Qualité et Test

Présentation du projet

Camille CONSTANT
Sophie ROUSSEAU

2015/2016

Plan

Plan

Plan

Objectifs visés

- Être capable de comprendre les besoins en test logiciel et qualité logicielle.
- Comprendre les besoins d'analyse statique et de couverture de codes sources logiciels.
- Acquérir une expérience dans la pratique d'outils d'analyse statique, de couverture de code et de test logiciel.

Compétences visées

- Être en mesure de cerner les besoins en qualité (règles à appliquer, rapport coût/bénéfice) et test logiciel (plan de test, cahier de test) pour un projet de développement.
- Savoir répondre aux exigences de qualité logicielle portant sur des aspects statiques du code (métriques, règles de codage, etc.).
- Être capable d'organiser et de planifier des tests logiciels au sein d'un projet.
- Être en mesure de sélectionner et d'appliquer les techniques de tests appropriées.
- Savoir répondre aux exigences de couverture de code lors de l'activité de test logiciel.

Plan

Modalités pédagogiques I

- Taille de chaque équipe de projet : 3

  (1 SE, 1 LD, 1 SE ou LD)

→ Remplir le document accessible sur le campus pour déterminer les équipes et le nom du projet pour chacune (CamelCase, lettres et chiffres)

Modalités pédagogiques II

- Répartition du volume horaire : 10 séances de 3h45 de projet parmi lesquelles :
  - au moins un audit qualité
  - des audits de test
- Évaluation
  - en séance (audits)
  - livrables
- Documents de projet disponibles sur la plateforme pédagogique.

Séquencement I

- 16 décembre :
  - Introduction du projet QT
  - Projet Monkeys : présentation du cahier des charges et des outils à utiliser
  - Création des projets sous Testlink
  - Création des comptes et projets sous Subversion
  - Écriture des CU et des tests de validation (Testlink)
- 4 janvier :
  - Écriture des CU et des tests de validation (Testlink)
  - Conception (à commencer avant)
  - Installation des outils (à faire avant, chez soi)

Séquencement II

- 6 janvier :
  - Conception
  - Configuration de Checkstyle et Findbugs
  - Développement Java (notamment les singes erratiques)
  - Présentation JUnit et Easymock
  - Tests du déplacement des singes erratiques (JUnit et Easymock)
  - Couverture de test (Emma)
- 11 et 13 janvier :
  - Développement Java (règles qualité suivies)
  - Tests (écriture, exécution et reporting)
  - Audits Qualité et Test

Séquencement III

- 18 janvier :
  - Présentation JMeter
  - Développement Java et tests
- 20 janvier :
  - Développement Java et tests
  - Audits Test et Qualité

Plan

Cahier des charges

Disponible sur le campus, à l'adresse du projet Qualité et Test.

→ vérifier à chaque début de séance s'il a évolué.

Plan

Objectifs

Objectif du projet : Réaliser, en respectant les exigences qualité demandées, et valider une première version du jeu de plateau multi-joueurs (incrément 1), nommé *Monkeys*.

Objectif du jeu : être le premier pirate à découvrir un trésor caché sur une île peuplée de singes *piratophages* et d'éventuels autres pirates.

L'île

Forme de damier contenant 2 types de cases :

- cases Mer,
- cases Terre.

Case Terre peut contenir :

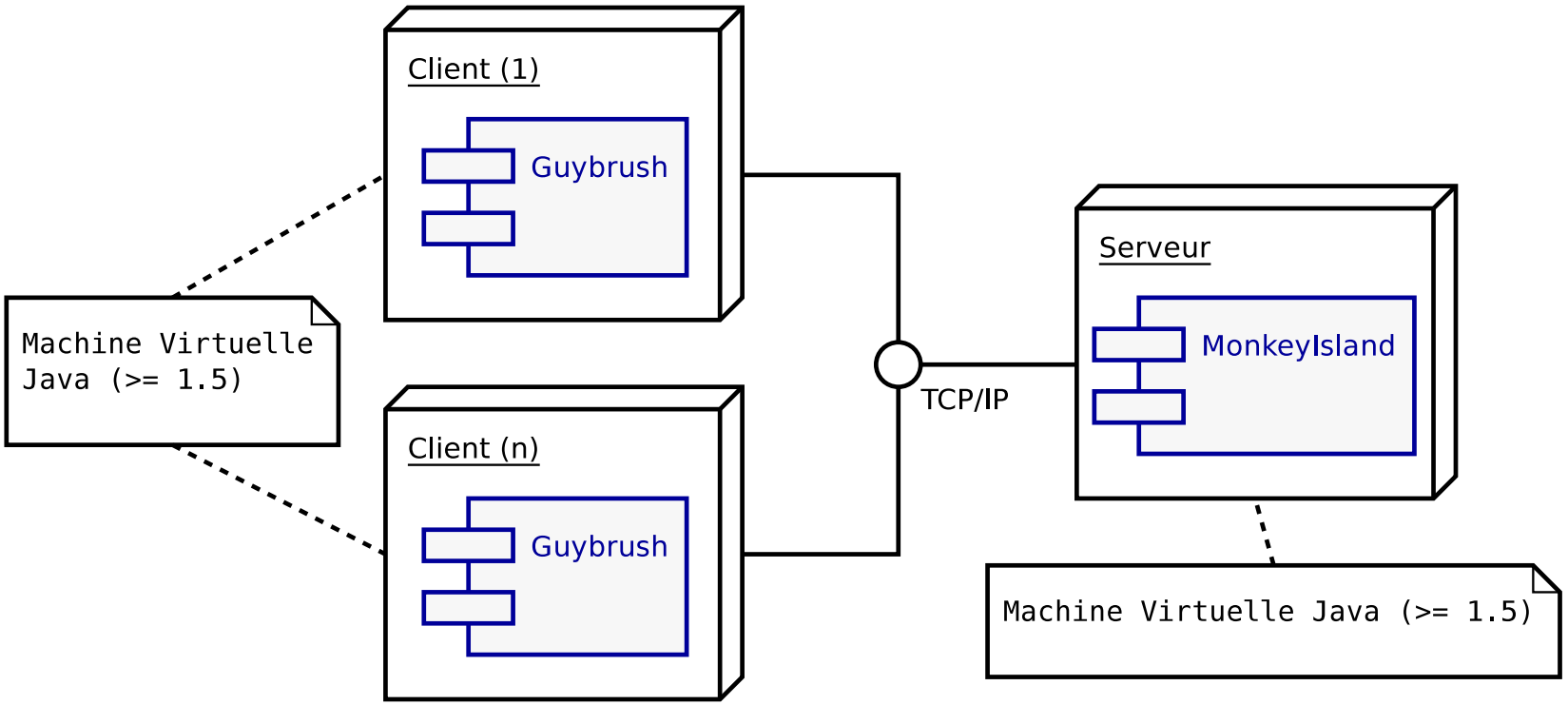
- un trésor caché,
- du rhum,
- un personnage.

Les personnages

3 types de personnages :

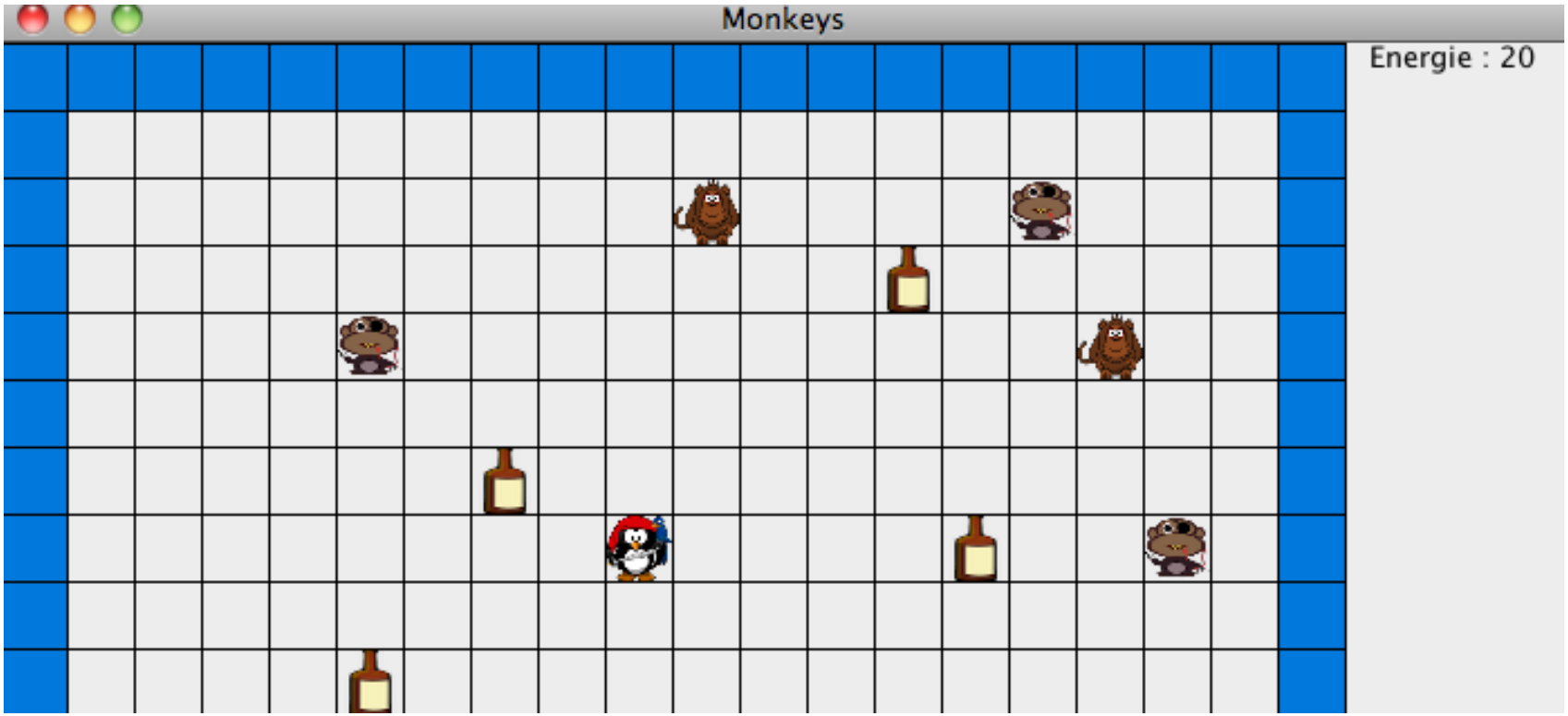
- les pirates :
  - 1 pirate par joueur,
  - niveau énergétique du pirate évolue en fonction de ses déplacements et de son alimentation ;
- les singes erratiques : piratophages se déplaçant aléatoirement ;
- les singes chasseurs : piratophages cherchant à atteindre le pirate le plus proche.

Déploiement



IHM

- Contrôles du pirate : clavier.
- Vue du jeu : carte de l'île, objets et personnages.



Plan

Incrément 1

Fonctionnalités :

- le déplacement des singes erratiques,
- la gestion des pirates (déplacement et décès),
- la gestion du trésor,
- la communication client-serveur.

Valider cet incrément avant de passer au suivant !

Incréments 2 et 3

Fonctionnalités incrément 2 :

- la gestion du fichier de configuration,
- la gestion de l'énergie de chaque pirate,
- la gestion des bouteilles de rhum,
- la gestion des parties.

Valider cet incrément avant de passer au suivant !

Fonctionnalités incrément 3 :

- la gestion des singes chasseurs.
- ...

Plan

Livrables

- Plan de test et cahier de tests (sous Testlink),
- Conception objet du projet (diagrammes de classes),
- Code Java de qualité,
- Code de test de qualité,
- Fichiers de configuration de Checkstyle,
- Plan de test et de travail de JMeter,
- Rapport de synthèse format PDF (fait en Latex ?).

Plan

Outils Qualité et Test

- Subversion
- Testlink
- Checkstyle
- Findbugs
- JUnit
- Easymock
- JMeter
- Emma
- (Eclipse)

Subversion

Gestionnaire de version disponible à l'adresse :
`http ://qualite-test.sea.eseo.fr/svn-nom1_nom2_nom3`

Les noms sont indiqués par ordre alphabétique.

Subversion - Création des comptes par équipe

→ Pour chaque équipe, effectuer les opérations suivantes :

- afin de générer le fichier contenant logins et mots de passe cryptés, taper dans un même terminal les commandes :
  - `htpasswd -c nom1_nom2_nom3.txt login1`
  - `htpasswd nom1_nom2_nom3.txt login2`
  - `htpasswd nom1_nom2_nom3.txt login3`
- envoyer par mail le fichier `nom1_nom2_nom3.txt` à `camille.constant@eseo.fr`

Les noms doivent être indiqués par ordre alphabétique.

Testlink

Outil de gestion et de suivi d'un projet de test.

Application web (basée sur PHP) disponible à l'adresse :
`http ://qualite-test.sea.eseo.fr/testlink`

Permet d'effectuer :

- une gestion de l'équipe de test,
- un cahier d'exigences,
- des campagnes de test avec plans de test,
- un cahier de test,
- un retour sur l'exécution des tests (métriques).

Une documentation est disponible sur le campus.

Checkstyle I

Outil d'analyse statique du code source Java permettant de s'assurer du respect de la qualité de code demandée (vérifie la forme du code, pas si celui-ci est correct).

Objectif de cette qualité : uniformiser le code de (différents) développeurs pour une meilleure lisibilité, une meilleure intégration et donc une meilleure maintenabilité.

Checkstyle II

Moyens mis en œuvre pour obtenir cette qualité (à configurer dans Checkstyle **avant de commencer à coder**) :

- présence et respect des règles de la Javadoc,
- respect des conventions de nommage Java,
- respect de la structure d'un code Java (ordre de déclaration, espaces, accolades, nombre de paramètres, etc.),
- respect des bonnes pratiques de codage Java (nombre de `return`, affectations conditionnelles, etc.),
- duplication de code,
- métriques (complexité du code).

Ces critères de qualité sont à définir selon le projet : une sur-qualité nuit à la productivité, un manque de qualité nuit à la maintenabilité d'un code.

Findbugs

Outil d'analyse statique du bytecode Java permettant la détection de *bugs* éventuels (via des *patterns* reconnus comme étant des *bugs*).

Findbugs est à configurer selon différents types de *bugs* :

- mauvaises pratiques de codage,
- vulnérabilité au code malicieux,
- sécurité,
- internationalisation,
- performance,
- etc.

Attention à analyser les bugs indiqués : de faux-positifs peuvent être signalés par l'outil.

JUnit : principe

Bibliothèque de test unitaire pour le langage Java, créée par Kent BECK et Erich GAMMA.

Deux types de fichiers de tests :

- **TestCase** : classes contenant des méthodes de test.
- **TestSuite** : pour exécuter des *TestCase* déjà définis.

Lancement des tests :

- **TestRunner** : classes de lancement des tests.

Utilisation :

- Ajout du *jar* de JUnit (junit.jar ou junit4.jar) dans le *classpath*.

JUnit : TestCase

Création d'un test :

```
public class <nom de la classe à tester[1]>Test
extends junit.framework.TestCase {}
```

Annotation des méthodes :

`@Test` : méthodes de test.
`@Before` : méthodes exécutées avant les méthodes de test.
`@After` : méthodes exécutées après les méthodes de test.

Méthodes de test :

```
public void test<nom de la méthode à tester>()
{
```

Vérification du comportement par des `assert*()`
(`junit.framework.Assert` ou `org.junit.Assert`).
(assertion non vérifiée → défaillance).

```
}
```

[1] par convention

JUnit : TestCase

La classe Assert :

- `assertEquals` :
  tester l'égalité de deux objets ou de deux types primitifs
  (*boolean, byte, char, double, float, int, long, short*)
  (delta possible sur les *double* et les *float*).
- `assertFalse`, `asserTrue` :
  tester une condition booléenne.
- `assertNotNull`, `assertNull` :
  tester si une référence est nulle ou non.
- `assertNotSame`, `assertSame` :
  tester si deux objets se réfèrent ou non au même objet.
- `fail` :
  faire échouer un test.

→ Message d'erreur personnalisable en premier argument de chacune des méthodes.

JUnit : TestRunner

Lancement d'un test en mode textuel :

```
java -classpath . :<path/to/>junit4.jar
     junit.textui.TestRunner <classe.de.test>
```

Lancement d'un test en mode graphique (JUnit < 4.x) :

```
java -classpath . :<path/to/>junit.jar
     junit.swingui.TestRunner <classe.de.test>
```

```
java -classpath . :<path/to/>junit.jar
     junit.awtui.TestRunner <classe.de.test>
```

JUnit : TestSuite

La classe *TestSuite* définit une méthode publique « *suite* » qui renvoie un objet de type *Test* pouvant contenir des *TestCase* ou d'autres *TestSuite*.

Principe :

```
public class <nom de classe>
{
     public static Test suite()
     {
          TestSuite suite = new TestSuite("<titre>");
          suite.addTest(new TestSuite(<classe de test>.class));
          suite.addTest(new TestSuite(<etc.>.class));
          return suite;
     }
}
```

Lancement (exemple) :

```
java -classpath . :<path/to/>junit4.jar
     junit.textui.TestRunner <classe.de.suite.de.tests>
```

JUnit : exemple sur Monix

```
package monix.modele.vente;

import junit.framework.TestCase;
import org.junit.*;
import monix.modele.stock.*;

public class AchatTest extends TestCase
{
	private Produit produit;

	@Before
	public void setUp() throws Exception
	{
		this.produit  = new Produit("A", "Libelle A", new Double(10.50), 1);
	}

	@Test
	public void testIncrementeQuantite()
	{
		Integer origine = 1, ajout = 2;
		Achat achat = new Achat(this.produit, origine);

		achat.incrementeQuantite(ajout);
		assertEquals((int)(origine + ajout), (int)achat.donneQuantite());
	}
}
```

JUnit : exemple sur Monix

```
package monix.modele.vente;

import junit.framework.TestCase;
import org.junit.*;
import monix.modele.stock.*;

public class AchatTest extends TestCase
{
	private Produit produit;

	@Before
	public void setUp() throws Exception
	{
		this.produit  = new Produit("A", "Libelle A", new Double(10.50), 1);
	}

	@After
	public void tearDown() throws Exception
	{
		/* ne rien faire */
	}

	@Test
	public void testPreconditions()
	{
		assertNotNull(this.produit);
	}
```

JUnit : exemple sur Monix

```
	@Test
	public void testIncrementeQuantite()
	{
		Integer origine = 1, ajout = 2;
		Achat achat = new Achat(this.produit, origine);

		achat.incrementeQuantite(ajout);
		assertEquals((int)(origine + ajout), (int)achat.donneQuantite());
	}

	@Test
	public void testDecrementeQuantite()
	{
		Integer origine = 3, retrait = 2;

		Achat achat = new Achat(this.produit, origine);

		achat.decrementeQuantite(retrait);

		assertEquals("Decremente quantite produit", (int)(origine - retrait), (int) ac
	}
}
```

JUnit : exemple sur Monix

```java
package monix.modele.vente;

import monix.modele.stock.*;

import junit.framework.TestCase;
import org.junit.Before;
import org.junit.After;
import org.junit.Test;

public class VenteTest extends TestCase
{
	/**
	 * Stock utilise pour les tests.
	 */
	private Stock stock = null;

	@Before
	public void setUp() throws Exception
	{
		this.stock = new StockBouchon();
	}

	@After
	public void tearDown() throws Exception
	{
		/* rien faire */
	}
```

JUnit : exemple sur Monix

```
@Test
public void testPreconditions()
{
	assertNotNull(this.stock);
}

@Test
public void testAjouteAchatProduit_exception_AchatImpossibleException()
{
	String id = "999999"; // id invalide, voir : modele.test.StockBouchon
	Integer quantite = 3;

	Vente vente = new Vente(this.stock);

	try {
		vente.ajouteAchatProduit(id, quantite);
		fail("Vente achat exception AchatImpossibleException");

	} catch (AchatImpossibleException e) {
		// passage attendu dans ce bloc.
	}
}
```

JUnit : exemple sur Monix

```
        @Test
        public void testAjouteAchatProduit()
        {
              String message = "Vente achat produit : ";
              String id = "11A"; // id valide, voir : modele.test.StockBouchon
              Integer quantite = 3;
              Vente vente = new Vente(this.stock);

              try {
                    vente.ajouteAchatProduit(id, quantite);

                    assertNotNull(
                          message + "achats null",
                          vente.donneAchats()
                    );
                    assertNotNull(
                          message + "produit non achete",
                          vente.donneAchats().get(id)
                    );
                    assertEquals(
                          message + "produit non conforme",
                          this.stock.donneProduit(id),
                          vente.donneAchats().get(id).donneProduit()
                    );
                    assertEquals(
                          message + "quantite non conforme",
                          (int)(quantite),
                          (int) (vente.donneAchats().get(id).donneQuantite())
                    );
              } catch (AchatImpossibleException e) {
                    fail(message + "catch AchatImpossibleException");
              }
        }
  }
```

JUnit : exemple sur Monix

```
package monix.modele.vente;

import junit.framework.Test;
import junit.framework.TestSuite;

public class VenteTestSuite
{
	public static Test suite()
	{
		  TestSuite suite = new TestSuite("Suite de tests pour le package vente.");

		suite.addTest(new TestSuite(AchatTest.class));
		suite.addTest(new TestSuite(VenteTest.class));

		/* alternative */
		/*
		Class<?>[] classesTest = {
			AchatTest.class,
			VenteTest.class,
		};

		TestSuite suite = new TestSuite(classesTest);

		*/

		return suite;
	}

}
```

JUnit : les limites

- les classes abstraites : tests possibles sur les classes filles (soit en créer une, soit tester les classes filles existantes) ;
- les méthodes et attributs privés : API de réflexion du langage (`Java.reflect`) (*cf. HelloJUnit*) ;
- les classes ayant des dépendances : doublures de test ou bouchons (`mocks`).

Doublure : principe

Plusieurs types de doublures (pour simuler le comportement d'un autre objet) :

- *dummy* (fantôme, bouffon) : objet « vide » sans fonctionnalité implantée.
- *stub* (bouchon) : classe qui renvoie une valeur en dur pour une méthode.
- *fake* (substitut, simulateur) : classe partiellement implantée
  (qui renvoie, par exemple, toujours les mêmes réponses selon les paramètres fournis).
- *spy* (espion) : classe qui vérifie son utilisation après exécution.
- *mock* (simulacre) : classe qui agit comme un *stub* et un *spy*.

Un objet de type *mock* permet :

1. de simuler le comportement d'un autre objet ;
2. de vérifier les invocations qui sont faites de cet objet
   (invocations des méthodes, paramètres fournis, ordre des invocations).

Mock : principe

Deux types de *mock*

- statique : classes Java écrites par le développeur ;
- dynamique : mis en œuvre via un *framework*.

→ avantage d'un *mock* dynamique : aucune classe implicite n'est écrite.

Solutions de *frameworks* de *mocking* :

- proxy : fourni à l'objet qui l'utilise via un constructeur ou un *setter* (nécessite un mécanisme d'injection de dépendance dans la classe à tester) (EasyMock, etc.).
- instrumentation : *classloader* spécifique qui charge dynamiquement une classe à la place d'une autre (utilisation de la classe java.lang.Instrument de Java 1.5) (JMockit)
- orienté aspect : invocation d'une méthode d'un *mock* à la place de celle d'une implémentation (sans utilisation d'interface ni de mécanisme d'injection de dépendances) (JEasyTest, AMock, etc.).

Mock : principe

Les *frameworks* de *mocking* permettent généralement de :

- créer dynamiquement des objets *mocks* (à partir d'interfaces) ;
- spécifier les méthodes invoquées (avec paramètres et valeur de retour) ;
- spécifier l'ordre des invocations de ces méthodes ;
- spécifier le nombre d'invocations de ces méthodes ;
- simuler des cas d'erreur en levant des exceptions ;
- valider des appels de méthodes ;
- valider l'ordre de ces appels.

Exemples de *frameworks* de *mocking* :

- EasyMock, JMockIt, Mockito, JMock, MockRunner, etc.

Mock : principe

Utilisation des *mocks* :

- Renvoyer des résultats déterminés (pour des tests unitaires automatisés) ;
- Obtenir un état difficilement reproductible (erreur d'accès réseau) ;
- Éviter d'invoquer des ressources longues à répondre (base de données) ;
- Invoquer un composant qui n'existe pas encore ;
- Etc.

Mise en œuvre des *mocks* :

1. Définir le comportement du mock (méthodes, paramètres, exceptions, etc.).
2. Exécuter le test en invoquant la méthode à tester.
3. Vérifier les résultats du test.
4. Vérifier les invocations du ou des *mocks* (nombre et ordre des invocations).

EasyMock : principe

Bibliothèque de *mocking* pour le langage Java.

Création de *mocks* :

- à partir d'interfaces (`org.easymock`).
- à partir de classes (`org.easymock.classextension`).
- classe **EasyMock** : méthodes statiques.

Utilisation :

- Ajout du *jar* de EasyMock (easymock.jar) dans le *classpath*.

EasyMock : principe

Création d'un *mock* :

```
<interface> mock =
    EasyMock.createMock(<interface>.class)
```

Comportement d'un *mock* :

`mock.<méthode>(<paramètres>)`

`EasyMock.expect(mock.<méthode>(<paramètres>))`
`.andReturn(<retour associé>)` (1 invocation)
`.andThrow(<exception associée>)` (1 invocation)
`.andStubReturn()` ou `.andStubThrow()` (0-n ? invocations)

Initialisation d'un *mock* :

`EasyMock.replay(mock)`

Vérification de l'invocation des méthodes :

`EasyMock.verify(mock)`
avec : `EasyMock.createMock()` (vérifie l'invocation des méthodes).
avec : `EasyMock.createStricMock()` (+ ordre des invocations).

EasyMock : exemple sur Monix (cf. Annexe Monix)

```
package monix.modele.vente;

import monix.modele.stock.*;

import junit.framework.TestCase;
import org.junit.*;
import org.easymock.EasyMock;

/**
 * Classe de test unitaire de la classe Vente.
 *
 * <p>Utilisation d'un mock (simulacre) de stock (construit avec EasyMock).</p>
 *
 * @see monix.modele.vente.Vente
 */
public class VenteTest extends TestCase
{
      private Stock stockMock;

      @Before
      public void setUp() throws Exception {
            this.stockMock = EasyMock.createMock(Stock.class);
      }

      ...
```

EasyMock : exemple sur Monix (cf. Annexe Monix)

```
@Test
public void testAjouteAchatProduit_quantite() {
      Integer quantite = 2;
      Vente vente = new Vente(this.stockMock);

      EasyMock.expect(
            this.stockMock.donneProduit("A")
      ).andReturn(
            new Produit("A", "l.A", new Double(1), 2)
      );

      EasyMock.replay(this.stockMock);

      try {
            vente.ajouteAchatProduit("A", quantite);
            assertEquals(
                  (int)(quantite),
                  (int)(vente.donneAchats().get("A").donneQuantite())
            );
      } catch (AchatImpossibleException e) {
            fail("Vente.AchatProduit catch AchatImpossibleException");
      }
}

...
```

EasyMock : exemple sur Monix (cf. Annexe Monix)

```
@Test
public void testAjouteAchatProduit_increment() {
      Integer quantite = 2, increment = 3;
      Vente vente = new Vente(this.stockMock);

      EasyMock.expect(
            this.stockMock.donneProduit("A")
      ).andReturn(
            new Produit("A", "l.A", new Double(1), 2)
      );
      EasyMock.expectLastCall().times(2);

      EasyMock.replay(this.stockMock);

      try {
            vente.ajouteAchatProduit("A", quantite);
            vente.ajouteAchatProduit("A", increment);
            assertEquals(
                  (int)(quantite + increment),
                  (int)(vente.donneAchats().get("A").donneQuantite())
            );
      } catch (AchatImpossibleException e) {
            fail("Vente.AchatProduit catch AchatImpossibleException");
      }
}
```

JMeter : présentation

Apache JMeter (*The Apache Software Foundation*).

A l'origine :

- test fonctionnel nominal et
- test de robustesse (performance : montée en charge, stress)

des applications serveur.

Aujourd'hui :

- étendu à tout type d'application et
- utilisable, par méthodologie, pour d'autres types de test.

JMeter : qualités (non exhaustives)

Mise en œuvre des tests :

- de nombreux protocoles supportés (TCP/IP, HTTP(S), IMAP, LDAP, etc.) ;
- simulation de plusieurs utilisateurs ;
- lancement de tests à partir de plusieurs postes.

Scénarios de tests :

- utilisation de JT externalisés ;
- enregistrement (et modification) de scénarios de requêtes (web) ;
- scénarios avec structures de contrôle (adaptables aux réponses serveur).

Résultats des tests :

- mesures et statistiques des temps de réponse ;
- détection d'erreurs (*assert*) ;
- plusieurs moyens de visualisation des résultats de test.

Extensibilité :

- de nombreux greffons (add-on, plugins) sont disponibles (gestion de protocoles, visualisation de résultats, etc.).

JMeter : qualités (non exhaustives)

Mise en œuvre des tests :

- de nombreux protocoles supportés (TCP/IP, HTTP(S), IMAP, LDAP, etc.) ;
- simulation de plusieurs utilisateurs ;
- lancement de tests à partir de plusieurs postes.

Scénarios de tests :

- utilisation de JT externalisés ;
- enregistrement (et modification) de scénarios de requêtes (web) ;
- scénarios avec structures de contrôle (adaptables aux réponses serveur).

Résultats des tests :

- mesures et statistiques des temps de réponse ;
- détection d'erreurs (*assert*) ;
- plusieurs moyens de visualisation des résultats de test.

Extensibilité :

- de nombreux greffons (add-on, plugins) sont disponibles (gestion de protocoles, visualisation de résultats, etc.).

JMeter : qualités (non exhaustives)

Mise en œuvre des tests :

- de nombreux protocoles supportés (TCP/IP, HTTP(S), IMAP, LDAP, etc.) ;
- simulation de plusieurs utilisateurs ;
- lancement de tests à partir de plusieurs postes.

Scénarios de tests :

- utilisation de JT externalisés ;
- enregistrement (et modification) de scénarios de requêtes (web) ;
- scénarios avec structures de contrôle (adaptables aux réponses serveur).

Résultats des tests :

- mesures et statistiques des temps de réponse ;
- détection d'erreurs (*assert*) ;
- plusieurs moyens de visualisation des résultats de test.

Extensibilité :

- de nombreux greffons (add-on, plugins) sont disponibles (gestion de protocoles, visualisation de résultats, etc.).

JMeter : qualités (non exhaustives)

Mise en œuvre des tests :

- de nombreux protocoles supportés (TCP/IP, HTTP(S), IMAP, LDAP, etc.) ;
- simulation de plusieurs utilisateurs ;
- lancement de tests à partir de plusieurs postes.

Scénarios de tests :

- utilisation de JT externalisés ;
- enregistrement (et modification) de scénarios de requêtes (web) ;
- scénarios avec structures de contrôle (adaptables aux réponses serveur).

Résultats des tests :

- mesures et statistiques des temps de réponse ;
- détection d'erreurs (*assert*) ;
- plusieurs moyens de visualisation des résultats de test.

Extensibilité :

- de nombreux greffons (add-on, plugins) sont disponibles (gestion de protocoles, visualisation de résultats, etc.).

JMeter : organisation

Deux emplacements :

- **Plan de test** : pour positionner les tests à exécuter.
- **Plan de travail** : pour stocker les tests non utilisés.

Remarques :

Enregistrement et ouverture des plans (test et travail) séparément :

- menu `Fichier` → `Ouvrir...` (idem `Enregistrer sous...`)
- clique droit → `Ouvrir...` (idem `Enregistrer sous...`)
- format : `jmx`

Ajout de composants :

- menu `Éditer` → `Ajouter`
- clique droit → `Ajouter`

JMeter : composants

Groupe d'unités

(Plan de test : `Moteurs d'utilisateurs` → `Groupe d'unités`)

- Paramètres de l'exécution des tests :
  - → comportement après une erreur (arrêt du test, de l'unité, etc.) ;
  - → nombre d'utilisateurs (qui se connectent) ;
  - → montée en charge (intervalle entre les connexions) ;
  - → nombre d'itérations (pour chaque utilisateur) ;
  - → programmateur de démarrage (date et heure des tests).
- Ajout de contrôleurs logiques et d'échantillons.

JMeter : composants

Contrôleurs logiques

- Élaboration d'un test (organisation et paramètres des requêtes) :
  - → simple (libre, organisation logique) ;
  - → aléatoire (envoi de requêtes dans un ordre aléatoire) ;
  - → structures de contrôle (boucles, conditions) ;
  - → exécution unique (requête unique - indép. nb. itérations) ;
  - → transaction (ensemble de requêtes) ;
  - → durée d'exécution (des requêtes) ;
  - → débit (nombre d'exécutions - total ou pourcentage) ;
  - → module (substitution d'un contrôleur lors de l'exécution) ;
  - → inclusion (utilisation d'un jmx externe) ;
  - → enregistreur (capture de requêtes HTTP - cf. Proxy HTTP).

JMeter : composants

Échantillons

- Élaboration des requêtes :
  - → action test (arrêt ou suspension du test) ;
  - → construction de requêtes (HTTP, FTP, TCP, etc.) ;
  - → utilisation de scripts (BeanShell, BSF, JSR223) ;
  - → utilisation de journaux d'accès (log serveur) ;
  - → maîtrise des classes clientes (Java, client TCP) ;
  - → test unitaire (JUnit).

JMeter : composants

Configuration

- Paramètres généraux des requêtes et variables :
  - → connexions (HTTP (autorisation, cookies...), FTP, TCP, etc.) ;
  - → lecture de données dans un fichier (CSV) ;
  - → variables (aléatoires, pré-définies) ;
  - → etc. (identification, compteurs, requêtes java...).

Compteurs de temps

- Écoulement de temps avant (ou après) une requête :
  - → déterministe (fixe, débit constant) ;
  - → aléatoire (gaussien, uniforme) ;
  - → synchronisation (entre différentes unités de test) ;
  - → compteurs de scripts (BeanShell, BSF, JSR223).

JMeter : composants

Configuration

- Paramètres généraux des requêtes et variables :
  - → connexions (HTTP (autorisation, cookies...), FTP, TCP, etc.) ;
  - → lecture de données dans un fichier (CSV) ;
  - → variables (aléatoires, pré-définies) ;
  - → etc. (identification, compteurs, requêtes java...).

Compteurs de temps

- Écoulement de temps avant (ou après) une requête :
  - → déterministe (fixe, débit constant) ;
  - → aléatoire (gaussien, uniforme) ;
  - → synchronisation (entre différentes unités de test) ;
  - → compteurs de scripts (BeanShell, BSF, JSR223).

JMeter : composants

Pré-processeurs

- Modification des requêtes avant exécution :
  - → données HTML (liens, formulaires, paramètres) ;
  - → données HTTP (URL, paramètres) ;
  - → paramètres utilisateur (individuels) ;
  - → utilisation de scripts (BeanShell, BSF, JSR223).

Post-processeurs

- Manipulation des réponses serveur :
  - → extraction de données (expression régulière, XPath) ;
  - → arrêt d'une unité de test (si un test ne passe pas) ;
  - → accès à une base de données (JDBC) ;
  - → utilisation de scripts (BeanShell, BSF, JSR223).

JMeter : composants

Pré-processeurs

- Modification des requêtes avant exécution :
  - → données HTML (liens, formulaires, paramètres) ;
  - → données HTTP (URL, paramètres) ;
  - → paramètres utilisateur (individuels) ;
  - → utilisation de scripts (BeanShell, BSF, JSR223).

Post-processeurs

- Manipulation des réponses serveur :
  - → extraction de données (expression régulière, XPath) ;
  - → arrêt d'une unité de test (si un test ne passe pas) ;
  - → accès à une base de données (JDBC) ;
  - → utilisation de scripts (BeanShell, BSF, JSR223).

JMeter : composants

Assertions

- Vérification sur les réponses aux requêtes :
  - → chaînes de caractères (comparaison, données, patrons) ;
  - → taille, temps de réponse (limite) ;
  - → HTML (syntaxe), XML (schéma, XPath) ;
  - → Hash (MD5) ;
  - → utilisation de scripts (BeanShell, BSF, JSR223).

Récepteurs

- Exploitation des résultats des tests :
  - → visualisation (arbres, tableaux, graphiques) ;
  - → envoi de mail, enregistrement (données, réponses) ;
  - → utilisation de scripts (BeanShell, BSF, JSR223).

JMeter : composants

Assertions

- Vérification sur les réponses aux requêtes :
  - → chaînes de caractères (comparaison, données, patrons) ;
  - → taille, temps de réponse (limite) ;
  - → HTML (syntaxe), XML (schéma, XPath) ;
  - → Hash (MD5) ;
  - → utilisation de scripts (BeanShell, BSF, JSR223).

Récepteurs

- Exploitation des résultats des tests :
  - → visualisation (arbres, tableaux, graphiques) ;
  - → envoi de mail, enregistrement (données, réponses) ;
  - → utilisation de scripts (BeanShell, BSF, JSR223).

JMeter : composants

Éléments hors test (Plan de travail)

- Affichage des propriétés (système et JMeter).
- Serveur HTTP Proxy (capturer une session HTTP) :

  → Mise en place du serveur proxy :

  `Plan de test` : `Ajouter` → `Groupe d'unités`;
  `Groupe d'unité` : `Ajouter` → `Contrôleur enregistreur`;
  `Plan de travail` : `Ajouter` → `Serveur HTTP Proxy`;
  `Serveur HTTP Proxy` : `Lancer`.

  → Utilisation du serveur proxy :

  Navigateur Web : configuration du proxy (`host`, `port`);
  JMeter : requêtes capturées dans le `Contrôleur enregistreur`.
- Serveur HTTP mirroir (renvoi des données envoyées).

JMeter : exemple sur Monix – `morix` (cf. Annexe Monix)

(1) Création des variables pour le protocole
(`Ajouter` → `Configurations` → `Variables pré-définies`)
`Nom` : INSCRIPTION_CLIENT, `Valeur` : /I
`Nom` : MESSAGE_INSCRIPTION, `Valeur` : Inscription OK

(2) Création d'une unité de test
(`Ajouter` → `Moteur d'utilisateurs` → `Groupe d'unités`)

(3) Création d'un contrôleur simple (dans l'unité de test)
(`Ajouter` → `Contrôleurs` → `Contrôleur simple`)

(4) Renseignement des paramètres TCP (du contrôleur)
(`Ajouter` → `Configurations` → `Paramètres TCP par défaut`)
`Nom ou IP du serveur` : 127.0.0.1
`Numéro de port` : 13579
`Expiration (millisecondes)` : 500

(5) Ajouter un compteur de temps fixe (au contrôleur)
(`Ajouter` → `Compteurs de temps` → `Compteur de temps fixe`)
`Délai d'attente (en millisecondes)` : 500

JMeter : exemple sur Monix – `morix` (cf. Annexe Monix)

(6) Ajouter une requête TCP (au contrôleur)
(`Ajouter` → `Échantillons` → `Requête TCP`)
`Ré-utiliser la connexion` : ✓
`Texte à envoyer` : ${INSCRIPTION_CLIENT} (suivi d'un retour à la ligne)

(7) Ajouter une assertion (à la requête)
(`Ajouter` → `Assertions` → `Assertion réponse`)
`Appliquer sur` : ✓ `L'échantillon`
`Section de réponse à tester` : ✓ `Texte de réponse`
`Motif à tester` : ${MESSAGE_INSCRIPTION}

(8) Ajouter un récepteur (au contrôleur)
(`Ajouter` → `Récepteurs` → `Arbre de résultats`)

(9) Sauver le plan de test
(`Fichier` → `Enregistrer le plan de test`)

(10) Lancer le plan de test (`morix` étant lancé)
(`Lancer` → `Lancer`)

(à suivre) Développement d'une classe TCPClientMorix (implements TCPClient)…

JMeter : exemple sur Monix – `morix` (cf. Annexe Monix)

(6) Ajouter une requête TCP (au contrôleur)
(`Ajouter` → `Échantillons` → `Requête TCP`)
`Ré-utiliser la connexion` : ✓
`Texte à envoyer` : `${`INSCRIPTION_CLIENT`}` (suivi d'un retour à la ligne)

(7) Ajouter une assertion (à la requête)
(`Ajouter` → `Assertions` → `Assertion réponse`)
`Appliquer sur` : ✓ `L'échantillon`
`Section de réponse à tester` : ✓ `Texte de réponse`
`Motif à tester` : `${`MESSAGE_INSCRIPTION`}`

(8) Ajouter un récepteur (au contrôleur)
(`Ajouter` → `Récepteurs` → `Arbre de résultats`)

(9) Sauver le plan de test
(`Fichier` → `Enregistrer le plan de test`)

(10) Lancer le plan de test (`morix` étant lancé)
(`Lancer` → `Lancer`)

(à suivre) Développement d'une classe TCPClientMorix (implements TCPClient)…