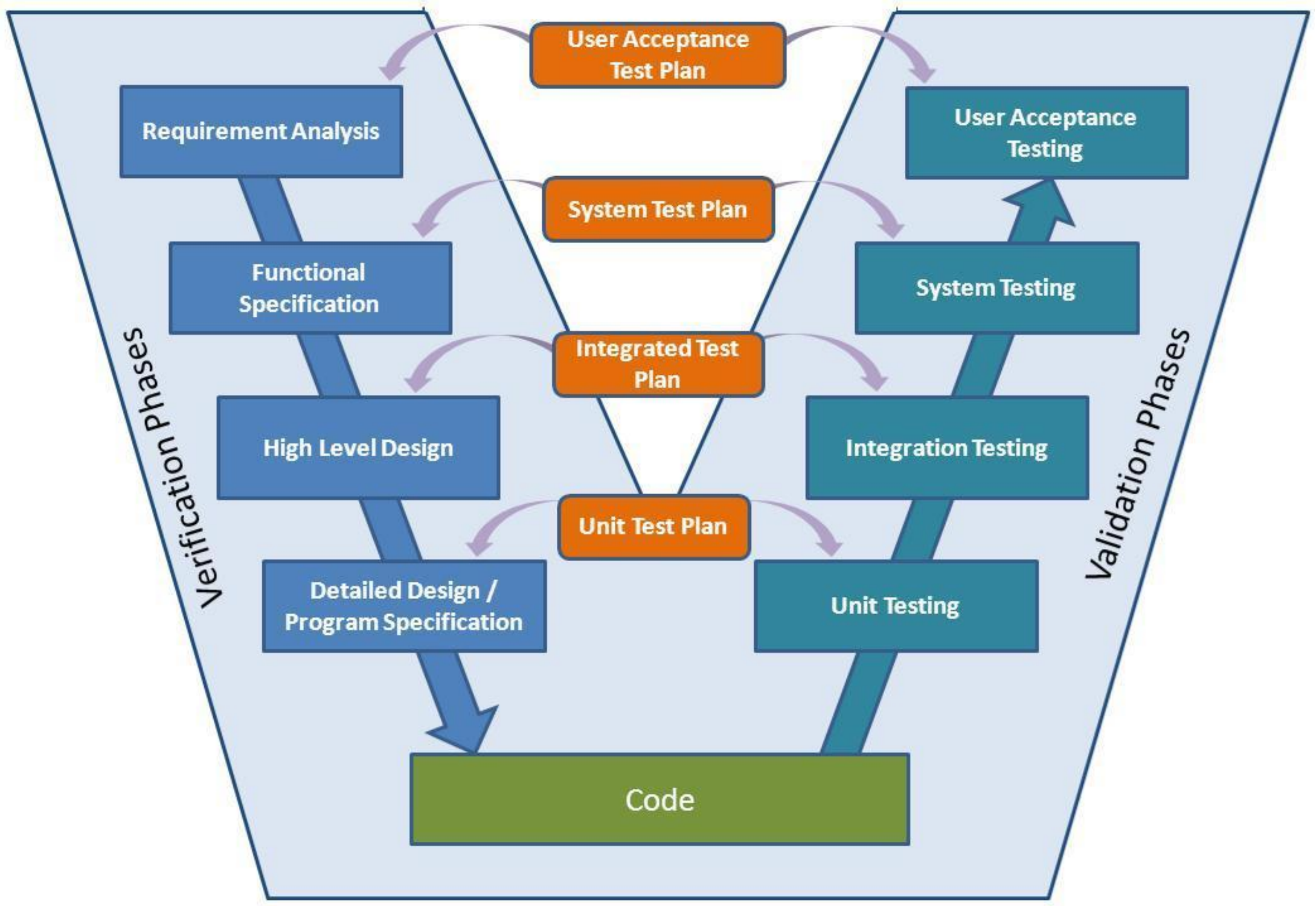
Tests in V-Model

UML and Java

Reminders

V-model


User Acceptance Test Plan
System Test Plan
Integrated Test Plan
Unit Test Plan
Requirement Analysis
Functional Specification
High Level Design
Detailed Design / Program Specification
Code
Unit Testing
Integration Testing
System Testing
User Acceptance Testing
Verification Phases
Validation Phases

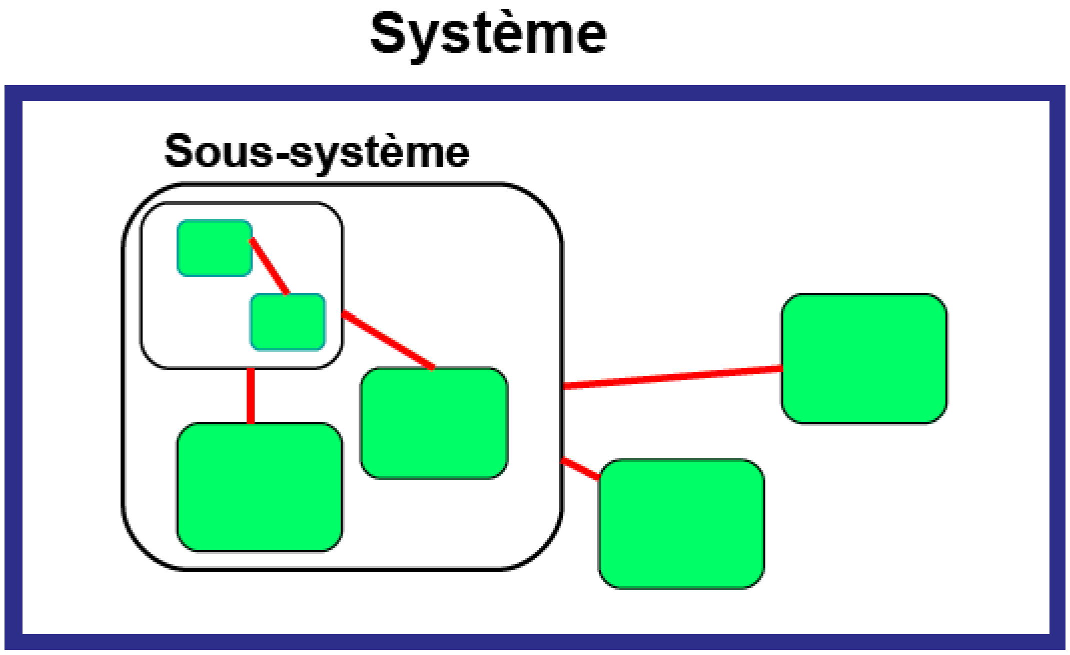
Testing Consistency (not uniquely OO)

- 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!

test level for each subsystem

- Validation Tests
- Integration Tests
- Unit Tests



Test in V-model

UML models

UML – the digrams

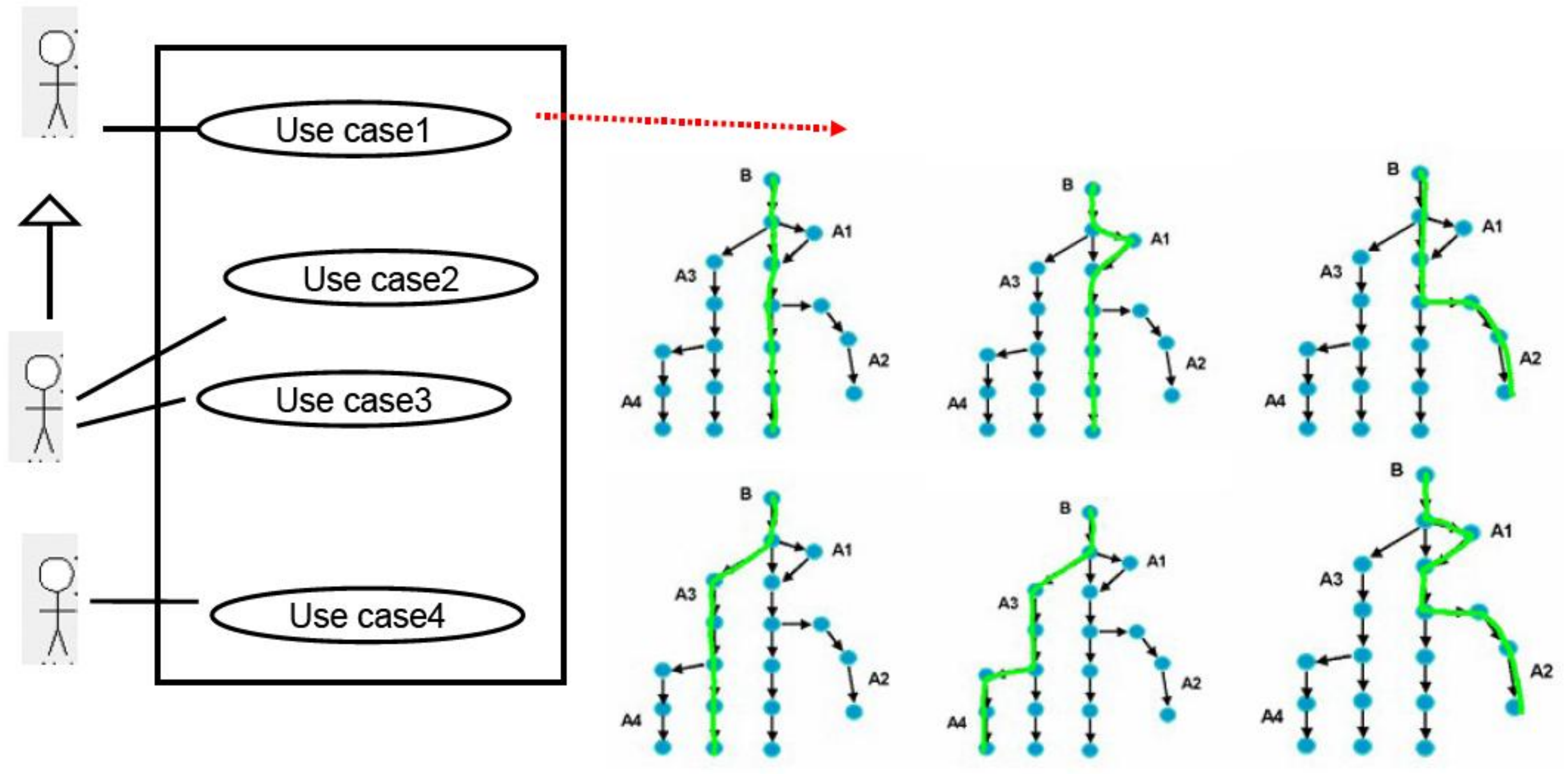
- Specification
  - Use cases diagrams ➔ Validation tests
  - Subsystem level (design and analysis) ➔ integration tests
    - Association and aggregation test (class diagrams)
    - Sequence tests (communication and sequence diagrams)
    - Exception tests
- Detailed design
  - Detailed class (state machine diagrams) ➔ Unit test

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.

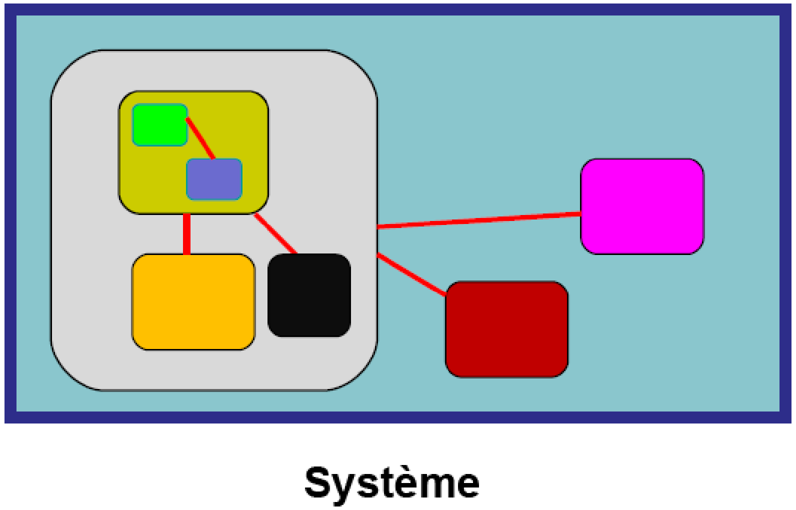
Validation

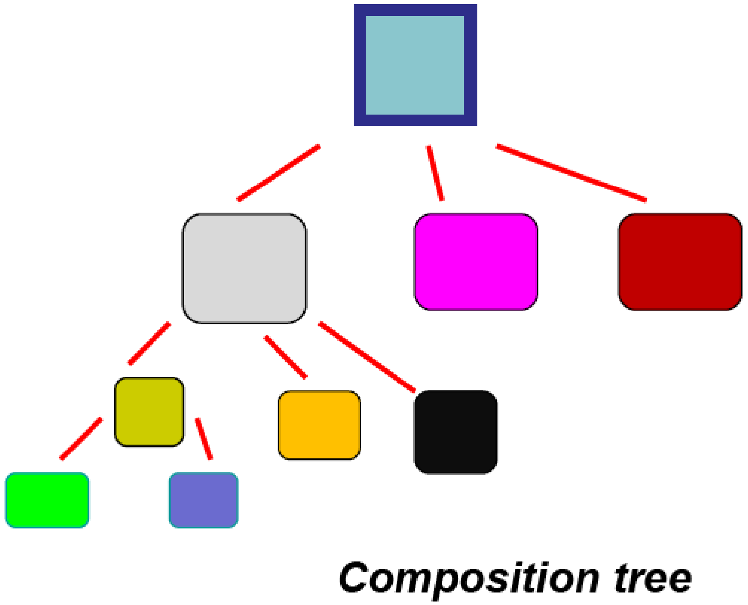
- Use Case Diagrams – for each use case examine possible scenarios, and choose a subset of alternative paths for testing. For example:



Integration : composition tree

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





- 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

Unit Tests

- We must test the invariants of each class
- We must test the functionality of each method of each class
- classes with sequential constraints on the activation methods may have sequencing errors.

➔The required behavior should be tested using a model state machine

An example

Sequencing

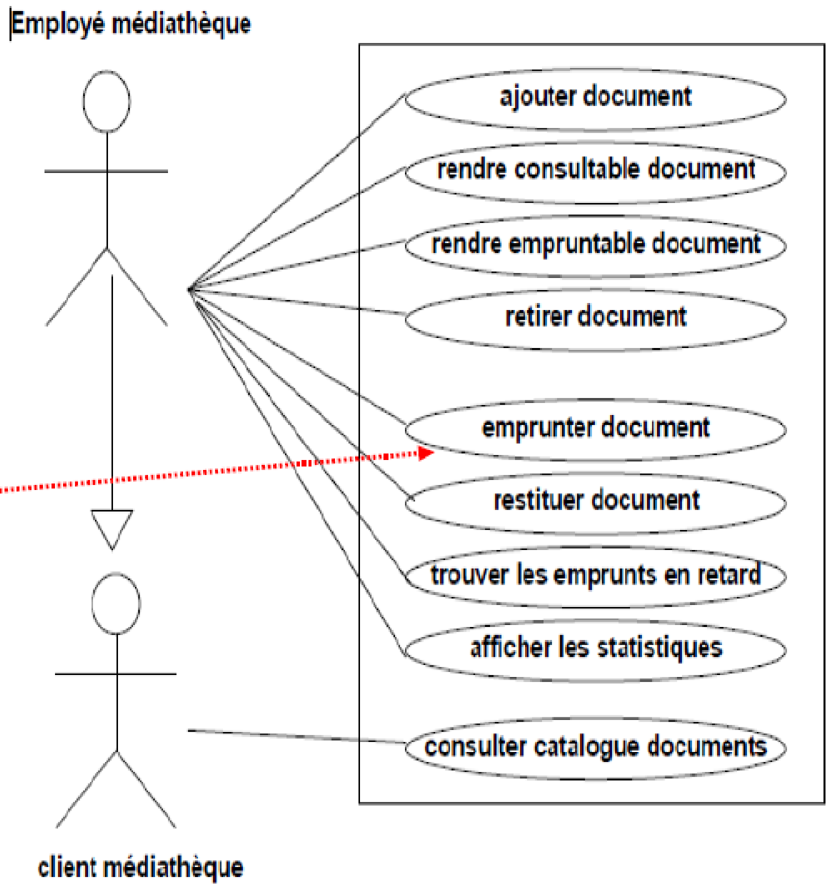
	seq
A) Preparation	
• Validation tests	1st
• Integration tests	3rd
• Unit tests	5th
B) Runtime and coding	
• Unit tests with Junit	6th
• Integration tests	4th
• Validation tests	2nd

Validation Tests

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


Employé médiathèque
ajouter document
rendre consultable document
rendre empruntable document
retirer document
emprunter document
restituer document
trouver les emprunts en retard
afficher les statistiques
consulter catalogue documents
client médiathèque

Preparation Validation tests

- Input data
  - Client: can be registered or not;
  - Borrowed items: already made by the client
  - There is a delay in a borrowed item?
  - the number of items borrowed corresponds to the maximum number of this customer?
- document:
  - exist?
  - Borrowable or just viewable?
  - already available or borrowed?

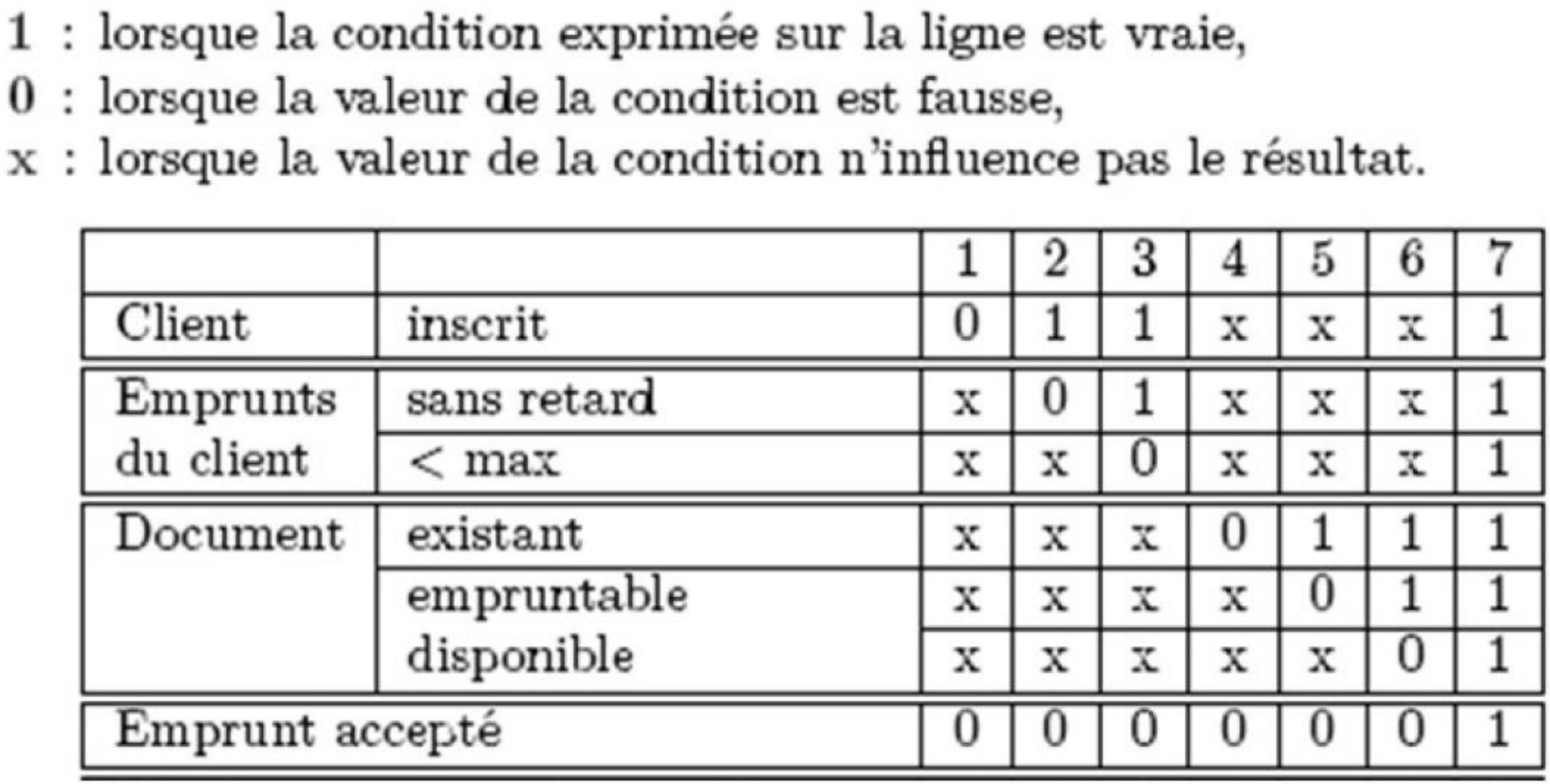
Preparation Validation tests

- **Output Data**
  - `Borrow` accepted or refused.
  - **Remark:** the definition of validation testing for the use case borrowdocument can lift at least the following questions (to ask the client):
    - a subscriber who has not paid his registration can still borrow a document?
    - should he be considered as a customer at the normal rate until he renewe its subscription?
    - or must re-subscribe before he can borrow a document?
- In general, validation test preparation allows remove ambiguities and gaps in the specification.

NOTE : Tests prepared earlier implies cheaper corrections

Preparation Validation tests

- **Decision table**



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

Preparation Validation tests

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

Test 1 – the client is not registered

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)

Preparation Validation tests

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

Test 2 – client has a borrowed document in delay

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)

Sequencing

A) Preparation	seq
• Validation tests	1st
• Integration tests	3rd
• Unit tests	5th
B) Runtime and coding	
• Unit tests with Junit	6th
• Integration tests	4th
• Validation tests	2nd

Validation Tests
Coding

- 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.

Validation Tests Coding

Test 1 – a client is not registred

```
/**
* 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

Validation Tests Coding

```
@Beforepublic
void setUp() throws Exception {
// un test de validation est un test unitaire sur la classe
     Mediathequem1 = 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("nom", "prenom", "Test_Client_Address2",cat);
}
@Afterpublic void tearDown() throws Exception {m1 = null;}
```

Validation Tests
Coding

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");
}
```

Sequencing

A) Preparation	seq
• Validation tests	1st
• Integration tests	3rd
• Unit tests	5th
B) Runtime and coding	
• Unit tests with Junit	6th
• Integration tests	4th
• Validation tests	2nd

Integration test preparation

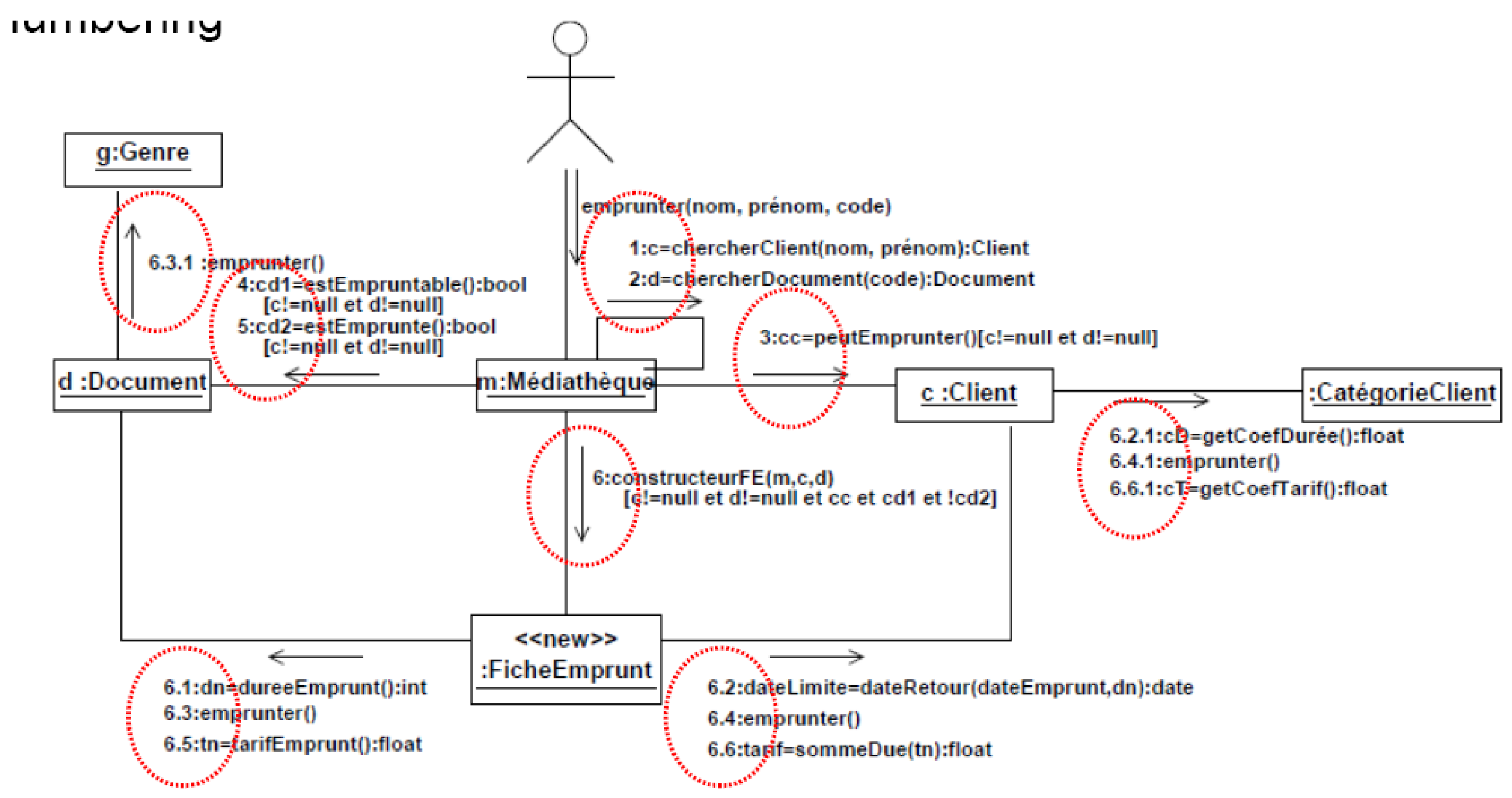
- The operation « emprunter » requires co-ordination between the
  - `client`,
  - `mediatheque`,
  - `document`, and
  - `ficheEmprunt` objects

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 …

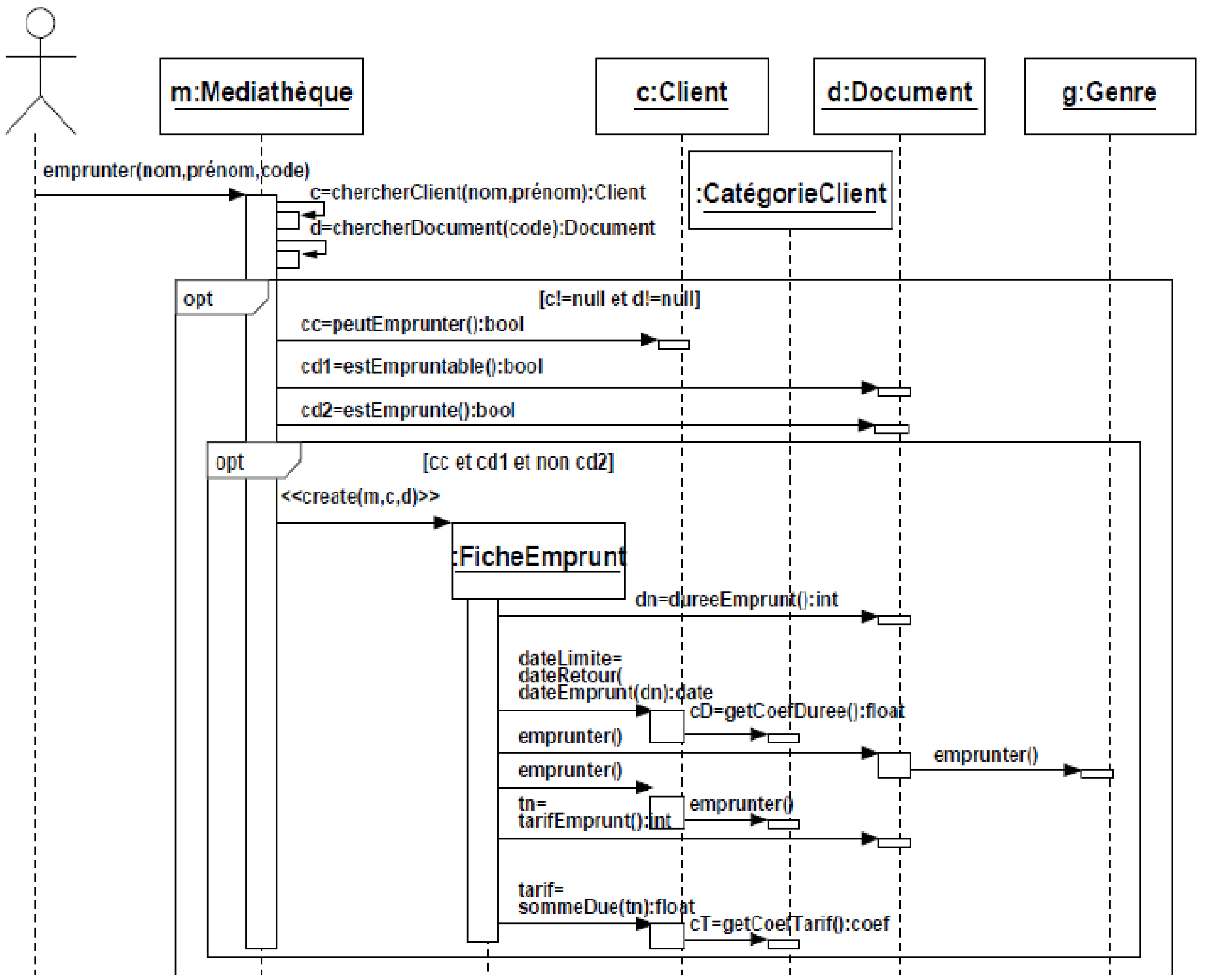
Integration test preparation

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



g:Genre
emprunter(nom, prénom, code)
1:c=chercherClient(nom, prénom):Client
2:d=chercherDocument(code):Document
6.3.1 :emprunter()
4:cd1=estEmpruntable():bool [c!=null et d!=null]
5:cd2=estEmprunte():bool [c!=null et d!=null]
3:cc=peutEmprunter()[c!=null et d!=null]
d :Document
m:Médiathèque
c :Client
:CatégorieClient
6.2.1:cD=getCoefDurée():float
6.4.1:emprunter()
6.6.1:cT=getCoefTarif():float
6:constructeurFE(m,c,d) [c!=null et d!=null et cc et cd1 et !cd2]
<<new>> :FicheEmprunt
6.1:dn=dureeEmprunt():int
6.3:emprunter()
6.5:tn=tarifEmprunt():float
6.2:dateLimite=dateRetour(dateEmprunt,dn):date
6.4:emprunter()
6.6:tarif=sommeDue(tn):float

Integration test preparation



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**

Integration test preparation

- 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

Integration test preparation

- 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 not been changed

Integration test preparation

- 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

Sequencing

	seq
A) Preparation	
• Validation tests	1st
• Integration tests	3rd
• Unit tests	5th
B) Runtime and coding	
• Unit tests with Junit	6th
• Integration tests	4th
• Validation tests	2nd

Etude de cas –Tests d’Integration – Codage et Execution

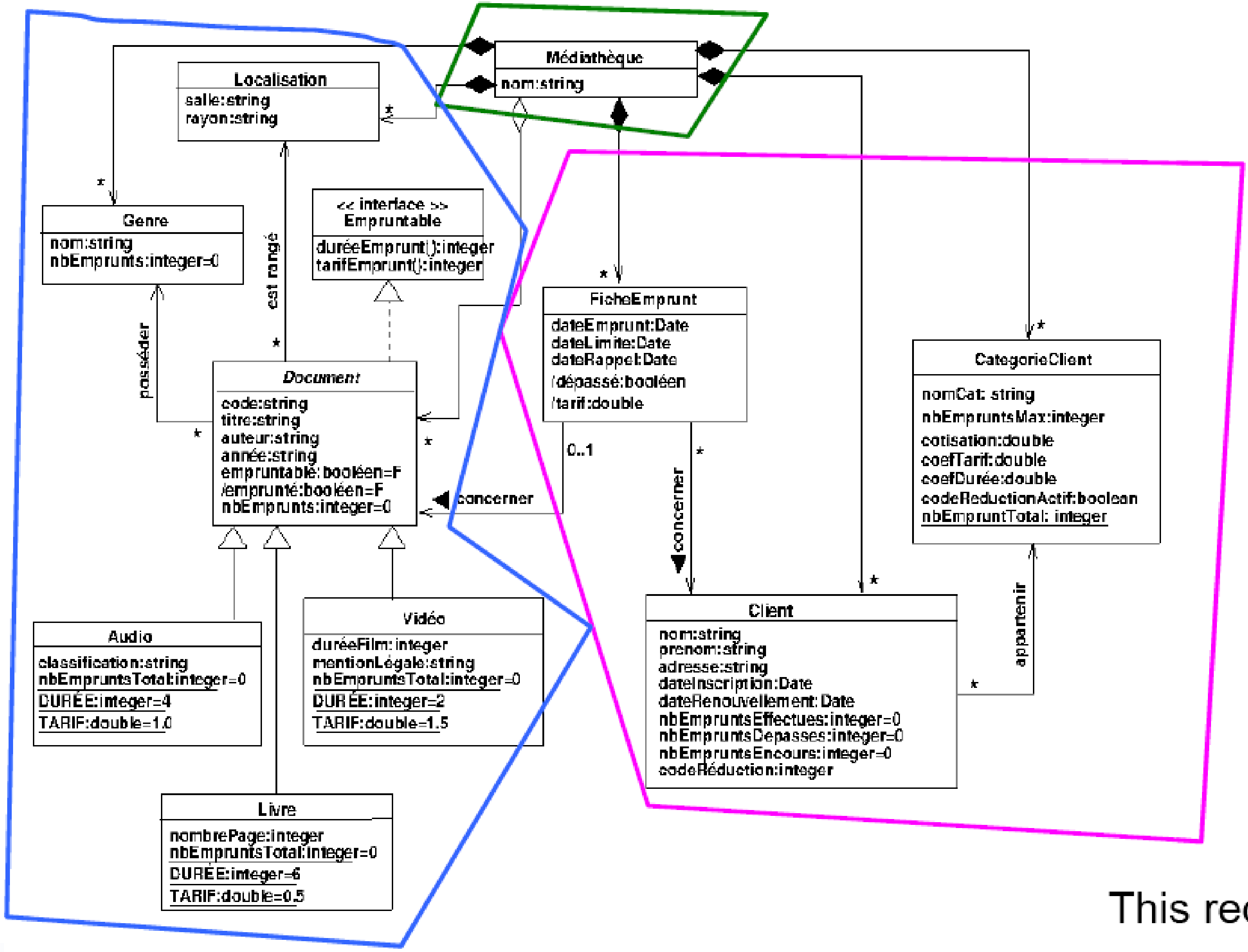
Integration Test1 Code: Verify correct co-ordination by FicheEmprunt

- 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

Etude de cas –Tests d'Integration – Codage et Execution

- Integration Tests: Typical/Example Development

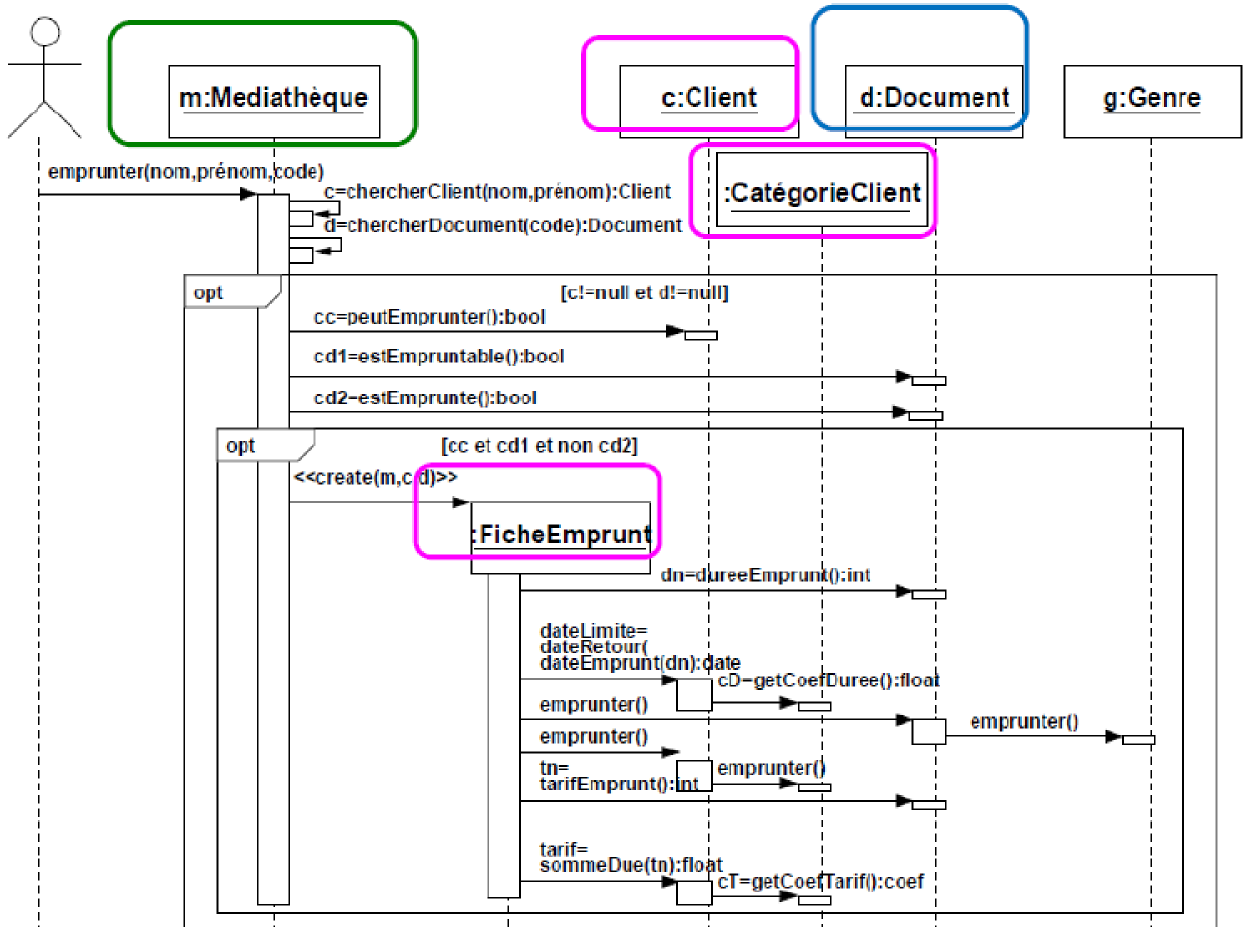


For example:
Completed
Partial
Not yet implemented

This requires further analysis

Etude de cas –Tests d'Integration – Codage et Execution

Integration Test 1



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

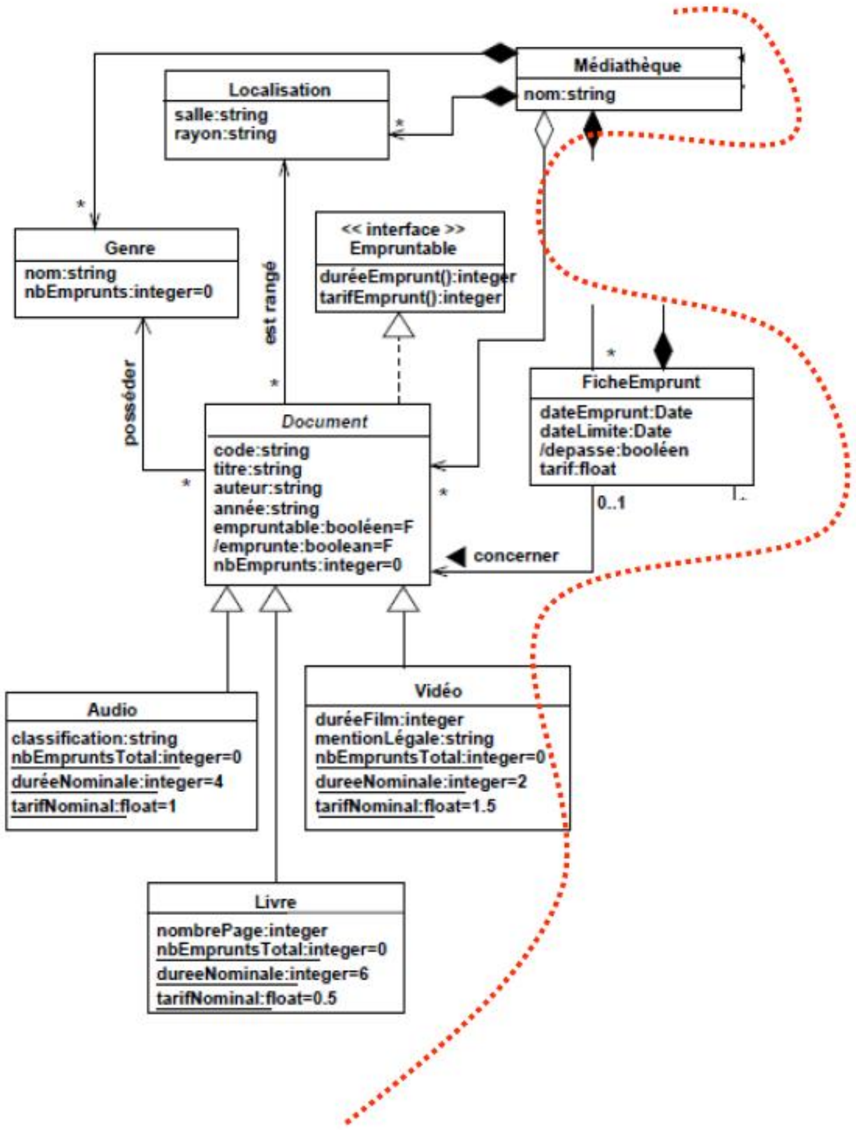
Sequencing

A) Preparation	seq
• Validation tests	1st
• Integration tests	3rd
• Unit tests	5th
B) Runtime and coding	
• Unit tests with Junit	6th
• Integration tests	4th
• Validation tests	2nd

Unit test preparation

- 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 black box 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.

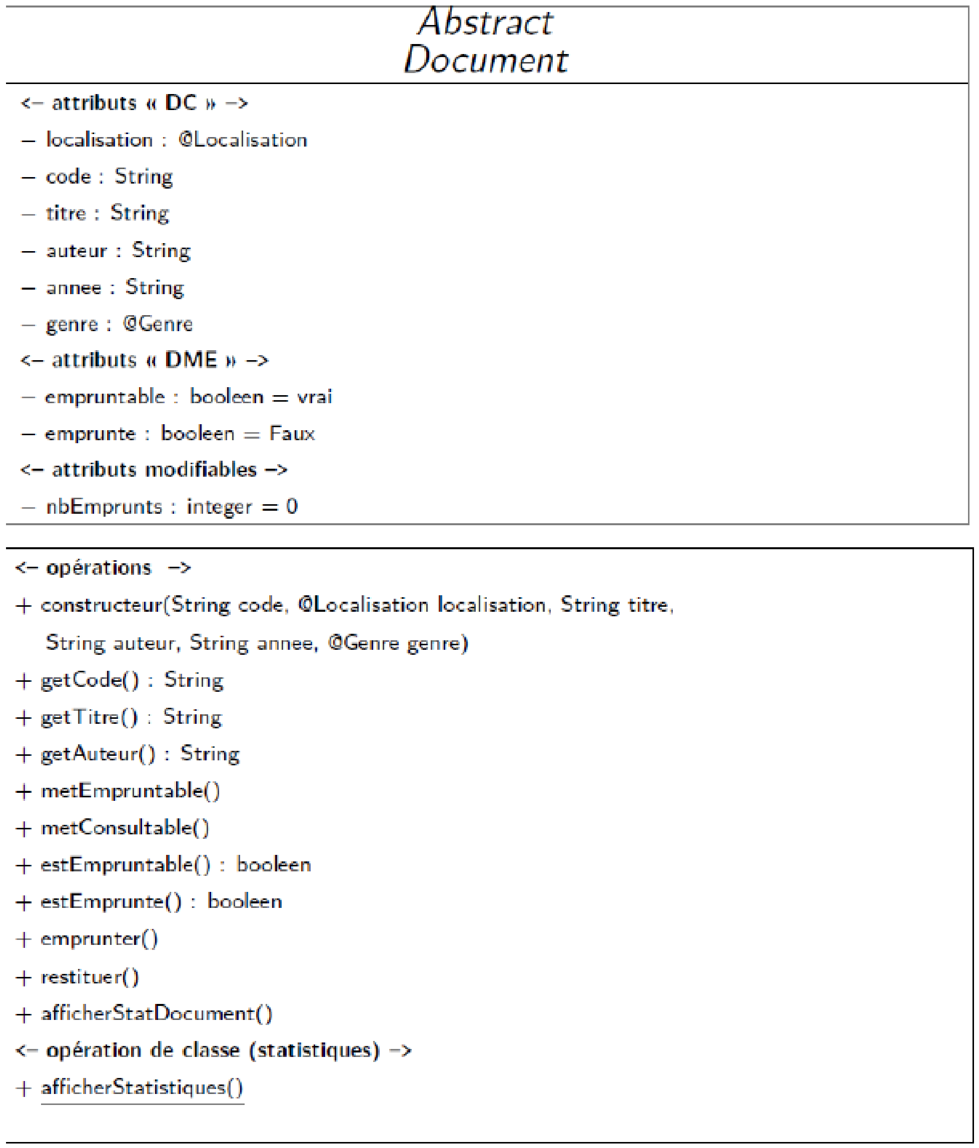
Unit test preparation



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

Unit test preparation



```
Abstract
Document
<– attributs « DC » –>
– localisation : @Localisation
– code : String
– titre : String
– auteur : String
– annee : String
– genre : @Genre
<– attributs « DME » –>
– empruntable : booleen = vrai
– emprunte : booleen = Faux
<– attributs modifiables –>
– nbEmprunts : integer = 0

<– opérations –>
+ constructeur(String code, @Localisation localisation, String titre,
  String auteur, String annee, @Genre genre)
+ getCode() : String
+ getTitre() : String
+ getAuteur() : String
+ metEmpruntable()
+ metConsultable()
+ estEmpruntable() : booleen
+ estEmprunte() : booleen
+ emprunter()
+ restituer()
+ afficherStatDocument()
<– opération de classe (statistiques) –>
+ afficherStatistiques()
```

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.

Unit test preparation

<– attributs « DME » –>

– empruntable : booleen = vrai

– emprunte : booleen = Faux

<– attributs modifiables –>

– nbEmprunts : integer = 0

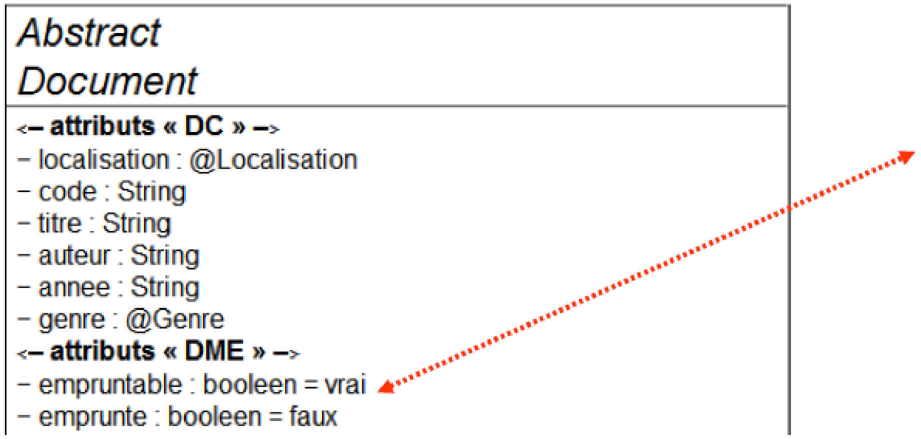
The ***invariant*** property is defined on the attributes of the class in order to say when a `Document` instance/object is in a SAFE state, i.e. a state which is meaningful/allowable.

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 »*

Unit test preparation


Abstract Document

<– attributs « DC » –>
- localisation : @Localisation
- code : String
- titre : String
- auteur : String
- annee : String
- genre : @Genre

<– attributs « DME » –>
- empruntable : booleen = vrai
- emprunte : booleen = faux

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`.)

Unit test preparation

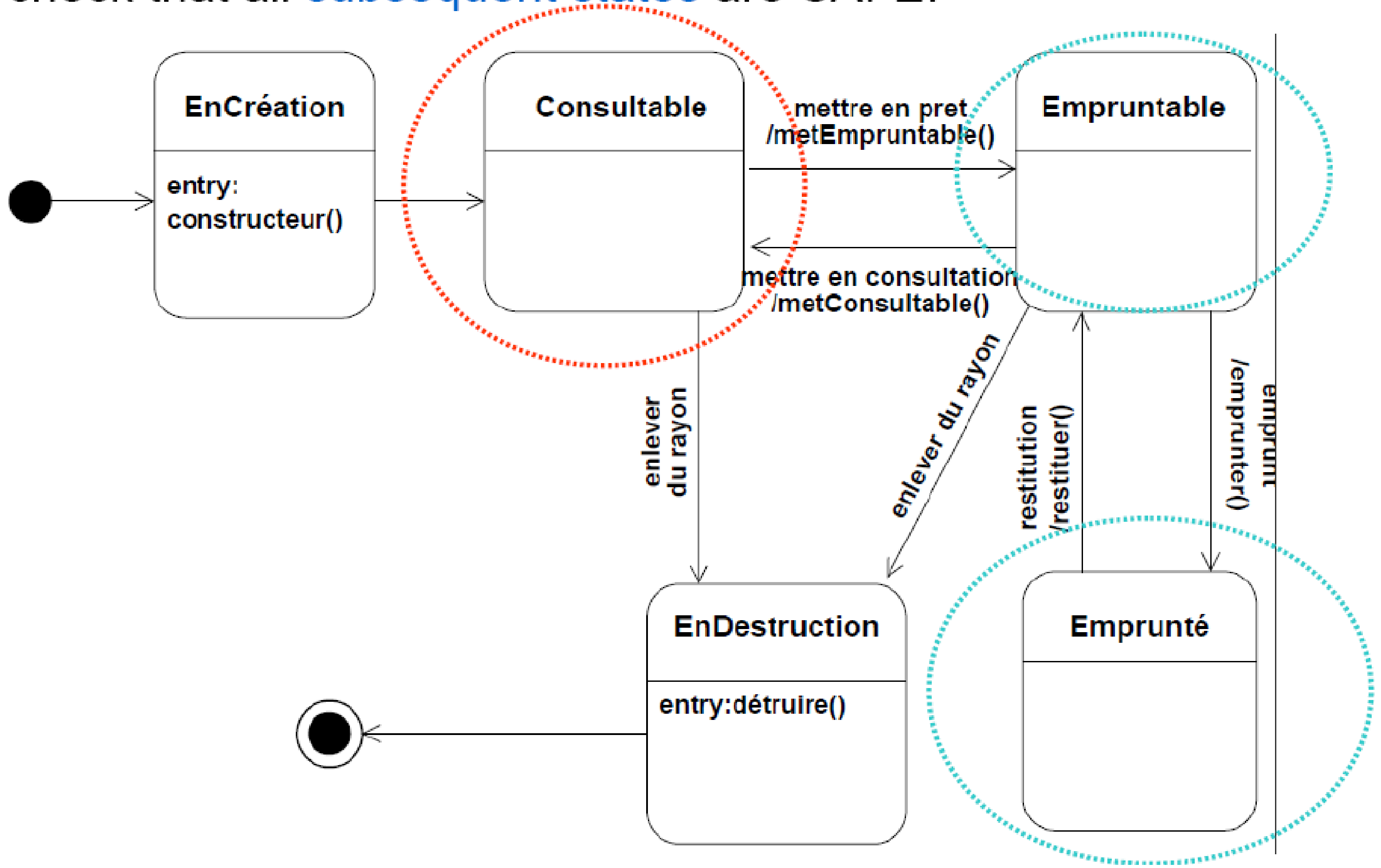
All design decisions involve compromise.

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 »

Unit test preparation

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.



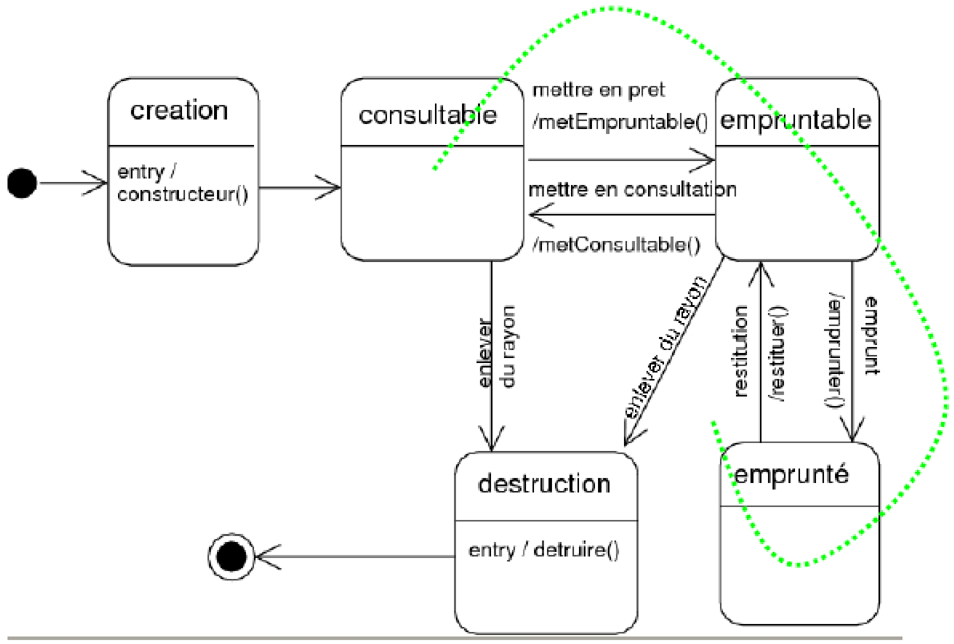
Unit test preparation

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

Unit test preparation

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**

Unit test preparation

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.

Unit test preparation

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

Unit test preparation

***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**

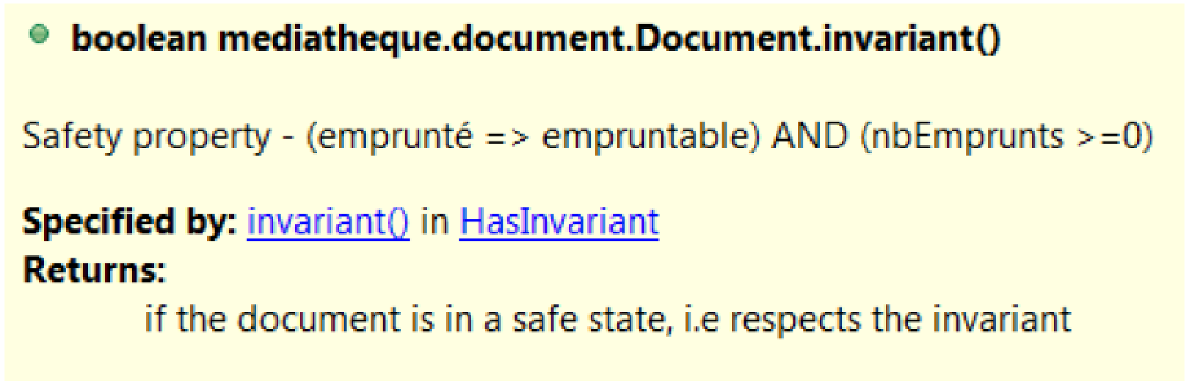
Note: some testers chose to do this test *first*

Sequencing

A) Preparation	seq
• Validation tests	1st
• Integration tests	3rd
• Unit tests	5th
B) Runtime and coding	
• Unit tests with Junit	6th
• Integration tests	4th
• Validation tests	2nd

Unit tests
Coding with JUnit

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;
```

Unit tests
Coding with JUnit

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{
emprumtable = true;
if (!invariant())
   throw new InvariantBroken("Document -"+this);
}
```

Unit tests
Coding with JUnit

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;
}
```

Unit tests
Coding with JUnit

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 )
...
```

Unit tests
Coding with JUnit

```
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
}
```

Unit tests
Coding with JUnit

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**

Unit tests
Coding with JUnit

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

Unit tests
Coding with JUnit

```
@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;
}
```

Unit tests
Coding with JUnit

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

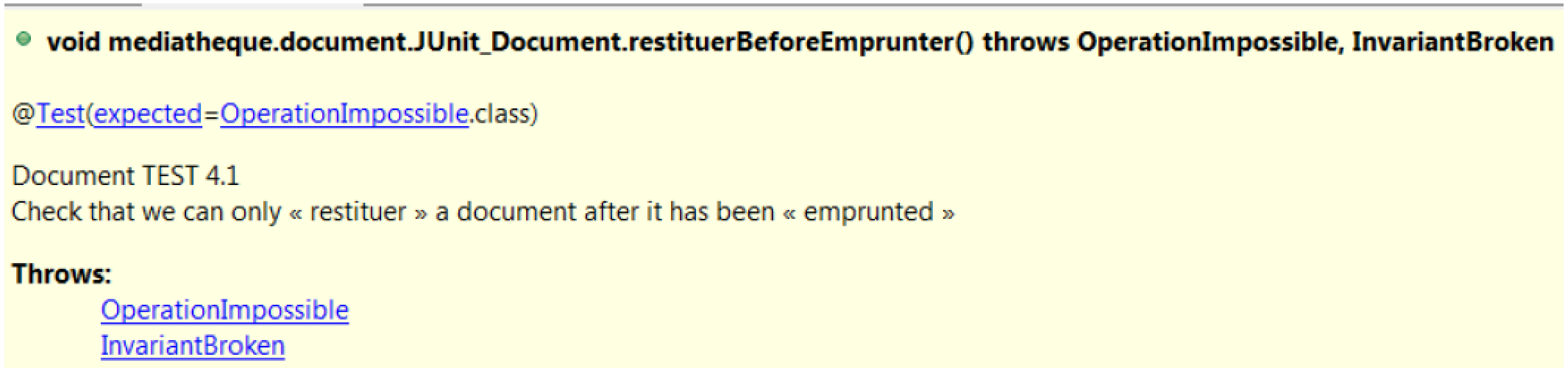
```
@Test
public void constructorInvariant( ) throws
OperationImpossible, InvariantBroken{

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());
}
```

Unit tests
Coding with JUnit

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)

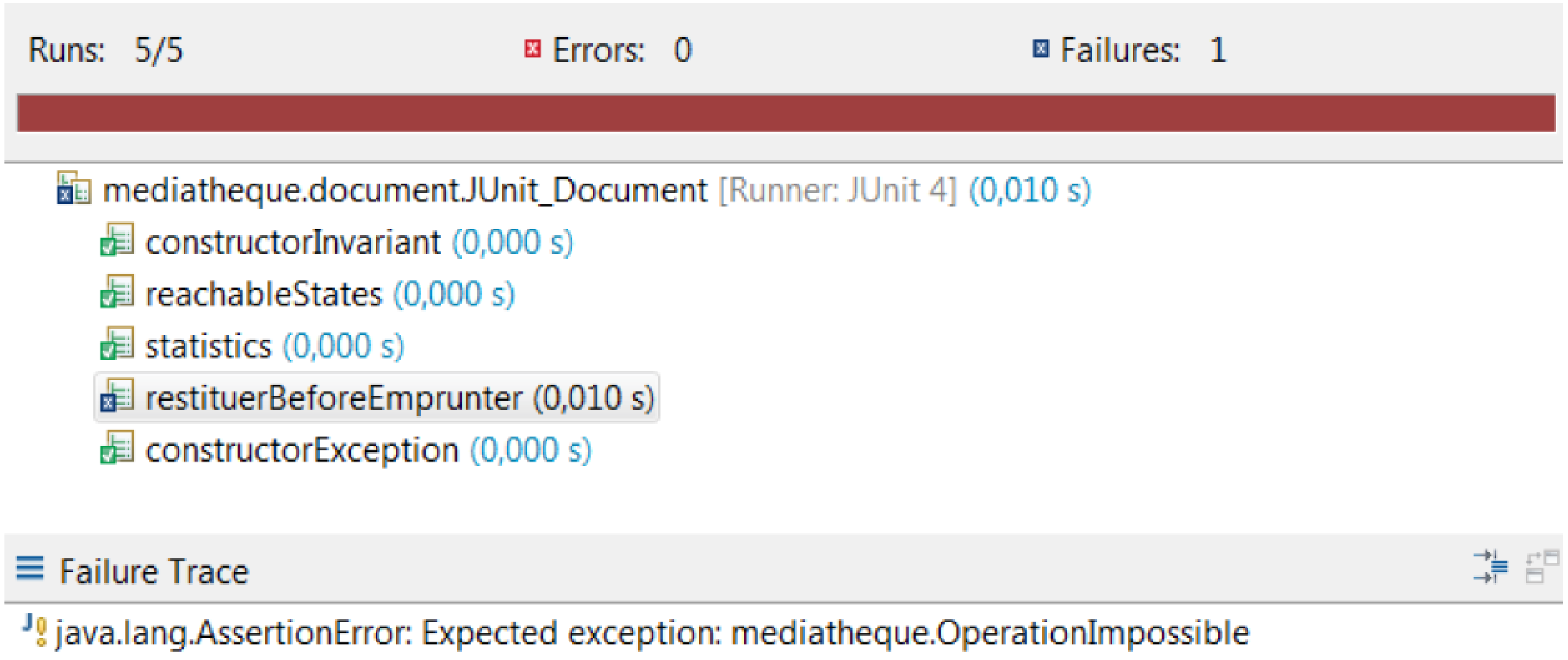
Document TEST 4.1
Check that we can only « restituer » a document after it has been « emprunted »

Throws:
OperationImpossible
InvariantBroken

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

Unit tests
Coding with JUnit

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



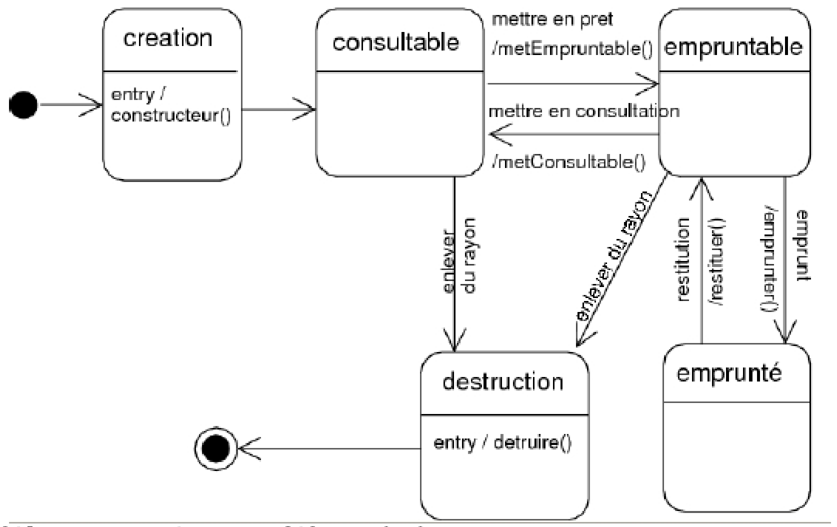
```
Runs:  5/5          Errors:  0          Failures:  1

mediatheque.document.JUnit_Document [Runner: JUnit 4] (0,010 s)
    constructorInvariant (0,000 s)
    reachableStates (0,000 s)
    statistics (0,000 s)
    restituerBeforeEmprunter (0,010 s)
    constructorException (0,000 s)

Failure Trace
java.lang.AssertionError: Expected exception: mediatheque.OperationImpossible
```

JUnit shows us if an expected exception was not thrown

Unit tests
Coding with JUnit

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)

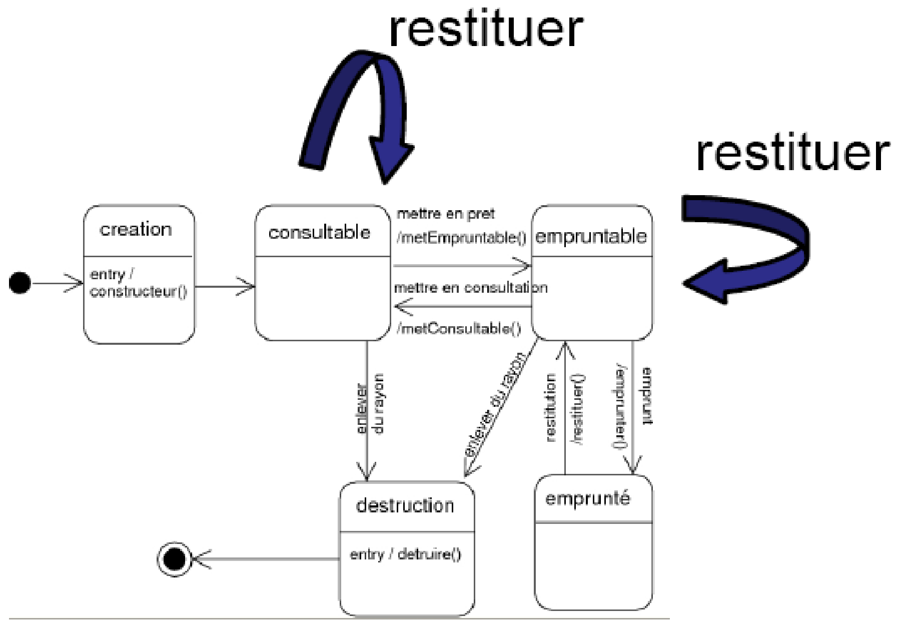
OPTION 2: Update Document code

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

Unit tests
Coding with JUnit

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

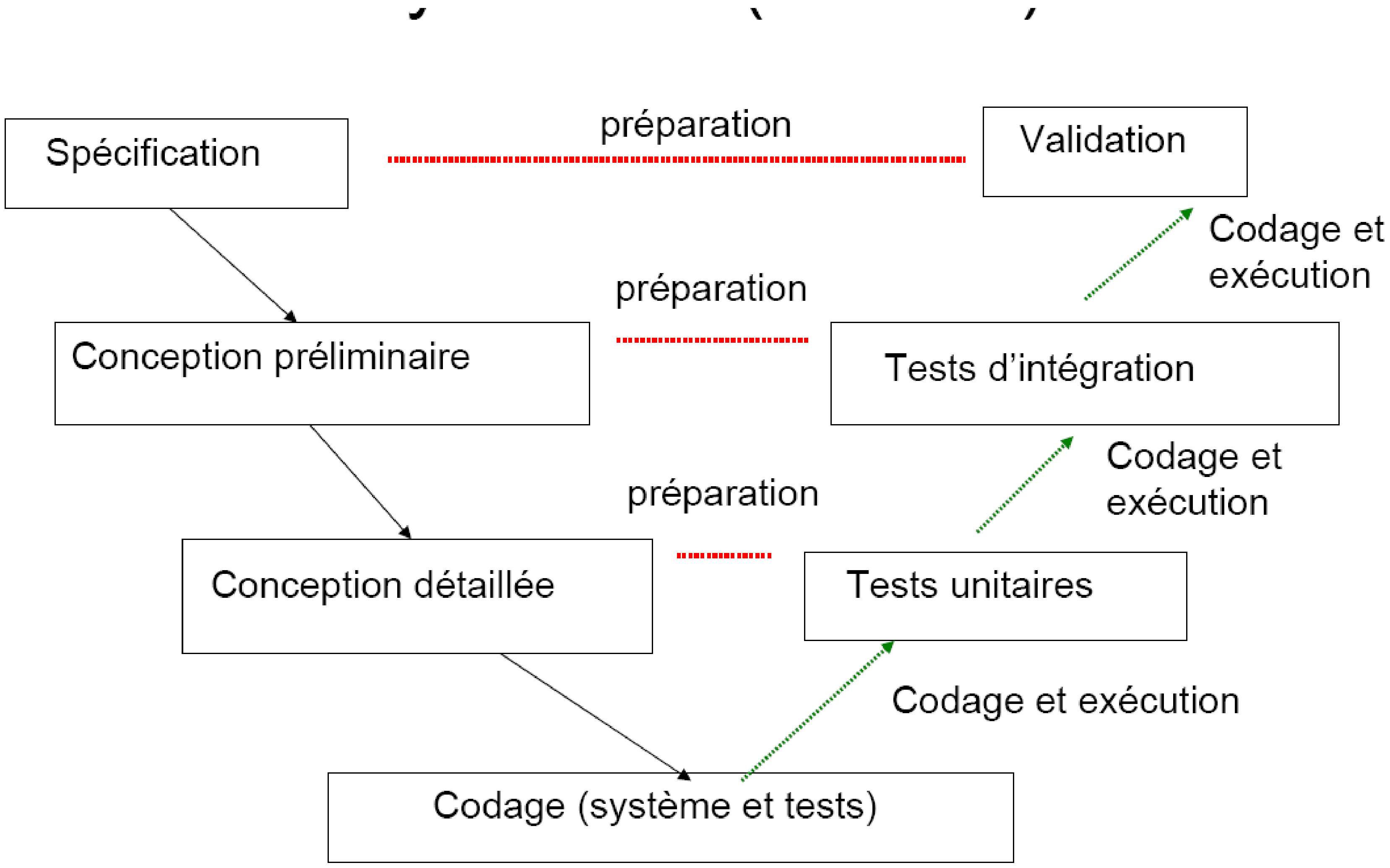
Unit tests
Coding with JUnit

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);
}
```

V-Model terminated



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