Techniques et Outils d'Ingénierie

Exercices

2014–2015

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

Stéphane Lopes

Chapitre 1

Environnement de développement

CONTRAINTES
- Utilisation de l'environnement de développement ECLIPSE.
- Vous pouvez consulter :
  - l'aide d'ECLIPSE sur le *workbench* et le développement Java,
  - une liste des principaux raccourci d'ECLIPSE.
  - un tutoriel sur le développemen Java avec ECLIPSE.

Exercice 1.1 (Découverte d'Eclipse)

1. Procédez au paramétrage initial d'Eclipse suivant (*Window* > *Preferences*) :
   - ajoutez un dictionnaire français (*General* > *Editors* > *Text Editors* > *Spelling* en ISO-8859-1),
   - modifiez l'encodage des caractères en *UTF-8* et les fins de ligne en *Unix* (*General* > *Workspace*),
   - vérifiez que l'option *Build automatically* est cochée (*General* > *Workspace*),
   - vérifiez les versions du JDK installées sur la machine et sélectionnez la plus récente (*Java*),
   - attachez les sources du JDK à la bibliothèque `rt.jar` (*Source Attachment...* dans les propriétés du JRE),
   - définissez les chemins des sources (`src/main/java`) et du bytecode (`target/classes`) pour les projets,
   - vérifiez la version du compilateur utilisé et sélectionez la plus récente,
   - activez la mise en forme du code lors des sauvegardes (*Java* > *Editor* > *Save Actions*).
2. Créez le projet Java DESSIN en conservant les options par défaut ;
3. Importez l'archive `Dessin.zip` (à partir d'e-campus) dans le répertoire des sources du projet.
4. Explorez le projet :
   - en utilisant la vue *Package Explorer*, naviguez dans les sources pour ouvrir le fichier `DrawingApp.java` dans un éditeur,
   - dans la vue *Outline*, utilisez la barre d'icones pour masquer/voir les différents éléments (membres statiques, ...),
   - quel impact la vue *Outline* a-t-elle sur l'éditeur ?
   - dans l'éditeur, faites apparaître la vue *Quick outline* (*Ctrl+o*) puis les membres hérités (*Ctrl+o*) et sélectionnez la méthode `java.lang.Object.toString()`,
   - dans l'éditeur, sur le fichier `DrawingApp.java`, placez le curseur sur le type `Shape` et faites apparaître la *hiérarchie des types* (*Ctrl+T* pour *Quick Type Hierarchy* ou *F4* pour la vue *Type Hierarchy*),
   - quelles classes héritent de `Shape` ?
5. Visualisez les erreurs du projet :
   - comment les erreurs dans le projet sont-elles signalées (vue et moyen) ?
   - placez le pointeur de la souris sur les différents symboles d'erreur dans l'éditeur,
   - placez le curseur sur l'erreur concernant le type `LogFactory`, faites apparaître la vue *Quick Fix* (*Ctrl+1*) et sélectionnez *Fix project setup...*.

6. Ajoutez la librairie de journalisation dans le projet :
   — créez un répertoire `lib` dans le projet,
   — récupérez et décompressez l'archive de la librairie *Apache Commons logging* (http://commons.apache.org/logging/) dans ce répertoire,
   — ajoutez la librairie au projet (*Project* > *Properties*) en y associant les sources et la javadoc.
7. Exécutez le projet :
   — lancez le programme comme une application Java,
   — éditez la configuration d'exécution pour définir les propriétés systèmes suivantes (*VM arguments*) :
   ```
   -Dorg.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
   -Dorg.apache.commons.logging.simplelog.defaultlog=trace
   ```
   — lancez à nouveau le programme. Que remarquez-vous ?
8. Editez le code :
   — dans la classe `DrawingApp`, ajoutez la méthode `public void trace()`,
   — utilisez *Content Assist* pour saisir le corps de la méthode suivant (après `log.`, tapez *Ctrl+espace*, puis `tr`, ...) :
   ```
   log.trace(shapes);
   ```
   — ajoutez la méthode `public void translate()`,
   — saisissez le corps suivant en utilisant les *templates de code* (après `for`, tapez *Ctrl+espace*, puis sélectionner *foreach*, puis *Tab* entre les champs) :
   ```
   for (Shape s : shapes) {
     s.translate(2.0, 3.0);
   }
   ```
   — Organisez les importations de module :
   — dans la vue *Outline*, rendez visible les déclarations d'*import* (*View Menu* > *Filters...*),
   — dans la même vue, supprimez les déclarations d'*import* du fichier Java,
   — enfin, dans l'éditeur, ajoutez automatiquement les déclarations d'*import* (*Source* > *Organize Imports* ou *Shift+Ctrl+o*).
9. Apportez les modifications suivantes au projet (*Refactoring*) :
   — changez le nom du package `forms` en `shapes`,
10. Générez un Jar exécutable du projet contenant les dépendances :
    — exportez le projet comme un Jar exécutable,
    — vérifier en ligne de commande que l'archive fonctionne de façon autonome.

Exercice 1.2 (Autres IDE)
Vous pouvez reprendre l'exercice précédent avec l'IDE de votre choix (IntelliJIDEA, NetBeans IDE, ...).

Chapitre 2

Style de codage, documentation et analyse statique

CONTRAINTES
- Sauf information contraire, l'ensemble de ce TD se déroule sous l'environnement de développement ECLIPSE.
- Vous pouvez consulter :
  - un tutoriel sur Checkstyle,

Exercice 2.1 (Javadoc)
Dans cet exercice, vous travaillerez sur le même projet `Dessin` que pour le TP précédent.

1. Documentez la classe `Point` à l'aide de commentaires Javadoc respectant les conventions de codage SUN ;
2. Définissez le chemin de la Javadoc du projet à `target/docs/apidocs` (*Project* > *Properties*) ;
3. Générez la Javadoc (Menu *Project*) en plaçant les fichiers résultants dans `target/docs/apidocs` ;
4. Exportez le répertoire de la Javadoc sous la forme d'un Jar. Le fichier `index.html` doit se trouver à la racine du Jar.

Exercice 2.2 (Checkstyle)
Dans cet exercice, vous travaillerez sur le même projet `Dessin` que pour le TP précédent. De plus, cet exercice nécessite l'installation du plugin eclipse-cs sous ECLIPSE.

1. Définissez par défaut les conventions « Sun Conventions (Eclipse) » pour Checkstyle (*Windows* > *Preferences*) ;
2. Activez Checkstyle pour le projet ;
3. Consultez la liste des avertissements dans la vue *Problems* puis dans la vue *Checkstyle violations* ;
4. Créez une configuration personnalisée :
   - effectuez une copie de la configuration « Sun Conventions (Eclipse) »,
   - configurez cette copie en désactivant la vérification des commentaires Javadoc,
   - sélectionnez cette configuration pour le projet.

Exercice 2.3 (Findbugs)
Dans cet exercice, vous travaillerez sur le même projet `Dessin` que pour le TP précédent. De plus, cet exercice nécessite l'installation du plugin Findbugs sous ECLIPSE.

1. Activez Findbugs pour qu'il s'exécute automatiquement ;
2. Ajoutez la ligne suivante au début de la méthode `void run(String[] args)` :

   ```
   shapes = null;
   ```

3. Ouvrez la vue *Bug Explorer* pour consulter les détails sur le problème.

Chapitre 3

Gestion de versions/Travail collaboratif

CONTRAINTES
- Utilisation de l'environnement de développement ECLIPSE.
- Pour chaque commande réalisée dans l'IDE, vous donnerez la ligne de commande correspondante.
- Vous aurez besoin d'un compte sur GitHub pour la fin du TD.

Exercice 3.1 (Subversion)
Dans cet exercice, vous travaillerez sur une copie du projet `Dessin` nommée `DessinSVN`.

1. Dans la vue *SVN Repositories*, créez un dépôt local nommé `Dessin.svn` dans le répertoire de votre choix ;
2. Partagez le projet `DessinSVN` dans ce dépôt (uniquement le répertoire `src` et son contenu) ; Les fichiers devront se trouver dans le sous-répertoire `trunk` du dépôt ;
3. Créez une étiquette pour marquer la version 1.0 du projet ;
4. Créez une branche nommée `avec_triangle` ;
5. Commuter la copie de travail vers cette nouvelle branche ;
6. Ajouter une classe `Triangle` au projet et validez les modifications ;
7. Commuter la copie de travail vers la ligne principale ;
8. Fusionner la copie de travail avec la branche `avec_triangle` ;
9. Vérifier que le programme compile et fonctionne normalement puis valider les changements ;
10. Supprimez la branche.

Exercice 3.2 (Git)
Dans cet exercice, vous travaillerez sur une copie du projet `Dessin` nommée `DessinGIT`.

1. Configurer EGit avec votre nom (*user.name*) et votre email (*user.email*) ;
2. Partagez le projet `DessinGIT` en créant un dépot local nommé `Dessin.git` dans le répertoire de votre choix ;
3. Ajoutez les répertoires `lib` et `bin` aux fichiers ignorés par *git* (fichier `.gitignore`) ;
4. Dans la vue *Git Staging*, ajoutez le répertoire `src` aux fichiers suivis puis validez ;
5. Créez une étiquette pour marquer la version 1.0 du projet ;
6. Ouvrez la vue *Git Repositories* pour examiner la structure du dépôt ;
7. Créez une branche nommée `avec_triangle` et commutez vers cette dernière ;
8. Ajouter une classe `Triangle` au projet et validez les modifications ;
9. Commuter la copie de travail vers la ligne principale ;
10. Fusionner la copie de travail avec la branche `avec_triangle` puis valider les changements ;
11. Consultez l'historique des révisions ;

12. Supprimez la branche ;
13. Partager ce projet sur GitHub
    (a) créez un dépôt nommé `dessin` dans votre compte,
    (b) « poussez » votre projet dans ce dépôt,
    (c) effectuez une modification sur le projet, validez-la puis envoyer les modifications sur le dépôt distant.
14. Essayez de provoquer un conflit dans le projet et réglez-le.

Chapitre 4

Gestion de la compilation

CONTRAINTES
- Utilisation de l'environnement de développement ECLIPSE.
- Pour chaque commande réalisée dans l'IDE, vous donnerez la ligne de commande correspondante.

Exercice 4.1 (Ant)
Dans cet exercice, vous travaillerez sur une copie du projet `Dessin` nommée `DessinAnt`.

1. Créez un fichier `build.xml` dans la racine du projet ;
2. Intégrez les cibles suivantes : compile, jar, javadoc ;
3. Testez les différentes cibles à partir d'ECLIPSE.

Exercice 4.2 (Maven)
Dans cet exercice, il pourra être nécessaire de configurer le proxy (`http://wwwcache.uvsq.fr:3128`) pour accéder au dépôt Maven.

1. Créez un nouveau projet Maven en sélectionnant l'archétype `maven-archetype-quickstart` et en définissant le groupe à `ini1.toi.<login>` et l'artéfact à `dessin` ;
2. Importez l'archive `Dessin.zip` (à partir d'e-campus) dans le répertoire des sources du projet ;
3. Assurez-vous que le projet respecte les conventions Maven pour l'arborescence des répertoires ;
4. Assurez-vous que le JRE utilisé par le projet est un JDK 1.7 ;
5. Dans le `pom.xml`, ajoutez la dépendance vers la librairie de journalisation ; Vous ne devez pas télécharger manuellement la bibliothèque en question ;
6. En consultant la documentation du plugin Maven Compiler Plugin, éditez le `pom.xml` pour que le compilateur utilise une version 1.7 des sources ;
7. Exécutez (*Run As...*) Maven pour générer un jar du projet ;
8. Modifiez le `pom.xml` pour fabriquer les rapports pour `Javadoc`, `checkstyle`, et `findbugs` ;
9. Générez le site du projet ;
10. En utilisant le plugin `assembly`, générez une archive du projet contenant ses dépendances.

Chapitre 5

Assertions et tests

CONTRAINTES
— Utilisation de l'environnement de développement ECLIPSE.

Exercice 5.1 (Assertions)
On veut implémenter une classe représentant un compte bancaire. Les opérations que devra supporter le compte sont :

— l'initialisation avec un solde initial,
— la consultation du solde,
— le crédit,
— le débit et,
— le virement vers un autre compte.

La réalisation devra tenir compte des contraintes suivantes :

— un compte n'est jamais à découvert,
— seules des sommes positives peuvent être passées en paramètre des opérations,
— l'invocation du débit ou du virement ne peut se faire qu'avec une somme inférieure au solde du compte concerné.

Pour cet exercice, vous utiliserez de façon appropriée les assertions du langage Java. L'implémentation des contraintes ne fera donc pas appel à la gestion d'erreurs.

1. Créez la classe ainsi que les squelettes de chaque méthode
2. Pour chaque opération, précisez les pré et post-conditions et insérez-les dans le code
3. Ecrivez une méthode `invariant` vérifiant l'invariant du compte et ajouter les appels nécessaires dans le code
4. Implémentez chaque méthode

Exercice 5.2 (Tests unitaire/TDD)
Pour cette exercice, vous reprendrez les spécifications du compte bancaire de l'exercice précédent.

Vous utiliserez une approche pilotée par les tests pour la réalisation, i.e. écrivez toujours un test avant le code et pensez au refactoring. La prise en compte des contraintes se fera donc ici par de la gestion d'erreurs.

1. Réalisez la classe compte

Chapitre 6

Débogage et profilage

CONTRAINTES
— Utilisation de l'environnement de développement ECLIPSE.

Exercice 6.1 (Débogage)
Pour cet exercice, vous utiliserez le débogueur d'*Eclipse*. L'exemple utilisé est issu du tutoriel de Cay Horstmann. La classe `WordAnalyser` permet d'analyser un mot en invoquant diverses méthodes. Cependant, cette classe contient plusieurs bogues.

Récupérez le fichier WordAnalyzer.java ainsi que les programmes de test (1, 2, 3). Chaque programme de test vérifie le fonctionnement d'une méthode.

1. Créez un projet Eclipse pour la classe `WordAnalyser`
2. Créez la classe de test JUnit `TestWordAnalyser` qui contiendra les tests de `WordAnalyser`
3. Transformez chaque test des programmes de test en test JUnit
4. Si un test échoue, déboguez le programme pour identifier et corriger le problème

Chapitre 7

Synthèse

CONTRAINTES
— Utilisation de l'environnement de développement ECLIPSE.

L'objectif de ce TD est de concevoir et implémenter une bibliothèque pour manipuler une carte formée de cases carrées. Cette bibliothèque pourra, par exemple, servir de base pour un jeu de wargame.

Le langage de programmation utilisé est le **langage Java**. Les outils de développement sont intégrés à l'**IDE Eclipse**. Les plugins suivants sont déployés sous Eclipse :

— m2eclipse pour maven,
— subversive pour subversion,
— EGit pour git,
— eclipse-cs pour checkstyle,
— pmd-eclipse pour PMD,
— findbugs, et
— moreunit pour améliorer le support des tests unitaires.

Une archive d'eclipse intégrant tous ces plugins est téléchargeable pour les systèmes Linux 32 bits (`http://www.prism.uvsq.fr/~hal/EclipseJunoLinux32.zip`) ou 64 bits (`http://www.prism.uvsq.fr/~hal/EclipseJunoLinux64.zip`).

Pour ce TD, vous respecterez une **approche pilotée par les tests** (test, implémentation, refactoring).

7.1 Paramétrer l'IDE

Dans cette section, vous vérifierez le paramétrage d'eclipse (menu « Window/Preferences ») pour les points suivants :

1. la version du JRE Java utilisée par eclipse (page « Java »),
2. le *build* automatique (page « General »),
3. les répertoires des sources (`src/main/java`) et des binaires (`target/classes`) (page « Java »),
4. l'option *Report problem as you type* est cochée (page « Java »),
5. la version du compilateur correspond au JRE sélectionné (page « Java »),
6. le paramétrage de *MoreUnit* (cf. `http://moreunit.sourceforge.net/#documentation/gettingstarted`)

7.2 Créer le projet

Cette étape consistera à créer un projet maven sous eclipse, à le configurer avec les bibliothèques utilisées par la suite et enfin à l'importer dans un dépôt.

1. Créer un projet maven (*Maven Project*) basé sur l'archétype *mavenarchetypequickstart*. Vous définirez *group id* à `polytech.toi` et *artifact id* à *maplib*.
2. Modifier le POM pour utiliser la version 1.6 de Java. Exécuter le *build* maven du projet.

3. Ajouter dans le POM la dépendance avec JUnit 4.10. Adapter la classe de test pour JUnit4. Vérifier que les tests se lancent correctement (CTRL+r).
4. Ajouter dans le POM une dépendance vers la bibliothèque Google Guava. Ajouter la classe `Preconditions` dans les favoris pour les *import static* (cf. Working With Static Imports in Eclipse). Compléter la classe `App` avec un exemple de précondition et exécuter le programme (CTRL+F11) pour en vérifier le bon fonctionnement.
5. Expérimenter l'activation de *Checkstyle*, *Findbugs* et *PMD* dans l'IDE (pas dans le POM).
6. Importer votre projet au choix dans un dépôt *subversion* ou *git* (cf. menu *Team/Share Project...*. Pour terminer l'importation, il faudra valider les changements.

7.3 Le modèle

L'idée est de permettre la manipulation d'une carte formée de cases (carrées). Chaque case possède un *type de terrain* (plaine, colline, ...) et peut accueillir une *unité* (fantassin, char, ...).

La bibliothèque devra permettre les manipulation suivantes :
— création d'une carte d'une taille donnée,
— définition du type d'une case,
— récupération du type d'une case,
— installation d'une unité sur la carte,
— récupération de la position d'une unité,
— récupération de l'unité se trouvant sur une case,
— déplacement d'une unité sur une case adjacente.

Le module `polytech.tod.maplib.modele` contiendra les classes du modèle de la bibliothèque. Il comportera également des classes de test JUnit4. Les classes `App` et `AppTest` peuvent maintenant être supprimées du projet.

1. Lister quelques tests permettant d'aboutir aux fonctionnalités ci-dessus.
2. Pour le premier test de la liste,
   (a) implémenter le test (*New JUnit Test Case*),
   (b) implémenter le minimum pour que le test compile (utilisez les *Quick fix* CTRL+1),
   (c) vérifier que le test échoue (CTRL+R),
   (d) implémenter le minimum pour que le test passe (CTRL+J pour passer du test au code),
   (e) vérifier que le test passe,
   (f) appliquer le refactoring sur les tests et sur l'implémentation si nécessaire,
   (g) vérifier que le test passe,
   (h) valider les changements sous Subversion.
3. Mettre à jour la liste des tests et recommencer en 2)

7.4 Evolution du modèle

On veut ajouter la prise en compte du type de terrain dans le déplacement d'unité. Chaque type de terrain nécessite un certain coût de déplacement pour être franchi. Une unité doit pouvoir se rendre d'une case à une autre en empruntant le chemin de coût minimum.

La bibliothèque devra permettre les manipulation suivantes :
— découvrir le chemin de coût minimum pour atteindre une destination,
— déplacer une unité en empruntant ce chemin.

1. Ajouter ces fonctionnalités en suivant la même approche que dans la section précédente.

7.5 Ressources

— Java™ Platform, Standard Edition 6 API Specification
— The Java Tutorials
— Développement dirigé par les tests
— TDD et productivité
— Tests unitaires automatisés avec JUnit4

Annexe A

Tables

Table des exercices

Table des figures