Slide 1

OCETJ : un outil d'aide à la correction de programmes Java basé sur JUnit . . . et ses implications possibles pour nos cours de base

Séminaire départemental

Dépt. d'informatique, UQAM

2 mai 2003

É. Labonté
Dépt. d'informatique
Cégep du Vieux-Montréal

G. Tremblay
Dépt. d'informatique
UQAM

Slide 2

Aperçu

- Qu'est-ce que JUnit?
- Qu'est-ce que OCETJ — ou comment JUnit peut être utilisé pour aider à la correction de programmes Java?
- Quelles sont les implications possibles pour nos cours?
  - Quelles "pratiques" du GL devraient être intégrées dans les cours de base?
  - Pourquoi les pratiques suivantes sont importantes?
    - Séparer la présentation et le modèle logique.
    - Documenter à l'aide d'assertions.
    - Automatiser les tests unitaires.

Slide 3

1 JUnit : un cadre de tests pour Java

Outil popularisé par les promoteurs de la programmation extrême (XP) :

- Support pour l'automatisation des tests
  (support pour les tests de régression, pour l'intégration continue, etc.).
- Permet l'organisation structurée des jeux d'essais
  (cas de tests, suites de tests, classes de tests).
- Mécanismes pour construction d'échafaudages de tests (`setUp`, `tearDown`).
- Librairies pour les tests d'objets complexes
  (par ex., *mock objects* pour objets `Output`, servlets, BDs, GUIs, etc.).

Remarque importante :

- `xUnit` = famille de cadres de tests avec variantes pour divers langages
  (Ada, C, C++, Eiffel, Java, Perl, Python, etc.)
- JUnit = Instance Java du cadre `xUnit` *(variante la plus connue)*

Slide 4

Caractéristique importante :

- Basé sur l'utilisation d'*assertions*, plutôt que sur la vérification de résultats textuels :

```
assertEquals( expectedResult, value )
assertEquals( expectedResult, value, precision )
assertTrue( booleanExpression )
assertNotNull( reference )
etc.
```

  $\Rightarrow$ Aucun résultat n'est produit si le test ne détecte pas d'erreur.

Slide 5

Un petit exemple

La classe à tester :

```
class Account {
  private Customer cstm;
  private int bal;

  public Account( Customer c, int initBal )
  { cstm = c; bal = initBal; }

  public int balance()
  { return( bal ); }

  public Customer customer()
  { return( cstm ); }

  public void deposit( int amount )
  { bal += amount; }

  public void withdraw( int amount )
  { bal += amount; }                // Error!
}
```

Slide 6

La classe de test :

```
import junit.framework;

public class AccountTest extends TestCase {
  private Customer c1, c2;
  private Account acc1;

  public AccountTest( String nom ) {
    super(nom);
  }

  protected void setUp() {
    c1 = new Customer( "Tremblay" );
    c2 = new Customer( "Labonte" );
    acc1 = new Account( c1, 100 );
  }

  public void testGetCustomer() {
    assertEquals( c1, acc1.customer() );
  }
```

Slide 7

```
  public void testNewAccount() {
    Account acc = new Account( c2, 200 );
    assertTrue( acc.customer() == c2 &&
                acc.balance()  == 200 );
  }

  public void testTransfer() {
    int initBal = acc1.balance();
    acc1.deposit ( 50 );
    acc1.withdraw( 50 );
    assertEquals( initBal, acc1.balance() );
  }

  public static Test suite() {
    return new TestSuite(AccountTest.class);    // "reflection"
  }

  public static void main( String[] args ) {
    junit.textui.TestRunner.run( suite() );
  }
}
```

Slide 8

Compilation et exécution de la classe de test `AccountTest` :

```
% javac Account.java
% javac AccountTest.java
% java AccountTest

There was 1 failure:
1) testTransfer(AccountTest)
    junit.framework.AssertionFailedError:
    expected:<100> but was:<200>
at AccountTest.testTransfer(AccountTest.java:31)
at AccountTest.main(AccountTest.java:39)

FAILURES!!!
Tests run: 3,  Failures: 1,  Errors: 0
```

Slide 9

Des *vues graphiques* des mêmes jeux d'essai sont aussi disponibles (`swingui` ou `awtui`).

Seule modification au programme principal (classe de test) :

```
public static void main( String[] args ) {
  junit.swingui.TestRunner.run( suite() );
}
```

Remarque pédagogique : Illustre comment des *vues différentes* sur un même composant logiciel peuvent être disponibles lorsque la partie présentation est clairement séparée de la partie logique.

Slide 10

Autres caractéristiques :

- Plusieurs suites de tests peuvent être définies et organisées de façon hiérarchique : une suite de tests est formée de cas de tests ou de *suites* de tests.

- Les exceptions peuvent aussi être testées, c'est-à-dire, on peut vérifier qu'elles sont bien signalées (le comportement du module est erroné si une exception n'est pas signalée) :

```
public void deposit( int amount ) throws InvalidAmountException;

public void testInvalidAmountException() {
  try {
    acc1.deposit ( -50 );
    fail( "InvalidAmountException has not been thrown." );
  } catch( InvalidAmountException ) {
  }
}
```

2 OCETJ : Un outil d'aide à la correction de travaux Java

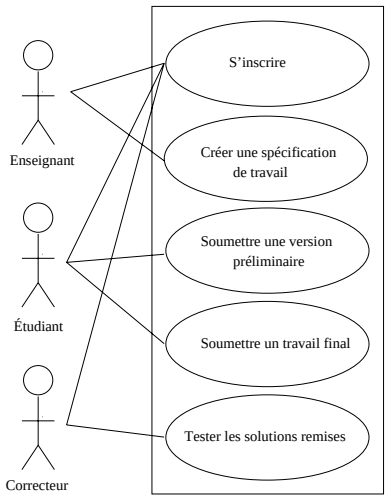
OCETJ = Outil de Correction et d'Évaluation de Travaux Java

Deux objectifs majeurs :

- Fournir du support aux enseignants dans leur travail de correction des travaux de programmation, plus précisément, faciliter le travail associé aux tests des travaux des étudiants.
- Fournir du *feedback* aux étudiants, et ce rapidement, si possible avant même la date finale de remise des travaux.

Slide 11

2.1 Principales fonctionalités d'OCETJ



Création d'une spécification de travail :

- Nom du travail.
- Suites publiques et privées de tests.
- Date limite de remise.
- Nombre maximum d'essais.

Slide 12

Slide 13

2.2 Mise en oeuvre d'OCETJ

Application (style) *web* réalisée à l'aide de diverses technologies Java
(Dépt. d'informatique, Cégep du Vieux-Montréal) :

- Accès au système : navigateur *web*.
- Présentation (pages *web* et scripts) : servlets Java.
- Logique de l'application : code Java et fichiers `.bat`.
- Données persistentes :
  - Suites (privées et publiques) de tests de l'enseignant et solutions des étudiants : réseau local (Novell) avec dossiers spéciaux de remise.
  - Données créées et gérées par OCETJ : base de données Oracle (manipulée avec JDBC).

Slide 14

Approche pédagogique utilisée (prog.-*in-the-small* sans E/S) combinée à l'utilisation de JUnit =>

- Code soumis par un étudiant . . .
  - contient la mise en oeuvre de diverses classes
  - *ne contient* aucun *programme principal*
- Programme principal = classe de tests (de l'étudiant ou de l'enseignant).

Vérification *d'une* solution soumise par un étudiant =

- Compiler (avec `javac`) le code de l'étudiant.
- Exécuter (avec `java`) le programme résultant
  (code de l'étudiant + classe de tests de l'enseignant).
- Conserver (dans la BD) les résultats des tests
  (nombre et nom des tests échoués).

Et . . . ceci doit être répété *pour chacun des programmes remis*.

Mise en oeuvre : fichier `.bat` exécuté (par l'intermédiaire du servlet) en créant un `Process` avec la méthode `exec` de la classe `Runtime`.

Slide 15

État de la mise en oeuvre :

- Pour des raisons techniques et de sécurité, OCETJ ne fonctionne actuellement que sur l'intranet du Cégep du Vieux-Montréal (accès aux répertoires de remise), et non pas comme une véritable application *web* (par ex., soumission par courriel).
- Est utilisé cette session dans le cadre du cours 420-A32-VM au Cégep du Vieux-Montréal.

Slide 16

3 Construction de logiciels et tests unitaires

La place du GL dans les cours de base

Nombreuses façons d'introduire le GL dans les cours de base :

- Production de documents d'analyse et de conception.
- Imposition de standards de documentation.
- Utilisation d'un langage *de haut niveau* avec garde-fous.
- Utilisation d'environnements et d'outils *industriels*.
- Introduction de techniques de gestion du temps (à la PSP).
- ...

Mais ... il faut tenir compte *du niveau* des étudiants (concret avant abstrait)

$\approx$ ne pas mettre *la charrue devant les boeufs*

Questions clés à examiner =

- À quelle étape du cycle de développement correspondent les cours de base?
- Quelles sont les principales caractéristiques des artéfacts produits au cours de cette étape (aspect *produit*)?
- Quelles sont les pratiques clés de cette étape (aspect *processus*)?

Slide 17

Étape clé = Construction de logiciels :

- Production de code (avec un langage de programmation).
- Développement de *tests unitaires*.
- Intégration de modules (construction d'un exécutable (*build*)).

Caractéristiques d'un bon produit :

- Simple et modulaire.
- Interfaces claires et bien documentées (par ex., *Design By Contract*).
- Bonne séparation entre la présentation et la logique d'affaire.
- Testable et modifiable

Slide 18

Pratiques modernes clés (récemment *popularisées* par les méthodes agiles) :

- Intégration des tests *dans le livrable*

  +
- Automatisation des tests

Slide 19

Parenthèse sur les méthodes agiles

- Méthode agile = processus plus léger, mais quand même *discipliné*, de développement de logiciels. Par ex. : XP (*eXtreme Programming*).
- On reconnaît de plus en plus que les méthodes agiles *ont leur place*.

  > *Agile and plan-driven methods both form part of the planning spectrum. Despite certain extreme terminology, each is part of the* responsible center *rather than the radical fringe. Indeed, agile methods perform a valuable service by drawing erstwhile cowboy programmers toward a more responsible center.*
  >
  > [B. Boehm, "*Get Ready for Agile Methods, With Care*", 2002]

- Particulièrement adaptées pour les petits et moyens projets.
- Fait : Les TPs (cours de base) sont des petits projets.

  En d'autres mots : pourquoi imposer un cadre de développement (approche *plan-driven*) à des projets pour lesquels ce cadre ne convient pas?

Slide 20

Séparation présentation/logique d'affaire

> [La] séparation de la présentation du modèle est l'une des heuristiques les plus fondamentales pour une bonne conception logicielle. Cette séparation est importante pour plusieurs raisons [:]
>
> [M. Fowler, "*Patterns of enterprise application architecture*"]

- Différentes présentations des mêmes informations.
- Présentation et modèle sont deux problèmes tout à fait différents (IPM vs. règles d'affaire, accès BD), utilisant des outils et librairies différents (i.e., spécialisations différentes).
- Les objets graphiques et visuels sont plus difficiles à tester.

  Séparation $\Rightarrow$ possibilité de tester la logique du domaine sans outils spéciaux pour les interfaces graphiques.

Slide 21

Implications pédagogiques possibles?

$\Rightarrow$ À bas les entrées–sorties textuelles dans les cours de base

- De moins en moins le style d'interaction auquel sont habitués les étudiants.
- Pas réaliste par rapport aux logiciels modernes.
- Plus difficile à tester.

. . . et surtout . . .

- Conduit souvent à une mauvaise façon d'organiser un programme.

Slide 22

Contrats clairs et bien documentés

- Bien documenter un composant logiciel
  $\neq$ écrire de lourds documents
  = décrire clairement le comportement du composant, ses limites, etc.
  $\Rightarrow$ Conception par contrat (au sens de B. Meyer)
- Un *contrat* définit les responsabilités et limites à l'aide d'assertions :
  - Pré/post-conditions
  - Invariants (de classe, de boucle, etc.)
- De nombreux langages permettent maintenant . . .
  - de spécifier les pré/post-conditions et invariants à l'aide d'asssertions.
  - de vérifier les assertions à l'exécution.
- Autre avantage : l'utilisation d'assertions facilite/réduit le débuggage.
  [Hunt & Thomas, "*The Pragmatic Programmer*"]
  - "*Crash Early*"
  - "*If It Can't Happen, Use Assertions to Ensure That It Won't*"

Slide 23

Tests unitaires

Tests unitaires = tests qui vérifient le bon fonctionnement *d'un module*.

- Base (fondation) de tous les autres tests :
  module pas testé $\Rightarrow$ pas *pertinent* d'effectuer les autres tests.

- Il est vrai que "*Testing can show the presence of bugs, never their absence*" [E. Dijkstra], mais c'est malgré tout *la* technique *de base*, utilisée par tous les programmeurs.

- [Hunt & Thomas, "*The Pragmatic Programmer*"]
  - "*Test Your Software, or Your Users Will*"
  - "*Coding Ain't Done 'Til All the Tests Run*"

Slide 24

Reconnaissance de plus en plus grande de l'importance des tests :

- Automatisation des tests.

  $\Rightarrow$ commande simple pour exécuter, de façon *automatique*, l'ensemble des tests.

  $\Rightarrow$ il ne doit pas être nécessaire de vérifier, *à bras* (*à l'oeil*), les résultats produits pour déterminer si tout est correct.

  Exemple : les modules `perl` (CPAN) $\Rightarrow$

  ```
  make install
  make
  make test
  ```

- Permet de supporter
  - L'intégration continue.
  - Le remodelage (*refactoring*) $\Rightarrow$ tests de régression.

Hypothèse fondamentale concernant le livrable "code" d'un logiciel :

- Le code *inclut* les tests unitaires (exécutables).

Slide 25

Test-Driven Development (TDD)

TDD $\Rightarrow$ les tests devraient être développés *avant même* d'écrire le code :

- *Write new code only if an automated test has failed.*
- *Eliminate duplication.*

[K. Beck, "*Test-Driven Development — By Example*"]

L'objectif = "*Clean code that works*."

Corollaire de la première règle :

- *Write a failing automated test before you write any code.*

Implication de la deuxième règle :

- Remodelage fréquent du programme (*refactoring*) (= *design-in-the-small*).

Slide 26

Affirmation (à première vue *paradoxale*!?) de K. Beck, concernant TDD :

Test-first coding is not a testing technique [K. Beck, "*Aim, Fire*"]

Explications :

- Aide à clarifier ce qui doit être programmé (i.e., les résultats attendus)
  $\Rightarrow$ forme de spécification
- Force à préciser l'interface
  - Test $\approx$ virus $\approx$ forme *inverse* du code qu'il faut écrire.
  - Les tests unitaires, intégrés à un composant logiciel, fournissent des exemples illustrant comment utiliser le composant.

  $\Rightarrow$ forme de documentation
- Améliore la modularité (augmente la cohésion, réduit le couplage).
  $\Rightarrow$ forme de technique de conception

Slide 27

Implications pédagogiques possibles?

Une approche *inspirée* de TDD pourrait (?) permettre de ...

- Introduire une approche *disciplinée*, mais légère, de construction de logiciels, adaptée *au niveau* des étudiants (concret).
- Introduire *tôt* les techniques de tests unitaires (et leur automatisation).
- Mettre l'accent sur la mise en oeuvre de la logique d'affaires.

  $\Rightarrow$ rend secondaires les entrées-sorties (textuelles ou non).

  $\Rightarrow$ illustre l'indépendance face à la *présentation* :
  classe de tests $\approx$ *vue* alternative.
- Introduire tôt, de façon "légère", les assertions.
- Introduire tôt les notions d'objets dans un cours de POO :

  - Objets définis par l'enseignant, puis créés et manipulés par les étudiants (classes de tests).

Slide 28

4 Conclusion

Les deux motivations initiales de OCTEJ étaient les suivantes :

1. Aider à *réduire* (mais non *supprimer*) la charge de correction dans des cours de programmation.
2. Fournir du *feedback* rapide aux étudiants.

Toutefois, l'utilisation d'un outil style OCETJ et d'un cadre de tests à la `xUnit` permettrait aussi d'introduire certaines idées et pratiques intéressantes du GL :

- Il faut séparer la présentation et la logique d'affaires.
- Les tests unitaires jouent un rôle clé dans la construction de logiciels corrects.

  = Les tests font partie *intégrante* du composant logiciel.

  $\Rightarrow$ L'exécution des tests devrait se faire de façon automatique.
- Les assertions permettent de bien décrire des composants logiciels.

Travaux futurs (outil de correction) :

- Modifier OCETJ pour en faire une véritable application *web*.
- Intégrer diverses analyses (métriques) pour évaluer la "*qualité*" des programmes.

Slide 29

Discussions futures (enseignement des cours de base et de GL) :

- Quelle place donner aux tests unitaires (automatisés)?
- Devrait-on introduire une approche de style TDD?
- Comment favoriser la pratique de séparation présentation/couche logique?
- Comment généraliser un peu plus l'utilisation des assertions `(` pas juste en INF3140 `;(`?
- ...