

Tests

(JUnit , Mockito ,
DbUnit , Selenium,
Jbehave, JMeter, ...)

Table des matières

I - Tests de composants logiciels.....	5
1. Types de tests et doublures.....	5
1.1. Différents types de tests.....	5
1.2. Tests unitaires.....	7
1.3. Tests d'intégration.....	9
1.4. Doublures et Mocks.....	10
II - JUnit (tests unitaires java).....	13
1. Tests unitaires avec JUnit (3 ou 4).....	13
1.1. Présentation de JUnit.....	13
1.2. Structure d'une classe de test (ancienne version JUnit3).....	13
1.3. Structure d'une classe de test (version actuelle JUnit4).....	14
1.4. Fonctionnement de JUnit.....	15
1.5. @BeforeClass et "static" (optimisations).....	16
1.6. Suite de tests (@Suite).....	17

1.7. Lancement des tests unitaires.....	18
---	----

III - Mockito (un des frameworks "mock" java).....20

1. Positionnement et intérêts des "mocks".....	20
2. Mockito.....	21
2.1. Présentation de Mockito.....	21
2.2. Dépendance maven nécessaire.....	21
2.3. Initialisation du Mock au niveau d'un test Junit:.....	21
2.4. Mockito as stub.....	23
2.5. Mockito as "spy" et "mock = stub + spy".....	25

IV - Tests de persistance (h2, dbUnit, ...).....28

1. Embedded database (hsqldb , h2).....	28
1.1. Fonctionnalités.....	28
1.2. Modes "in memory" et "local".....	29
1.3. hsqldb et h2.....	30
2. DbUnit.....	32
3. Automatisation tests de persistance.....	35

V - Tests d'intégration (contexte, Selenium, ...).....38

1. Test d'intégration (contexte , généralités).....	38
1.1. Contexte , cadre général et points à préparer.....	38
1.2. Pistes technologiques pour les tests d'intégration.....	40
1.3. Types classiques de tests d'intégration.....	42
2. Tests web (http/html) via selenium.....	43
2.1. Présentation et installation de Selenium.....	43
2.2. Enregistrement de séquence via selenium-IDE.....	44
2.3. Selenium au sein de tests d'integration "maven".....	46

VI - Tests de comportements / fonctionnels.....48

1. Tests comportementaux (fonctionnels).....	48
2. Tests comportementaux avec JBehave.....	51
2.1. Environnement et démarrage.....	51
2.2. Exemple "Step jBehave".....	53
2.3. Résultats d'un test jBehave.....	55

VII - Tests de performances.....56

1. Rôles des tests et bénéfices attendus.....	56
2. Types de tests (principes , intérêts).....	57
2.1. Différents types d' analyses (vue externe, sondes, ...).....	57
2.2. Activation des mesures et conséquences.....	58

3. Profiling (application / serveur).....	59
4. Mesures externes (non intrusives).....	60
5. Simulation et test de charge.....	61
6. Topologie de la plateforme de test (charge).....	62
7. Configuration des requêtes nécessaires à la simulation.....	63
8. Synthèse des résultats obtenus.....	63
9. Présentation de JMeter.....	65
10. Configuration élémentaire de JMeter.....	66
10.1. Configuration d'un groupe d'unités (clients simultanés).....	66
10.2. Configuration d'une requête à lancer (plusieurs fois).....	67
10.3. configuration de la récupération des résultats (synthèse, courbe, ...).....	67
10.4. (re-)lancement du test et observation des résultats.....	68
11. Très important: tester tout d'abord le test !!!.....	69
12. Gestion correcte des sessions HTTP (cookies).....	70
13. Successions de requêtes http liées entre elles.....	70
13.1. Principe du "repost-hidden".....	70
13.2. Cas concret du framework JSF.....	73
14. Configurations diverses.....	75
14.1. Test de charge (SQL) vers un serveur de base de données.....	75
14.2. Variété au niveau des requêtes pour plus de réalisme.....	75
14.3. Plugins utiles.....	76
14.4. JMeter à la source:.....	76
14.5. Plugin "JMeterPlugins-WebDriver-1.2" pour "selenium".....	77
15. Principe du "profiling".....	78
16. JaMon (Java Api for Monitor).....	78
16.1. Intérêt d'une api compatible avec les api de logs.....	78
16.2. Utilisation élémentaire de l'api "Jmon".....	79
16.3. Activation et désactivation des mesures.....	80
16.4. Intégration au sein de Spring 2 ou 3 (pour mesurer les temps d'exécution de la partie "back-office" (services + dao).....	81
16.5. Intégration au sein d'un module EJB3 (alternative à Spring) :.....	83
16.6. Utilisation de l'api "JaMon" sur la partie JDBC/SQL.....	83
16.7. Utilisation de l'api "JaMon" sur la partie Web (via filtre web).....	85
16.8. Temps théoriques à calculer :.....	86

VIII - Annexe – Méthodologies avec tests.....88

1. Processus unifiés (UP).....	88
1.1. Meilleures pratiques communes.....	88
2. Principaux Processus.....	89
2.1. RUP (Rational UP – origine d'UP).....	89
2.2. 2TUP (2 tracks UP) – Pocessus en Y.....	89
2.3. XP (eXPerience & eXtreme Programing).....	90

3. Présentation de RUP.....	91
4. Approche itérative.....	91
5. Vocabulaire.....	92
6. 4 grandes phases.....	93
6.1. Phase de conceptualisation / commencement (<i>inception</i>).....	94
6.2. Phase d'élaboration.....	95
6.3. Phase de construction.....	95
6.4. Phase de transition.....	95
7. XP (Extreme Programming).....	96
7.1. Principales caractéristiques de XP.....	96
7.2. Cycle de développement XP.....	96
7.3. Valeurs de XP.....	97
7.4. Pratiques (extrêmes?) de XP.....	97
7.5. Autres pratiques extrêmes et variantes.....	98
7.6. Zoom sur la planification adaptative.....	99

I - Tests de composants logiciels

1. Types de tests et doublures

1.1. Différents types de tests

Classification des tests par niveaux

Niveau de Tests	Caractéristiques
Test unitaire de composant	Tester de façon isolée un seul composant logiciel
Test d'intégration technique	Tester un assemblage de composants au sein d'une application et vérifier "bonnes communications , pas d'incompatibilités, pas d'effets de bords , ..."
Test "système fonctionnel" (test d'intégration fonctionnel , VABF)	Tester la validité fonctionnelle de tout un sous système informatique (Database + ESB + applications + ...)
Test d'acceptation (UAT = User Acceptance Test) alias " <i>recette</i> "	Acceptation du logiciel dans le contexte "client" ou "moa" .

Classification selon de niveau d'accessibilité :

- Test "**boîte noire**" (sans connaître la structure interne du composant à tester)
- Test "**boîte blanche**" (en connaissant la **structure** interne et en vérifiant certaines caractéristiques internes : intégrité , ...)

Classification selon une caractéristique (attribut de qualité , ...) :

- tests de **montée en charge** et de **performance** : temps de réponse corrects ? Combien de clients simultanés au maximum ? ...
- test **fonctionnel** : fonctionnalités demandées sont bien supportées, ...
- test de **robustesse** : valider la stabilité et la fiabilité du logiciel dans le temps.
- test de **vulnérabilité** : vérification de sécurité du logiciel.
-

Autres catégories de tests ou synonymes :

- **Revue de code** (souvent automatisée) et rapport qualimétrique (code bien écrit ? respectant certaines normes/conventions ? , pas trop de copier/coller ? , ...)
- **Tests de non-régression** : pour vérifier que des modifications n'ont pas altérées le fonctionnement de l'application.
- **Tests IHM** (charte graphique respectée ? Navigations ? ...)
- **Tests d'adaptation à différentes configurations ou contextes** selon système hôte (linux, windows , ...) , résolution écran, ...
- **Tests fonctionnels de bout en bout** (des processus métier au sein d'un sous système informatique).

VABF = (Vérification d'aptitude au Bon Fonctionnement)

- **Tests d'exploitabilité** (dans un contexte de (pré-)production)
Qualité de services (performances , sécurité, ...) correctes ?
- **VSR** = **V**érification en **S**ervice **R**égulier (surveillance/pilotage).

1.2. Tests unitaires

4 phases d'un test unitaire

- 1) **Initialisation** (méthode *setUp* ou *init*): définition d'un environnement de test complètement reproductible (une "fixture" avec par exemple une instance de composant bien initialisée et un éventuel jeux de données)
- 2) **Exécution du code à tester**: appel d'une méthode avec certains paramètres d'entrée bien choisis (valides ou invalides, ...).
- 3) **Vérification (assertions)**: comparaison du résultat obtenu avec la valeur de réponse attendue. Ces assertions définissent le résultat du test: SUCCÈS ou ÉCHEC .
- 4) **Désactivation/terminaison** (méthode *tearDown* ou ...): désinstallation des "fixtures" pour retrouver l'état initial du système, dans le but de ne pas polluer/perturber les tests suivants. Tous les tests doivent idéalement être indépendants et reproductibles.

NB: Selon le degré de sophistication/complexité du test unitaire, les phases 1 et 4 pourront être triviales ou très évoluées (ex : avec *dbUnit*).

Utilités/objectifs des tests unitaires

- **Bien appréhender/formaliser le contrat technico/fonctionnel d'un composant** logiciel (en lisant le code d'un test bien écrit , on comprend le comportement attendu des opérations/méthodes du composant à tester).
- **Trouver les erreurs rapidement et simplifier la maintenance** :
Une fois le test unitaire écrit, on peut le relancer automatiquement sans effort des milliers de fois pour *vérifier l'absence de régression* et pour *localiser rapidement une erreur* en cas de problème .
- **Développer consciencieusement** (en testant naturellement l'absence de bug et le bon fonctionnement) au fur et à mesure de la programmation.
- S'assurer que tout le code écrit (et prévu pour être appelé/invoqué) soit **couvert** par un nombre suffisant de tests unitaires.
- Ne pas se contenter de la boutade "*tester c'est douter*" !

Stratégie habituelle de test (Test Driven Development)

- 1) **définir la structure du composant à tester** (exemple: *modèle UML* , *interface java* , ...) et un éventuel embryon d'implémentation.
- 2) **écrire la classe de test** (en s'appuyant sur une structure connue et définie du composant à tester mais sur une implémentation qui n'est pas encore opérationnelle) .
- 3) **lancer une première fois les tests unitaires** et vérifier normalement que "sans code d'implémentation finalisé" les tests remontent bien des "échecs" .
- 4) **écrire le code d'implémentation du composant à tester**. Relancer les tests qui devraient normalement réussir.
- 5) coder et tester des **variantes** au niveau des tests (paramètres d'entrées volontairement erronés , vérification de remontée d'exception , ...)
- 6) poursuivre de manière incrémentale et itérative (nouvelle méthode,...)

Bonnes pratiques sur tests unitaires

- Tester essentiellement les méthodes publiques (pas les méthodes privées)
- Factoriser (lorsque c'est possible) le code de quelques tests (en appelant des sous méthodes de la classe de test , en héritant de classes utilitaires sur les Tests , ..)
- En cas de test qui ne passe plus , d'abord remettre en question le code du composant à tester , puis ensuite le code du test lui même (qui peut également être bogué ou bien trop simpliste).
- Utiliser éventuellement des "mocks" (composants en arrière plan simulés) pour de multiples bonnes raisons qui doivent cependant se justifier selon le contexte . Trop de "mocks" ralentissent quelquefois le développement et rendre le code des tests moins lisible .

1.3. Tests d'intégration

Portées et contraintes des tests d'intégration

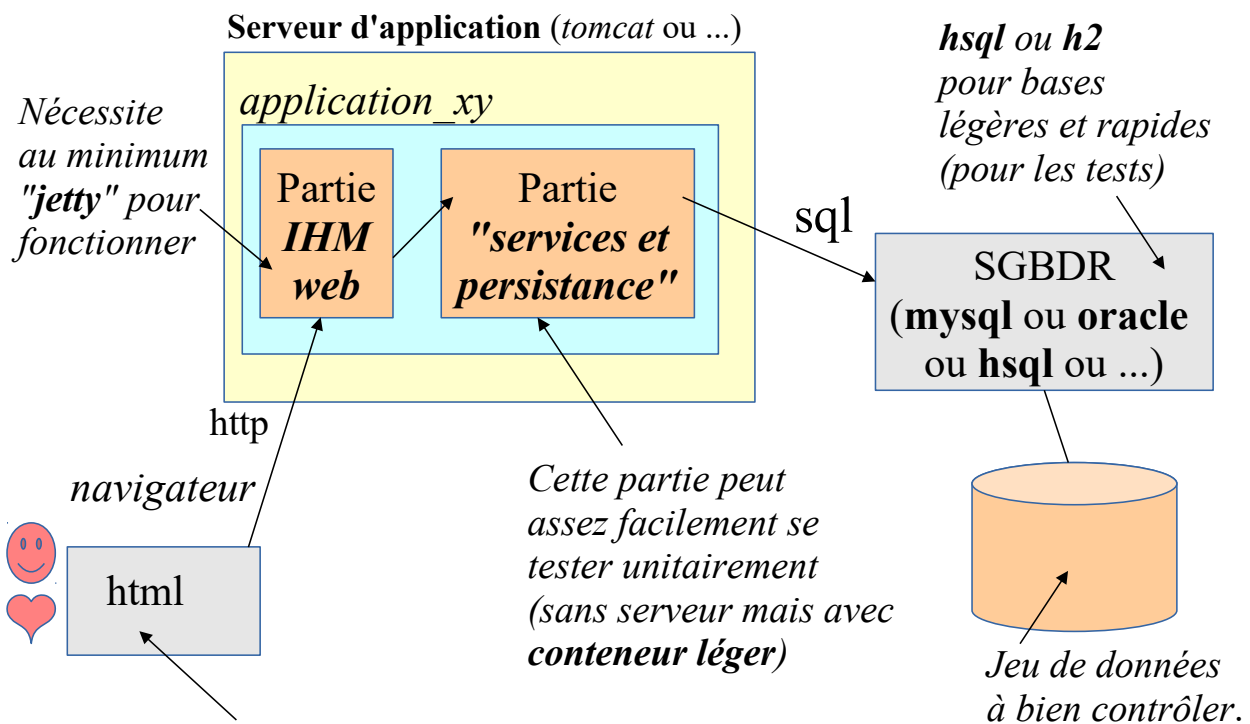
Un **test d'intégration technique** porte généralement sur une application entière qui est quelquefois elle même prise en charge par un serveur d'application (*ex: serveur JEE tomcat , jboss , websphere , ...*).

Une application informatique est généralement composée de plusieurs sous modules complémentaires (partie "IHM/web" , partie "services en arrière plan" ,) et chaque module correspond souvent à un sous projet séparé (géré par "maven" ou autre) .

Un **test d'intégration** nécessite souvent d'**automatiser** les points suivants :

- construction/compilation et assemblage/packaging des modules de l'application
- déploiement et démarrage de l'application au sein d'un serveur d'application (ex : tomcat ou ...)
- Tester la partie web (qui elle même appelle un service qui lui même envoie des requêtes à la base de données)
- Arrêter l'application (et éventuellement le serveur)

Illustrations de certains contextes récurrents



La technologie "**selenium**" permet d'enregistrer certaines séquences "web" pour les rejouer ensuite de façon automatisée

1.4. Doublures et Mocks

Intérêts de "doublures" pour tests (ou ...)

- Le terme *doublure* se comprend bien dans le cadre d'un "*crash test automobile*" : le passager n'est pas une vraie personne humaine mais un mannequin ressemblant (*morphologie proche* , ...) et bourré de "capteurs" pour mesurer l'impact du choc .
- dans le cadre d'un test logiciel , certaines parties peuvent être simulées via des doublures lorsqu'elles sont pas encore prêtes , inaccessibles ou trop lentes.
- on peut simplifier énormément de code d'une doublure par rapport à celui d'un vrai composant fonctionnel tout en rendant son comportement identique : on peut ainsi simplifier l'initialisation des jeux de données et coder le test plus rapidement.
- on peut simuler/forcer des événements exceptionnels (déconnexions , service inaccessible, ...) pour vérifier les remontées d'exceptions .

Principaux types de "doublures"

- **dummy** (*fantôme, bouffon*) : objets avec implémentations "vides" .
- **stub** (*bouchon*) : classe alternative (codée très rapidement) qui renvoie en dur une valeur pour chaque méthode invoquée
- **fake** (*substitut, simulateur*) : sorte de "stub" assez réaliste qui renvoie des valeurs de retour dépendant des paramètres fournis
- **spy** (*espion*) : classe qui espionne les appels entrants pour vérifier (en fin de test) l'utilisation qui en est faite après l'exécution du code.
- **mock** (*simulacre*) : classes qui agissent comme un *stub* et un *spy*

NB: Beaucoup d'aspects (fonctionnalités) des "mocks" sont en général pris en charge dynamiquement par un framework spécialisé (ex : **Mockito**) . ce qui évite de devoir écrire manuellement tout le comportement simulé.

Un bon paramétrage de mock (dans un test unitaire) permet de bien séparer le code réel d'un composant (toujours utile et sans trace) , du code supplémentaire qui n'est temporairement utile que lors d'une phase de test.

Idée du "mock" pour des tests les plus unitaires possibles

Un composant logiciel peut être de granularité plus ou moins fine :

- un sous service "DAO" (Data Access Object avec méthodes CRUD) codé par exemple avec JPA ou JDBC.
- un service métier (utilisant en interne ou en arrière plan des composants "DAO" et "DataSource" pour la persistance et la connexion à la base)
- Un test global/externe (de type boîte "noire") du gros composant "service métier" permettra de tester d'un seul coup "le service , les "DAO" et le "dataSource" avec une vraie base de données en arrière plan)
- Dans certains cas complexes, il peut être difficile de localiser un bug dans tout cet assemblage de sous composants et on sera amené à tester plus finement/unitairement certaines méthodes de la classe principale du service en simulant le comportement des composants "dao" en arrière plan).
- Autrement dit, en simulant certains éléments périphériques, on concentre un test unitaire sur une portion de code pourtant dépendante des autres.

Test comportemental (d'acceptation fonctionnelle)

TDD = **T**est **D**riven **D**evelopment

du côté "*développement*" : JUnit + ...

du côté "*acceptation fonctionnelle*" : *JBehave* ou *easyb* ou *cucumber* ou

ATDD = **A**cceptance **T**est Driven Development

BDD = **B**ehavior Driven Development (synonyme de ATDD)

BDD (ou ATDD):

users_stories (fichiers "*.story*" idéalement *accrochés aux Uses Cases UML*)

= *liste de scénarios rédigés de la façon suivante:*

scénario xyz:

Given *contexte*

When *événement ou condition*

Then *comportement attendu*

*Au départ: simple *partie des spécifications fonctionnelles*

*Au final (avec *technologie annexe telle que jBehave*): *réel test (exécutable) d'acceptation fonctionnelle*

II - JUnit (tests unitaires java)

1. Tests unitaires avec JUnit (3 ou 4)

1.1. Présentation de JUnit

JUnit est un *framework* simple permettant d'effectuer des tests (unitaires , de non régression, ...) au cours d'un développement java . [Projet Open source ---> <http://junit.sourceforge.net/> , <http://junit.org>] . JUnit est intégré au sein de l'IDE Eclipse . JUnit existe en versions 3 et 4.

La version 4 utilise des annotations pour son paramétrage (@Test , @Before ,)

1.2. Structure d'une classe de test (ancienne version JUnit3)

(Ancien) JUnit 3

héritage

Conventions
de noms sur
méthodes
setUp(),
testXy()
et
tearDown()

```
import junit.framework.TestCase;

public class CalculateurTest extends TestCase {
    private Calculateur c; //objet/composant à tester

    protected void setUp(){ //initialisation du composant à tester
        c = new Calculateur(); //setUp() appelée avant chaque testXy()
    }

    public void testAdd() {
        assertEquals( c.add(5,6) , 11 , 0.000001 );
        // ou assertTrue(condition_a_vérifier) .
    }

    public void testMult() {
        assertEquals( c.mult(5,6) , 30 , 0.000001 );
    }

    protected void tearDown(){
        // éventuel code de terminaison réinitialisant certaines valeurs
        // d'un jeu de données (méthode facultative) }
    }
}
```

JUnit 3 est basée sur des conventions de nommage:

- La méthode *setUp()* sera appelée automatiquement pour initialiser les valeurs de certains objets qui seront ultérieurement utilisés au sein des tests.
- Chaque test correspond à une méthode de type "*testXxx()*" ne retournant rien (**void**) mais effectuant quelques assertions (*Assert.assertEquals(...)*)
- On peut éventuellement programmer une méthode *tearDown()* qui sera alors appelée après chaque terminé (ex: pour ré-initialiser le contenu d'une base après).

1.3. Structure d'une classe de test (version actuelle JUnit4)

JUnit4 (avec annotations)

Plus besoin
d'hériter de
TestCase
mais
@Before
@After
et
@Test
attendus

```
import org.junit.Assert;
import org.junit.Test; import org.junit.Before;

public class CalculateurTest
{ private Calculateur c;

  @Before /* comportement proche d'un constructeur par défaut*/
  public void initialisation(){
    c = new Calculateur(); // déclenché avant chaque @Test .
  }

  @Test
  public void testerAdd() {
    Assert.assertEquals( c.add(5,6) , 11 , 0.000001 );
    //ou Assert.assertTrue(5+6==11);
  }

  @Test
  public void testerMult() {
    Assert.assertEquals( c.mult(5,6) , 30 , 0.000001 );
  }
}
```

Méthode statique

Il existe @Before , @After (potentiellement déclenchés plusieurs fois [avant/après chaque test]) et @BeforeClass , @AfterClass (pour initialiser des choses "static")

NB: il faut que **JUnit-4...jar** soit dans le classpath /

La démarche conseillée consiste à

- * coder un embryon des classes à programmer (code incomplet)
- * coder les tests (voir précédemment) et les déclencher une première fois (==> échecs normaux)
- * programmer les traitements prévus
- * ré-effectuer les tests (==> réussite ???)
- * améliorer (peaufiner) le code
- ré-effectuer les tests (==> non régression ???) * ...

1.4. Fonctionnement de JUnit

Une instance de "Test JUnit" pour chaque test unitaire !!!

La technologie JUnit (en version 3 ou 4) créer automatiquement une instance de la classe de test pour chaque méthode de test à déclencher.

→ constructeurs , setUp() et méthodes préfixées par @Before seront donc potentiellement appelés plusieurs fois !!!!
 → la notion d'ordre d'appel des méthodes est inexistante (non applicable) sur une classe de test JUnit.

Ceci permet d'obtenir des tests unitaires complètement indépendants mais ceci peut quelquefois engendrer certaines lenteurs ou lourdeurs.

Certains contextes (plutôt "stateless") peuvent se prêter à **des optimisations "static"** (**@BeforeClass** , **@AfterClass**) .

En combinant JUnit4 avec d'autres technologies (ex : SpringTest) , on peut également effectuer quelques optimisations (initialisations spéciales) au cas par cas selon le(s) framework(s) utilisé(s).

Assert.assertNotNull() et **Assert.fail("message")** peuvent être pratiques dans certains cas (exceptions non remontées, ...)

Ordre des méthodes de test sur une classe JUnit4

Une classe de test **JUnit4** peut comporter plusieurs méthodes de test. Chacune de ces méthodes correspond à un **test unitaire** (censé être indépendant des autres) et donc par défaut , pour garantir une bonne isolation entre les différents tests unitaires :

- l'ordre des méthodes de tests déclenchées est non déterministe
- chaque méthode de test est exécutée avec une instance différente de la classe de test

Dans certains cas rares et pointus, on peut préférer **contrôler l'ordre des méthodes qui seront appelées**. Les tests seront alors un peu **moins "unitaires"** et un peu plus "liés".

Ceci peut s'effectuer de deux manières :

- * avec l'annotation **@FixMethodOrder** que l'on trouve sur les versions récentes de JUnit4 .
- * En écrivant une classe contrôlant le déclenchement d'une **série ordonnée de tests unitaires** (avec **@Suite**) .

1.5. @BeforeClass et "static" (optimisations)

JUnit4 (avec "*static*" et "*@BeforeClass*")

@BeforeClass

@AfterClass

attendus pour
gérer des
éléments
"static"

```
import org.junit.Assert;
import org.junit.Test; import org.junit.Before;

public class CalculateurTest
{ private static Calculateur c;

  @BeforeClass /* appelée une seule fois */
  public static void initialisation(){
    c = new Calculateur(); // initialisation "static".
  }

  @Test
  public void testerAdd() {
    Assert.assertEquals( c.add(5,6) , 11 , 0.000001 );
    //ou Assert.assertTrue(5+6==11);
  }

  @Test
  public void testerMult() {
    Assert.assertEquals( c.mult(5,6) , 30 , 0.000001 );
  }
}
```

Méthode statique

1.6. Suite de tests (@Suite)

Suite ordonnées de tests (JUnit4)

Quelques usages potentiels (tests ordonnés):

séquence Create/insert , select , update , select , delete , ...

variantes au niveau intégration continue :

- Suite_tests_essentiel (pour build rapides)
- Suite_complete_tests (pour build de nuit)

exemple :

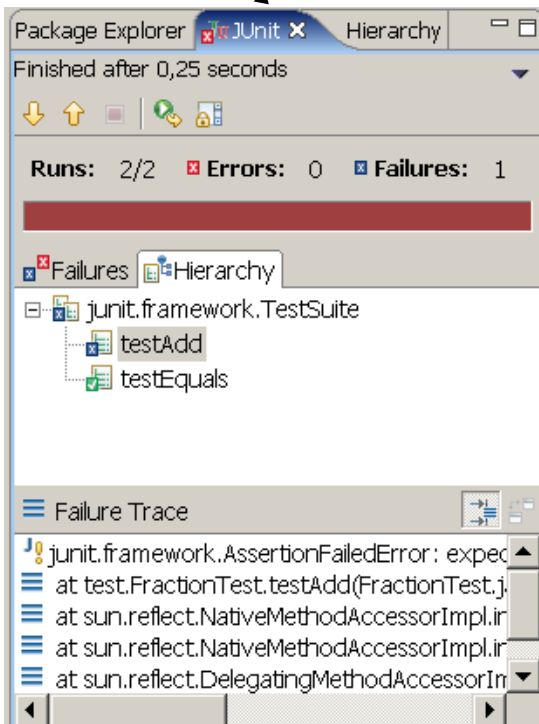
```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
    JunitTest1.class,
    JunitTest2.class
})
public class JunitTestSuite {
}
```

1.7. Lancement des tests unitaires

Lancement des tests unitaires

Depuis l'IDE eclipse:

Run as ... / JUnit test



TestSuite en JUnit3

et lancement après une sélection de **package** en JUnit4 pour lancer d'un coup toute une série de tests.

Comptabilisations :

Error(s) : exceptions java non rattrapées.

Failure(s) : assertions non vérifiées.

VERT si aucune erreur.

Depuis "maven" :

coder des classes nommées "**Test**Xy" ou "XY**Test**" dans **src/test/java** et lancement via **mvn test** ou autre.

Combinaisons de frameworks (de tests) via **@RunWith**

```

import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
...
// nécessite spring-test.jar et junit4.11.jar dans le classpath
@RunWith(SpringJUnit4ClassRunner.class)
// Chargement automatique de "/mySpringConf.xml" pour la configuration
@ContextConfiguration(locations={"/mySpringConf.xml"})
public class TestGestionComptes {

    // injection/initialisation du composant à tester contrôlée par @Autowired (de Spring)
    @Autowired //ou bien @Inject
    private InterfaceServiceXY serviceXy = null;

    @Test
    public void testTransferer(){
        serviceXy.transferer(...); Assert.assertTrue( ... );
    }
}

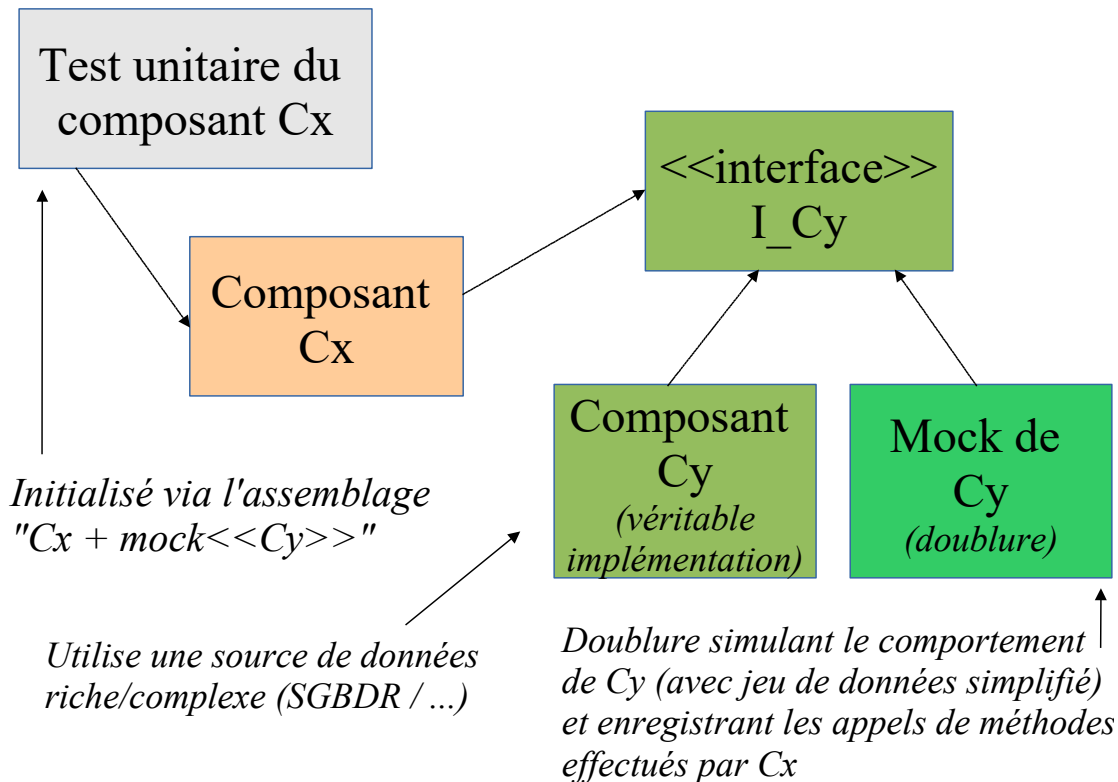
```

Il existe aussi **@RunWith(MockitoJUnitRunner.class)** et autres.

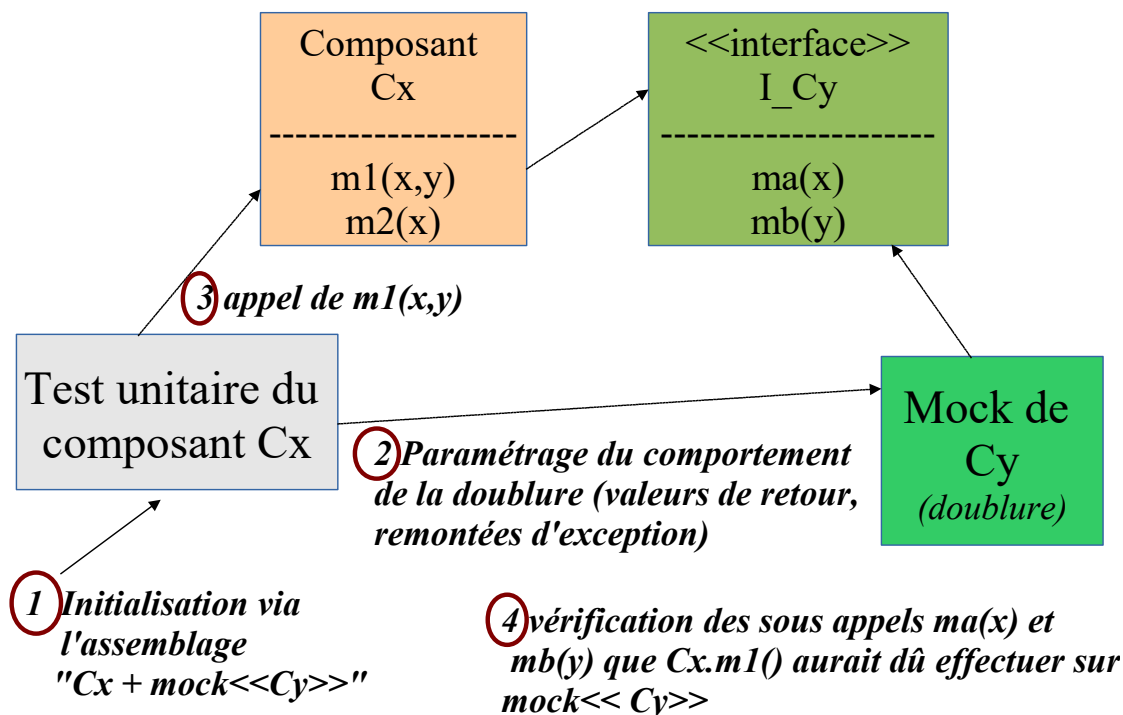
III - Mockito (un des frameworks "mock" java)

1. Positionnement et intérêts des "mocks"

Positionnement des "mocks"



intérêts des "mocks"



2. Mockito

2.1. Présentation de Mockito

Mockito est une technologie java assez populaire pour mettre en œuvre des "Mocks" (simulacres).

2.2. Dépendance maven nécessaire

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <!-- <artifactId>mockito-all</artifactId> -->
  <version>1.10.19</version>
</dependency>
```

2.3. Initialisation du Mock au niveau d'un test Junit:

Solution1 (par annotation "**@Mock**" interprétée du fait de **@RunWith(MockitoJUnitRunner.class)**) :

```
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class UserLoginMockTest {

    @Mock
    private static UserLogin userLogin;
    ....
```

Solution2 (par annotation "**@Mock**" interprétée du fait de l'appel à **MockitoAnnotations.initMocks(this)**; lors de l'initialisation) :

```
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

//rien ou @RunWith(SpringJUnit4ClassRunner.class)
public class UserLoginMockTest {

    @Mock
    private UserLogin userLogin;

    @Before
    public void init() /* or setUp or ... */ {
        MockitoAnnotations.initMocks(this);
    } ...
```

Solution3 (*sans annotation* Mockito , avec un appel explicite à **Mockito.mock()**) :

```
import org.mockito.Mockito;

public class UserLoginMockTest {

    private static UserLogin userLogin;

    @BeforeClass
    public static void init() {
        userLogin = Mockito.mock(UserLogin.class);
    }

    ....
}
```

Soit l'interface fonctionnelle suivante :

```
public interface UserLogin {

    public boolean verifyLogin(String username,String password);
    public String goodPasswordForUser(String username);
    public void setSize(int size);
    //pour tester valeur de retour par défaut:
    public String getAuteur();
    public int getSize(); //return nbAccount ( >=0)
    public double getDoubleValue(double x); //return x * 2
}
```

et soit une classe d'implémentation basique suivante :

```
public class UserLoginImpl implements UserLogin {

    private int size=10; //par défaut
    public boolean verifyLogin(String username, String password) {
        boolean res=false;
        if(password !=null && password.equals("pwd_"+username))
            return true;
        return res;
    }
    public String goodPasswordForUser(String username) {
        return "pwd_" +username;
    }
    public void setSize(int size) { this.size=size;
}
```

```

    }
    public String getAuteur() { return "didier";
    }
    public int getSize() { return size ;
    }
    public double getDoubleValue(double x) { return 2 *x;
    }
}

```

Le comportement de Mockito est alors le suivant :

2.4. Mockito as stub

Comportement par défaut d'un mock (géré par Mockito)

Avec aussi bien

userLogin = ***Mockito.mock***(***UserLogin.class***); //interface

que

userLogin = ***Mockito.mock***(***UserLoginImpl.class***); //classe d'implémentation

tout appel de méthode sur l'objet "mock" géré par mockito retourne une valeur par défaut de type "null" , false , 0 ou 0.0 selon le type de retour :

```

public void displayReturnValues(){
    boolean pwdOk= userLogin.verifyLogin("toto", "pwd_toto");
    System.out.println("pwdOk="+pwdOk);

    String goodPwd= userLogin.goodPasswordForUser("toto");
    System.out.println("goodPwd="+goodPwd);

    String auteur = userLogin.getAuteur();
    System.out.println("auteur="+auteur);

    int taille = userLogin.getSize();
    System.out.println("taille="+taille);

    double val = userLogin.getDoubleValue(3.2);
    System.out.println("val="+val);
}

```

Comportement vraie classe:

pwdOk=true
 goodPwd=pwd_toto
 auteur=didier
 taille=10
 val=6.4

Comportement du mock:

pwdOk=false
 goodPwd=null
 auteur=null
 taille=0
 val=0.0

Préciser (forcer) une valeur de retour via Mockito :

Mockito.when(userLogin.getSize()).**thenReturn**(5);

```
int taille = userLogin.getSize();  
System.out.println("taille="+taille); → affiche toujours taille=5  
userLogin.setSize(20);    taille = userLogin.getSize();  
System.out.println("taille="+taille); → affiche toujours taille=5 (et pas 20 !)
```

On peut forcer un **retour d'exception** selon par exemple certaines valeurs en entrée :

Mockito.when(userLogin.setSize(**Mockito.eq**(-1)))
 .thenReturn(new **IllegalArgumentException**("message xy"));

Lorsque l'on "mock" une classe (et pas une interface), on peut explicitement demander à Mockito de rétablir le comportement de la véritable classe d'implémentation sur certaines méthodes :

Mockito.when(userLogin.getSize()).**thenCallRealMethod**();
Mockito.doCallRealMethod().when(userLogin).setSize(**Mockito.anyInt**());
// il existe aussi Mockito.anyString() , ...

```
userLogin.setSize(20) ;    taille = userLogin.getSize();  
System.out.println("taille="+taille); → affiche taille=20
```


2.5. Mockito as "spy" et "mock = stub + spy"

Comportement par défaut d'un mock initialisé via Mockito.spy()

Fonctionnellement : stub = bouchon (comportement simulé) avec 0,null,0.0 par défaut
.spy() = espionnage (enregistrement des appels pour vérifications ultérieures).
.mock() = comportement "stub" + fonctionnalité de .spy()

`userLogin = Mockito.spy(UserLogin.class); //interface`
 ==> même comportement que via Mockito.mock() car pas de code par défaut

`userLogin = Mockito.spy(new UserLoginImpl()); //classe d'implémentation`

On obtient alors un **comportement normal** (identique à la classe d'origine) **sur toutes les méthodes sauf sur celles où l'on demande explicitement à redéfinir le comportement** :

`Mockito.when(userLogin.getSize()).thenReturn(5);`

Résultats (par défaut)
depuis code précédent :
 pwdOk=true
 goodPwd=pwd_toto
 auteur=didier
 taille=5 (à la place de taille = 10)
 val=6.4

On peut désactiver le comportement d'un setter (ou d'une méthode en void) :

`Mockito.doNothing().when(userLogin).setSize(Mockito.anyInt());`

Autrement dit :

- sémantiquement **mock** = "stub + spy"
- pragmatiquement pas de différence notable entre Mockito.mock() et Mockito.spy() en partant d'une interface car si pas de code d'implémentation ---> toujours comportement "mock=stub+spy" .

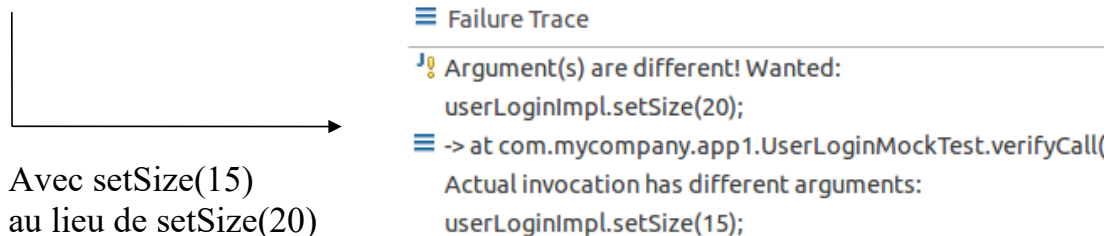
Si par contre Mockito.spy() est appelé en partant d'une réelle instance de composant alors :

- spy --> espionnage seulement pour ultérieur "verify"
 sans changer comportement de l'application
 et mock (= stub + spy)--> verify possible ET implémentation de départ
 écrasée par Mockito.when().then...

Vérification des appels effectués sur un mock (spy) :

Mockito.spy(...) porte bien son nom lorsque l'on sait que l'on peut demander à Mockito d'espionner les appels effectués sur un "mock" et vérifier si certaines méthodes ont bien été appelées (avec certaines valeurs attendues de paramètres en entrée) :

```
@Test
public void verifyCall(){
    userLogin.setSize(15); //userLogin.setSize(20);
    //appel habituellement indirect effectué depuis
    // le code caché d'un composant à tester
    // vers le "spy" ou "stub+spy=mock" d'un sous composant
    Mockito.verify(userLogin).setSize(Mockito.eq(20));
}
```



Quelques exemples de vérifications via Mockito

```
// vérifie que la méthode m1 a été appelée sur obj,
// avec une String strictement égale à "s1" :
Mockito.verify(obj).m1(Mockito.eq("s1"));
// note : ici, le matcher n'est pas indispensable, la ligne suivante est équivalente :
Mockito.verify(obj).m1("s1");

// vérifie que la méthode m2 n'a jamais été appelée sur l'objet obj :
Mockito.verify(obj, Mockito.never()).m2();

// vérifie que la méthode m3 a été appelée exactement 2 fois sur l'objet obj :
Mockito.verify(obj, Mockito.times(2)).m3();

// idem avec un nombre minimum et maximum d'appels :
Mockito.verify(obj, Mockito.atLeast(3)).m3();
Mockito.verify(obj, Mockito.atMost(10)).m3();

// vérifie que la méthode m4 a été appelée sur obj,
// avec un objet similaire à celui passé en argument :
Mockito.verify(obj).m4(Mockito.refEq(obj2));
```

Quelques "matchers" pour vérifier ou paramétrer les valeurs des paramètres :

Mockito. eq (...)	Égal à ...
Mockito. refEq (obj2)	Égal à cet objet
Mockito. anyString() , anyInt() , anyFloat() ,	Chaîne quelconque , entier quelconque, ..
Mockito. anyObject ()	Objet quelconque
Mockito. any (Class<T> c)	Objet d'un certain type
Mockito. anyList ()	Toute implémentation de List
Mockito. argThat (new MyMatcher())	Vérifiant matcher spécifique
...	

On peut définir de nouveaux "matcher" via des classes qui héritent de **ArgumentMatcher<T>**

Exemple de "matcher" personnalisé/spécifique:

```
import org.hamcrest.Description;    import org.hamcrest.Matcher;

public class MyIntegerBetween implements Matcher<Integer>{
    private double inclusiveMini;
    private double exclusiveMaxi;

    public MyIntegerBetween() { super();
        this.inclusiveMini = 0;    this.exclusiveMaxi = 100;
    }

    public MyIntegerBetween(double inclusiveMini, double exclusiveMaxi) {
        super();    this.inclusiveMini = inclusiveMini;
        this.exclusiveMaxi = exclusiveMaxi;
    }

    @Override
    public boolean matches(Object arg0) {
        Integer x= (Integer) arg0;
        if(x>= inclusiveMini && x < exclusiveMaxi)
            return true;
        /*else*/
        return false;
    }
    ...}

```

Utilisation :

```
Mockito.when(serviceTauxCourants.tauxMensPctCourant(
    Mockito.intThat(new MyIntegerBetween(97,1000)))).thenReturn(0.1);
```

IV - Tests de persistance (h2, dbUnit, ...)

1. Embedded database (hsqldb , h2)

1.1. Fonctionnalités

Caractéristiques des bases embarquées

- * pas de serveur (SGBDR) séparé , ni de communication réseau .
- * **fonctionnement directement en mémoire dans le processus client qui effectue les appels JDBC/SQL.**
- * comportement pourtant classique (traitement ordinaire des requêtes SQL avec transactions et verrous).
- * **très rapide**
- * pratique pour les phases de test.

1.2. Modes "in memory" et "local"

2 modes (classiques) d'utilisation (bases embarquées)

* seulement **en mémoire** (et à ré-initialiser à chaque fois) pour des séquences de test :

- a) create table
- b) insert into (jeux de données)
- c) opérations SQL diverses
- d) (éventuellement) drop table (pas nécessaire)

* **avec une réelle persistance locale (sur disque)**

- même comportement (et fonctionnalités) qu'un SGBDR classique mais avec un fonctionnement local (pas d'accès réseau , meilleur performances)
- structures et valeurs conservées après un arrêt et redémarrage.

1.3. hsqldb et h2

Bases embarqués "java" (H2 et HsqlDB)

	H2	HsqlDb
Editeur	Thomas Mueller (H2 signifie Hypersonic 2)	The HSQL Development Group
Driver jdbc	org.h2.Driver	org.hsqldb.jdbc.JDBCDriver
URL (memory)	jdbc:h2:mem <i>ou bien</i> jdbc:h2:mem:dbName	jdbc:hsqldb:mem:mymemdb
URL (disk)	jdbc:h2:file:/opt/dbXy (file : par défaut) jdbc:h2:~/testDB (où ~ désigne \$HOME)	jdbc:hsqldb:file:/opt/testdb
Default auth.	username="sa" password=""	username="SA" password=""
...		

Principales spécificités de HsqlDB

Dépendance "maven" pour HsqlDB :

```
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>2.3.3</version>
</dependency>
```

Principales spécificités de H2

Dépendance "maven" pour H2 :

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.4.190</version>
</dependency>
```

Lancement console web de H2 (url=http://localhost:8082):

```
cd h2/bin
java -jar h2*.jar (ou bien h2.bat ou bien double click sur h2*.jar)
```

Lancement d'un script sql (h2) :

```
java org.h2.tools.RunScript -url jdbc:h2:~/test -user sa -script xy.sql
```

2. DbUnit

Présentation de DbUnit

- * *DbUnit est une extension pour JUnit qui, comme son nom l'indique, est dédiée aux tests de persistance.*
- * *DbUnit sert essentiellement à vérifier que les ordres java (jdbc/sql ou jpa/hibernate ou ...) sont bien répercutés en base (valeurs bien actualisées dans les tables?).*
- * DbUnit permet de **facilement manipuler** des "dataSet" (jeux de données) et tout particulièrement de
 - *initialiser les valeurs en base au début des tests*
 - *récupérer des valeurs de certains enregistrements pour effectuer des comparaisons.*
 - *convertir automatiquement des jeux de valeurs au format XML (ce qui est pratique et lisible)*
- * *En d'autres termes, avec DbUnit (qui ne s'appuie que sur JDBC) on peut tester si les valeurs insérées en base via JPA sont correctes (sachant qu'il ne vaut mieux pas vérifier avec la même technologie que le code effectif de l'application pour éviter des erreurs auto-compensées).*

Principales spécificités de DbUnit

Dépendance "maven" pour DbUnit :

```
<dependency>
  <groupId>org.dbunit</groupId>
  <artifactId>dbunit</artifactId>
  <version>2.5.1</version>
</dependency>
```

En complément de :

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <!-- <scope>test</scope> -->
</dependency>
```

Exemple de jeux de données (en XML)

Jeu de données initial (*src/test/resources/dataset/initialDataSet.xml*):

```
<dataset>
  <!-- <TableName columnName="value1_ofRow1" column2Name="value2_ofRow1"/> -->
  <!-- <TableName columnName="value1_ofRow2" column2Name="value2_ofRow2"/> -->
  <!-- <Table eventuellement sans pkColumn="..." if auto_increment /> -->
  <!-- Attention: il faut respecter un ordre cohérent avec les contraintes d'intégrité référentielles →

  <Compte numCpt="1" label="compte courant" solde="1200.0" />
  ...

  <Operation numOp="1" dateOp="2010-12-24" label="achat xy"
    montant="-50" ref_compte="1" />
</dataset>
```

(sous) jeu de données (pour vérifier mise à jour):

```
<dataset>
  <Client prenom="alex" nom="therieur" dateNaissance="1980-02-11"
    ref_adressePrincipale="1" password="pwd2" telephone="0504030201"
    email="alex.therieur@ici_ou_la" />
</dataset>
```

Exemple d'initialisation de valeurs (java + dbUnit)

```

import org.dbunit.DataSourceDatabaseTester;
import org.dbunit.IDatabaseTester;
import org.dbunit.dataset.IDataset;
import org.dbunit.dataset.xml.FlatXmlDataSetBuilder;

public class BasicDBUnitHelper {

    protected String initialDataSetFileName="initialDataSet.xml"; //by default
    protected String dataSetDirectory="src/test/resources/dataset"; //by default for maven
    protected IDatabaseTester databaseTester;
    protected IDataset initialDataSet=null;
    protected DataSource dataSource; //+get/set

    public void reInitDataBaseFromInitialDataSet() throws Exception{
        //databaseTester = new JdbcDatabaseTester("org.hsqldb.jdbcDriver","jdbc:hsqldb:sample", "sa", "");
        this.databaseTester = new DataSourceDatabaseTester(this.dataSource);

        // initialize your dataset here
        String initialDataSetPathName = this.dataSetDirectory+"/"+this.initialDataSetFileName;
        FlatXmlDataSetBuilder fxdsb = new FlatXmlDataSetBuilder();
        this.initialDataSet = fxdsb.build(new File(initialDataSetPathName));
        databaseTester.setDataSet( this.initialDataSet );
        databaseTester.onSetup(); // will call default setUpOperation (clean_insert by default)
    }
}

```

3. Automatisation tests de persistance

Pistes pour l'automatisation des tests de persistance

- * préparer idéalement un "**DAO générique**" de façon à ce que les ordres CRUD élémentaires puissent être lancés d'une manière uniforme .
Chaque DAO concret de l'application pourra ainsi hériter de ce "DAO générique" et n'ajouter que le code supplémentaire spécifique (findByXy,...) .
- * coder ensuite un *mini framework de "test de persistance"* en s'appuyant à fond sur DbUnit . Ce framework pourra ainsi tester automatiquement les ordres CRUD élémentaires via une séquence automatisée du type :
 - 1) **ré-initialiser les valeurs en bases** via un "*initial-dataSet.xml*"
 - 2) récupérer les valeurs d'un fichier "*new-xy.xml*" pour peupler (par réflexion/introspection) les valeurs d'un objet persistant java .
 - 3) **appeler automatiquement une méthode** de type *persist(entity)* ou *saveNewEntity(entity)* sur le DAO (héritant d'une partie générique).
 - 4) **vérifier l'absence d'exception , relire en java (via le dao) et comparer.**
 - 5) **récupérer via dbUnit un dataSet de l'état d'une table Xy** de la base de donnée . Effectuer une **comparaison** entre le *dataSet* récupéré et la *fusion des dataSet* "*initial-dataSet.xml*" et "*new-xy.xml*" (*coïncidence attendue*).
 - 6) idem pour test de "mise à jour", de "suppression" , de "recherche" , ...

Exemple de DAO générique

```
package org.mycontrib.generic.persistence; //dans generic-jee-back-utils-abstract
...
public interface GenericDao<T,ID extends Serializable> {
    public void deleteEntity(ID pk) throws GenericException; // remove entity from pk
    public void removeEntity(T e) throws GenericException; // remove entity
    public T updateEntity(T e) throws GenericException; // update entity (and return it ref)
    public T getEntityById(ID pk) throws GenericException; //return null if not found
    public T persistNewEntity(T e) throws GenericException; // persist and return e with pk
    public ID persistIdNewEntity(T e) throws GenericException; // persist and return id/pk
}
```

Exemple d'utilisation par héritage

```
public interface DaoDevise extends GenericDao<_Devise,String> {
    public _Devise getDeviseByName(String deviseName);
    public List<_Devise> getAllDevise();
}

public class DaoDeviseJpa extends GenericDaoJpaImpl<_Devise,String>
    implements DaoDevise {
    ...
}
```

Séquence théoriquement re-déclenchable sans conflit**Séquence CRUD classique :**

- 0) *jeu initial de données*
- 1) **ajout** d'une entité
- 2) *relecture (en mémoire) d'une liste d'entité pour vérification de ajout (+vérif. DbUnit/en base)*
- 3) **modification**
- 4) *relecture (en mémoire) de l'entité modifiée pour vérification mise à jour (+vérif. DbUnit/en base)*
- 5) **suppression** de l'entité ajoutée
- 6) *tentative de relecture de l'entité retirée qui doit normalement remonter une exception.*

Exemple de code source au bout de l'URL suivante (pour **git clone**) :

<https://github.com/didier-mycontrib/mycontrib-utils/tree/master/generic-jee-back-test-common/src/main/java/org/mycontrib/generic/test>

Exemple de test de persistance basé sur le mini-framework

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/serviceSpringConf.xml","/dataSourceForTestSpringConf.xml"})
@TransactionConfiguration(transactionManager="txManager",defaultRollback=false)
public class TestDaoDeviseWithGenericAndDbUnit
    extends GenericDaoTestWithDbUnit<_Devise,String>{

    private DaoDevise dao = null; // dao à tester via test CRUD hérité

    @Override
    public String getPkOfEntity(_Devise entity){ return entity.getCodeDevise();
    }

    @Inject
    public void setDao(DaoDevise dao) { this.dao = dao;  this.setGenericDao(dao);
    }

    @Inject
    public void setDataSource(DataSource ds){ super.setDataSource(ds);
    }

    @Test
    public void test_DaoDevise_specific_methods() {...
    }
}

```

En entrées:

▼ > src/test/resources

▼ dataset

initialDataSet.xml

newDevise.xml

updatedDevise.xml

```

<dataset>
  <Devise codeDevise="C" dChange="1.0"
    monnaie="credit international" />
</dataset>

```

Résultats:

Finished after 6,112 seconds

Runs: 3/3 Errors: 0 Failures: 0

▼ tp.app.zz.impl.persistence.dao.test.TestDaoDeviseW

test_DaoDevise_specific_methods (0,303 s)

testGenericDao_CRUD (0,614 s)

testGenericDao_CRUD_InOneTx (0,122 s)

**** test CRUD sur T avec plusieurs petites transactions (via DAO GenericDao<T>) ****

**** test CRUD sur T en une seule transaction (via DAO GenericDao<T>) ****

id(pk) du nouveau Compte créé: C

valeurs initiales de l'entité (créée):

_Devise(dChange=1.0,monnaie=credit international,codeDevise=C)

nouvelle valeur de l'entité (modifiée):

_Devise(dChange=1.0,monnaie=credit intergalactique,codeDevise=C)

entité bien supprimée

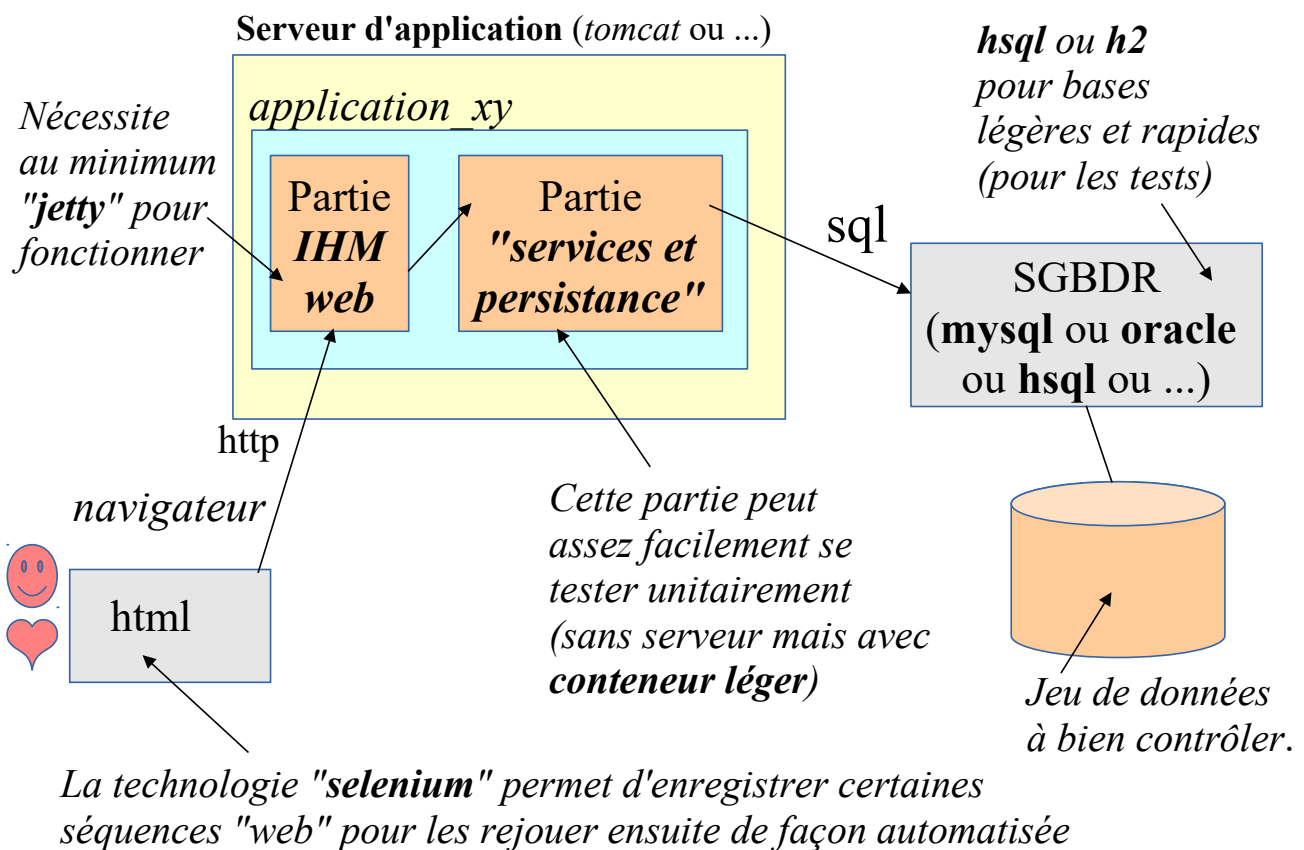
**** end of CRUD test en une seule transaction ****

V - Tests d'intégration (contexte, Selenium, ...)

1. Test d'intégration (contexte , généralités)

1.1. Contexte , cadre général et points à préparer

Rappel : illustrations de certains contextes récurrents



Portées et point d'entrée d'un test d'intégration

Un **test d'intégration technique** porte généralement sur une application entière qui est quelquefois elle même prise en charge par un serveur d'application (*ex: serveur JEE tomcat , jboss , websphere , ...*).

Une application informatique est généralement composée de plusieurs sous modules complémentaires (partie "IHM/web" , partie "services en arrière plan" ,) qu'il faudra assembler (via "maven" ou autre) .

Une fois l'application construite/assemblée , déployée et initialisée (avec si besoin un jeu de données préparé en base), **un test de la partie "en premier plan" (souvent IHM web) permet de tester indirectement l'ensemble de l'application.**

Par exemple, tester simplement si la page d'accueil est accessible permet de détecter un problème de démarrage (mauvaise configuration , incompatibilité dans les modules assemblés , ...).

Points à préparer (par automatisation "maven" ou ...)

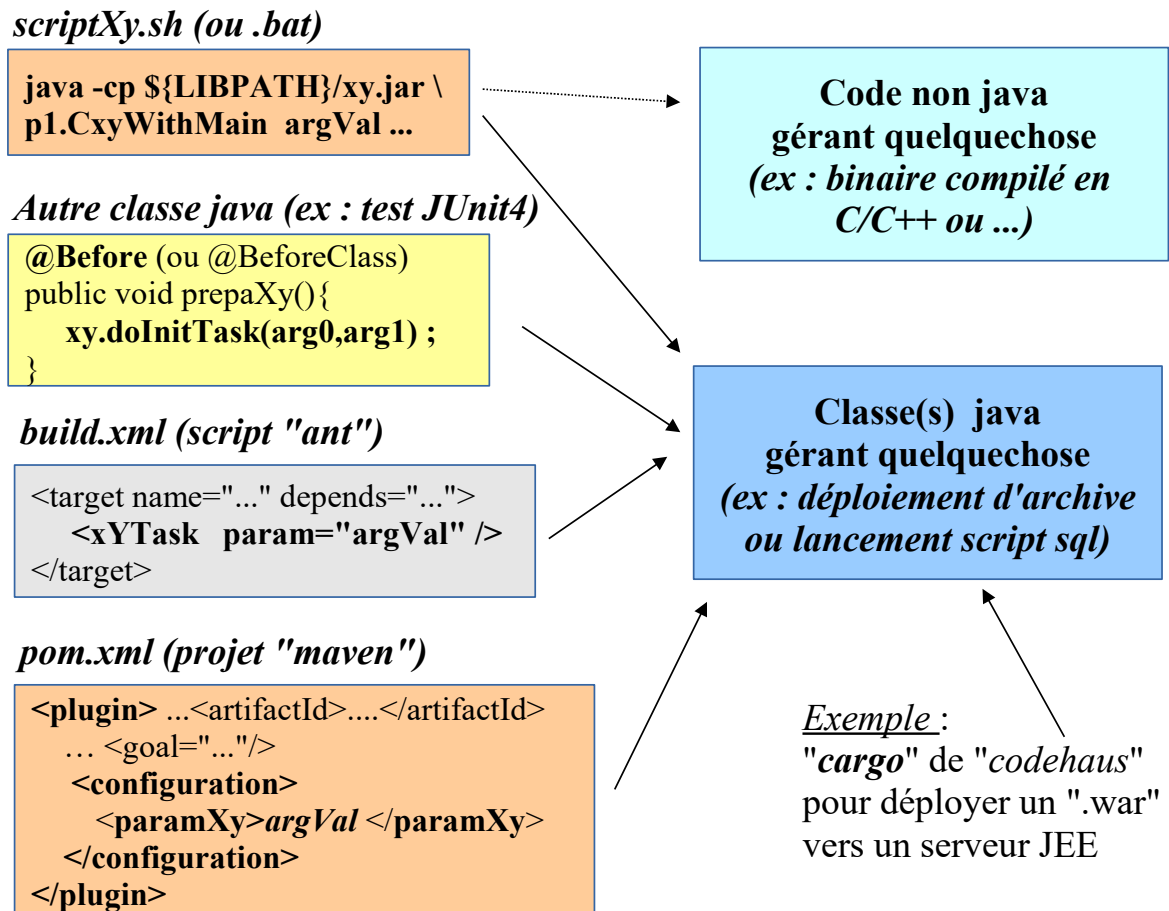
Un test d'intégration nécessite souvent d'**automatiser** la préparation des points suivants :

- construction/compilation et assemblage/packaging des modules de l'application
- Initialisation d'un jeu de données (dans H2 ou MySQL ou ...)
- déploiement et démarrage de l'application au sein d'un serveur d'application (ex : tomcat ou ...)

Les phases "integration-test" de la technologie "maven" ont été spécialement conçues pour automatiser la préparation et le lancement d'un test d'intégration dans le cadre d'une intégration continue d'application java .

1.2. Pistes technologiques pour les tests d'intégration

Pistes pour la préparation des tests d'intégration



Technologies (portables) pour lancer des scripts sql

* Le plugin maven "**sql-maven-plugin**" de "codehaus / mojo" est bien et pratique à utiliser au sein d'un projet **maven**.

* La tâche ant "sql" existe et s'utilise au sein de **build.xml** de la façon suivante : `<sql driver="org.database.jdbcDriver" url="jdbc:database-url" userid="sa" password="pwd" src="data.sql" />`

* Pour un déclenchement par code **java** (puis éventuellement indirectement via un script shell) on pourra coder puis utiliser une **classe utilitaire basée sur l'api JDBC** et en s'inspirant par exemple du code source de la classe "**ScriptRunner**" de **ibatis/mybatis**.

Utilisation du plugin maven "sql-maven-plugin"

```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>sql-maven-plugin</artifactId><version>1.5</version>
  <dependencies> <dependency>    <groupId>postgresql</groupId>
    <artifactId>postgresql</artifactId><version>8.1-407.jdbc3</version>
  </dependency> </dependencies>
  <configuration> <!-- common configuration shared by all executions -->
    <driver>org.postgresql.Driver</driver>
    <url>jdbc:postgresql://localhost:5432:yourdb</url>
    <username>postgres</username> <password>password</password>
    <skip>${maven.test.skip}</skip>
    <!--all executions are ignored if -Dmaven.test.skip=true-->
  </configuration>

  <executions> ...
    <execution>
      <id>create-db</id> <phase>process-test-resources</phase>
      <goals> <goal>execute</goal> </goals>
      <configuration>    <autocommit>true</autocommit>
        <sqlCommand>create database yourdb</sqlCommand>
      </configuration>
    </execution>
  ...

```

Utilisation du plugin maven "sql-maven-plugin" (suite)

```

<execution>
  <id>create-schema</id> <phase>process-test-resources</phase>
  <goals> <goal>execute</goal> </goals>
  <configuration> <autocommit>true</autocommit>
    <srcFiles> <srcFile>src/main/sql/your-schema.sql</srcFile> </srcFiles>
  </configuration>
</execution>
<execution>
  <id>insert-data-before-test</id> <phase>process-test-resources</phase>
  <goals> <goal>execute</goal> </goals>
  <configuration> <orderFile>ascending</orderFile>
    <fileset> <basedir>${basedir}</basedir>
      <includes> <include>src/test/sql/test-data2.sql</include>
        <include>src/test/sql/test-data1.sql</include>
      </includes> </fileset>
    </configuration>
</execution>
<execution>
  <id>drop-db-after-test</id> <phase>test</phase>
  <goals> <goal>execute</goal> </goals>
  <configuration> <autocommit>true</autocommit>
    <sqlCommand>drop database yourdb</sqlCommand>
  </configuration>
</execution>
</executions>
</plugin>

```

1.3. Types classiques de tests d'intégration

Quelques types classiques de tests d'intégration

- Tester une **séquence d'interaction utilisateur** (*rejouer un enregistrement d'utilisation d'un ensemble de pages web et vérifier le bon enchaînement des pages et certains résultats ou messages remontés*) .
Ceci peut (entre autres) être effectué via la technologie "selenium" .
- Tester un service web ("soap" ou "rest") d'une application web préalablement démarrée.
- Tester des communications asynchrones en *envoyant des messages* ("email" ou "JMS" ou ...) dans une destination (*file d'attente* ou ...) liée à une application censée renvoyer des réponses .
- ...

2. Tests web (http/html) via selenium

2.1. Présentation et installation de Selenium

Présentation et installation de Selenium

La suite "Selenium" (de *seleniumHQ*) permet de simplement mettre en place des **tests automatisés d'interfaces graphiques web** (basées sur *HTTP* et *HTML*).

Deux logiciels complémentaires et une extension optionnelle :

- * **Selenium-IDE** = *plugin pour firefox* permettant d'**enregistrer un dialogue HTTP** (pour le rejouer plus tard de façon paramétrable au sein des futurs tests).
- * **Selenium-WebDriver** = api utilisée pour lancer les tests.
- * Selenium-grid = produit facultatif de la suite Selenium pour lancer des tests en parallèle.

NB : Grâce aux enregistrements effectués par Selenium-IDE, on peut assez facilement produire des "tests web" sans avoir absolument à connaître une syntaxe de script particulière.

C'est cette facilité d'utilisation qui a fait la popularité du produit Selenium.

Depuis un navigateur "**Firefox**" récent lancer l'installation du **plugin "selenium-ide"** via une URL ressemblant à la suivante (ici en version 2.8) :
<http://release.seleniumhq.org/selenium-ide/2.8.0/selenium-ide-2.8.0.xpi>

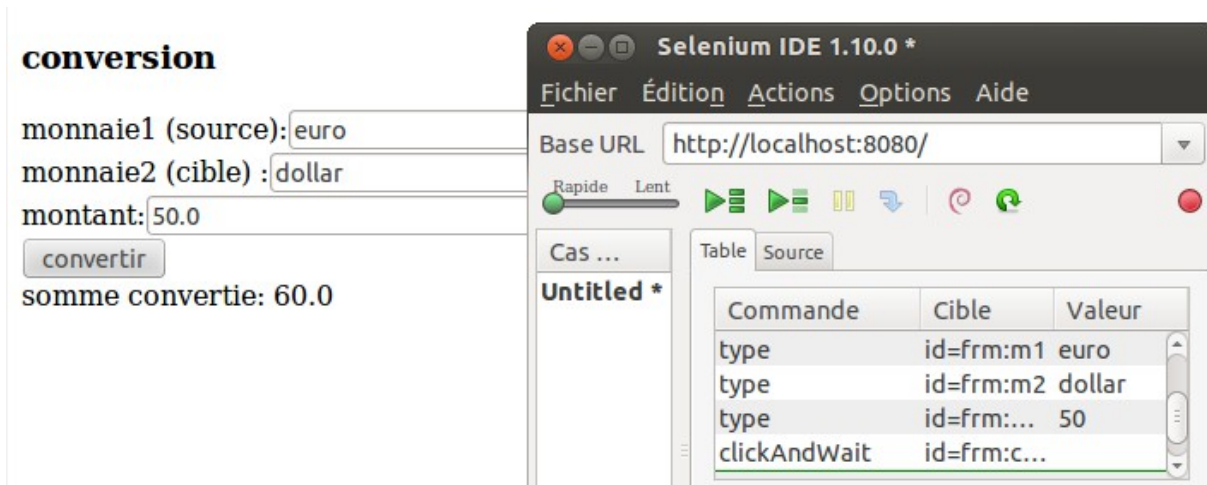
Attention (petit piège JSF et d'autres frameworks WEB) :

Il faut absolument définir des "id" explicites au niveau de chaque composant des pages ".xhtml" de JSF pour que ceux-ci soient stables entre l'enregistrement et les futurs tests (sinon id générés automatiquement et non stables) :

```
<h:form id="frm"> monnaie1 :<h:inputText id="m1" value="#{conv.monnaie1}"/> ...
```

2.2. Enregistrement de séquence via selenium-IDE

Enregistrement de séquence web via selenium-IDE



Etant positionné sur la page d'accueil d'une application web (ex : index.html) on peut déclencher un nouvel enregistrement de séquence "html/http" en activant le menu **"Outils / Selenium IDE"** de **Firefox** .

Il suffit ensuite d'utiliser normalement l'application (clicks sur liens hypertextes , saisies de valeurs dans des formulaires , ...) pour que toutes les actions effectuées par l'utilisateur soient au fur et à mesure enregistrées par le plugin firefox "Selenium IDE" .

Une fois la séquence terminée, il faut cliquer sur l'**icône rouge** "*en cours d'enregistrement, cliquer pour terminer l'enregistrement*".

Enregistrement de séquence web via selenium-IDE (suite)

Pour visualiser la séquence enregistrée, il suffit ensuite de :

- choisir une vitesse (Rapide ou lent)
- cliquer sur l'un des triangles verts ("rejouer l'enregistrement")

A partir de la séquence enregistrée , il est possible de générer directement une classe de test basée sur JUnit4 pour java via le menu "**Fichier / Exporter le test sous ... / Java / JUnit4 / web driver** " de "**Selenium-IDE**".

En choisissant un nom de type "**SequenceYyyyIT.java**" , cette classe de test pourra être utilisée (*de façon "à peine remaniée"*) en tant que test d'intégration déclenchable par maven .

NB : Les versions récentes de Selenium permettent d'utiliser "WebDriver" à la place de "Seelenium-server" !!!!

Cette évolution apporte deux intérêts :

- plus besoin de démarrer et arrêter "selenium-server" au niveau de la configuration du plugin maven pour selenium
- possibilité d'utiliser **HtmlUnitDriver** à la place de *FirefoxDriver* de façon à ce que les tests puissent s'exécuter sans réel navigateur internet (et donc sans aucune interface graphique) sur un serveur d'intégration potentiellement placé sur un ordinateur sans écran.

2.3. Selenium au sein de tests d'integration "maven"

Dépendance "maven" pour utiliser l'api "selenium-java" dans les classes de tests :

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <!-- <artifactId>selenium-server</artifactId> -->
    <artifactId>selenium-java</artifactId>
    <version>2.44.0</version>
    <scope>test</scope>
</dependency>
```

Ceci permet d'utiliser les types "HtmlUnitDriver" et "WebDriver" dans les classes de tests.

Exemple :

```
package tp.app.zz.it.test;

import static org.junit.Assert.fail; import java.util.concurrent.TimeUnit;
import org.junit.After; import org.junit.Before; import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
//import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.htmlunit.HtmlUnitDriver;

public class SequenceWebDriverIT {
    private WebDriver driver;
    private String baseUrl;
    private StringBuffer verificationErrors = new StringBuffer();

    @Before
    public void setUp() throws Exception {
        //driver = new FirefoxDriver();//visible browser during test
        driver = new HtmlUnitDriver(); //invisible browser (with limitations)
        ((HtmlUnitDriver)driver).setJavascriptEnabled(true);

        baseUrl = "http://localhost:8080/";
        driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
    }

    @Test
    public void testIT() throws Exception {
        driver.get(baseUrl + "/my-spring-jeeapp1-web/");
        driver.findElement(By.linkText("bienvenue")).click();
        driver.findElement(By.linkText("conversion")).click();
        driver.findElement(By.id("frm:m1")).clear();
        driver.findElement(By.id("frm:m1")).sendKeys("euro");
        driver.findElement(By.id("frm:m2")).clear();
        driver.findElement(By.id("frm:m2")).sendKeys("dollar");
        driver.findElement(By.id("frm:montantInput")).clear();
        driver.findElement(By.id("frm:montantInput")).sendKeys("60");
        driver.findElement(By.id("frm:convertButton")).click();
    }
}
```

@After

```
public void tearDown() throws Exception {  
    driver.quit();  
    String verificationErrorString = verificationErrors.toString();  
    if (!"".equals(verificationErrorString)) {  
        fail(verificationErrorString);  
    }  
}  
}
```

VI - Tests de comportements / fonctionnels

1. Tests comportementaux (fonctionnels)

Test comportemental (d'acceptation fonctionnelle)

TDD = **T**est **D**riven **D**evelopment

du côté "*développement*" : JUnit + ...

du côté "*acceptation fonctionnelle*" : *JBehave* ou *easyb* ou *cucumber* ou

ATDD = **A**cceptance **T**est Driven Development

BDD = **B**ehavior Driven Development (synonyme de ATDD)

BDD (ou ATDD):

users_stories (fichiers "*.story*" idéalement *accrochés aux Uses Cases UML*)
= *liste de scénarios rédigés de la façon suivante:*

scénario xyz:

Given *contexte*

When *événement ou condition*

Then *comportement attendu*

*Au départ: simple *partie des spécifications fonctionnelles*

*Au final (avec *technologie annexe telle que jBehave*): *réel test (exécutable) d'acceptation fonctionnelle*

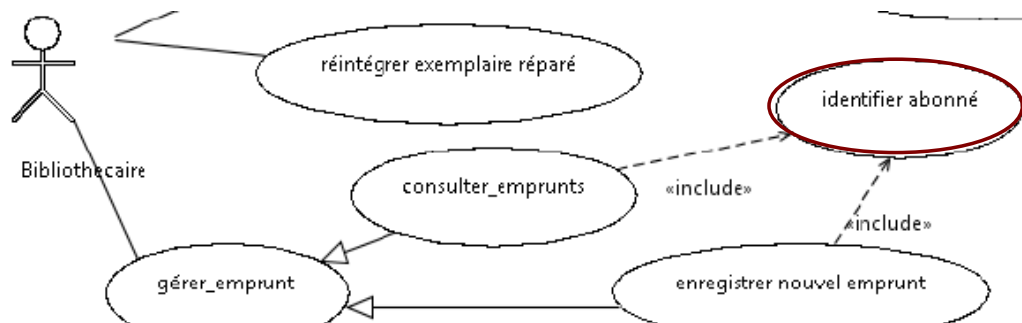
Test comportemental (d'acceptation fonctionnelle)

Certaines technologies (ex: JBehave en java , ... en ruby , ...) sont prévues pour ré-exploiter les fichiers **".story"** et déclencher tous les "test d'acceptation"

Un rapport est généralement produit automatiquement.

Dans le cas particulier de la technologie java "JBehave" , un développeur code une *classe java en arrière plan de chaque scénario* avec des annotations **@Given** , **@When** , **@Then** de façon à associer le scénario à de véritables **étapes (steps)** d'un test exécutable (que l'on peut facilement re-déclencher plusieurs fois).

Exemple (dans spécifications fonctionnelles)



Exemple de "user-stories" pour UC "identifier abonné" :

scenario abonnement valide

given abonne déjà enregistré avec dernier réabonnement depuis moins d'un an

when identifiant valide

then statut ABONNEMENT_VALIDE and message bien identifié

scenario abonnement expiré

given abonné déjà enregistré mais dernier réabonnement depuis plus d'un an

when identifiant valide

then statut ABONNEMENT_EXPIRE

and message abonnement expiré (à renouveler)

JBehave (utilisation)

stories/xy.story

```

Scenario xy :
  Given ctxVal a=6
  When evtMult b=5
  Then resMultAttendu=30

```

```

class MyAbstractJBehaveStories
  extends JunitStories {
    @Override
    public Configuration configuration() {...}
    @Override
    protected List<String> storyPaths() {
      return ... "**/*.story" ;
    }
    @Override
    public InjectableStepsFactory stepsFactory() {
      return .... new XySteps()...
    }
  }

```

steps/xySteps.java

```

public class XySteps {
  private ... contextData1 ;
  private ... statefulData2 ;
  @Given("ctxVal a=$a")
  public void initContextValue(...a)
    contextData1=a ; ...
  }
  @When("evtMult b=$b")
  public void lorsqueMultPar(...b)
    statefulData2=b*contextData1 ;
  }
  @Then("resMultAttendu=$val")
  public void verifResult(...val)
    Assert.assertTrue(statefulData2==val) ;
  }...}

```

⇒ JUnit **success** or **failure**
+ *HTML report*

2. Tests comportementaux avec JBehave

JBehave est l'une des technologies java qui permet de rendre exécutable un test fonctionnel (exprimé avec Given ,When , Then).

2.1. Environnement et démarrage

Jbehave (détails techniques)

Dépendance "maven" pour récupérer jbehave :

```
<dependency>
  <groupId>org.jbehave</groupId>
  <artifactId>jbehave-core</artifactId>
  <version>4.0.4</version>
</dependency>
```

Principaux "import" :

```
import org.jbehave.core.configuration.Configuration;
import org.jbehave.core.configuration.MostUsefulConfiguration;
import org.jbehave.core.failures.FailingUponPendingStep;
import org.jbehave.core.io.CodeLocations;
import org.jbehave.core.io.LoadFromClasspath;
import org.jbehave.core.io.StoryFinder;
import org.jbehave.core.junit.JUnitStories;
import org.jbehave.core.reporters.Format;
import org.jbehave.core.reporters.StoryReporterBuilder;
import org.jbehave.core.steps.InjectableStepsFactory;
import org.jbehave.core.steps.InstanceStepsFactory;
import org.jbehave.core.annotations.AfterScenario;
import org.jbehave.core.annotations.BeforeScenario;
import org.jbehave.core.annotations.Given;      import org.jbehave.core.annotations.Named;
import org.jbehave.core.annotations.Then;      import org.jbehave.core.annotations.When;
import org.jbehave.core.steps.Steps;
```

Classe abstraite MyAbstractJBehaveStories pour le démarrage des tests JBehave

```

public abstract class MyAbstractJBehaveStories extends JUnitStories {
    public MyAbstractJBehaveStories() {
        super();//global jbehave settings:
        this.configuredEmbedder().embedderControls()
            .doGenerateViewAfterStories(true)
            .doIgnoreFailureInStories(false)
            .doIgnoreFailureInView(false)
            .doVerboseFailures(true);
    }
    @Override
    public Configuration configuration() {
        return new MostUsefulConfiguration()
            .usePendingStepStrategy(new FailingUponPendingStep()) //fail if pending / not mapped java
            .useStoryLoader(new LoadFromClasspath(this.getClass()))
            .useStoryReporterBuilder(new StoryReporterBuilder().withDefaultFormats()
                .withFormats(Format.CONSOLE, Format.TXT).withFailureTrace(true)
                .withFailureTraceCompression(true) );
    }
    @Override
    protected List<String> storyPaths() {
        //return Arrays.asList("stories/s1.story", "stories/s2.story");
        URL searchInURL = CodeLocations.codeLocationFromClass(this.getClass());
        return new StoryFinder().findPaths(searchInURL, "**/*.story", "");
    }
}

```

Classe pour le démarrage d'un ou plusieurs tests JBehave

*Lancement de tous les tests "**/*.story" :*

```

public class AllJBehaveTest extends MyAbstractJBehaveStories /* JUnitStories*/ {
    @Override
    public InjectableStepsFactory stepsFactory() {
        return new InstanceStepsFactory(configuration(),
            new CarreSteps() ,
            new CompteBehaviorSteps());
    }
}

```

Lancement d'un seul et spécifique ".story" :

```

public class JBehaveTestCompte extends MyAbstractJBehaveStories /* JUnitStories*/ {

    public InjectableStepsFactory stepsFactory() {
        return new InstanceStepsFactory(configuration(), new CompteBehaviorSteps());
    }

    @Override // default is "stories/*.story" in MyAbstractJBehaveStories.storyPaths()
    protected List<String> storyPaths() {
        return Arrays.asList("stories/decouvert.story"); //seulement un ".story" ici
    }
}

```

2.2. Exemple "Step jBehave"

Exemple de classe dont le comportement sera testé

```
public class Compte {
    public static enum Status {OK, A_DECOUVERT };
    public static double DECOUVERT_AUTORISE = -300.0;
    private Long numero;    private String label;
    private Double solde;    private Status statut;

    public void debiter(double montant){
        double nouveauSolde = solde -montant;
        if(nouveauSolde >= DECOUVERT_AUTORISE){
            solde=nouveauSolde;
        }else
            throw new RuntimeException("decouvert trop important non accepté");
    }
    /*//version avec mauvais comportement fonctionnel détecté par jbehave:
    public void debiter(double montant){
        solde = solde -montant;
    }
    */
    public void crediter(double montant){ solde = solde +montant;
    }
    public Status getStatut() {
        if(solde>=0) statut = Status.OK;
        else  statut = Status.A_DECOUVERT;
        return statut;
    }
    ...}

```

Exemple partiel de classe "....Steps" pour un test jbehave (partie1)

<pre>public class CompteBehaviorSteps extends Steps { private Compte compte ; private RuntimeException eventuelleException; private double ancienSolde,nouveauSolde; private void initialiserCompte(double solde){ compte = new Compte(); compte.setNumero(123L); compte.setLabel("compte courant"); compte.setSolde(solde); } @Given("solde=\$solde positif") public void initialiserComptePositif(@Named("solde")double solde){ Assert.assertTrue(solde>=0); initialiserCompte(solde); } @When("debiter(montant=\$montant)") public void declencherDebit(@Named("montant")double montant){ eventuelleException=null; try { compte.debiter(montant);} catch (RuntimeException e) {eventuelleException= e; } } }</pre>	<p>Scenario: decouvert autorise</p> <p>Given solde=100 positif</p> <p>When avec montant=200>solde et \</p> <p style="padding-left: 20px;">solde-montant >= DECOUVERT_AUTORISE</p> <p>And debiter(montant=200)</p> <p>Then debit accepte</p> <p>And statut = A_DECOUVERT</p>
---	---

Exemple partiel de classe "**....Steps**" pour un test jbehave (partie2)

```

@When("avec montant=$montant>solde et solde-montant >= DECOUVERT_AUTORISE")
public void avecDecouvertAutorise(@Named("montant")double montant){
    Assert.assertTrue(montant > compte.getSolde());
    Assert.assertTrue(compte.getSolde()-montant >= Compte.DECOUVERT_AUTORISE);
}

@Then("debit accepte")
public void constaterDebitAccepte(){
    Assert.assertTrue(eventuelleException==null);
}

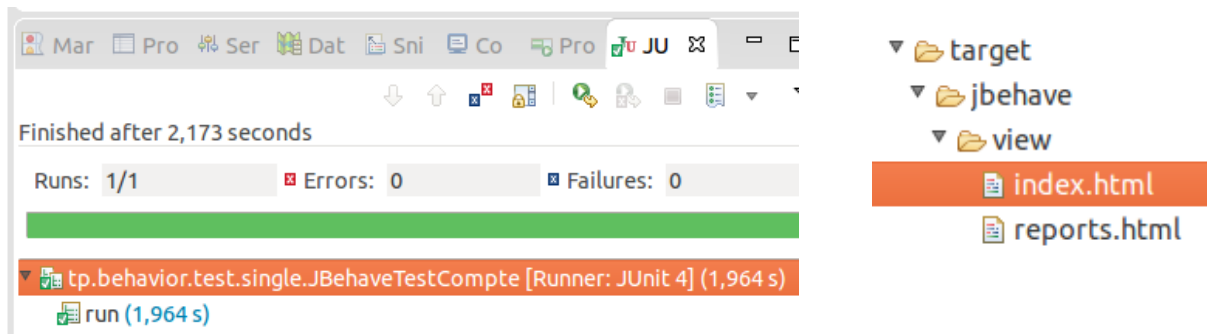
@Then("statut = A_DECOUVERT")
public void constaterStatutADecouvert(){
    Assert.assertTrue(compte.getStatut() == Compte.Status.A_DECOUVERT);
}

/* A priori (d'un point de vue pragmatique), pas de différence entre :
 * Given ... When ... When ... et Given ... When ... And ...
 * ni entre Then .. Then ... et Then .. And ...
 *
 * MAIS TOUS LES Mots clefs (Given , When , Then , And) doivent
 * absolument figurer EN TOUT DEBUT de LIGNE
 *
 * Une seule instance de la classe "...Steps" est utilisée pour gérer tous les scénarios d'un même fichier .story
 * logique "stateful" entre 2 steps d'un même scénario .
 */

```

2.3. Résultats d'un test jBehave

Résultats d'un test jBehave (exemples)



S'il manque une correspondance java :

```
And debiter(montant=70) (PENDING)
Then debit accepte (NOT PERFORMED)
And statut = OK (NOT PERFORMED)
@When("debiter(montant=70)")
@Pending
public void whenDebitermontant70() {
    // PENDING
}
```

Si le comportement attendu n'est pas respecté :

```
Scenario: decouvert non autorise
Given solde=-50 superieur au decouvert autorise
When debiter(montant=600) avec solde-montant <
DECOUVERT_AUTORISE
Then debit refuse (exception) (FAILED)
(java.lang.AssertionError)
And solde inchange (NOT PERFORMED)
```

```
java.lang.AssertionError
    at org.junit.Assert.fail(Assert.java:86)
```

VII - Tests de performances

1. Rôles des tests et bénéfices attendus

Objectifs des tests de performances et de montée en charge :

- **Arbitrer des choix de technologies** (*comparer les perfs de plusieurs technologies et choisir la plus efficace*)
- **Qualification technique** (*vérifier que le minimum attendu est au moins assuré pour garantir un fonctionnement normal / corriger l'application en cas de problème(s)*)
- **Optimisation des ressources** (*détecter et enlever certains goulots d'étranglement pour ne pas sous utiliser les autres ressources matérielles / ne pas gaspiller*).
- **Anticiper** sur des besoins de **redimensionnement** (*nécessité d'agrandir un cluster ?,*)

Différents types de mesures et de tests

Type de mesures et de tests	Moment(s) adéquat(s)	Objectifs / caractéristiques
Profilage (<i>en interne dans une application ou dans un serveur d'applications</i>)	Développement Intégration (dans serveur d'application)	Détecter mauvaise programmation (algorithmes lents, mémoire mal gérée, ...)
Mesures externes des rendements/capacités et de temps de réponse	Développement Qualification Production	Vérifier un fonctionnement correct et/ou déterminer le seuil de saturation
Simulation de charge et <i>courbe de réponses</i>	Qualification	Pas besoin de stresser le serveur en production !

Éléments requis pour la mise en œuvre des tests

- *Simuler/Extrapoler correctement:*
Proche réalité/infrastructure de production
ou bien extrapolations justes/contrôlées.
- *Bien mesurer:*
Savoir utiliser les logiciels de mesures
- *Diagnostiquer correctement:*
**Savoir interpréter les résultats (avoir des
éléments de comparaison)**
- *Résoudre les problèmes: Savoir comment agir .*

Règle (assez générale) des 80/20

- Environ **20% d'efforts** pour obtenir **les
premiers et principaux résultats (80%)**
- **80% d'efforts** pour difficilement obtenir
les derniers résultats (20%)

2. Types de tests (principes , intérêts)

2.1. Différents types d' analyses (vue externe, sondes, ...)

L'analyse des performances comporte divers aspects que l'on peut ranger dans les catégories suivantes:

- Les **mesures non intrusives**: Il n'y a pas de sonde à l'intérieur du serveur. Les traitements ne sont donc pas alourdis mais on ne peut pas connaître les réglages internes à optimiser.
Exemple: **test de montée en charge, Analyse du temps de réponse , analyse du débit supporté (throughput / TPS)**.
- Les **mesures intrusives**: On paramètre le serveur (et/ou l'application) de façon à ce qu'il(s) nous envoie(nt) régulièrement des mesures détaillées sur tel ou tel aspect. Ces sortes de sondes ont tendances à alourdir les traitements donc à fausser légèrement les mesures mais on peut en contrepartie bien cerner ce qu'il est nécessaire d'optimiser (ex: mémoire ou taille d'un pool de connexions, ...).

P.M.I. signifie *Performance Monitoring Infrastructure* .

2.2. Activation des mesures et conséquences

L'activation des mesures (sondes) a tendance à alourdir les traitements du serveur et donc à fausser les mesures elles mêmes (*principe d'incertitude d'Einsenberg*).

On aura donc tout intérêt à effectuer que des mesures ponctuelles et bien ciblées.

3. Profiling (application / serveur)

Profilage (interne) d'une application

Principe: *récolter* via des "*sondes*" tout un tas de *mesures internes précises* de façon à *associer/affecter* précisément des "*coûts*" (*en temps d'exécution ou ...*) aux différentes *fonctions* et sous fonctions du code.

--> **objectif**: *repérer* les parties "*traitements longs*" et "*attentes*" et *essayer de les expliquer et/ou optimiser*.

NB: *il faut avoir quelques éléments de comparaison pour pouvoir interpréter les résultats (bon ou pas ?)*

4. Mesures externes (non intrusives)

Mesures de capacités et de temps de réponse

Principe: effectuer des *mesures à l'extérieur de l'application* (et/ou du serveur logiciel) de façon à connaître ses *capacités (temps de réponse , % d'utilisation CPU , ...)* .

(Exemple: envoyer des requêtes SQL vers un SGBDR(Oracle/Mysql) ou bien des requêtes HTTP vers une application Web (Php ou Asp ou Java_EE) et mesurer le nombre de secondes (ou ms) qui s'écoulent en moyenne avant d'obtenir une réponse.)

Certaines mesures ne sont pertinentes que si le serveur travaille vraiment normalement (déjà entièrement initialisé et bien sollicité par des clients simultanés).

5. Simulation et test de charge

Simulation de charge

Principe: *Envoyer* via un logiciel spécialisé un **paquet de requêtes à un rythme assez élevé** vers un serveur logiciel de façon à **simuler la charge induite par un nombre bien défini de clients simultanés**.

Il s'agit en quelque sorte de "stresser" un peu un certain serveur pour observer son comportement et ses capacités.

Objectif: effectuer différents "tirs" en faisant varier le nombre de requêtes simultanés et synthétiser les mesures résultantes (temps de réponse, ...) au sein d'une courbe de réponses. *Charge maxi supportée = avant saturation*

Pour effectuer des mesures à un rythme assez régulier et pour que l'outil de stress ne fausse pas les mesures, il faut impérativement faire tourner l'utilitaire de test de montée en charge (ex: JMeter) sur un ordinateur différent de celui où s'exécute le serveur à tester.

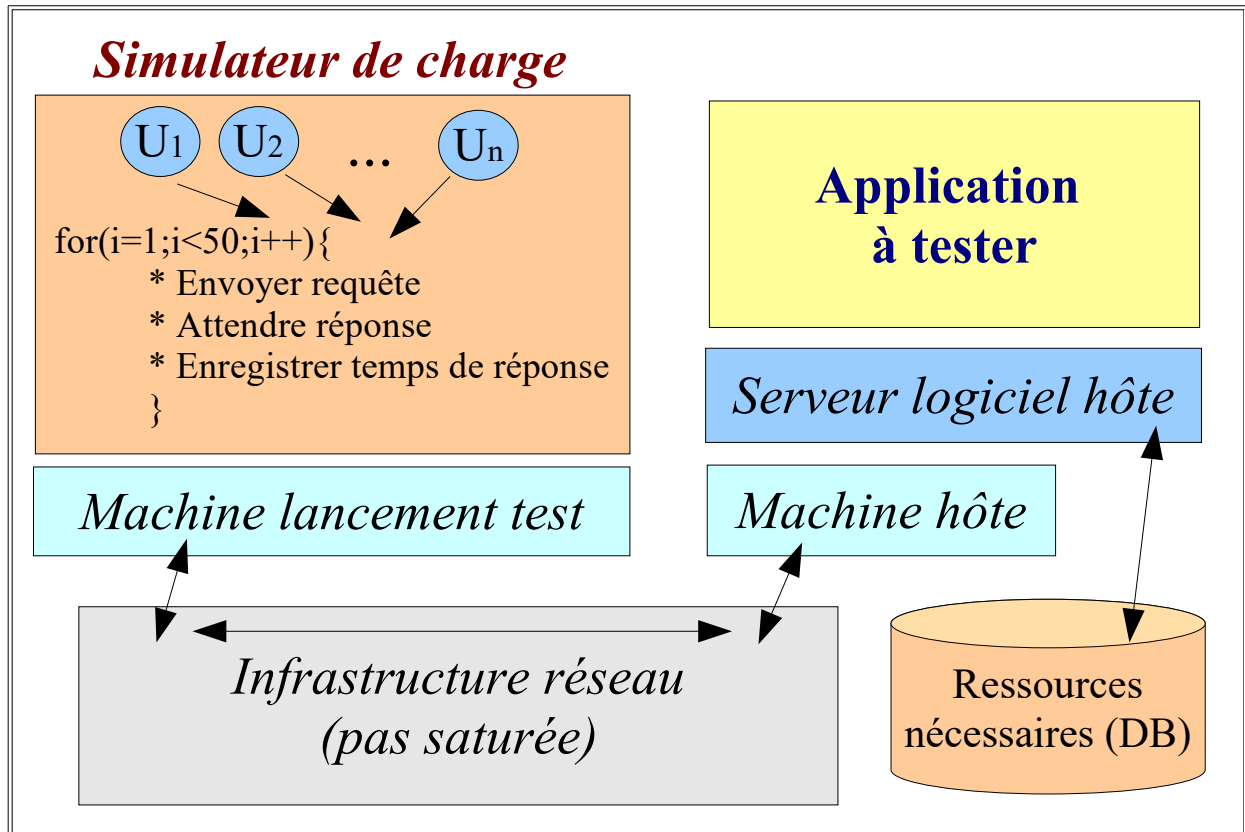
Dans une configuration de test idéale, chacun des éléments suivants doit fonctionner sur une machine séparée:

- JMeter ou équivalent (test de montée en charge , analyse des rythmes et temps des réponses).
- Le serveur d'application (Tomcat ou JBoss ou ...).
- Le SGBDR (MySQL , Oracle , ...) avec les bases de données applicatives.

Etant programmé en java (et devant donc gérer des phases de ramassage de la mémoire [**Garbage Collector**]) l'utilitaire **JMeter** peut quelquefois manquer de régularité et les mesures seront alors assez approximative .

Finalement , pour obtenir des mesures réalistes , il est vivement conseillé d'introduire une certaine variabilité au sein des requêtes envoyées au serveur pour que ce dernier ne se contente pas de nous renvoyer tout le temps le même résultat (éventuellement en cache ou construit d'une façon trop bien optimisée).

6. Topologie de la plateforme de test (charge)



NB: Certains produits sophistiqués sont capables de :

- déclencher simultanément des "tirs de requêtes" émis depuis plusieurs machines clientes
- synthétiser globalement des résultats obtenus .

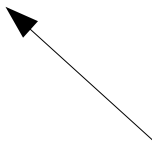
Ceci permet quelquefois d'ajouter un peu plus de réalisme et est avant tout utile si les requêtes chargent beaucoup le réseau ou si le simulateur de charge est installé sur une machine qui n'est pas assez puissante pour solliciter suffisamment le serveur (ou le cluster de serveurs).

7. Configuration des requêtes nécessaires à la simulation

Requêtes pour simulation de charge

- Bien formulées (*selon protocole, ...*)
- Techniquement acceptées (*avec cookies HTTP, champs cachés, ...*)
- Dans le bon ordre (*index, page1 , pageN*)
- Avec arguments variés
- ...

Mise au point par *enregistrement d'une séquence/session réelle* et/ou *configuration pas à pas*



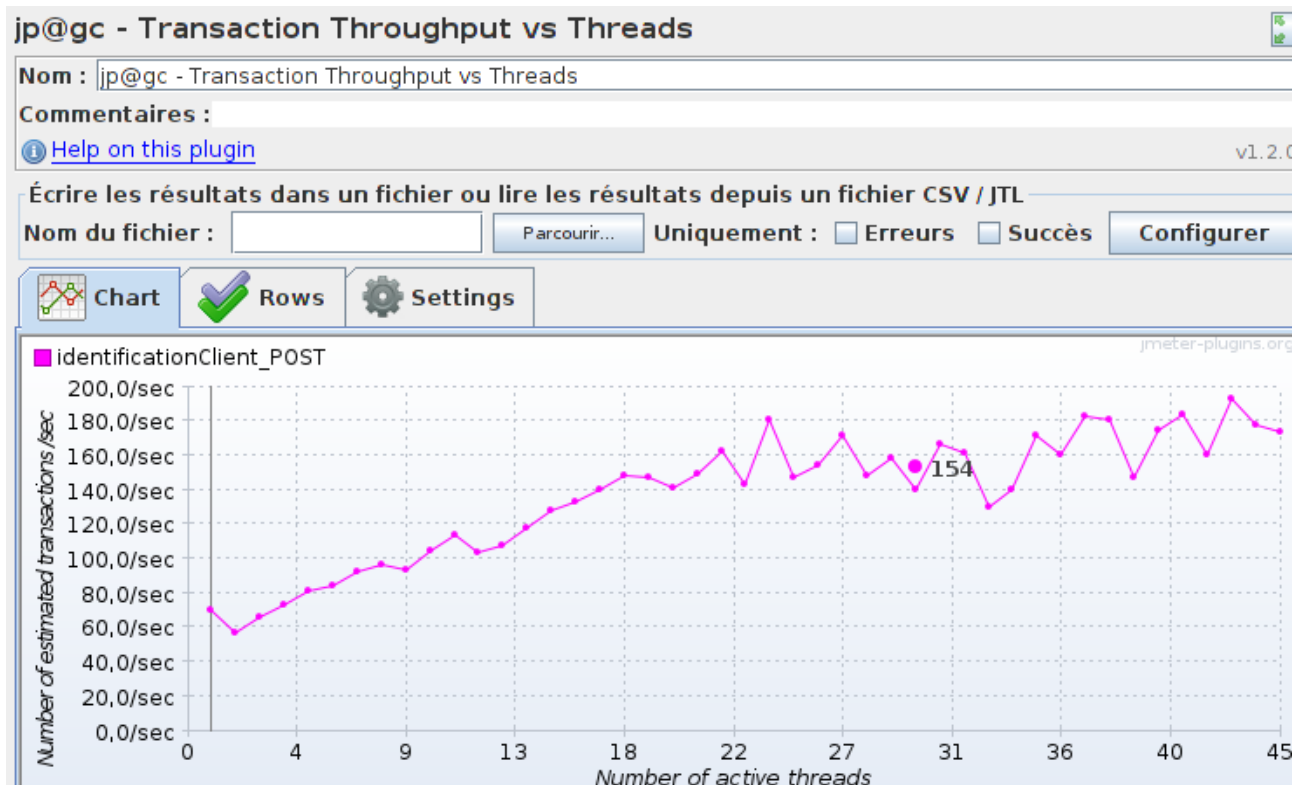
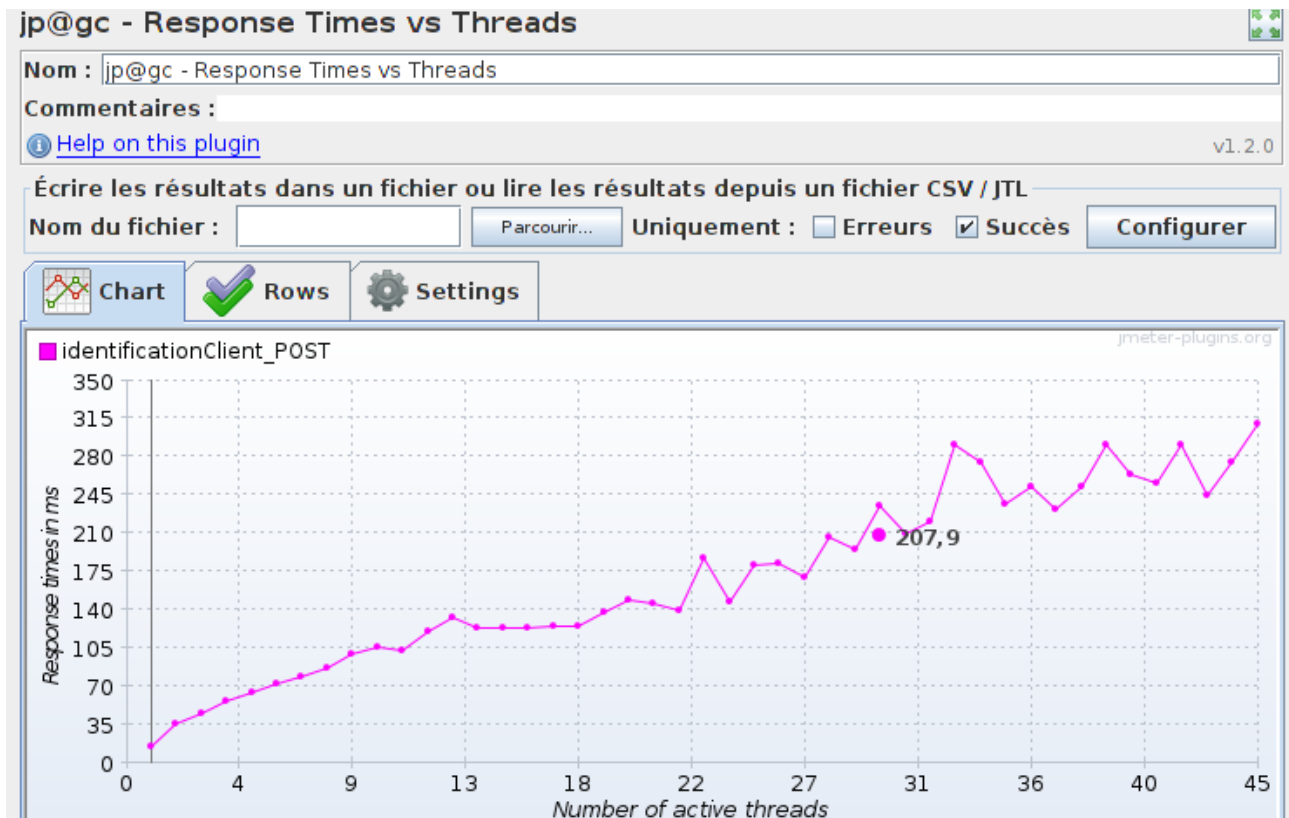
==> Pas si simple (voir même assez complexe avec certains outils).

8. Synthèse des résultats obtenus

Devoir faire soit même un rapport (avec courbe de réponse selon la charge) ou bien paramétrer un automatisme qui le génère automatiquement constitue l'une des différences importantes entre un produit d'entrée de gamme et un produit sophistiqué .

L'utilitaire open source de référence "**JMeter**" de la *fondation Apache* comporte un paquet de plugins dit "*extra*". Les **plugins** "**TransactionPerSecond vs Thread**" et "**ResponseTime vs Thread**" permettent d'obtenir simplement des rapports graphiques synthétiques (si l'on paramètre bien un démarrage progressif du nombre de theads dans le groupe d'unité d'exécution).

Exemples :



Dans l'exemple ci dessus , on visualise assez bien un **seuil de saturation** à partir d'environ 20 threads actifs en parallèle (associé à environ 150 TPS) .

9. Présentation de JMeter

JMeter est un produit **open source** de la communauté **Apache**.

JMeter est capable de **simuler une certaine charge** tout en **récoltant des mesures essentielles** (temps de réponses , débits,) .

JMeter est développé en **java** et est donc:

- multiplateforme (windows , linux, ...)
- soumis à certains cycles de "GC/ramasse-miettes" qui ont tendance à introduire un certain niveau d'approximation dans les mesures recueillies (==> ordre de grandeur correct mais précision limitée) .
- Simple à utiliser (interface graphique perfectionnée).

Principales fonctionnalités du produit "JMeter":

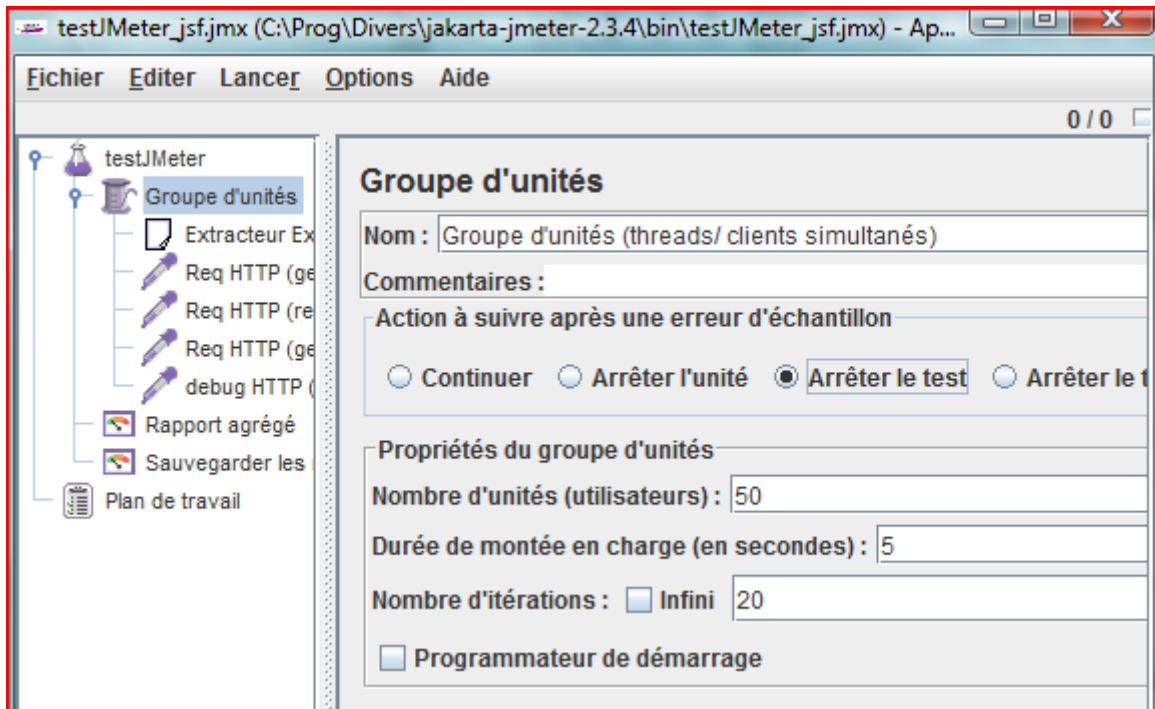
- requêtes HTTP , SOAP , ...
- requêtes SQL (JDBC), ...
- requêtes LDAP , FTP , TCP, ...
- requêtes JMS,
- résultats en graphes , tableaux, rapports,
- vérifications via assertions, comparaisons, sauvegardes fichiers,
- Timer, threads(clients simultanés simulés)
- Pré et post-processeurs (extraction de valeurs via reg-expr, ...)
- Variables, boucles , ...
- Gestion automatique des cookies HTTP (utile entre autres pour identifier les sessions des clients simultanés simulés)
- ...

10. Configuration élémentaire de JMeter

Commencer par renommer le plan de test (ex: MyApp ou TestXY)

10.1. Configuration d'un groupe d'unités (clients simultanés)

Ajouter / Groupe d'unités [Add / Thread group]



10 Threads = 10 clients simultanés

Période de chauffe [*Ramp-up period*] = 5

==> les threads démarrent un par un toutes les $5 / 10 = 0,5$ s.

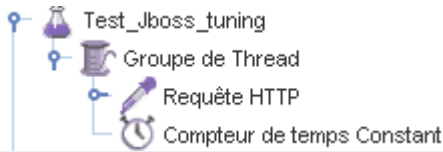
Une fois démarré, chaque thread enverra 20 requêtes (*Loop Count*). Entre chaque requête, chacun des threads attendra par défaut 0 ms.

Pour paramétrer (si besoin) une pause éventuelle (ex: **1000 ms**) entre les requêtes consécutives envoyées par un même thread il faut alors incorporer (sous le groupe de Thread) un **Compteur de temps fixe**[*Timer*].



10.2. Configuration d'une requête à lancer (plusieurs fois)

Sous ce "Groupe d'unités" ==> **Ajouter / Echantillon / Requête HTTP**



Requête HTTP

Nom: Requête HTTP / Conversion

Serveur Web

Nom ou IP du Serveur: localhost

Numéro de Port: 8080

Requête HTTP

Protocole: Méthode: ☒ GET ☐ POST

Chemin: deviseWeb/convlt.jsp ☐ Rediriger Automatiquement ☒ Suivre

Envoyer les Paramètres Avec la Requête:

Nom:	Valeur
somme1	100
monnaie1	Euro
monnaie2	Yen

10.3. configuration de la récupération des résultats (synthèse, courbe, ...)

==> Sous la "Requête HTTP" ou bien sous "le groupe d'unités"

==> **Ajouter / Récepteur / Résultats Graphiques**

==> **Ajouter / Récepteur / Rapport Agrégé (tableau récapitulatif)**

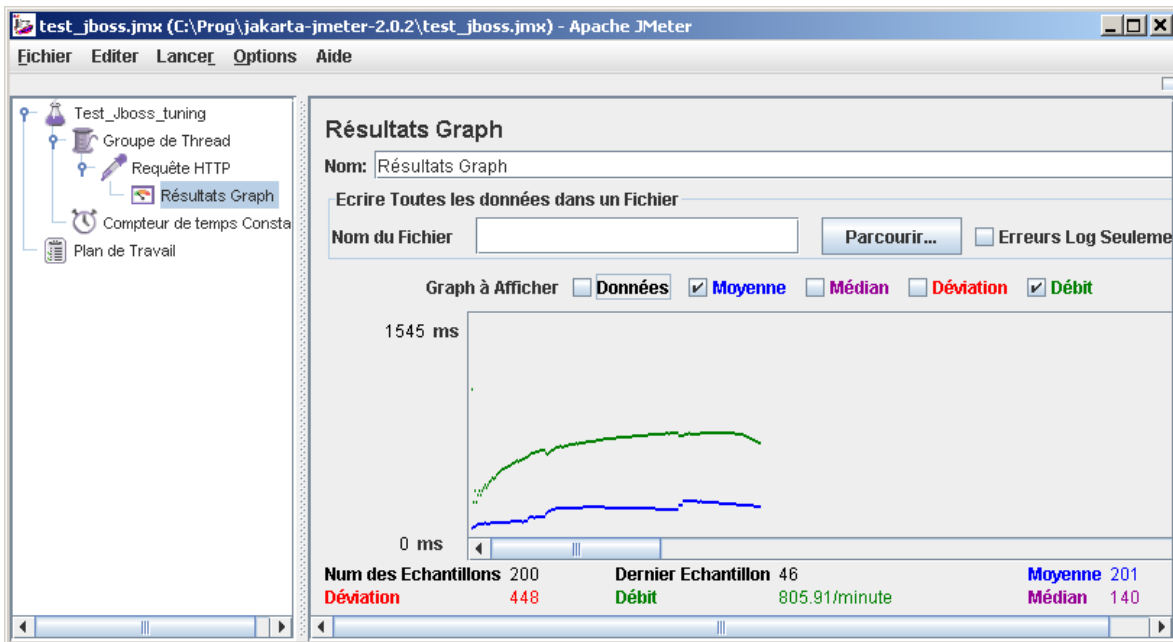
10.4. (re-)lancement du test et observation des résultats

Sauvegarder le "Test Plan" .

Ré-initialiser à zéro des courbes et résultats via "**Lancer / Nettoyer tout** [*Run / Clear All*]"

Activer la simulation de montée en charge via "**Lancer / Démarrer** [*Run / Start*]"

==>



Rapport Agrégé

Nom: Rapport Agrégé

Ecrire Toutes les données dans un Fichier

Nom du Fichier

Parcourir...

☐ Erreurs Log Seulement

URL	Count	Average	Min	Max	Error%	Rate
Requête HTTP	40	41	0	140	0,00%	4,6/sec
TOTAL	40	41	0	140	0,00%	4,6/sec

11. Très important: tester tout d'abord le test !!!

Si l'on ne vérifie pas l'exactitude des paramétrages au niveau des tests, on peut alors confondre une réponse normale avec une réponse de type "message d'erreur" retournée par le serveur. Des temps de réponses liés à des réponses négatives (messages d'erreurs) ne sont évidemment pas significatifs.

Pour être certain que la réponse retournée par le serveur est correcte/normale, il suffit de procéder de la manière suivante (durant la phase de mise au point):

- 1) diminuer (temporairement) au maximum le nombre de requêtes qui seront envoyées:

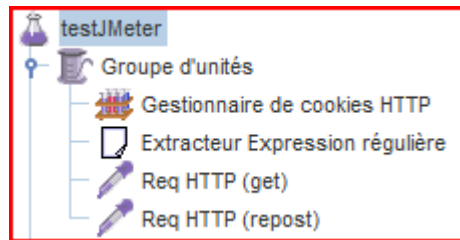
- 2) ajouter un récepteur de type "*Sauvegarder les réponses vers un fichier*"

- 3) **lancer le test et vérifier le contenu du ou des fichier(s) généré(s)**
- à refaire à chaque nouveau re-paramétrage des tests (pour vérifier une non-régression)
- 4) une fois la phase de mise au point terminée, il suffit de **rendre inactif** le récepteur de type "*Sauvegarder les réponses vers un fichier*" pour ne pas ralentir inutilement les tests. En jouant ainsi sur le menu contextuel "activer/désactiver", on peut basculer rapidement d'un mode "debug" vers un mode "test réaliste" et vice-versa. Il faudra également penser à ré-augmenter les nombres de boucles et d'utilisateurs simultanés au sein du groupe d'unité après avoir achever la phase de "debug".

12. Gestion correcte des sessions HTTP (cookies)

De façon à ce que JMeter se comporte exactement comme un navigateur internet au niveau de la gestion des sessions utilisateurs, il faut **absolument ajouter** la configuration "**Gestionnaire de cookies HTTP**" en dessous du niveau "groupe d'unités" .

Rien d'important n'est à paramétrer à ce niveau, mais le "**Gestionnaire de cookies HTTP**" doit simplement être présent .



13. Successions de requêtes http liées entre elles

Un utilisateur utilise généralement une application web en passant par une suite logique de pages (accueil , menu1, menu2 , sélection/recherche , résultats , action , statut ,).

Ces différentes pages activées sont souvent liées entre elles par un ou plusieurs champs cachés (véhiculant un id "utilisateur" ou bien "un état de session" ou "...").

Ces "id" sont générés dynamiquement et ne sont donc pas constants (différents pour chaque utilisateur et pour chaque session).

JMeter offre heureusement un moyen pour "récupérer et ré-injecter" ces "id" véhiculés sous forme de champs cachés **via un processeur d'extraction d'expressions régulières** (ou bien Xpath).

13.1. Principe du "repost-hidden"

repost_hidden.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>repost_hidden</title>
</head>
<%
String chX=request.getParameter("x");
String chLast=request.getParameter("last");
int x=1;
int last=1;
if(chX!=null) x=Integer.parseInt(chX);
if(chLast!=null)
    try{ last=Integer.parseInt(chLast);}
    %>
```

```

    catch (NumberFormatException ex) { last=0; }
int y=last*x;
%>
<body>
    <form>
        x=<input type="text" name="x" /> <br/>
        <input type="hidden" name="last" value="<%=y%>" />
        <input type="submit" value="mult by last value"/>
    </form>
</hr/>
x=<%=chX%><br/>
last=<%=chLast%><br/>
y=last*x=<%=y%> (new last)
</body>
</html>

```

Cet exemple de page jsp montre l'utilisation d'un champ caché permettant de mémoriser une certaine valeur entre plusieurs appels consécutifs émis depuis le même utilisateur.

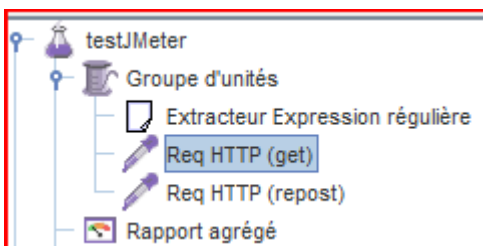
Résultats des ré-appels successifs:

<input type="text" value="x= 3"/> <input type="button" value="mult by last value"/>	<input type="text" value="x= 2"/> <input type="button" value="mult by last value"/>	<input "="" type="text" value="x="/> <input type="button" value="mult by last value"/>
x=null last=null y=last*x=1 (new last)	x=3 last=1 y=last*x=3 (new last)	x=2 last=3 y=last*x=6 (new last)

Nouvelle valeur de y = ancienne valeur multipliée par le x saisi .
L'ancienne valeur (last) est véhiculée via un champ caché (aller/retour "serveur-client-serveur").

```
<input type="hidden" name="last" value="1 , 3 puis 6"/>
```

Test JMeter associé:



Paramétrage de la première requête (en mode GET)

Requête HTTP

Nom : Req HTTP (get)

Commentaires :

Serveur web

Nom ou adresse IP : localhost Port : 8080

Requête HTTP

Protocole (défaut http) : Méthode : GET Encodage du c

Chemin : testJmeter/repost_hidden.jsp

☒ Rediriger automatiquement ☐ Suivre les redirections ☒ Connexions persistantes

Envoyer les paramètres avec la requête :

Nom :	Valeur :

Paramétrage de l'extracteur d'expression régulière (post response processor) :

Extracteur Expression régulière

Nom : Extracteur Expression régulière

Commentaires :

Champs réponse à cocher:

☒ Corps ☐ Corps (non échappé) ☐ Entêtes ☐ URL ☐ Code de

Nom de référence : last

Expression régulière : <input type="hidden" name="last" value="(.*")/>

Canevas : \$1\$

Correspond au num. (0 pour Aléatoire) : 1

Valeur par défaut : 0

==> la valeur du champ caché est ainsi récupérée dans la variable "last" ==> **`${last}`**

Cette valeur est ensuite ré-injectée dans les requêtes ultérieures:

Requête HTTP

Nom : Req HTTP (repost)

Commentaires :

Serveur web

Nom ou adresse IP : localhost Port : 8080

Requête HTTP

Protocole (défaut http) : Méthode : POST Encodage :

Chemin : testJmeter/repost_hidden.jsp

☒ Rediriger automatiquement ☐ Suivre les redirections ☒ Connexions persistantes

Envoyer les paramètres avec la requête :

Nom :	Valeur :
x	2
last	\${last}

13.2. Cas concret du framework JSF

Dans le monde Java, les deux frameworks "WEB" les plus utilisés sont STRUTS et JSF. Le framework JSF (Java Server Faces) aujourd'hui en standard dans JEE5 utilise systématiquement la logique du "repost-hidden" avec un champ caché technique appelé "ViewState":

exemple:

```
<form ...> .... <input type="hidden" name="f_SUBMIT" value="1" />
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
value="C6+wc1LPa1t7RmBc0gq2x7OOkLeUW00e0N6oI3nSK68L7t2CDtFOklqNvxrafMIhcH
TeV/zfHwOSd9Hr8zDSgA==" /> </form>
```

Extracteur Expression régulière

Nom : Extracteur Expression régulière

Commentaires :

Champs réponse à cocher:

☒ Corps ☐ Corps (non échappé) ☐ Entêtes ☐ URL ☐ Code de réponse ☐ Message de réponse

Nom de référence : jsfViewState

Expression régulière : <input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState" value="(.*?)"/>

Canevas : \$1\$

Correspond au num. (0 pour Aléatoire) : 0

Valeur par défaut : 0

expression régulière :

Pour JSF ancien :

```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState" value="(.*?)"/>
```

Pour JSF récent :

```
<input type="hidden" name="javax.faces.ViewState"
id="j_id__v_0:javax.faces.ViewState:1" value="(.*?)"/>
```

Le contenu de la variable *jsfViewState* est ensuite ré-injectée dans les requêtes ultérieures:



Requête HTTP

Protocole (défaut http) : Méthode : **POST**

Chemin :

☒ Rediriger automatiquement ☐ Suivre les redirections ☒ Connexion

Envoyer les paramètres

Nom :	
f.a	2
javax.faces.ViewState	\${jsfViewState}
f.username	didier
f.b	3
f_SUBMIT	1
f.doAddition	additionner

Encodage	Inclure égale ?
<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/>	<input checked="" type="checkbox"/>

avec

pour le paramètre *javax.faces.ViewState* qui comporte quelquefois dans sa valeur un caractère "+" ne devant pas être transformé en caractère espace (" ") par les mécanismes d'encodage HTML.

NB:

- Le paramètre `f_SUBMIT` (valant "1") est un autre champ caché des mécanismes JSF.
- Il ne faut pas oublier d'indiquer le paramètre lié au bouton poussoir (`<h:commandButton />` JSF / `<input type="submit" ..>` HTML) ici "f.doAddition" valant "additionner".
- Une gestion des sessions (via un gestionnaire de cookies HTTP ou ...) est absolument nécessaire pour que JSF fonctionne correctement.

Remarque importante (pour JSF):

De façon à ce que les noms des champs du formulaire html soient intelligibles (ex: "f:a", "f:username") il faut absolument que le développeur précise une valeur au niveau de l'attribut `id` facultatif des balises de JSF:

exemple:

```
<%@ page language="java" contentType="text/html"%>
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
```

```

<html><head><title>page1 (jsf)</title></head>
<body>
<f:view>
  <h3> page1.jsp (jsf) avec h:form et re-post automatique en mode post
</h3>
  <h:form id="f">
    username (sessionScope) = <h:inputText id="username"
                                value="#{myJsfbBean.mySession.username}" /> <br/>
    a (requestScope)= <h:inputText id="a" value="#{myJsfbBean.a}" /> <br/>
    b (requestScope)= <h:inputText id="b" value="#{myJsfbBean.b}" /> <br/>
    <h:commandButton id="doAddition" value="additionner"
                      action="#{myJsfbBean.additionner}" />
  </h:form>
</f:view>
</body>
</html>

```

Sans l'information `id="..."`, les champs du formulaire ont des identifiants par défaut qui sont très peu compréhensibles: `<input id="j_id_jsp_85527204_1:j_id_jsp_85527204_3" name="j_id_jsp_85527204_1:j_id_jsp_85527204_3" type="text" value="0" /> !!!!`

14. Configurations diverses

14.1. Test de charge (SQL) vers un serveur de base de données

NB: **JMeter** peut également effectuer des tests de montée en charge sur un SGBDR (MySQL, Oracle, ...). Il faut penser à ajouter le driver JDBC adéquat au CLASSPATH de JMeter [*Archives ".jar" à déposer dans jakarta-jmeter-2.x.y/lib*]. Ce type de tests (**JDBC Request**) est très pratique pour effectuer des ajustements au niveau des pools de connexions JDBC.

14.2. Variété au niveau des requêtes pour plus de réalisme

Si l'on déclenche en boucle une requête avec toujours les mêmes valeurs de paramètres en entrée, le serveur peut éventuellement optimiser ses traitements internes (via des caches) et retourner extrêmement rapidement la même réponse. Pour mieux simuler des clients simultanés envoyant des requêtes un peu différentes on a alors besoin d'injecter des valeurs variables dans les requêtes.

L'introduction d'un **contrôleur logique de type forEach** (en tant que *parent d'une requête*) permet de répéter celle-ci avec une valeur de paramètre variable (exemple: `${forEachOutputVar}`).

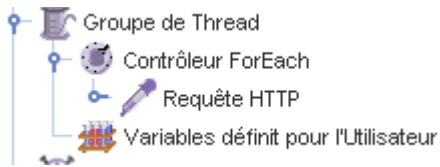
Requête HTTP

Protocole: Méthode: ☒ GET ☐ POST

Chemin: ☐ Rediriger Automatiquement ☒ Suivre

Envoyer les Paramètres Avec la Requête:

Nom:	Valeur
log_level	\${forEachOutputVar}



Contrôleur ForEach

Nom:

Préfix du paramètre en entrée

Nom du paramètre en sortie

La boucle ForEach sera effectuée sur les variables suivantes (*_1, _2, _3, ..., _n*) :

Variables définies pour l'utilisateur

Nom:

Variables définies pour l'utilisateur

Nom:	Valeur
forEachInputVar_1	TRACE
forEachInputVar_2	WARN

Le Listener "**voir l'arbre des résultats**" est dans ce cas très pratique pour contrôler les requêtes et les réponses .

14.3. Plugins utiles

JMeter peut être enrichi par certains plugins.

Le site officiel de JMeter comporte quelques paquets de plugins.

Le zip "**JmeterPlugins-Extras-1.2.0.zip**" (qu'il suffit d'extraire (en ajout) dans le répertoire de JMeter) comporte quelques plugins intéressants (dont "**TPS vs Threads**" et "**ResponseTime vs Threads**").

NB : bien penser à cliquer sur "*configurer*" et cocher "*nombre d'unités actives*"

et dimensionner au mieux la période de chauffe (ramp up period) du groupe d'unités pour que les threads puissent démarrer progressivement (via "plugin "*Stepping Thread Group*" ou ...).

14.4. JMeter à la source:

Pour télécharger **JMeter** ou approfondir certains de ses aspects , l'url de départ est :

<http://jmeter.apache.org/>

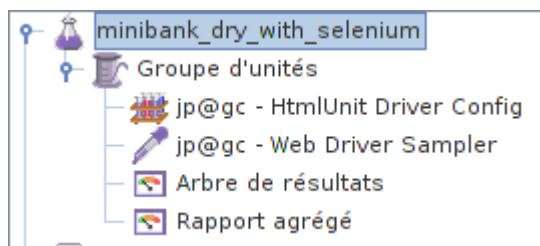
14.5. Plugin "JMeterPlugins-WebDriver-1.2" pour "selenium"

En téléchargeant et extrayant le contenu de *JmeterPlugins-WebDriver-1.2.0.zip* dans le répertoire de JMeter, on peut installer un plugin permettant de déclencher simplement depuis JMeter une séquence "selenium".

L'intérêt principal de ce plugin tient dans le fait qu'une séquence sélénium peut être simplement enregistrée via le plugin "selenium-ide" pour le navigateur "firefox". (voir annexe "selenium")

Mode opératoire :

- Installer si nécessaire le plugin "**selenium-ide**" dans firefox et utiliser celui-ci pour **enregistrer une séquence d'utilisation d'une suite de pages WEB** (navigation, soumission de formulaire, ...).
- Rejouer la séquence pour la vérifier et **exporter** le test sous les formats "**Java / Junit4 / WebDriver**" et "**python2 / unittest/ WebDriver**".
- Au sein d'un projet de test **jMeter**, configurer une arborescence classique ressemblant à celle-ci:



NB : on peut (au choix) utiliser "**firefox Driver Config**" ou bien "**HtmlUnit Driver Config**" (en prenant soin de supprimer les anciens ".jar" selon la documentation du plugin).

- La configuration la plus importante se situe au niveau de "**web Driver Sampler**". Il faut renseigner un script (en langage javascript) permettant de piloter un navigateur web ("firefox" ou "chrome" ou "..." ou l'invisible "htmlUnit").
Ce script peut en grande partie se rédiger en effectuant un copier/coller partiel de l'enregistrement "java" ou "python" du test selenium et en **adaptant** la syntaxe.

Exemple de script attendu par le plugin "Web Driver" de JMeter :

```

WDS.browser.get('http://localhost:8080/minibank-dry/')
WDS.sampleResult.sampleStart() //Start capturing the sampler timing
WDS.browser.findElementByLinkText("identification client").click()
WDS.browser.findElementById("identification:numClient").clear()
WDS.browser.findElementById("identification:numClient").sendKeys(["1"])
WDS.browser.findElementById("identification:btnIdentification").click()
WDS.sampleResult.sampleEnd() //Stop the sampler timing
if(WDS.browser.getTitle() != 'listeComptes.jsp') { // Verify the results:
    WDS.sampleResult.setSuccessful(false)
    WDS.sampleResult.setResponseMessage('Expected title to be listeComptes.jsp')
}
  
```

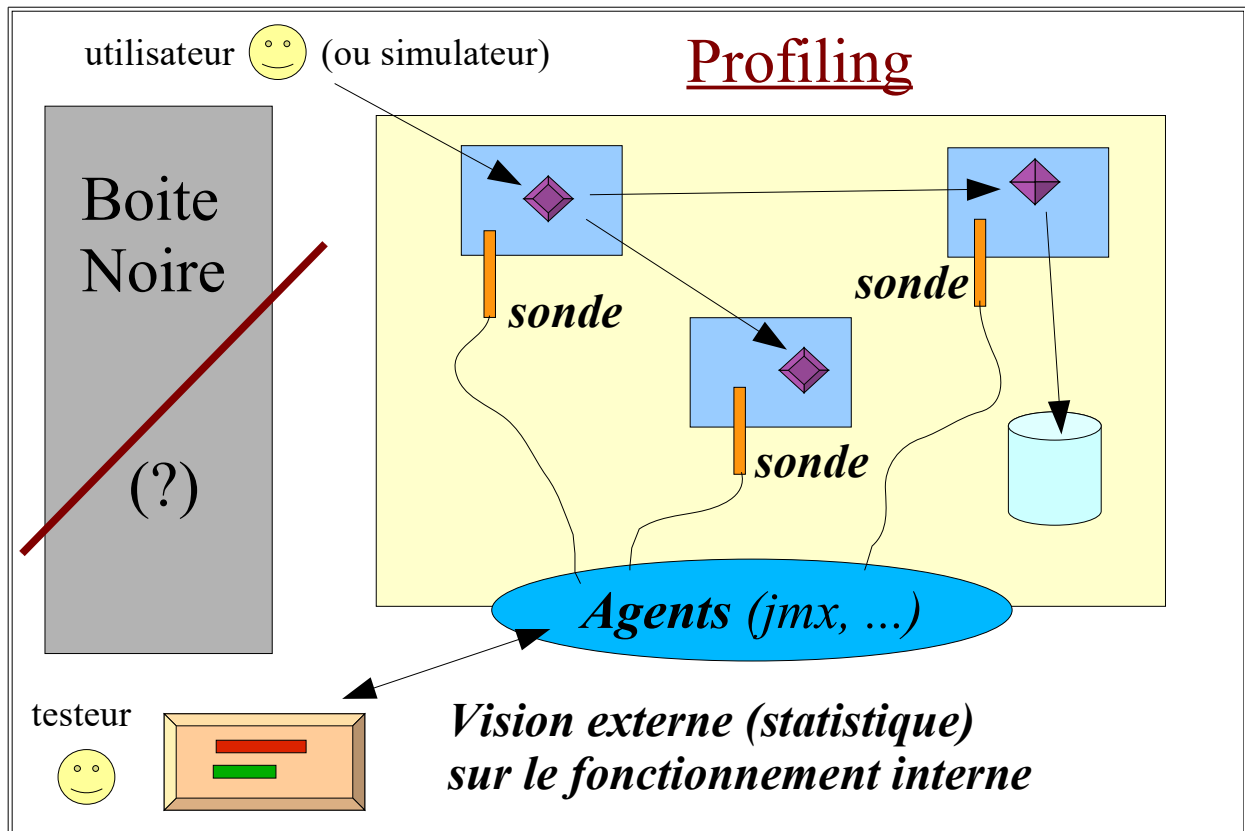
exemple d'enregistrement/export partiel (au format "python") effectué par selenium-IDE :

```

driver.get(self.base_url + "/minibank-dry/")
driver.find_element_by_link_text("identification client").click()
driver.find_element_by_id("identification:numClient").clear()
driver.find_element_by_id("identification:numClient").send_keys("1")
  
```

Attention : la version java/javascript de sendKeys() attend un tableau de string → ajouter des `[]`.

15. Principe du "profiling"



16. JaMon (Java Api for Monitor)

Il existe depuis longtemps une API open source java spécialisée dans les mesures des performances. Cette api s'appelle "**JaMon**" (*Java Api for Monitor*). Elle est simple à utiliser et peut facilement être associée à d'autres technologies java (spring, ejb3, ...).

Nb : il existe une autre api java du même genre (*JMeasurement*). Cependant *JMeasurement* semble être plus complexe et moins au point que *JaMon*.

Nb2 : l'api "**Commons-Monitoring**" (Counter, Gauge, Stopwatch, ...) de la fondation **Apache** est plus récente et pour l'instant un peu plus light/limitée. **Commons-Monitoring** reprend cependant petit à petit la même logique et les mêmes fonctionnalités que *jamonapi* et deviendra peut être le futur standard.

16.1. Intérêt d'une api compatible avec les api de logs

Le principal intérêt d'une api de mesures de performances (telle que *jamon*) réside dans le fait de pouvoir être utilisé à tous les stades (développement, intégration, recette, production) sans dépendre d'un environnement spécifique (pas de version d'eclipse imposée).

16.2. Utilisation élémentaire de l'api "Jmon"

Placer la librairie **jamon-2.79.jar** dans le *classpath* d'une application java (exemple: dans *WEB-INF/lib*).

```
import java.util.Iterator;
import com.jamonapi.Monitor;
import com.jamonapi.MonitorFactory;

public class TestAppJAMon {
    public static void main(String[] args) {
        Monitor mon=null,m=null;
        for (int i=1; i<=10; i++) {
            mon = MonitorFactory.start("myFirstMonitor");
            try { Thread.sleep(100 + i);
            } catch (Exception e) { e.printStackTrace();
            }
            mon.stop();
        }
        System.out.println(mon.toString());
        /*m=MonitorFactory.getMonitor("myFirstMonitor", "ms.");
        // m.reset();
        System.out.println(m);*/
        /*Iterator it = MonitorFactory.iterator();
        while(it.hasNext()){
            Monitor mm = (Monitor) it.next();
            System.out.println(mm);
        }*/
        System.out.println(MonitorFactory.getReport()); //tableau html
    }
}
```

Mesures affichées :

JAMon **Label=myFirstMonitor, Units=ms.**: (LastValue=112.0, Hits=10.0, Avg=105.9, Total=1059.0, Min=101.0, Max=112.0, Active=0.0, Avg Active=1.0, Max Active=1.0, First Access=Wed Nov 07 07:32:04 CET 2012, Last Access=Wed Nov 07 07:32:05 CET 2012)

```
<table border='1' rules='all'>
<th>Label</th><th>Hits</th><th>Avg</th><th>Total</th><th>StdDev</th><th>LastValue</th><
th>Min</th><th>Max</th><th>Active</th><th>AvgActive</th><th>MaxActive</th><th>FirstAcce
ss</th><th>LastAccess</th><th>Enabled</th><th>Primary</th><th>HasListeners</th><th>Labe
l</th>
<tr><td>myFirstMonitor,
ms.</td><td>10.0</td><td>105.9</td><td>1059.0</td><td>3.21282153600563</td><td>112.0</t
d><td>101.0</td><td>112.0</td><td>0.0</td><td>1.0</td><td>1.0</td><td>Wed Nov 07
07:32:04 CET 2012</td><td>Wed Nov 07 07:32:05 CET
2012</td><td>true</td><td>false</td><td>false</td><td>myFirstMonitor, ms.</td></tr>
</table>
```

Label	Hits	Avg	Total	StdDev	LastValue	Min	Max
myFirstMonitor, ms.	10.0	105.9	1059.0	3.21282153600563	112.0	101.0	112.0

...

Remarque: la mesure "Active" (et les variantes associées "*MinActive*", "*MaxActive*", "*AvgActive*") correspond à la notion de "**nombre d'appels simultanés**". "Active" vaut souvent "0" ou "1" en mode mono-thread et peut valoir "2" ou plus en mode multi-thread (ex: si fonctionnement au sein de tomcat avec beaucoup d'utilisateurs simultanés).

Dépendance maven pour jamonapi :

```
<dependency>
  <groupId>com.jamonapi</groupId>
  <artifactId>jamon</artifactId>
  <version>2.79</version>
</dependency>
```

16.3. Activation et désactivation des mesures

La méthode `MonitorFactory.setEnabled(true or false)` permet d'activer ou désactiver globalement toutes les mesures de jamon.

En mode "désactivé / `enabled=false`" les appels à `monitor.start()` et `monitor.stop()` ne font quasiment rien et l'on récupère ainsi un maximum de puissance CPU pour les fonctionnalités métiers de l'application. Ceci peut être quelquefois pratique en mode "production".

Etant donné que l'appel à `MonitorFactory.setEnabled()` peut être effectué lors de l'exécution de l'application (au runtime), l'activation/désactivation des mesures peut ainsi être pilotée par un administrateur (par exemple via la page `jamonadmin.jsp` de `jamon.war`).

Pour paramétrer l'activation de seulement certaines mesures (avec plus de finesse), il faudra au cas par cas régler certains paramètres techniques (url pour applications du filtre web, pointcut spring aop, ...).

16.4. Intégration au sein de Spring 2 ou 3 (pour mesurer les temps d'exécution de la partie "back-office" (services + dao))

```
package org.mycontrib.generic.profiler;
import org.aspectj.lang.ProceedingJoinPoint;

public interface GenericProfilerAspect {
    public abstract Object doProfiling(ProceedingJoinPoint pjp)
        throws Throwable;
}
```

```
package org.mycontrib.generic.profiler;
import org.aspectj.lang.ProceedingJoinPoint;
import com.jamonapi.Monitor;
import com.jamonapi.MonitorFactory;

/**
 * JamonGenericProfilerAspect est une classe d'aspect pour Spring-AOP
 * Elle enregistre les temps d'exécution des méthodes via l'api open source "JaMon" .
 * (pour récupérer les statistiques --> on peut partir de MonitorFactory.getMonitor("signature methode", "ms.");
 * ou de MonitorFactory.iterator() ou encore de MonitorFactory.getReport());
 *
 * @author Didier DEFRANCE
 *
 * exemple de configuration Spring :
 *
 * <bean id="jamonGenericProfilerAspectBean"
 *     class="org.mycontrib.generic.profiler.JamonGenericProfilerAspect"></bean>
 * <aop:config>
 *     <aop:pointcut id="execution_methodes_generic_dao"
 *         expression="execution(* org.mycontrib.generic.persistence.*.*(..))" />
 *     <aop:pointcut id="execution_methodes_dao"
 *         expression="execution(* tp.myapp.minibank.impl.persistence.dao.jpa.*.*(..))" />
 *
 *     <aop:aspect id="myProfilerAspect" ref="jamonGenericProfilerAspectBean">
 *         <aop:around method="doProfiling" pointcut-ref="execution_methodes_generic_dao" />
 *         <aop:around method="doProfiling" pointcut-ref="execution_methodes_dao" />
 *     </aop:aspect>
 * </aop:config>
 * */

public class JamonGenericProfilerAspect implements GenericProfilerAspect {
    public Object monitor(ProceedingJoinPoint pjp) throws Throwable {
        Monitor monitor = MonitorFactory.start(pjp.getSignature().toShortString());
        Object objRes = pjp.proceed();
        monitor.stop();
        return objRes;
    }
}
```

exemple de résultat (getReport() / html) :

Label	Hits	Avg	Total	StdDev
GenericDao.getEntityById(..), ms.	4.0	15.75	63.0	31.5
GenericDao.persistNewEntity(..), ms.	1.0	156.0	156.0	0.0

NB : L'exemple ci-dessus date de 2013.

depuis 2014, les nouvelles versions **2.76** et **2.79** de jamon intègrent maintenant une classe "com.jamonapi.aop.spring.**JamonAspect**" qui est à peu près équivalente à la classe *JamonGenericProfilerAspect* de la page précédente (JamonAspect n'implémente pas l'interface GenericProfilerAspect).

Exemple de configuration spring :

```
<import resource="profilingSpringConf.xml" /> <!-- dans applicationContext.xml -->
```

profilingSpringConf.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .....>

  <bean id="jamonAspectBean" class="com.jamonapi.aop.spring.JamonAspect" />

  <aop:config>
    <aop:pointcut id="execution_methodes_dao"
      expression="execution(* tp.myapp.minibank.impl.persistence.dao.jpa.*(..))" />
    <aop:pointcut id="execution_methodes_services"
      expression="execution(* tp.myapp.minibank.impl.domain.service.*(..))" />

    <aop:aspect id="myJamonAspect" ref="jamonAspectBean">
      <aop:around method="monitor" pointcut-ref="execution_methodes_dao" />
      <aop:around method="monitor" pointcut-ref="execution_methodes_services" />
    </aop:aspect>
  </aop:config>
</beans>
```

16.5. Intégration au sein d'un module EJB3 (alternative à Spring) :

META-INF/ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
  <assembly-descriptor>
    <!-- Default interceptor that will apply to all methods for all
beans in deployment -->
    <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>
        com.jamonapi.aop.JAMonEJBInterceptor
      </interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

16.6. Utilisation de l'api "JaMon" sur la partie JDBC/SQL

NB: le pseudo driver jdbc "JAMonDriver" (dont le code est à l'intérieur de jamon.jar) peut être utilisé pour **enregistrer automatiquement des mesures de performances sur les requêtes SQL** . Celles ci sont à récupérer via MonitorFactory.(.getReport() ou ...)

Paramétrages :

Le nom complet du pseudo driver jdbc de JaMon est `com.jamonapi.proxy.JAMonDriver`

Si l'url jdbc ordinaire est de type `jdbc:mysql://localhost/minibank_db`

et si le driver jdbc ordinaire est `com.mysql.jdbc.Driver`

alors l'URL jdbc à fournir pour jamon est

`jdbc:jamon:mysql://localhost/minibank_db?jamonrealdriver=com.mysql.jdbc.Driver`

Exemple de configuration JDBC complète utilisant le pseudo-driver "JAMonDriver" :

```
<bean id="myDataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <!-- <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost/minibank_db" /> -->

  <property name="driverClassName" value="com.jamonapi.proxy.JAMonDriver" />
  <property name="url" value=
"jdbc:jamon:mysql://localhost/minibank_db?jamonrealdriver=com.mysql.jdbc.Driver" />

  <property name="username" value="root" />
  <property name="password" value="root" />
</bean>
```

Exemple (partiel) de résultats :

Label	Hits	Avg	Total	StdDev	LastValue	Min	Max
MonProxy-SQL-PreparedStatement: insert into Client (ref_adressePrincipale, dateNaissance, email, nom, password, prenom, telephone) values (?, ?, ?, ?, ?, ?, ?), ms.	1.0	15.0	15.0	0.0	15.0	15.0	15.0
MonProxy-SQL-Statement: delete from CLIENT, ms.	1.0	16.0	16.0	0.0	16.0	16.0	16.0
MonProxy-SQL-Type: All, ms.	19.0	13.157894736842104	250.0	24.56796278330048	0.0	0.0	78.0
MonProxy-SQL-Type: delete, ms.	5.0	34.4	172.0	40.33360881448621	0.0	0.0	78.0
MonProxy-SQL-Type: insert, ms.	8.0	9.75	78.0	11.76860229593982	15.0	0.0	32.0
MonProxy-SQL-Type: select, ms.	6.0	0.0	0.0	0.0	0.0	0.0	0.0

16.7. Utilisation de l'api "JaMon" sur la partie Web (via filtre web)

De façon à récupérer automatiquement des mesures de performances sur la partie web (fabrication des pages html via servlet ou jsp) on peut activer un filtre web prédéfini de jamon qui interceptera automatiquement les requêtes http et qui générera des mesures sur les temps d'exécutions .

Ce filtre web (dont le code est dans jamon.jar) se configure dans **WEB-INF/web.xml** de la façon suivante :

```
<web-app>
...
<filter>
  <filter-name>JAMonFilter</filter-name>
  <filter-class>com.jamonapi.JAMonFilter</filter-class>
</filter>






<filter-mapping>
  <filter-name>JAMonFilter</filter-name>
  <url-pattern>/*</url-pattern>  <!-- ou ... -->
</filter-mapping>
</web-app>
```

Comme d'habitude avec jamon, les mesures sont à récupérer via **MonitorFactory.(getReport() ou ...)** .

Etant donné que l'on situe alors forcément dans un environnement web java (ex : tomcat) , on pourra indirectement récupérer (et afficher) les mesures via **jamonadmin.jsp** (de **jamon.war**) ou bien via un équivalent (copie adaptée).

NB : Dans le cas où l'application jamon.war est installée à côté de notreAppliWeb.war dans tomcat , il faudra prévoir une installation de jamon.jar dans les librairies partagées de tomcat (tomcat/lib ou ...) .

Exemple de résultats (ici affichés via une copie de **jamonadmin.jsp**) :

JAMon Action	Mon Proxy Action	Output	Range/Units	Display Columns	Col			
Refresh	No Action (currently=TTTF)	HTML	AllMonitors	Basic Cols Only	###			
Modify	Label ↑	Hits	Avg	Total	StdDev	LastValue	Min	Max
	/minibank-jpa/pages/identificationClient.jsf, ms.	2	66	132	72	117	15	117
	/minibank-jpa/pages/listeComptes_href.jsf, ms.	2	16	32	0	16	16	16
	/minibank-jpa/pages/operations_get.jsf, ms.	2	54	107	1	53	53	54
	/minibank-jpa/pages/virement.jsf, ms.	2	40	81	35	65	16	65
	/minibank-jpa/pages/welcome.jsf, ms.	5	3	15	7	15	0	15

Dans la plupart des cas, c'est un même thread (de tomcat ou ...) qui exécute :

- le code d'un servlet ou d'une page jsp (et les éventuels suppléments struts ou jsf)
- le code d'un service spring (ou ...) en arrière plan
- le code jdbc/sql pour accéder à la base de données

Le filtre web fournira donc des temps d'exécution globaux (front-office/ihm + back-office).

Ces mesures devront théoriquement être assez proches des mesures de "temps de réponse" que l'on peut récupérer via des outils externes de type "JMeter" .

Attention, attention :

Une application java/web démarre généralement très lentement car beaucoup d'aspects dynamiques nécessitent un grand nombre d'éléments à initialiser :

- pages JSP à transformer en servlet et à compiler
- configuration Spring/Hibernate à initialiser (introspection , aop , ...)
- ajustement des zones mémoires (allocations dans le "heap") .
-

Lorsque tout est bien initialisé , une application java fonctionne ensuite très rapidement.

Tout ceci induit donc une période initiale de chauffe où les mesures ne sont pas du tout significatives. Conseil : effectuer un "reset" des mesures après la période de chauffe pour ne pas comptabiliser les mesures non significatives dans les moyennes .

16.8. Temps théoriques à calculer :

tR : temps de réponse observé dans jMeter

tHttp : temps que les requêtes et réponses HTTP passent à traverser le réseau

tGlobalJee : temps d'exécution complet (jee+sql) récupéré par le filtre web de JaMon

tWeb : temps d'exécution purement "ihm web" (fabrication des pages html sans compter les sous-temps back-office)

tReqSQL : temps d'exécution (SQL+JDBC) récupéré par le pseudo driver jdbc de jaMon

tTraitService : temps d'exécution des services métiers (code java Spring ou EJB avec éventuels DAO et avec éventuelles conversions <<entity>>/<<dto>>) sans compter le temps d'exécution "SQL+DataBase"

tGlobalBackOffice : temps complet de la partie back-office (service + sql / db) (se mesure via intercepteur AOP Spring ou EJB3)

Temps récupérés par mesures directes de jaMon (ou JMeter):

tR , tGlobalJee , tReqSQL , tGlobalBackOffice

Temps récupérés par calculs indirects:

tTraitService = **tGlobalBackOffice** - **tReqSQL**

tWeb = **tGlobalJee** - **tGlobalBackOffice**

tHttp = **tR** - **tGlobalJee**

et **tempGlobalPurementJava** = **tGlobalJee** – **tReqSQL**

ANNEXES

VIII - Annexe – Méthodologies avec tests

1. Processus unifiés (UP)

1.1. Meilleures pratiques communes

Processus unifiés (UP)

Processus unifiés (UP) : Il s'agit en fait de processus basés sur une **trame commune** issue des **meilleures pratiques de développements**.

Quelques exemples de processus unifiés:

RUP (Rational U.P.) (origine de UP)

2TUP (2 Tracks U.P.) (Processus en Y)

U.P. basé sur **XP** (eXpérience / eXtreme Programing)

Points communs des U.P.

- * **macro-processus incrémental** (*ex: proto1, proto2, alpha, bêta, v1, v2*)
- * **piloté par les risques** (*technique et architecture appropriée, satisfaire les besoins des utilisateurs, anticiper les Pb, prototypes, tests, validation, ...*).
- * **construit autour de la création et de la maintenance d'un modèle** (*ex: Diagrammes UML, ...*)
- * **itératif** (*itérer sur une succession d'étapes = micro-processus n° i effectué, ré-effectué, peaufiné, ...*).
- * **orienté composant** (*réutilisabilité, déploiement, standardisation, ...*).
- * **centré sur l'architecture du système**

2. Principaux Processus

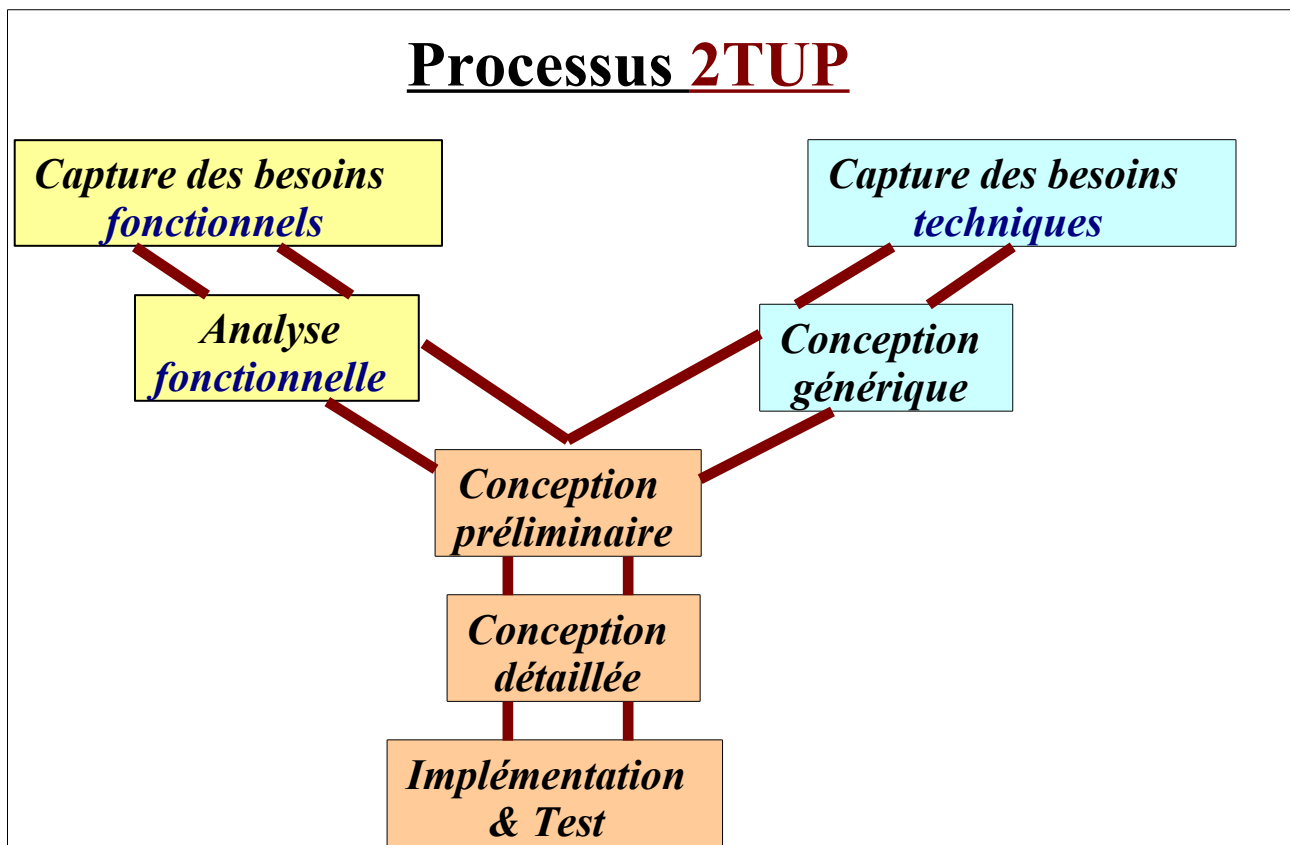
2.1. RUP (Rational UP – origine d'UP)

RUP (Rational U.P.) est le processus U.P. proposé par la **société Rational** (maintenant rachetée par **IBM**). RUP est un précurseur, il est à l'origine même de UP.

Les principales caractéristiques de RUP sont les suivantes:

- Processus très détaillé (beaucoup de choses doivent être *explicitées* sur des *documents à produire*).
Ceci en fait un processus assez "*bureaucratique*" qui convient bien sur des (très) **gros projets**.
- Processus nécessitant des outils sophistiqués et des équipes de travail assez importantes.

2.2. 2TUP (2 tracks UP) – Pocessus en Y



L'originalité de ce **processus en Y** tient dans ses **2 pistes (tracks) initiales bien distinctes**:

- la **branche** de gauche dite **fonctionnelle** ne se focalise que sur les aspects "métiers"
- la **branche** de droite dite **technique** se focalise quant à elle sur les aspects technologiques génériques (ex: sécurité, stockage, transactions, ...).

Une partie délicate de la conception préliminaire consiste alors à **PROJETER** les aspects FONCTIONNELS dans la Solution TECHNOLOGIQUE choisie lors de la conception générique.

[NB: Le livre "UML en action" (entièrement basé sur *2TUP*) permet d'approfondir le sujet]

2.3. XP (eXPerience & eXtreme Programing)

eXPerience

- **Tenir compte de l'expérience (critiques, retours, ...).**
- Via un découpage très fin (parties de UC) on définit des **incréments et des livraisons par parties à des dates fixes**(ex: tous les 3 mois).
- **Planification fine et non figée tenant compte des priorités et des risques**
 → **Renégociations partielles portant sur le contenu à livrer.**
 → **Ré-évaluer souvent la planification du projet en fonction des changements et des retours .**

Pragmatisme (eXtreme Programming)

- Ne pas anticiper sur des évolutions qui n'interviendront peut être jamais.
- **Tester très fréquemment.**
- Effectuer du **refactoring** (refonte & réadaptation du code).
- **Test de non régression.**
- (Typage faible & souplesse) primant sur rigidité.
- Eventuelle programmation par équipe de 2 (une personne qui code , une personne qui corrige immédiatement les erreurs , qui aide , qui donne son avis , ...).

3. Présentation de RUP

R.U.P. (Rational Unified Process) est un des membres de la famille des **processus unifiés (U.P.)**. A ce titre, il se conforme aux bonnes pratiques de développement suivantes:

- **Itératif et incrémental.**
- **Centré sur une architecture modulaire à base de composants.**
- **Focalisé sur les véritables besoins (contraintes techniques, fonctionnalités, ...).**
- **Basé sur des modèles que l'on crée visuellement et que l'on affine.**
- **Vérifications qualitatives (tests, ...).**
- **Intégrer la gestion des changements** (gestion des versions, maintien de la cohérence entre code et modèle, ...)

R.U.P. est développé par la société "**Rational Software**".

R.U.P. est concrètement constitué de:

- Un **important ensemble de pages "html"** constituant une **base de connaissance** et une **sorte de guide (démarche) sur la gestion de projet "orientée objet"**. Ces pages "html" détaillent essentiellement les activités à réaliser pour mener à bien les différentes étapes classiques (Conceptualisation, Analyse & conception, ...).
- Nb: la consultation de ces pages (depuis le site internet Rational Rose) nécessite une inscription préalable (une sorte de période d'évaluation gratuite est prévue).
- Quelques modèles de document (templates)
- Quelques conseils
- Des indications sur l'utilisation des outils de la société Rational (**Tools Mentors**)
- ...

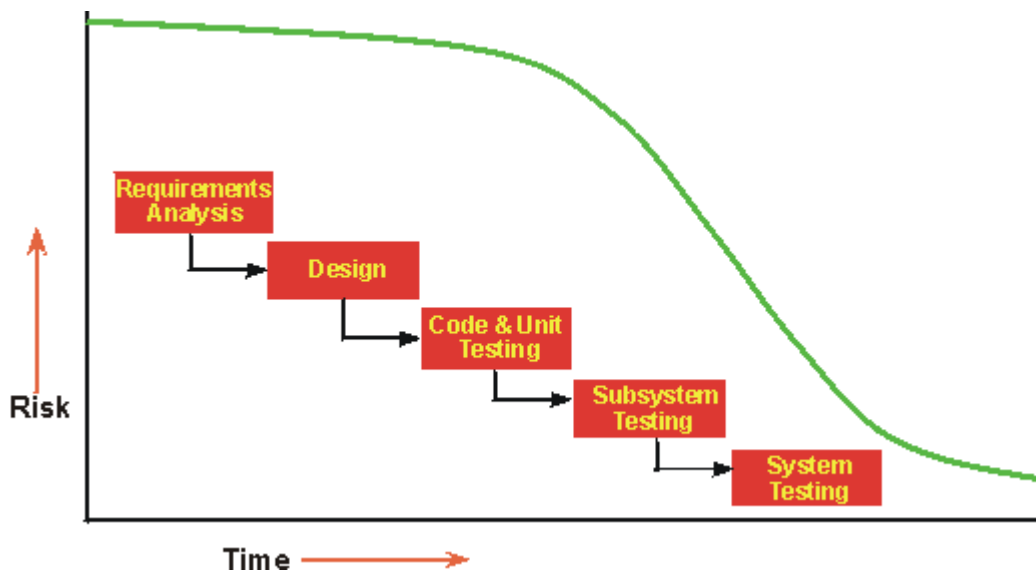
NB: La société "**Rational**" a été rachetée par **IBM**.

RUP fait maintenant partie de **RMC (Rational Method Composer)** qui s'intègre dans eclipse.

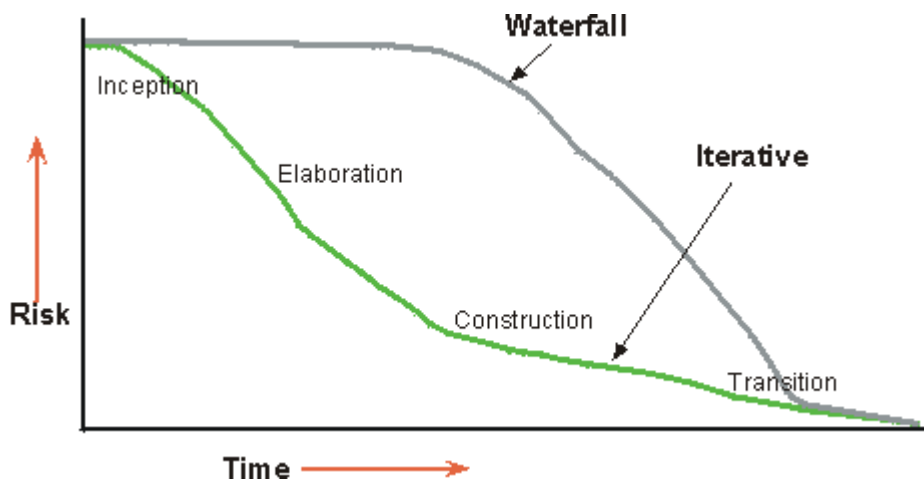
4. Approche itérative



Evolution des risques sans itération:



Evolution comparée des risques via une approche itérative:

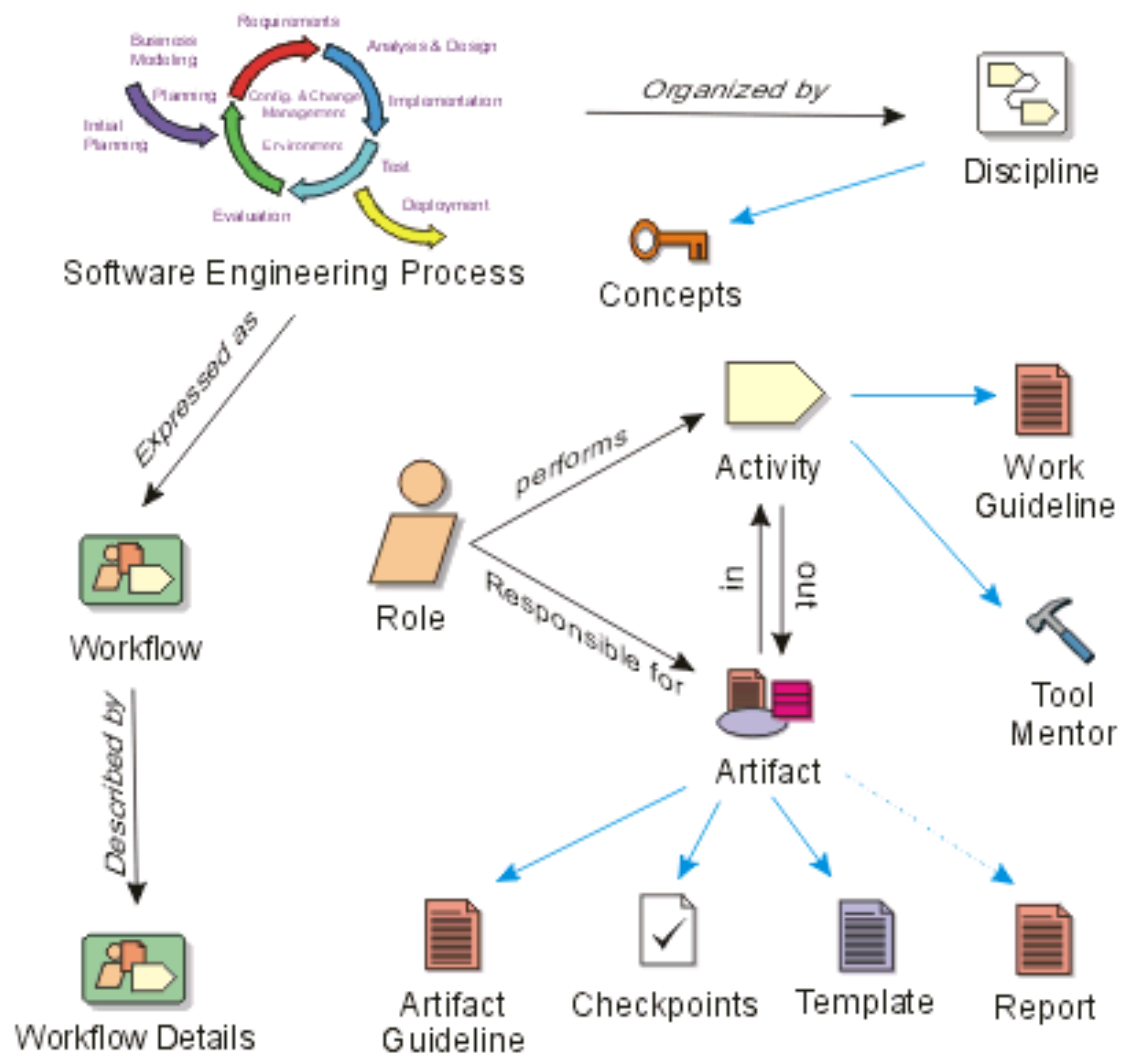


5. Vocabulaire

termes	définitions	exemples
Worker / Role (travailleur/acteur) (fonction)	Rôle ou fonction d'une personne (ou d'un groupe de personnes) travaillant sur un projet.	Concepteur, Architecte, Développeur,...
Activities (activités)	Activité (unité de travail) avec un objectif bien défini.	"Trouver les acteurs et les UC" , "Revoir la conception" , "Effectuer les tests de performance"
Artifact	Document / chose produite	"Modèle" , "Classe" , "Doc" , "Composant" , ...
Workflows	Enchaînements (séquences d'activités réalisées par différents acteurs)	"Capture des besoins" , "Analyse et conception" ,
Disciplines	Collection d'activités ...	Analyse, conception ,

- Qui (Who) ? ==> Workers / Roles

- **Comment (How) ? ==> Activities** (Dans quel contexte ? ==> **Disciplines**)
- **Quoi (What) ? ==> Artifact** (Avec quel outils ? ==> **Tool Mentor**)
- **Quand (When) ? ==> Workflows**



Disciplines de R.U.P. :



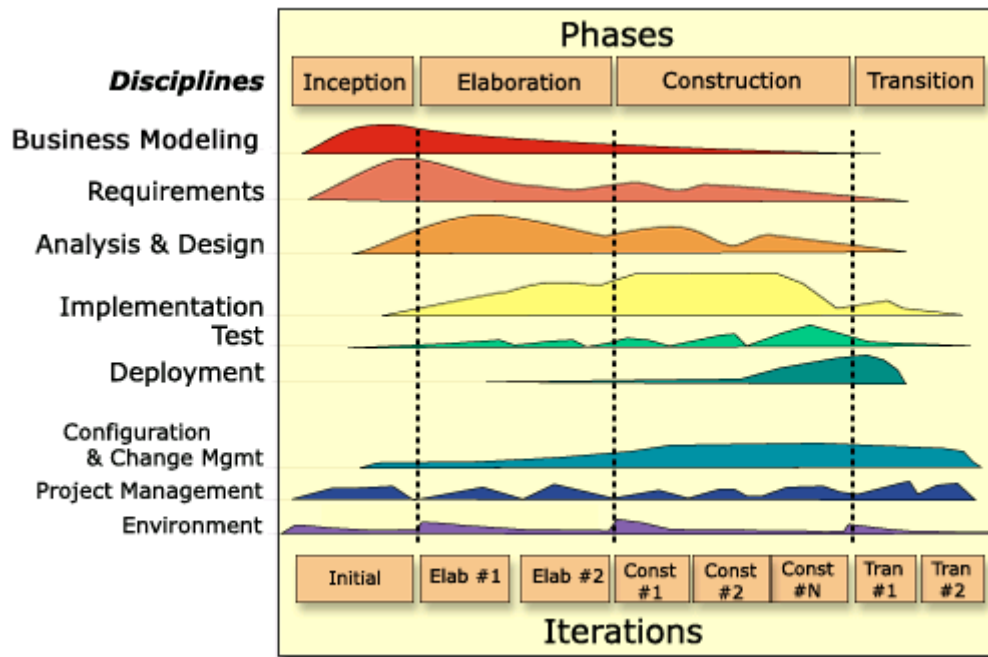
Disciplines
de base (Core)

Disciplines
d'encadrement

6. 4 grandes phases

L'élaboration d'un logiciel est schématisée par:

- Un ou plusieurs **cycle(s)**. Chaque nouveau cycle correspondant alors à une nouvelle génération du logiciel.
- Un cycle est décomposé en 4 grandes **phases**. La fin d'une phase est délimitée par une **borne** de fin de phase (**milestone**). Chaque borne est franchie via **une prise de décision** (est-on prêt à passer à la suite ?).
- Une phase peut éventuellement conduire à effectuer plusieurs **itérations** sur les différentes activités . Une itération s'achève généralement avec la génération d'un **prototype**.



6.1. Phase de conceptualisation / commencement (inception)

Objectifs:

- Vue d'ensemble sur le projet (besoins , caractéristiques clefs , principales contraintes).
- Une ébauche des diagrammes des Uses Cases (Dégrossi à hauteur de 10% - 25%).
- Un glossaire initial (termes, définitions).
- Une première version des processus métiers (avec analyse marché , prévisions sur les gains ,).
- Une première estimation des risques.
- Une ébauche de la planification du projet (phases, itérations).
- Un ou plusieurs Prototype(s) (éventuellement succin(s)).

Borne de fin de phase:

1. Crédibilité sur l'estimation des coûts , risques et du temps de développement ?
2. Pertinence du prototype (technicité , étendue, ...) ?
3. Compréhension claire des besoins ?
4. ...

6.2. Phase d'élaboration

Objectifs:

- Compléter et détailler le modèle des Uses Cases (ajout de diag. de séquences).
- Modélisation objet (Diagrammes = fruits de l'analyse).
- Description de l'architecture du logiciel (avec conception préliminaire et prototype).
- Une ré-estimation des risques et de la planification du projet.

Borne de fin de phase:

- Vision stable sur l'aspect fonctionnel du produit ?
- Architecture stable (prototype concluant) ?
- Planification suffisamment détaillée et précise ?
- Principaux risques écartés ?
- Accord des commanditaires sur les choix effectués (analyse, architecture) ?

6.3. Phase de construction

Objectifs:

- Concevoir , implémenter et tester le système par parties (itérations / incréments).
- Fabriquer un manuel "utilisateur" et une documentation technique .
- Intégration du logiciel (quasi complet) sur la bonne plate-forme (version bêta).

Borne de fin de phase:

- Le produit est-il assez mature et stable pour être déployé sur les postes des utilisateurs ?
- Commanditaires prêts à accepter le déploiement de la version bêta ?
- ...

6.4. Phase de transition

Objectifs:

- Feed-back sur la version bêta , correction de bugs.
- Déploiement du nouveau logiciel en parallèle de l'ancien (si possible).
- Optimisations, maintenance applicative.
- Assistance vis à vis des utilisateurs.
- ... Packaging , commercialisation éventuelle, ...

Borne de fin de phase:

- Recette
- Acceptation (et maîtrise) du produit par les utilisateurs ?
- Acheter la version finale dans les délais et avec des coûts acceptables ?

7. XP (Extreme Programming)



La méthode agile "XP (eXtreme Programming)" est née officiellement en octobre 1999 avec le livre *Extreme Programming Explained* de [Kent Beck](#).

7.1. Principales caractéristiques de XP

(selon wikipédia)

Dans le livre *Extreme Programming Explained*, la méthode est définie comme :

- une tentative de réconcilier l'humain avec la productivité ;
- un style de développement ;
- une discipline de développement d'applications informatiques.

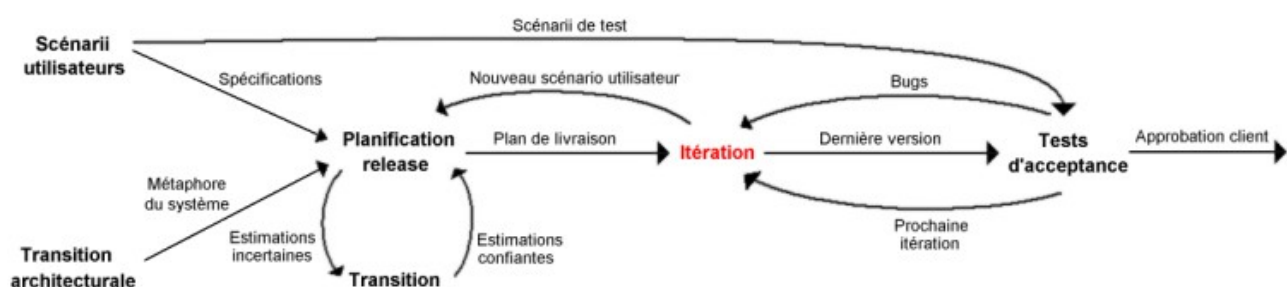
Son but principal est de réduire les coûts du changement. Dans les méthodes traditionnelles, les besoins sont définis et souvent fixés au départ du projet informatique ce qui accroît les coûts ultérieurs de modifications. **XP s'attache à rendre le projet plus flexible et ouvert au changement en introduisant des valeurs de base, des principes et des pratiques.**

Les principes de cette méthode ne sont pas nouveaux : ils existent dans l'industrie du logiciel depuis des dizaines d'années et dans les méthodes de management depuis encore plus longtemps.

L'originalité de la méthode est de les pousser à l'extrême :

- puisque la revue de code est une bonne pratique, elle sera faite en permanence (par un binôme) ;
- puisque les tests sont utiles, ils seront faits systématiquement avant chaque implantation ;
- puisque la conception est importante, elle sera faite tout au long du projet ([refactoring](#)) ;
- puisque la simplicité permet d'avancer plus vite, nous choisirons toujours la solution la plus simple ;
- puisque la compréhension est importante, nous définirons et ferons évoluer ensemble des métaphores ;
- puisque l'intégration des modifications est cruciale, nous l'effectuerons plusieurs fois par jour ;
- puisque les besoins évoluent vite, nous ferons des cycles de développement très rapides pour nous adapter au changement.

7.2. Cycle de développement XP



L'Extreme Programming repose sur des cycles rapides de développement (des itérations de quelques

semaines) dont les étapes sont les suivantes :

- une phase d'exploration détermine les scénarios clients ("user stories" / mini "use case") qui seront à prendre en charge pendant cette itération ;
- l'équipe transforme les scénarios en tâches à réaliser et en tests fonctionnels ;
- chaque développeur s'attribue des tâches et [les réalise avec un binôme](#) ;
- lorsque tous les tests fonctionnels passent, le produit est livré.

Le cycle se répète tant que le client peut fournir des scénarios à implémenter. Généralement le cycle de la première livraison se caractérise par sa durée et le volume important de fonctionnalités embarquées. Après la première mise en production, les itérations peuvent devenir plus courtes (une semaine par exemple).

7.3. Valeurs de XP

L'eXtreme Programming repose sur cinq valeurs fondamentales :

- **La communication** : c'est le moyen fondamental pour éviter les problèmes.
- **La simplicité** : la façon la plus simple d'arriver au résultat est la meilleure. Anticiper les extensions futures est une perte de temps.
- **Le feed-back** : le retour d'information est primordial pour le programmeur et le client. Les tests unitaires indiquent si le code fonctionne. Les tests fonctionnels donnent l'avancement du projet. Les livraisons fréquentes permettent de tester les fonctionnalités rapidement.
- **Le courage** : certains changements demandent beaucoup de courage. Il faut parfois changer l'architecture d'un projet, jeter du code pour en produire un meilleur ou essayer une nouvelle technique.
- **Le respect**

7.4. Pratiques (extrêmes?) de XP

- **Client sur site** : un représentant du client doit, si possible, être présent pendant toute la durée du projet. Il doit avoir les connaissances de l'utilisateur final et avoir une vision globale du résultat à obtenir. Il réalise son travail habituel tout en étant disponible pour répondre aux questions de l'équipe.
- **Jeu du Planning** ou [Planning poker](#) : le client crée des scénarios pour les fonctionnalités qu'il souhaite obtenir. L'équipe évalue le temps nécessaire pour les implémenter. Le client sélectionne ensuite les scénarios en fonction des priorités et du temps disponible. On joue avec les plannings (réaffectation glissante des aspects secondaires et introduction surprise d'un nouvel élément fonctionnel dans le cahier des charges).
- **Intégration continue** : lorsqu'une tâche est terminée, les modifications sont immédiatement intégrées dans le produit complet. On évite ainsi la surcharge de travail liée à l'intégration de tous les éléments avant la livraison. Les tests facilitent grandement cette intégration : quand tous les tests passent, l'intégration est terminée.
- **Petites livraisons** : les livraisons doivent être les plus fréquentes possible. L'intégration continue et les tests réduisent considérablement le coût de livraison.
- **Rythme soutenable** : l'équipe ne fait pas d'heures supplémentaires. Si le cas se présente, il faut revoir le planning. Un développeur fatigué travaille mal.
- **Tests de recette (ou tests fonctionnels)** : À partir des scénarios définis par le client, l'équipe crée des procédures de test qui permettent de vérifier l'avancement du développement. Lorsque tous les tests fonctionnels passent, l'itération est terminée. Ces tests sont souvent automatisés mais ce n'est pas toujours possible.
- **Tests unitaires** : avant d'implémenter une fonctionnalité, le développeur écrit un test qui vérifiera que son programme se comporte comme prévu. Ce test sera conservé jusqu'à la fin

du projet, tant que la fonctionnalité est requise. À chaque modification du code, on lance tous les tests écrits par tous les développeurs, et on sait immédiatement si quelque chose ne fonctionne plus.

- **Conception simple** : plus l'application est simple, plus il sera facile de la faire évoluer lors des prochaines itérations.
- **Utilisation de métaphores** : on utilise des métaphores et des analogies pour décrire le système et son fonctionnement. Le fonctionnel et le technique se comprennent beaucoup mieux lorsqu'ils sont d'accord sur les termes qu'ils emploient.
- **Refactoring (ou remaniement du code)** : amélioration régulière de la qualité du code sans en modifier le comportement. On retravaille le code pour repartir sur de meilleures bases tout en gardant les mêmes fonctionnalités.
- **Appropriation collective du code** : l'équipe est collectivement responsable de l'application. Chaque développeur peut faire des modifications dans toutes les portions du code, même celles qu'il n'a pas écrites. Les tests diront si quelque chose ne fonctionne plus.
- **Convention de nommage** : puisque tous les développeurs interviennent sur tout le code, il est indispensable d'établir et de respecter des normes de nommage pour les variables, méthodes, objets, classes, fichiers, etc.
- **Programmation en binôme** : la programmation se fait par deux. Le premier appelé *pilote* tient le clavier. C'est lui qui va travailler sur la portion de code à écrire. Le second appelé *partner* (ou *co-pilote*) est là pour l'aider en gardant un œil critique et en suggérant de nouvelles possibilités ou en décelant d'éventuels problèmes (correction des erreurs, avis différent, aide, ...). Les développeurs changent fréquemment de partenaire ce qui permet d'améliorer la connaissance collective de l'application et d'améliorer la communication au sein de l'équipe.

7.5. Autres pratiques extrêmes et variantes

Sur des petits projets, on travaille généralement en équipe réduite. Il faut alors savoir un peu tout faire. On se forge ainsi une bonne expérience où les aspects pragmatiques l'emportent sur les grands discours théoriques. L'apprenti dépasse le maître et devient virtuose. Commencent alors les pratiques extrêmes :

- on jongle avec les générateurs de code, le copier-coller et l'inspiration du moment.
- ...

Bien qu'assez extrêmes et pas toujours applicables, ces pratiques permettent quelquefois :

- de pouvoir aller très vite.
- d'être plus réactif
- ...

citation (de S.G. ou ...) : le tact dans l'audace c'est de savoir jusqu'où on peut aller trop loin.

Critiques et variantes/adaptations :

Quand on connaît le coût (assez élevé) d'une journée de développement (salaire du développeur + charges sociales ,) , on peut se demander si le principe du travail en binôme est réellement applicable.

Le principe du travail en binôme est généralement judicieux que si l'est utilisé au bon moment (et pas systématiquement / tout le temps).

Lorsqu'il y a du "turn over" dans une équipe , le fait de travailler à deux permet de bien intégrer un nouveau développeur au sein de l'équipe existante :

- les premiers jours , le nouvel arrivant observe (en tant que co-pilote) les manières de développer au niveau du projet (environnement de dev , convention de nommage , ...)
- et ensuite , c'est "tiens , prends le volant ". Le nouvel arrivant code de son mieux et le développeur rodé au projet vérifie si c'est bien fait et préconise des ajustements aussitôt.

On peut éventuellement s'autoriser des variantes par rapport à ce qui a été rédigé/formalisé au sein de la méthode XP. Le principe "0" serait : ne pas appliquer systématiquement XP tel quel mais seulement ce qui semble utile dans la méthode XP.

7.6. Zoom sur la planification adaptative

A partir d'un **découpage très fin** en "*User Stories*" (sorte de "mini *Use Cases*" auxquels on attache des priorités , des risques et des estimations de temps de développement) , on établit un planning temporaire avec des dates fixes (mais des contenus/livrables qui pourront partiellement changer).

====> Le planning initial est régulièrement remanié en fonction des :

- imprévus (Pb technique, ...) coté développement
- changements des besoins (nouvelles priorités) coté client

planification prévisionnelle initiale:

Livrables prévus sous les 20 premiers jours	UC1 , UC6 , UC4
Livrables prévus sous les 20 jours suivants (globalement 40 jours après le début)	UC3 , UC7 , UC5
Livrables prévus sous les 20 jours suivants (globalement 60 jours après le début)	UC2 , UC8

planification revue au bout de 20 jours ouvrés:

[*retard* sur UC4 (2 jours) + UC9 = *nouveau besoin prioritaire du client* (5j/h)]

Livrables réalisés et livrés sous les 20 premiers jours	UC1 , UC6
Livrables prévus sous les 20 jours suivants (globalement 40 jours après le début)	fin _UC4 , UC9 , UC3 , UC7
Livrables prévus sous les 20 jours suivants (globalement 60 jours après le début)	UC5 , UC2 , UC8

planification revue au bout de 40 jours ouvrés:

[retard sur UC7 (2 jours) + UC10 = nouveau besoin prioritaire du client (6j/h)]

Livrables réalisés et livrés sous les 20 premiers jours	UC1 , UC6
Livrables réalisés et livrés sous les 20 jours suivants (globalement 40 jours après le début)	UC4 , UC9 , UC3
Livrables réalisés et livrés sous les 20 jours suivants (globalement 60 jours après le début)	UC10, fin_UC7 , UC5 , UC2
<i>Eléments jamais livrés ou bien livrés plus tard si avenant et budget .</i>	UC8 (le moins prioritaire !!!)