first thing is that the initial state must be SAFE. Then we check that all subsequent states are SAFE.



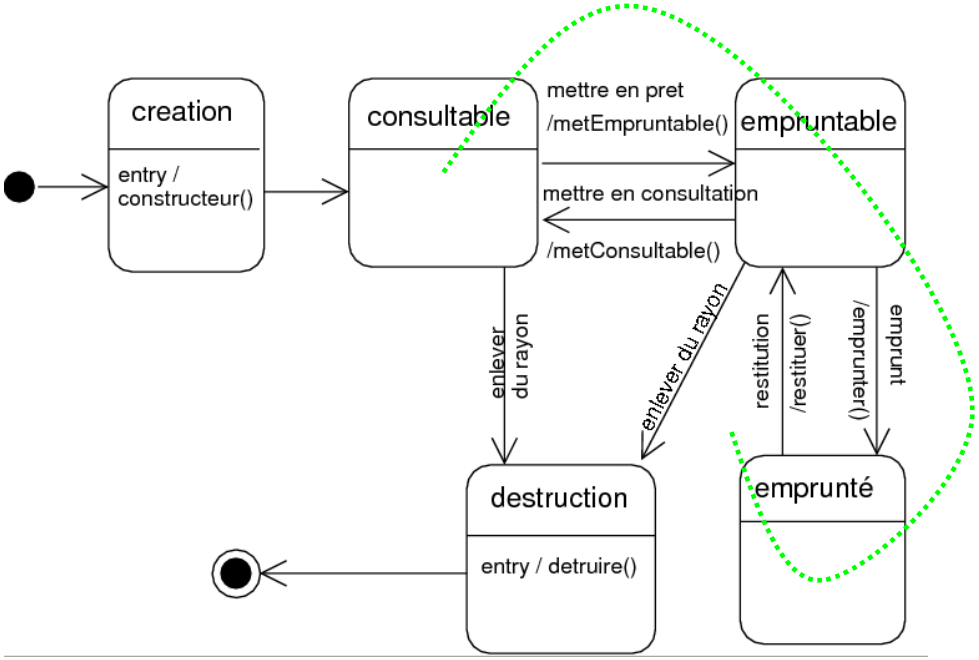
Document

Test1: check that the construction of a `Document` respects the invariant

1. Create a `DocumentTest` class with a single `main` method for testing a concrete instance of a `Document` subclass –
   - `Video`, or
   - `Audio`, or
   - `Livre`
2. Create a `test` method that gets called in the `main` method.
3. Ensure that the test can be checked –
   - output results to screen, or
   - print results to file, or
   - include test oracle code that knows what the results should be and *asserts* these when the test executes, automating the test process

Document

Test 2: Check all reachable states to be SAFE, i.e. respect the invariant



Code Design:
Complete state coverage can be achieved by a **single execution trace** after creation –

metEmpruntable()
emprunter()

NOTE: Testing all states respect the invariant does not check the *correct temporal behaviour* of the Document class …We will see this problem later with **Test4**

Document

We need to write at least one test for each public operation/method of the `Document` class.

For conciseness, we illustrate this by looking at the single `Emprunter` operation

> ***« emprunter** : … Pour tous les documents, les statistiques sont mises à jour à savoir : le nombre d'emprunts de ce document, le nombre d'emprunts de ce type de document et le nombre d'emprunts total »*

Test3 - check statistics are updated correctly

Code design steps: « emprunt » a document 5 times and, each time, check that the individual document « statistiques » are incremented. At the end of the loop, check that the total « statistiques » have increased by 5.

Document

Test4: correct temporal sequencing

1. **Check that we can only « restituer » a document after it has been « emprunté »**

2. **Check that we can only « mettre en consultation » if the document is « empruntable »**

Code Design: such sequences of events should produce exceptions

Document

Final Test

After testing all other important temporal properties of the Document class, we should conclude the `Document` unit tests by testing how the constructor method(s) behaves when one tries to construct a `Document` using « *invalid* » component parameters. For example:

- `Null Genre` or `Localisation` or …
- `UNSAFE Genre` or `Localisation` or …

Final Test Java (code) design steps:

1. **Attempt to construct a Document with a null Genre**
2. **Check that the exception is generated and handled as required**

Note: some testers chose to do this test *first*

Comment Tester la Médiathèque ?

A) Préparation

1. Tests de validation, e.g.: `Emprunter`
2. Tests d'intégration, e.g.: `Emprunter`
3. Tests unitaires, e.g.: `Document`

B) Codage et Exécution

4. **Tests unitaires avec Junit**
5. Tests d'intégration
6. Tests de validation

Coding step - Write code for `invariant` in `Document`

boolean mediatheque.document.Document.invariant()

Safety property - (emprunté => empruntable) AND (nbEmprunts >=0)

Specified by: invariant() in HasInvariant
Returns:
if the document is in a safe state, i.e respects the invariant

```
public boolean invariant (){
return !(emprunte && !empruntable)   && nbEmprunts >=0;
```

Document

Coding step : Update all Document operations/methods that may change state of a Document to check invariant and throw exception when it is broken.

```
Mediatheque –
   metEmpruntable,
   metConsultable
Audio – emprunter
Livre – emprunter
Video - emprunter
```

For example, **metEmpruntable:**

```
public void metEmpruntable() throws InvariantBroken{
empruntable = true;
if (!invariant())
   throw new InvariantBroken("Document -"+this);
}
```

Coding step - Update `toString` method to display if Document is SAFE or UNSAFE (depending on whether invariant is respected)

```
public String toString() {
String s = "\"" + code + "\" " + titre + " " + auteur + " " + annee
                + " " + genre + " " + localisation + " " + nbEmprunts;
if (empruntable) {
    s += " (emp ";
    if (emprunte) s += "O";
    else s += "N";
    s += ")"; }
if (invariant()) s += " SAFE "; else s +=" UNSAFE ";
return s;
}
```

Using **JUnit tool** – assertions and failures
(see `http://junit.org/javadoc/4.10/`)

```
assertTrue(boolean)
assertFalse(boolean)
assertArrayEquals( _ , _ )
assertEquals( _ , _ )
assertNull(_)
assertSame( _ , _ )
fail()
fail(java.lang.String )
...
```

Document

```
package mediatheque.document;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import mediatheque.Genre;
import mediatheque.Localisation;
import mediatheque.OperationImpossible;
import util.InvariantBroken;

public class JUnit_Document {

protected Localisation l;
protected Genre g;
protected Document d1;

// setUP    method      Annotation @Before
// tearDown method      Annotation @After
// test     methods     Annotation @Test
}
```

Call Sequences : Simplest Case

The call sequence for a class with two test methods – test1 and test2 is:

Call **@Before setUp**
Call **@Test** method **test1**
Call **@After tearDown**
Call **@Before setUp**
Call **@Test** method **test2**
Call **@After tearDown**

Call Sequences : More Complex Case

When setting up and tearing down is “expensive” then we can use @BeforeClass and @AfterClass annotations/methods to ensure that these are executed only once for each class.

For example, the call sequence for a class with two test methods – test1 and test2 is:

Call **@BeforeClass setUpClass**
Call **@Test** method **test1**
Call **@Test** method **test2**
Call **@AfterClass tearDownClass**

(See the JUnit documentation for more details on call sequences when we mix simple and complex cases)

Note: we use the simplest case in the following examples

Document

```
@Before
public void setUp() throws Exception{
        g = new Genre("Test_nom1");
        l = new Localisation("Test_salle1","Test_rayon1");
        d1 = new Video ("Test_code1", l, "Test_titre1",
                        "Test-auteur1", "Test-annee1", g,
                        "Test_duree1", "Test-mentionLegale1");
}
```

```
@After
public void tearDown() throws Exception {
l=null; g=null; d1=null;
}
```

Document

Test1: check the construction of a `Document` respects the invariant

```
@Test
public void constructorInvariant( ) throws
OperationImpossible, InvariantBroken{

Assert.assertTrue(d1.invariant());

}
```

Test2: Check all reachable states to be SAFE, i.e. respect the invariant

```
@Test
public void reachableStates () throws OperationImpossible,
InvariantBroken {
Assert.assertTrue(d1.invariant());
d1.metEmpruntable();assertTrue(d1.invariant());
d1.emprunter();assertTrue(d1.invariant());
}
```

Test4.1 : Check that we can only « restituer » a document after it has been « emprunté »

void mediatheque.document.JUnit_Document.restituerBeforeEmprunter() throws OperationImpossible, InvariantBroken

@Test(expected=OperationImpossible.class)

Document TEST 4.1
Check that we can only « restituer » a document after it has been « emprunted »

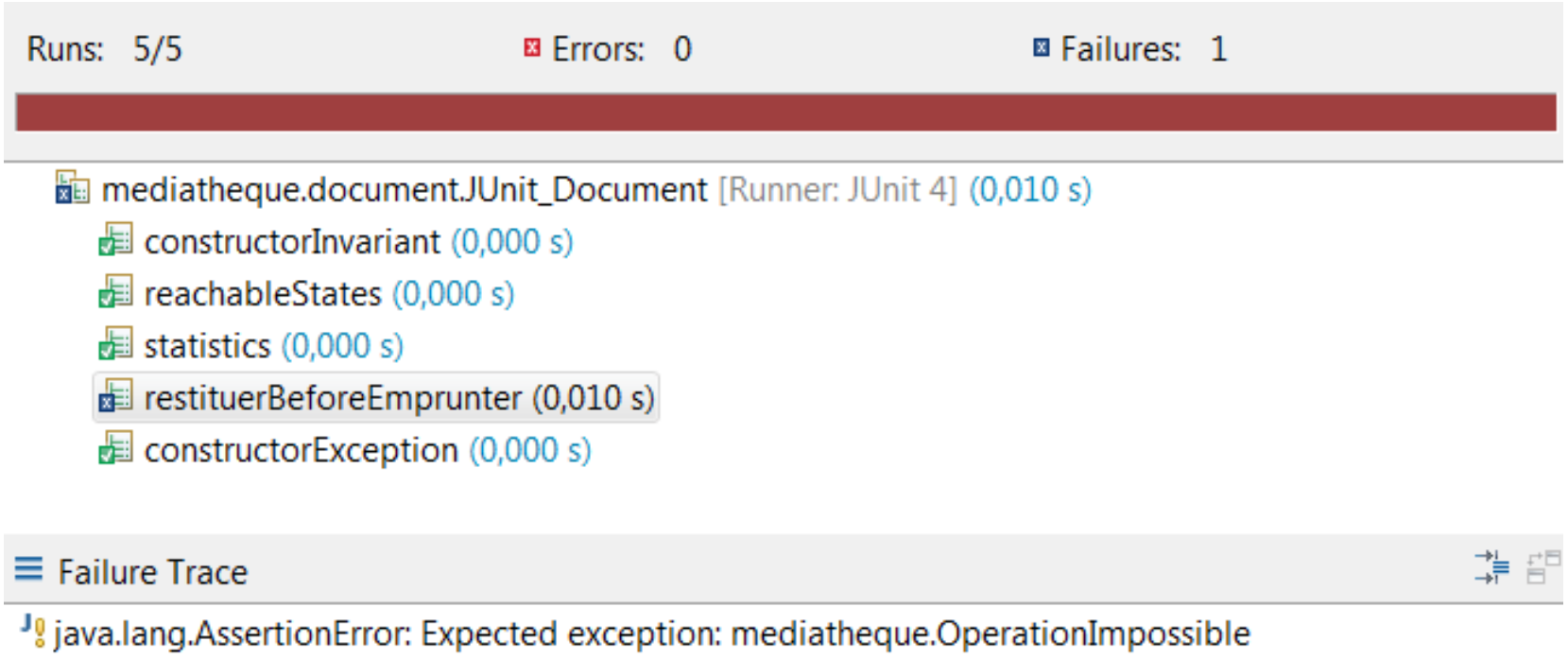
Throws:

OperationImpossible
InvariantBroken

```
@Test(expected=OperationImpossible.class)
public void restituerBeforeEmprunter () throws
OperationImpossible, InvariantBroken {
    Assert.assertTrue(d1.invariant());
    Assert.assertTrue(!d1.estEmprunte());
    d1.restituer();
}
```

Document

Test4.1 : Check that we can only « restituer » a document after it has been « emprunté »

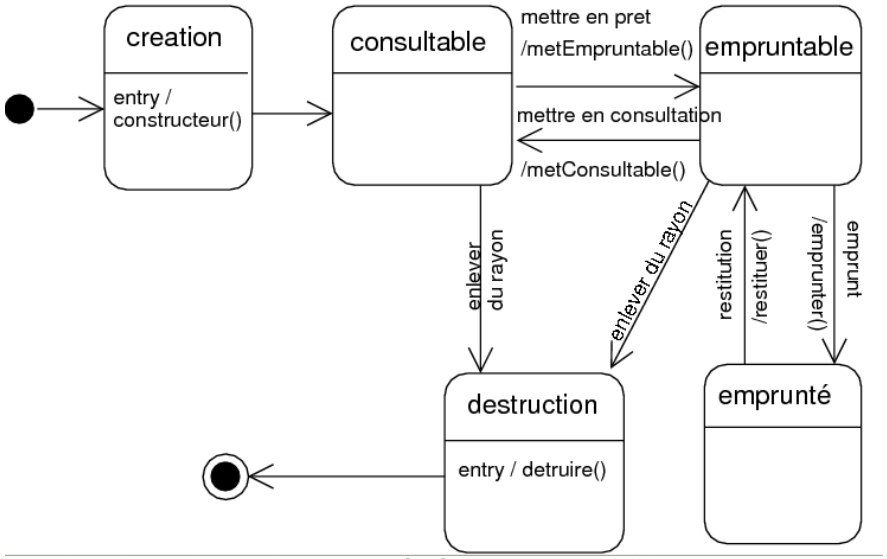


JUnit shows us if an expected exception was not thrown

Document

Test4.1 : Check that we can only « restituer » a document after it has been « emprunté »

Q: restituer in states consultable and empruntable?



Q: How to Fix This

OPTION 1: Update UML diagram with new transition(s)

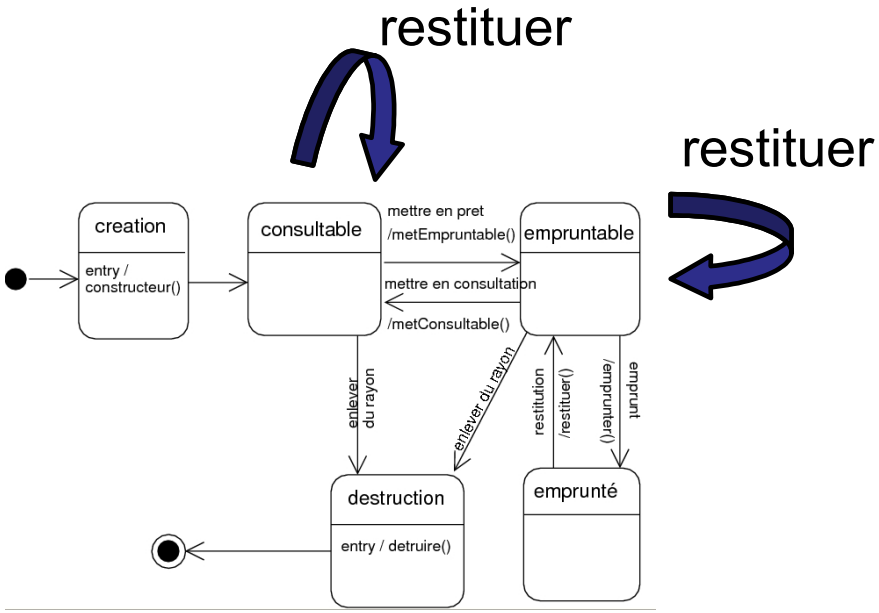
OPTION 2: Update Document code

OPTION 3 … : Can You Think Of Any Other Options?

Document

Test4.1 : Check that we can only « restituer » a document after it has been « emprunté »

OPTION 1: Update UML diagram with new transition(s)



No longer require an exception to be thrown in consultable and empruntable states when restituer is called

Document

Test4.1 : Check that we can only « restituer » a document after it has been « emprunté »

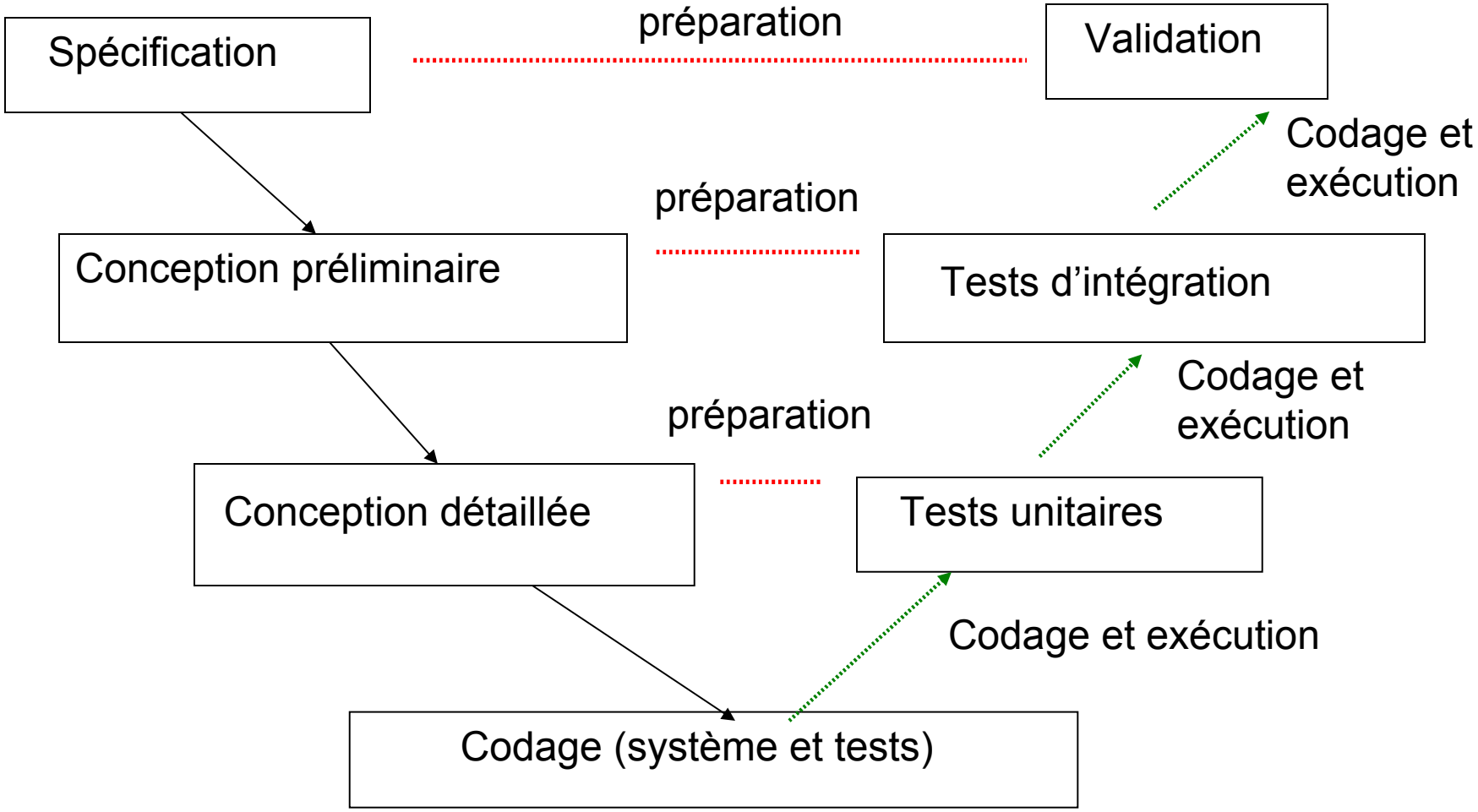
OPTION 2: Update Document code

```
public void restituer() throws InvariantBroken, OperationImpossible{

if (!emprunte) throw new OperationImpossible("Document -"+this);

emprunte = false;
System.out.println("Document: ranger \"" + titre + "\" en " +
localisation);
if (!invariant())
   throw new InvariantBroken("Document -"+this);
}
```

Le cycle en V (terminé)



SOME MORE ADVANCED ISSUES

INHERITANCE

Inheritance complicates the testing process:

- Should we also have a test inheritance hierarchy?
- How do we test the subclassing relationship?

EXCEPTIONS

Exceptions complicate the testing process:

- Exception handling testing requires generating the exceptions and where/how this should be done is not always obvious

GUIs are difficult to test as you often need to simulate user actions

To Come (In the TP)

Unit Tests – *Document*

Integration Tests – *FicheEmprunt*

Complete the validation tests : use case *emprunter document*

Unit tests on *Document subclasses* (using inheritance)

QUESTIONS?????