DevOps

Test (automatisé) d'un logiciel – JUnit

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2026

Agenda

Introduction aux tests

JUnit

Agenda

Introduction aux tests

JUnit

Des questions

- Est-ce que mon logiciel fonctionne?
- Est-ce que mon logiciel correspond aux spécifications?
- Je viens de faire une modification, est-ce que ça fonctionne toujours?
- Comment puis-je devenir un meilleur programmeur?

Des questions

- Est-ce que mon logiciel fonctionne?
- Est-ce que mon logiciel correspond aux spécifications?
- Je viens de faire une modification, est-ce que ça fonctionne toujours?
- Comment puis-je devenir un meilleur programmeur?

La réponse: En testant

Exemples d'erreurs logicielles

Mars Climate Orbiter (NASA, 1998)

- Sonde spatiale destinée à étudier la météorologie sur Mars
- Coût: Plus de 50 millions de dollars
- Perdue lors de l'insertion en orbite
  - Erreur logicielle
  - Le module de calcul de trajectoire utilisait le système métrique
  - Un autre module lui fournissait des données dans le système de mesure anglo-saxon

Exemples d'erreurs logicielles

Explosion d'Ariane 5 (vol 501)

- Premier vol d'Ariane 5
- Coût: Transportait 370 millions de dollars de charge utile
- Autodestruction de la fusée pour cause de mauvaise trajectoire
  - Erreur logicielle
  - Protection inadéquate contre le dépassement d'un entier
  - Réutilisation d'un module du système de guidage d'Ariane 4
  - Problème: Accélération beaucoup plus importante d'Ariane 5 au décollage

Les tests

Les test sont à la base de toute activité d'ingénierie:

- Médicaments
- Conception d'un avion
- ...

Pourquoi?

- Systèmes complexes
- Incapacité de prévoir exactement le comportement de ce que nous avons créé
- Nous faisons des erreurs

Tester un logiciel est complexe

Exemple de test

- Un ascenseur: Tester avec une charge de 100kg
- Pas besoin de tester avec une charge de 70kg

Test d'un algorithme de tri

- Test avec 256 éléments
- Est ce qu'il fonctionne avec 3 éléments? avec 1024? si je change les 256 éléments?

Utilité des tests

Program testing can be used to show the presence of bugs, but never to show their absence! [E. Dijkstra]

- Il a raison (tous les logiciels ont des bugs)
- Mais ce n'est pas une raison pour ne pas tester
  - Si on peut supprimer une majorité de bugs, c'est déjà pas mal

Dans un monde idéal . . .

Dans un monde idéal, on prouverait qu'un logiciel est correct

- En utilisant des méthodes formelles (pas en raisonnant sur papier)
- Extrêmement difficile pour des petits programmes (quelques centaines de lignes de code)
- Infaisable dans la plupart des cas

Utilisation d'un modèle abstrait pour faire du *model checking*:

- Peut marcher pour des propriétés simples
  - Il est impossible d'atteindre cet état particulier (deadlock)
- Mais plus difficile pour des propriétés complexes
  - Ce système de fichier fonctionne correctement

En dernier recours

À défaut d'autres solutions, on peut essayer d'exécuter le code pour voir si il fonctionne.

- C'est le *software testing* (Test)
- En fait, tester est nécessaire dans tous les cas
  - Une preuve formelle n'est en général pas faite directement sur le code

Beware of bugs in the above code; I have only proved it correct, not tried it. [Knuth]

Principles of testing

- Les tests mettent en évidence des problèmes.
  - Ils ne peuvent pas prouver qu'il n'y a pas de problèmes
- Les tests exhaustifs sont impossibles.
  - Tester tous les combinaisons possibles d'entrées est en général infaisable
  - Example: 15 champs avec 5 valeurs possibles $\rightarrow$ 5^{15} possibilités
- Tester au plus tôt
  - Dans le processus de développement logiciel, les tests doivent démarrer aussi tôt que possible
  - Plus un problème est détecté tôt, moins il est coûteux à traiter
- Localité des problèmes
  - La plupart des problèmes observés pendant les tests sont liés à un (petit) sous-ensemble des modules (Pareto)

Principles of testing

- Le paradoxe des pesticides
  - Plus vous exécutez un test, moins il y a de chance qu'il mette en évidence un bug
- Tester dépend du contexte
  - Les méthodes et types de tests à utiliser sont liés au type d'applications
  - Un site web devra passer des tests de performance (en fonction du nombre d'utilisateurs)
  - Une application contrôlant des appareils médicaux devra répondre à des exigences spécifiques
- L'argument fallacieux de l'absence d'erreurs
  - Ça ne veut pas dire que le logiciel est prêt à être livré
  - Est-ce que les tests ont été correctement conçus?
  - Est-ce que le logiciel répond aux besoins des clients?

Objectifs des tests

2 objectifs principaux

- Validation: Est-ce que nous développons le bon produit?
  - Est ce que le produit répond aux besoins des utilisateurs?
- Vérification: Est-ce que le produit développé est bon?
  - Est-ce que le produit répond aux spécifications?

Plus généralement

- Trouver des malfaçons
- Avoir confiance dans la qualité du produit
- Fournir un retour sur les spécifications
- Prévenir des problèmes futurs

Types de tests

International Software Testing Qualifications Board

- Tests unitaires (*Component testing*): Teste les fonctions/modules de manière indépendante (pendant qu'on écrit le code)
- Tests d'intégration: Teste les interactions entre les modules
- Tests systèmes: Test du système intégré dans les conditions normales d'utilisation. Vérifie la conformité aux exigences spécifiées.

Types de tests

International Software Testing Qualifications Board

- Tests de validation (*Acceptance test*): Test si la solution proposée correspond aux besoins de l'utilisateur (implique en général l'utilisateur). Vérifie si le système est utilisable et prêt à être déployé.
- Tests de régression: Test d'un programme préalablement testé, après une modification, pour s'assurer que des défauts n'ont pas été introduits ou découverts dans des parties non modifiées du logiciel

Acceptance test: Alpha, Beta

Alpha testing

- Par l'entreprise développant le produit mais pas par les développeurs

Beta testing

- Par les clients ou les clients potentiels au sein de leur entreprise
- Procédure précise pour rapporter des bugs
- Il peut être embarrassant de donner un système buggé aux beta-testeurs

Niveaux de test

- Black-box testing
- Grey-box testing
- White-box testing

Black-box testing

Principe

- Est ce que le comportement du système correspond à sa spécification?
- Fourni une entrée et compare la sortie à la spécification
  - Sans spécification, tout résultat est valide

- Concevoir au moins un test par fonctionnalité
- Itérer sur le code jusqu'à passer tous les tests

Grey-box testing

Principe

- Utiliser la connaissance de l'architecture du système pour concevoir des tests "boite noire" plus complets

- Vérifier les informations des logs
  - Est-ce que pour chaque opération, l'état interne du système est bien mis à jour?
- Vérifier des informations spécifiques au système
  - Sommes de contrôle, estampilles
- Vérifier le "nettoyage"
  - Suppression des fichiers temporaires, fuites mémoire, ...

White-box testing

Principe

- Écriture de tests à partir d'une connaissance complète du code

- Tester toutes les parties du code (couverture)
- Erreurs traitées correctement
- Utilisation des ressources

Tests fonctionnels et non fonctionnels

Tests fonctionnels

- Evalue les sorties en fonction des entrées (black-box testing)

Tests non fonctionnels

- Fiabilité
- Utilisabilité
- Efficacité
- Maintenabilité
- Portabilité
- Sécurité

L'oracle

Comment décider si un test est passé avec succès?

- Approches automatisables:
  - Le programme ne doit pas échouer
  - Les assertions doivent être vérifiées
  - Comparer les résultats avec ceux d'un programme (supposé) correct
- En dernier recours:
  - Inspecter les résultats à la main
  - Note: certains tests sont difficilement automatisables (ex: utilisabilité)

Tests et spécification

La conception de tests peut aider à améliorer la spécification.

Exemple

Ajouter un enfant au dossier de Mme Martin.

- Test 1: Vérifier que le nombre d'enfants de Mme Martin a été incrémenté de 1
- Test 2: Vérifier aussi le nombre d'enfants de Mr Martin
- Test 3: Vérifier que le nombre d'enfants des autres personnes n'a pas changé

Equivalence Paritioning

Principe

- Découper le domaine d'entrée en groupes supposés avoir un comportement similaire
- Choisir une entrée dans chaque groupe
- Le découpage peut être appliqué aux entrées valides et non valides

Exemple

`int pgcd(int x, int y)` (plus grand commun diviseur)

Equivalence Paritioning

Principe

- Découper le domaine d'entrée en groupes supposés avoir un comportement similaire
- Choisir une entrée dans chaque groupe
- Le découpage peut être appliqué aux entrées valides et non valides

Exemple

`int pgcd(int x, int y)` (plus grand commun diviseur)

- $x = 6$, $y = 9$: $result = 3$ (cas normal)
- $x = 2$, $y = 4$: $result = 2$ ($x = GCD$)
- $x = 3$, $y = 5$: $result = 1$ (deux nombres premiers)
- $x = 9$, $y = 0$: $result =?$ (test avec 0)
- $x = -3$, $y = 9$: $result =?$ (test nombre négatif)

Boundary-value analysis

Principe

- Les valeurs aux frontières ont une probabilité plus importante de mettre en évidence un comportement non valide
- Peut être combiné avec du partitionnement

Exemple

Une fonction prend en entrée des valeurs entre 0 et 100.

- Tester avec -1, 0 et 1
- Tester avec 99, 100 et 101

Agenda

Introduction aux tests

JUnit

JUnit

xUnit: Outil permettant de réaliser des tests unitaires dans un langage donné

- SUnit: Smalltalk (Framework originel)
- CUnit: C (un parmi plusieurs)
- CppUnit: C++
- EUnit: Erlang
- LuaUnit: Lua

JUnit[1]: Tests unitaires pour Java

- Chercher les erreurs dans un sous-système isolé
- Le sous-système est une classe/un objet

[1] `junit.org`

En 2 mots

- Pour une classe `Foo`, on crée une autre classe `FooTest` pour la tester
- La classe `FooTest` contient plusieurs méthodes implémentant des cas de test (*test case*)
- Chaque méthode vérifie un résultat particulier, et peut réussir ou échouer (*pass/fail*)
- JUnit fournit des méthodes assert pour écrire les tests

Écrire une classe de test avec JUnit

JUnit 4

- Basé sur l'utilisation des annotations (@Annotation)
  - Méta-données pour enrichir le code source
- C'est ce que nous allons utiliser dans la suite

JUnit 5

- Quelques fonctionalités en plus par rapport à Junit 4
- Quelques evolutions dans les annotations

JUnit 6 vient de sortir

Premier exemple

```
import org.junit.*;
import static org.junit.Assert.*;

public class name {
    ...

    @Test
    public void name() {  // a test case method
        ...
    }
}
```

- Une méthode annotée avec `@Test` est marquée comme un cas de test JUnit
- Toutes les méthodes `@Test` sont exécutées lorsque JUnit exécute une classe de tests

Vocabulaire

- Test runner: Le logiciel qui exécute les tests et rapporte les résultats.
  - Ligne de commande, intégré à un IDE, ...
- Test suite: Une collection de cas de tests
- Test case: Test la réponse d'une méthode à une entrée donnée
- Assertion: Fonction ou macro vérifiant une condition à l'exécution, si la vérification échoue, une exception est levée/arrêt du test courant.
- Test Fixture: Initialisation/terminaison commune à tous les tests unitaires

Structure d'une classe de test

- La classe `ListTest` définit un ensemble de tests unitaires pour la classe `List`

```
public class ListTest{
}
```

- Cette classe a un constructeur par défaut

```
public ListTest(){ }
```

Structure d'une classe de test

- `@Before public void init()`
  - Définit une fixture à exécuter avant les tests
  - Création et initialisation des objets
  - Exécutée avant chaque test
- `@After public void cleanUp()`
  - Définit une fixture à exécuter après les tests
  - Libérer les ressources après un test
  - Est exécutée même si le test échoue/lève une exception
- `@Test public void testIsEmpty()`, `@Test public void testGet()`, `...`
  - Définit les cas de test à exécuter
  - Note: toutes ces méthodes doivent avoir la visibilité `public`

Les assertions

Au cours d'un test:

- Appeler la méthode à tester et récupérer le résultat
- *Assert* (Vérifier) une propriété qui doit être vraie sur le résultat

Si une propriété n'est pas vérifiée:

- L'assertion échoue et lève une `AssertionError`
- JUnit attrape les exceptions, enregistre le résultat des tests et affiche les résultats

Les assertions

Liste non exhaustive de méthodes disponibles[1]:

- `assertTrue(test)`: échoue si *test* $\neq$ *true*
- `assertFalse(test)`: échoue si *test* $\neq$ *false*
- `assertEquals(expected, actual)`: échoue si *expected* $\neq$ *actual* (utilise `equals()` si défini, $==$ sinon)
- `assertSame(expected, actual)`: échoue si *expected* et *actual* ne référencent pas le même objet
- `assertNull(test)`: échoue si *test* $\neq$ *null*
- `assertNotNull(test)`: échoue si *test* $==$ *null*
- `fail()`: échoue immédiatement

[1] `junit.org/javadoc/latest/index.html`

Exemple

```
import org.junit.*;
import static org.junit.Assert.*;

public class TestArrayIntList {
    @Test
    public void testAddGet() {
        ArrayIntList list = new ArrayIntList();
        list.add(-3);
        list.add(15);
        assertEquals("get first element", -3, list.get(1));
        assertEquals("get second element", 15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue("new list", list.isEmpty());
        list.add(123);
        assertFalse("list after one add", list.isEmpty());
        list.remove(0);
        assertTrue("list after one add+remove", list.isEmpty());
    }
...
```

Bonnes pratiques

- Ajouter un message décrivant les tests
  - Chaque méthode `assert` a une version qui prend un message en paramètre
- Ne pas afficher la valeur de `expected` et `actual` dans le message
  - Leur valeur sera affichée par JUnit en cas d'erreur
- La valeur de comparaison (`expected`) doit être à gauche

```
assertEquals("get first element", list.get(1), -3); //wrong!!
```

- Donner aux méthodes de cas de tests des noms explicites

```
@Test
public void test_remove_without_add(){...}
```

Comparer des objets

First version

```
@Test
public void test1() {
    Date d = new Date(2050, 2, 15);
    d.addDays(4);
    assertEquals(2050, d.getYear());
    assertEquals(2, d.getMonth());
    assertEquals(19, d.getDay());
}
```

Better version

```
@Test
public void test2() {
    Date d = new Date(2050, 2, 15);
    d.addDays(4);
    Date expected = new Date(2050, 2, 19);
    assertEquals("date after 4 days", expected, d);
}
```

Comparer des objets

Nécessite de définir la méthode equals()

```
public boolean equals( Object anObject ){
  if( anObject instanceof Date ) {
    Date d= (Date)anObject;
    return d.getYear() == getYear()
              && d.getMont()==getMonth()
              && d.getDay() == getDay();
  }
  return false;
}
```

Utilisation de timeouts

Que se passe-t-il si un test bloque ou si il met trop de temps à s'exécuter?

- Utilisation de `timeout`
- Paramètre de l'annotation `@Test`
- Définit un temps en ms après lequel le test échoue si il n'est pas terminé

```
@Test(timeout = 5000)
public void name() { ... }
```

Utilisation de timeouts

Affecter un timeout à tous les cas de test:

- Utilisation d'une variable statique

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_withinSameMonth() {
      ...
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_wrapToNextMonth() {
      ...
    }

    private static final int DEFAULT_TIMEOUT = 2000;
}
```

Utilisation de timeouts

Affecter un timeout à tous les cas de test:

- Timeout rule

```
public class DateTest {
    @Rule
    public Timeout globalTimeout = Timeout.seconds(10);
}
```

Tester les exceptions

Comment tester que notre code lève bien l'exception attendue?

```
@Test(expected = ExceptionType.class)
public void name() throws Exception{
  ...
}
```

- Le test passe si l'exception est levée
- Le test échoue sinon

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() throws Exception{
      ArrayIntList list = new ArrayIntList();
      list.get(4);   // should fail
}
```

Setup and teardown

- Méthodes appelées avant/après chaque méthode test
  - `@Before`
  - `@After`
- Méthodes appelées une fois avant/après l'exécution des méthodes de test
  - `@BeforeClass`
  - `@AfterClass`

Éviter les redondances

Définition de méthodes *helpers*

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_withinSameMonth_1() {
        addHelper(2050, 2, 15, +4, 2050, 2, 19);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_wrapToNextMonth_2() {
        addHelper(2050, 2, 15, +14, 2050, 3, 1);
    }

    // use lots of helpers to make actual tests extremely short
    private void addHelper(int y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date act = new Date(y, m, d);
        actual.addDays(add);
        Date exp = new Date(y2, m2, d2);
        assertEquals("after +" + add + " days", exp, act);
    }
...
```

Manipulation de structures de données

- Besoin de tableaux à passer en paramètres

```
public void exampleMethod(int[] values) { ... }
...
exampleMethod(new int[] {1, 2, 3, 4});
exampleMethod(new int[] {5, 6, 7});
```

- Besoin d'une `ArrayList`: Pensez à `Arrays.asList`

```
List<Integer> list = Arrays.asList(7, 4, -2, 3, 9, 18);
```

- Besoin d'un set, file, tas: On peut souvent partir d'une liste

```
Set<Integer> list = new HashSet<Integer>(
                            Arrays.asList(7, 4, -2, 9));
```

De bonnes méthodes de test

- Tester une chose à la fois
  - 10 petits tests $>$ un gros test
- Chaque cas de test doit avoir peu d'assertions (1?)
  - La première assertion fausse arrête le test
  - Avec plusieurs assertions, on ne sait pas si les suivantes auraient échoué
- Éviter les structures conditionnelles dans les tests (if, switch/case, loops, . . .)
  - Éviter aussi try/catch (utilisation du paramètre `expected`)
- Éviter que plusieurs méthodes manipulent le même objet
  - Que ce passe-t-il si la première échoue et corrompt l'objet?

Quelques conseils en plus

- Les tests doivent être *silencieux*
  - Pas de `System.out.println()`
  - Utiliser des assertions
- Lorsque vous êtes tenté d'ajouter un `println` dans le code de votre application:
  - Ajoutez plutôt un test
- Comment peut-on tester une méthode qui n'a pas de valeur de retour?
  - Tester les effets de bord

Test-driven development (TDD)

Développement piloté par le test

Les tests unitaires peuvent être écrits après, pendant ou même avant de coder

- TDD: Ecrire les tests, puis écrire le code pour les passer
- Concevoir les tests permet de clarifier le fonctionnement attendu d'une méthode

JUnit5

Introduction

- Stable depuis 2017
- Utilise des annotations comme JUnit4
  - Certaines annotations changent avec le changement de version (ex: `@Before` devient `@BeforeEach`)
- Tire partie des fonctionnalités introduites dans Java8 (lambda expressions)

Nouvelles fonctionnalités

- Tests paramétrés
- Exécution conditionnelle de tests
- Répétition de tests
- Tests imbriqués
- etc.

Nouvelle manière de tester les exceptions (JUnit5)

Utilisation de la fonction `assertThrows()` avec une expression lambda.

```
Exception exception = assertThrows(
                MyException.class,
                () -> myMethod(), "message if assertion fails");

assertEquals("exception message", exception.getMessage());
```

- Le test passe si l'exception est levée
- Le code ci-dessus permet en plus de vérifier que le message associé à l'exception est celui attendu

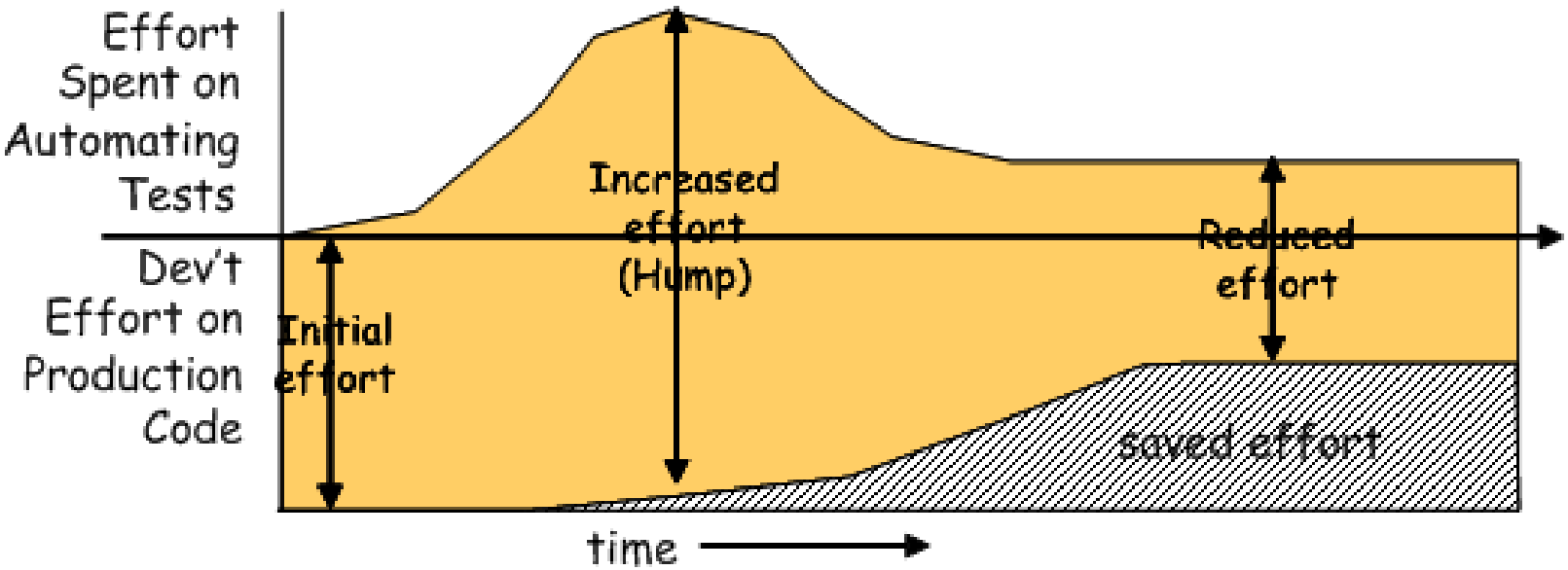
```
@Test
public void testBadIndex(){
      ArrayIntList list = new ArrayIntList();
      assertThrows( ArrayIndexOutOfBoundsException.class, () -> {
                  list.get(4)
      });
}
```

Résumé

- Les tests doivent permettre de savoir ce qui a échoué
  - Un test doit avoir un nom descriptif
  - Une assertion doit avoir un message clair
  - Écrire pleins de petits tests, plutôt qu'un gros
- Toujours mettre un `timeout`
- Tester les cas d'erreur/exception
- Utiliser l'assertion appropriée, pas toujours `assertTrue()`
- Éviter les structures de contrôle
- Utiliser les helpers et `@Before` pour éviter les redondances entre les tests

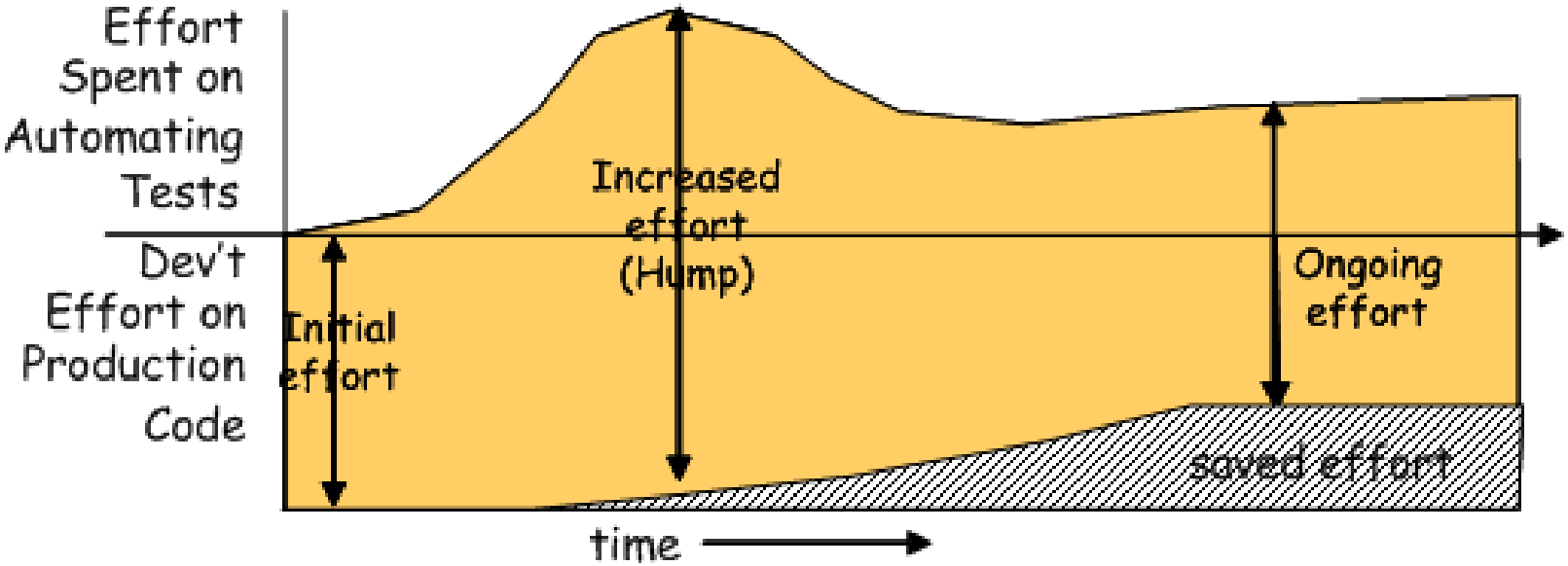
Objectifs des tests automatisés

xUnit Test Patterns – Refactoring Test Code



A éviter

xUnit Test Patterns – Refactoring Test Code



Causes de coûts qui restent importants dans la mise en place de tests automatisés

- Tests illisibles ou trop compliqués (difficiles à maintenir)
- Tests fragiles (ne fonctionnant plus au moindre petit changement dans la spécification, ajout de fonctionnalités, etc.)

Objectifs des tests automatisés (v2)

xUnit Test Patterns – Refactoring Test Code

- Nous aider à améliorer la qualité des logiciels
  - Améliorer la spécification (TDD)
  - Éviter les bugs
  - Localiser facilement les problèmes
- Nous aider à comprendre le logiciel
  - Les tests unitaires font office de documentation *exécutable*
- Réduire les risques
- Les tests doivent être faciles à utiliser (automatisés, reproductibles)
- Les tests doivent être faciles à écrire et à maintenir
- Les tests doivent nécessiter peu de changements quand le système évolue

Références

- Notes de A. Groce
- Notes de J. Pearson
- Notes de K. Anderson
- Notes de M. Stepp sur JUnit
- Notes de P. Labatut sur JUnit
- International Software Testing Qualification Board Syllabus
- *xUnit Test Patterns – Refactoring Test Code* de G. Meszaros

Des slides en plus

Suite de tests

Test Suite: Une classe qui exécute plusieurs tests.

- Exécuter tous les tests d'une application
- `@RunWith(Suite.class)`: Exécute la classe dans l'exécuteur Suite
- `@Suite.SuiteClasses({class1, class2})`: Spécifie les classes à exécuter

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class MyTestSuite{}
```

Exécuter une suite de tests

Exécution à la ligne de commande

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(MyTestSuite.class);
    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }
    System.out.println(result.wasSuccessful());
  }
}
```

- Compiler avec `javac`
- Exécuter avec `java`
- Penser à ajouter `junit.jar` au classpath

JUnit et Ant

Possibilité d'ajouter une tâche JUnit à un fichier `build.xml`

- Ajouter une cible "test" qui dépend de la cible "compile"
- Modifier le classpath pour y ajouter `junit.jar`
- Ajouter la tâche JUnit qui va exécuter les tests

Attributs de la tâche JUnit

- haltonerror: S'arrêter quand un test échoue
- haltonfailure: S'arrêter en cas de problème
- printsummary: Demande à Ant d'afficher des statistiques sur les tests
  - Utiliser un `formatter` pour formater la sortie

Exemple de build.xml

```
<project name="JunitTest" default="test" basedir=".">
   <property name="testdir" location="test" />
   <path id="classpath.test">
      <pathelement location="/lib/junit-4.10.jar" />
   </path>

   ...
   <target name="compile" depends="clean">
      <javac srcdir="${srcdir}" destdir="${testdir}">
         <classpath refid="classpath.test"/>
      </javac>
   </target>

   <target name="test" depends="compile">
      <junit haltonfailure="true" printsummary="true">
         <classpath refid="classpath.test" />
         <formatter type="brief" usefile="false" />
         <test name="mypackage.DateTest" />
      </junit>
   </target>
</project>
```