COMPLÉMENT JAVA

PUBLIC CONCERNÉ : formation initiale, 2^e année.

NOM DE L'AUTEUR : V. Thomas

DATE 2012/2013

UNIVERSITÉ NANCY 2
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE
2 ter boulevard Charlemagne
CS 5227
54052 NANCY cedex

Tél : 03.83.91.31.31
Fax : 03.83.28.13.33
http://www.iuta.univ-nancy2.fr

Contents

I Eclipse comme environnement de développement **2**

1 Présentation générale d'Eclipse **3**
1.1 Présentation . . . 3
1.2 Installation . . . 3
1.3 Philosophie d'Eclipse . . . 3
1.4 Interface d'Eclipse . . . 4
1.4.1 Perspectives . . . 4
1.4.2 Vues . . . 5
1.5 Utilisation d'Eclipse . . . 5
1.5.1 Créer un projet . . . 5
1.5.2 Créer des entités . . . 5
1.5.3 Compilation . . . 5
1.5.4 Exécution . . . 6
1.5.5 Fermeture des projets . . . 6
1.6 Éditeur de code . . . 6
1.6.1 Complétion automatique . . . 7
1.6.2 Correction automatique des erreurs . . . 7
1.6.3 Génération de code . . . 7
1.6.4 Modification de code . . . 7
1.6.5 Le *refactoring* . . . 8
1.6.6 Mise en garde . . . 8

2 Debugger avec Eclipse **9**
2.1 Définition . . . 9
2.1.1 Debugger . . . 9
2.1.2 Utilisation . . . 9
2.2 Debugger sous Eclipse . . . 9
2.2.1 Accéder au debugger . . . 9
2.2.2 Description de la perspective debugger . . . 9
2.2.3 Lancer un programme avec le debugger . . . 10
2.3 Outils . . . 10
2.3.1 Les points d'arrêts . . . 10
2.3.2 Exécution pas à pas . . . 11
2.3.3 Le suivi d'expressions . . . 11
2.3.4 L'affichage du contenu de la mémoire . . . 12
2.3.5 Bilan . . . 12

3 Packages **14**
3.1 Présentation . . . 14
3.1.1 Définition d'un package . . . 14
3.1.2 Objectif des packages . . . 14
3.2 Construction d'un package . . . 15
3.2.1 Le mot-clef package . . . 15
3.2.2 Structure d'un package . . . 15
3.2.3 Visibilité d'un package . . . 15
3.2.4 Localisation des packages . . . 16
3.3 Notion de classpath . . . 16

3.3.1 Principe de structuration d'une application JAVA . . . 16
3.3.2 Définition du classpath . . . 17
3.3.3 Utilisation du classpath sans package . . . 17
3.3.4 Classpath et packages de base JAVA . . . 17
3.3.5 Classpath et vos packages . . . 18
3.3.6 Classpath sous Eclipse . . . 18
3.4 Fichiers .jar . . . 18
3.4.1 Structure d'un fichier .jar . . . 18
3.4.2 Création d'un fichier .jar . . . 19
3.4.3 Spécification d'un point d'entrée . . . 19
3.4.4 Utilisation d'un fichier .jar . . . 19
3.5 Utilisation d'Eclipse . . . 20
3.6 Dernière remarque . . . 20

II Tests et vérifications **21**

4 Réflexions sur les erreurs **22**
4.1 Qu'est ce qu'une erreur ? . . . 22
4.1.1 Définition . . . 22
4.1.2 Plusieurs types d'erreurs . . . 22
4.1.3 Quelles sont les conséquences d'une erreur ? . . . 22
4.2 Compromis du traitement des erreurs . . . 23
4.2.1 A l'exécution . . . 23
4.2.2 Pendant la phase de test . . . 23
4.2.3 Paradoxe . . . 23
4.3 Comment prévenir correctement des erreurs . . . 24
4.3.1 A la conception . . . 24
4.3.2 De manière online - pendant l'exécution de l'application . . . 24
4.3.3 De manière offline - au cours d'une phase de test . . . 24
4.4 Bilan . . . 25

5 Assertions **26**
5.1 Présentation . . . 26
5.1.1 L'objectif des assertions . . . 26
5.1.2 Principe . . . 26
5.1.3 Définition . . . 26
5.2 Utilisation . . . 27
5.2.1 Déclarer une assertion . . . 27
5.2.2 Fonctionnement d'une assertion . . . 27
5.2.3 Exécuter avec les assertions . . . 27
5.2.4 Exemples . . . 28
5.3 Où mettre des assertions . . . 28
5.3.1 Documenter du code . . . 28
5.3.2 Pre-conditions d'une méthode privée . . . 29
5.3.3 Post-conditions . . . 29
5.3.4 Invariants de classe . . . 30
5.3.5 Invariants de flux . . . 30
5.3.6 Invariants logiques . . . 31
5.4 Bonne pratique des assertions . . . 31
5.4.1 Pas d'effet de bord . . . 31

5.4.2 Garder à l'esprit qu'elles sont debrayables . . . 31
5.4.3 Assertion versus Exception . . . 32
5.5 Conclusion . . . 32

6 Tests unitaires **34**
6.1 Présentation des tests unitaires . . . 34
6.1.1 Objectif . . . 34
6.1.2 Définition . . . 34
6.1.3 Démarche . . . 34
6.1.4 Spécification des tests . . . 35
6.1.5 Mise en oeuvre des tests unitaires . . . 36
6.2 Junit . . . 36
6.2.1 Accés à JUnit . . . 36
6.2.2 Principe . . . 37
6.3 Mise en oeuvre . . . 37
6.3.1 Squelette de la classe applicative . . . 37
6.3.2 Créer un test . . . 38
6.3.3 Créer une batterie de tests . . . 39
6.3.4 Lancer les tests . . . 40
6.4 Compléments à JUnit . . . 41
6.4.1 Preparation et liberation des ressources . . . 41
6.4.2 Tester la levée d'exceptions . . . 42
6.4.3 Tester une classe générique . . . 42
6.4.4 Resultats de test . . . 42
6.5 Tests unitaires sous eclipse . . . 42
6.5.1 Ajouter Junit à Eclipse . . . 42
6.5.2 Créer un test . . . 43
6.5.3 Creer une batterie de tests . . . 43
6.5.4 Lancer les tests . . . 43
6.6 JUnit 4 . . . 44
6.6.1 Méthode de test . . . 44
6.6.2 Interception d'exception . . . 44
6.6.3 Ignorer des tests . . . 45
6.6.4 Nouveaux tests possibles . . . 45
6.7 Bilan . . . 45
6.7.1 Intérêts . . . 45
6.7.2 Resolution de probleme . . . 45
6.7.3 Bonne Pratique . . . 46

7 Bilan sur les tests **47**
7.1 Distinction assertion, test unitaire . . . 47
7.2 Utilité des tests . . . 47

III Outils pour la conception d'applications **48**

8 Approche Modèle - Vue - Contrôleur **49**
8.1 Problème lié aux Interfaces graphiques . . . 49
8.2 Modèle Vue Contrôleur . . . 49
8.2.1 Principe . . . 49
8.2.2 Les différents blocs . . . 49

8.2.3 Diagramme d'interaction . . . 51
8.3 Observer / Observable . . . 51
8.3.1 le design Observeur-Observé . . . 51
8.3.2 Les classes Observer et Observable . . . 52
8.3.3 La classe Observable . . . 52
8.3.4 L'interface Observer . . . 53
8.3.5 Mise en place d'un MVC avec Observer et Observable . . . 53
8.4 Exemple . . . 53
8.4.1 Description des éléments MVC . . . 53
8.4.2 Mise en place du modèle . . . 54
8.4.3 Mise en place de la Vue Textuelle . . . 54
8.4.4 Mise en place de la vue Graphique . . . 55
8.4.5 Mise en place du contrôleur textuel . . . 55
8.4.6 Mise en place du contrôleur graphique . . . 57
8.4.7 Agencement des blocs . . . 58
8.5 Synthèse de l'approche MVC . . . 59
8.5.1 Démarche . . . 59
8.5.2 Intérêt . . . 59
8.6 Une autre approche pour MVC . . . 59
8.7 Ouverture : Les patrons de conception . . . 60

9 Généricité **61**
9.1 Vocabulaire . . . 61
9.2 Définition . . . 61
9.3 Utilisation . . . 61
9.3.1 Déclaration . . . 61
9.3.2 Création d'instance . . . 64
9.3.3 Manipulation . . . 64
9.4 Variables de type . . . 65
9.4.1 Récursivité . . . 65
9.4.2 Variables de type contraintes . . . 65
9.4.3 Classe générique avec plusieurs variables de type . . . 66
9.5 Sous-typage . . . 67
9.5.1 Sous-typage entre classes génériques . . . 67
9.5.2 Sous-typage entre classes paramétrées . . . 67
9.5.3 les jokers . . . 67
9.6 Raw types . . . 68
9.7 'Template method' pattern . . . 68
9.7.1 Présentation . . . 68
9.7.2 Exemple sur ArrayList . . . 69
9.7.3 Conséquence . . . 70

IV Conclusion et Perspectives **71**

Introduction

Ce module a pour objectif de présenter des outils et technologies utilisables dans le cadre de projets JAVA.

Il est décomposé en trois grandes parties :

- la partie **'Eclipse comme environnement de développement'** présente Eclipse, un outil disponible à l'IUT et très pratique pour développer du code JAVA.
- la partie **'Tests et vérification'** présente quelques éléments pour vérifier une application et s'assurer de son bon fonctionnement :
  - les **assertions** pour suivre le comportement d'une application au cours de l'exécution et détecter l'apparition de bug
  - et les **tests unitaires** et le framework `JUnit` qui permettent d'automatiser la phase de test d'une application.
- la partie **'Outils pour la conception'** présente quelques éléments importants pour développer des applications complexes :
  - les **packages** pour développer du code structuré,
  - la **généricité** pour développer des classes paramétrées (templates),
  - et **l'approche MVC** qui propose une architecture flexible et modulaire pour gérer l'interface graphique d'une application.

Part I

Eclipse comme environnement de développement

1 Présentation générale d'Eclipse

Pré-requis

Avant d'aborder cette partie, vous devez savoir

- compiler des classes à partir d'une console
- exécuter des classes à partir d'une console
- lancer des programmes avec des bibliothèques (`classpath`)

Contenu

Cette section présentera

- le fonctionnement d'Eclipse
- la notion de vue et de perspective
- la manière d'utiliser au mieux Eclipse

1.1 Présentation

Eclipse est un **environnement de développement intégré** (`IDE`), c'est à dire un programme qui regroupe

- un éditeur de texte pour écrire et modifier du code source
- un compilateur pour produire les applications
- des outils de fabrication automatique (structure pré-remplie pour définir une classe, des *getters*, des *setters*, ...)
- et souvent un debugger pour pouvoir suivre l'exécution du programme pas à pas et détecter des bugs (cf partie 2)

De plus, Eclipse est une plate forme modulaire à laquelle il est possible de rajouter de nombreux plug-ins proposant de nouvelles fonctionnalités (modélisation UML, accès aux bases de données, XML, ...). Ainsi, Eclipse a été au départ conçu pour du développement JAVA mais de nombreux plug-ins permettent d'utiliser Eclipse comme environnement de développement pour de nombreux autres langages (PHP, C++, Python, ...).

1.2 Installation

Eclipse est installé sur les différentes machines de l'IUT.

Eclipse est gratuit et opensource. Il se trouve facilement sur internet. La page `http://www.eclipse.org/` regroupe toute l'information liée à Eclipse (plateforme et plugins) ainsi qu'une page de téléchargement où vous pourrez obtenir la dernière version.

1.3 Philosophie d'Eclipse

Eclipse fonctionne par **projet**. Un projet correspond à une application à part entière constituée

- d'un ensemble de packages
- d'un ensemble de classes/interfaces dans ces packages
- d'un ensemble de ressources (fichiers sonores, images, fichiers textes, ...)

Les fichiers associés aux différents projets sont stockés dans un espace de travail (*workspace*). A l'IUT, sous Windows, ce workspace se trouve par défaut dans le répertoire `u:\workspace`, mais il est possible de changer la localisation du workspace au lancement d'Eclipse.

Chaque projet correspond à un sous-répertoire de l'espace de travail.

1.4 Interface d'Eclipse

Eclipse est basé sur des **perspectives**. Une perspective correspond à une utilisation particulière d'Eclipse et présente le code selon certains aspects.

Une perspective contient plusieurs **vues**, chaque vue présentant un aspect particulier de l'application.

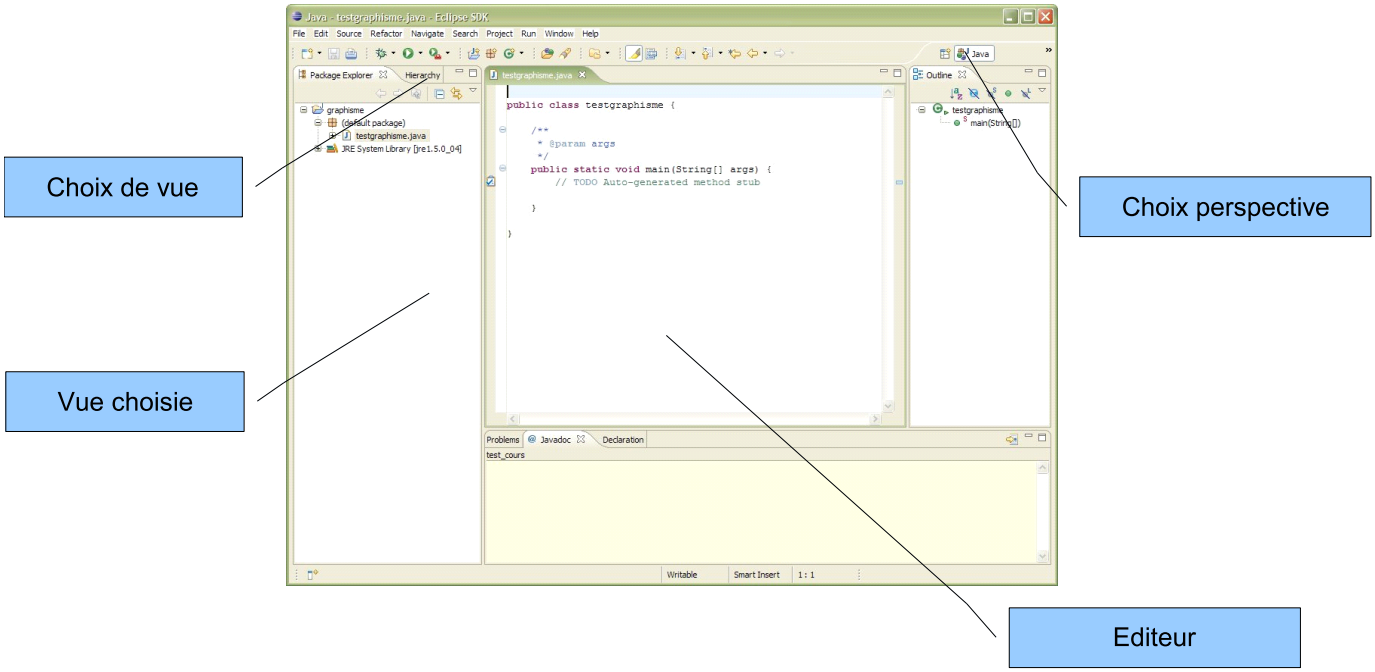


Figure 1: Interface générale d'Eclipse

1.4.1 Perspectives

Eclipse propose les perspectives suivantes:

- **Java** une perspective utilisée pour développer du code JAVA, c'est celle que vous serez amenés à utiliser le plus fréquemment
- **Navigation JAVA** une perspective utilisée pour parcourir rapidement les différentes classes d'un projet
- **Hiérarchie JAVA** une perspective utilisée pour parcourir les classes en fonction de l'arbre de hiérarchie entre les classes (relation d'héritage)
- **Debuggage** une perspective utilisée pour suivre les variables en cours d'exécution de l'application et localiser des bugs (cf chapitre 2 sur le debugger)
- **Ressource** une perspective utilisée pour afficher les ressources associées au projet (fichiers)

Les autres perspectives ne seront pas abordées, Il reste néanmoins

- **Développement de Plug-ins** une perspective utilisée pour développer des plugins d'Eclipse
- **Synchronisation** une perspective utilisée pour travailler à plusieurs
- **CVS** une perspective utilisée pour le travail collaboratif.

1.4.2 Vues

Eclipse propose de nombreuses vues associées aux différentes perspectives. Parmi les vues possibles, les suivantes peuvent se révéler intéressantes pour vos projets

- **Package explorer** Cette vue permet d'afficher l'ensemble des classes, méthodes, attributs d'un projet et de pouvoir accéder directement au code de la classe ou de la méthode en cliquant sur le nom
- **Hierarchy** Cette vue permet d'afficher la hiérarchie d'une classe et de naviguer dans les sous classes et les super classes
- **Javadoc** Cette vue affiche la javadoc de la méthode surlignée dans l'éditeur
- **Déclaration** Cette vue affiche la déclaration de la méthode surlignée dans l'éditeur

Il est possible d'afficher une vue spécifique en utilisant le menu `'Window --> Show view'`.

1.5 Utilisation d'Eclipse

Développer une application sous Eclipse se fait en plusieurs étapes

1. Créer un projet
2. Remplir le projet avec des entités (packages, classes, interfaces, ...)
3. Compiler l'application
4. Exécuter l'application

1.5.1 Créer un projet

Pour créer un projet, il suffit de sélectionner le menu `'File --> New --> Project'` ou de sélectionner l'icône `'new project'`.

Il suffit ensuite de sélectionner projet JAVA parmi tous les types de projets proposés. Une fenêtre présentant les différentes caractéristiques du projet s'ouvre. Il est possible de spécifier les bibliothèques à inclure dans le projet (cf chapitre 3).

1.5.2 Créer des entités

Une fois qu'un projet est défini, il suffit de créer des classes, des interfaces et des packages qui sont automatiquement ajoutés au projet. Pour créer une entité, il suffit de sélectionner le menu `'File --> New --> Entite_a_créer'`.

Au moment de la création d'une classe, il est possible de lui spécifier différentes caractéristiques (comme le fait que la classe puisse être exécutable ou non).

Une fois l'entité validée, Eclipse crée le fichier et écrit le squelette de la classe (avec éventuellement un `main` si la classe a été spécifiée comme exécutable).

1.5.3 Compilation

Eclipse effectue une compilation dès qu'une modification du code source est faite. Vous pouvez donc suivre à tout moment les erreurs générées à la compilation.

Ces erreurs sont surlignées en rouge dans l'éditeur. Toutes les erreurs associées aux projets en cours sont répertoriées dans la vue `'Problems'`. En cliquant sur l'erreur, on accède directement dans l'éditeur à l'endroit où l'erreur a été détectée.

Si aucune erreur n'a été détectée, le code est correctement compilé et est prêt à être exécuté.

1.5.4 Exécution

Pour exécuter un projet, le plus simple consiste à effectuer un clic droit sur la classe constituant le point d'entrée du projet et choisir le menu `'Run as --> JAVA application'`.

Une fenêtre s'ouvre et permet de choisir diverses options (y compris les options transmises à la JVM)

Au lancement de l'application, la vue `'Console'` permet de suivre ce qui se passe à l'écran.

1.5.5 Fermeture des projets

Dés que vous ne travaillez plus sur un projet, pensez à le fermer. Dans le cas contraire, la vue `'Problems'` présentera toutes les erreurs liées à tous les projets ouverts et peut vite devenir illisible.

1.6 Éditeur de code

L'éditeur de code permet de modifier du code source. Il est constitué de deux éléments

- le code source lui même
- une ligne d'icônes sur la gauche répertoriant les différentes erreurs et avertissements pour y accéder plus rapidement
- une ligne sur la droite présentant une vue d'ensemble du fichier.

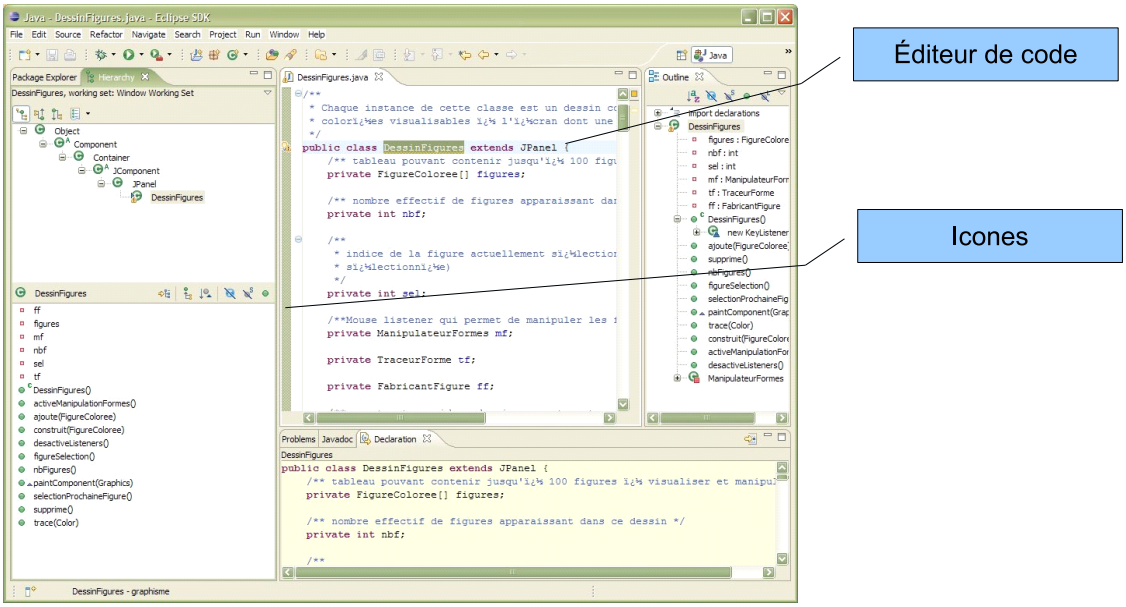


Figure 2: Éditeur de code

L'éditeur offre de nombreuses fonctionnalités. Ce document en aborde un certain nombre.

1.6.1 Complétion automatique

Il est possible de demander à Eclipse des suggestions pour compléter un début de code. Il suffit pour cela de commencer à écrire et d'appuyer sur les touches **CTRL + espace**. Eclipse propose alors plusieurs possibilités pour compléter le mot.

> **ATTENTION** Il s'agit d'un des outils les plus pratiques d'Eclipse qui font qu'il est très agréable de l'utiliser. Mais une utilisation abusive de cet outil peut être très dangereuse. Évitez à tout prix de sélectionner les suggestions d'Eclipse sans être sûr de ce que vous faites.

La complétion automatique fonctionne

- sur les méthodes : il suffit d'écrire le début de la méthode et Eclipse complète en fournissant les noms des méthodes compatibles;
- sur les classes anonymes : il suffit d'écrire le début de la classe Anonyme et Eclipse complète en ajoutant les méthodes à redéfinir;
- sur certaines structures : il suffit d'écrire `'if'` et Eclipse construit un squelette correspondant à une condition;
- sur certaines macro prédéfinies : il suffit d'écrire `'sysout'` et d'appuyer sur control+espace et Eclipse remplace la chaîne par `System.out.println`.

Certaines macros (`'sysout'` par exemple) et certaines structures (`'if'`, ...) sont définies et modifiables dans les options de l'éditeur. Ces options sont accessibles par le menu `'Window --> Preferences --> JAVA --> Editor'`.

1.6.2 Correction automatique des erreurs

Lorsqu'Eclipse détecte une erreur de compilation, la colonne à gauche de l'éditeur présente un panneau rouge. Lorsqu'on clique sur ce panneau, Eclipse propose certaines réponses classiques pour résoudre le problème.

> **ATTENTION (bis)** Il s'agit à nouveau d'un outil très pratique, mais soyez bien sûr de comprendre ce qu'Eclipse vous propose avant de sélectionner une réponse.

1.6.3 Génération de code

Eclipse peut aussi générer automatiquement des morceaux de code comme

- les importations : sélectionner la classe à ajouter et appuyer sur **control+M**.
- les constructeurs : dans le menu `'Source --> Generate constructor using fields'`
- les *getter* et *setter* : dans le menu `'Source --> Generate getters et setters'`

1.6.4 Modification de code

Eclipse propose aussi comme fonctionnalité

- le formatage de code : sélectionner le texte à formater et sélectionner le menu `'Source --> format'`. Les règles de formatage peuvent être modifiées dans le menu `'Window --> preferences --> JAVA'`
- la mise en commentaire d'une partie de code : sélectionner la portion de code et sélectionner le menu `'Source --> Commentaires'`
- les blocs try/catch : sélectionner le bloc à protéger et choisir le menu `'Source --> surround with --> bloc try/catch'`

1.6.5 Le *refactoring*

Eclipse permet aussi de renommer automatiquement une classe, un attribut ou une méthode. Dans ce cas, ce n'est pas seulement la déclaration de la classe ou de la méthode qui est modifiée mais aussi toutes les références contenues dans tous les fichiers du projet.

On accède à cette option par le menu `'Refactor --> Rename'`.

1.6.6 Mise en garde

L'utilisation d'Eclipse permet d'automatiser un certain nombre d'opérations et cela constitue un de ses intérêts. Méfiez vous cependant de l'utilisation abusive de ces fonctionnalités qui peuvent conduire à de nombreuses erreurs.

Par exemple, la complétion automatique des noms de méthodes est très pratique, mais il faut que vous connaissiez à l'avance la méthode à rechercher. En choisir une en fonction de son nom sans connaître l'API peut conduire à des désastres.

Objectif pédagogique

A l'issue de cette section, vous devez savoir

- comment construire un projet sous Eclipse
- comment naviguer entre les classes écrites sous Eclipse
- comment exécuter un projet sous Eclipse

Références externes

Si vous souhaitez plus d'informations sur Eclipse, voici quelques références

- le site web d'Eclipse
  `http://www.eclipse.org/`
- le document très complet 'développons en JAVA avec Eclipse' de JM Doudoux
  `http://www.jmdoudoux.fr/accueil_java.htm#dejae`

2 Debugger avec Eclipse

Contenu

Dans cette partie, on expliquera

- ce qu'est un debugger et à quoi il sert
- comment fonctionne le debugger sous Eclipse
- comment mettre des points d'arrêts (avec conditions éventuelles)
- comment faire du suivi de variable
- comment parcourir le contenu de la mémoire

2.1 Définition

2.1.1 Debugger

Un debugger est un programme qui permet de suivre l'exécution d'un autre programme, de l'arrêter à certains endroits, de vérifier la valeur de certaines expressions et de sonder l'état de la mémoire.

Il s'agit d'un programme très pratique pour permettre au concepteur d'une application de détecter des erreurs et de les corriger

2.1.2 Utilisation

Le programme à debugger s'exécute par l'intermédiaire du debugger. Le debugger permet alors de contrôler le programme

- il est possible de mettre en pause le programme à tout instant
- il est possible de suivre le contenu de certaines variables et l'état complet de la mémoire
- lorsque le programme à debugger est mis en pause, le debugger spécifie l'endroit du code source où le programme a été interrompu.

2.2 Debugger sous Eclipse

2.2.1 Accéder au debugger

Eclipse intègre un debugger. Pour accéder au débugger, il suffit simplement de sélectionner la perspective 'debugger' représentée par l'icône Debug

Figure 3: Icone debug

2.2.2 Description de la perspective debugger

La perspective debugger propose par défaut plusieurs vues

- le cadre en haut à gauche présente la Vue `'Debug'` qui affiche la liste des processus JAVA en cours d'exécution (cf partie 2.3.2). Il est possible de sélectionner un processus, de le mettre en pause et d'effectuer une exécution pas à pas.

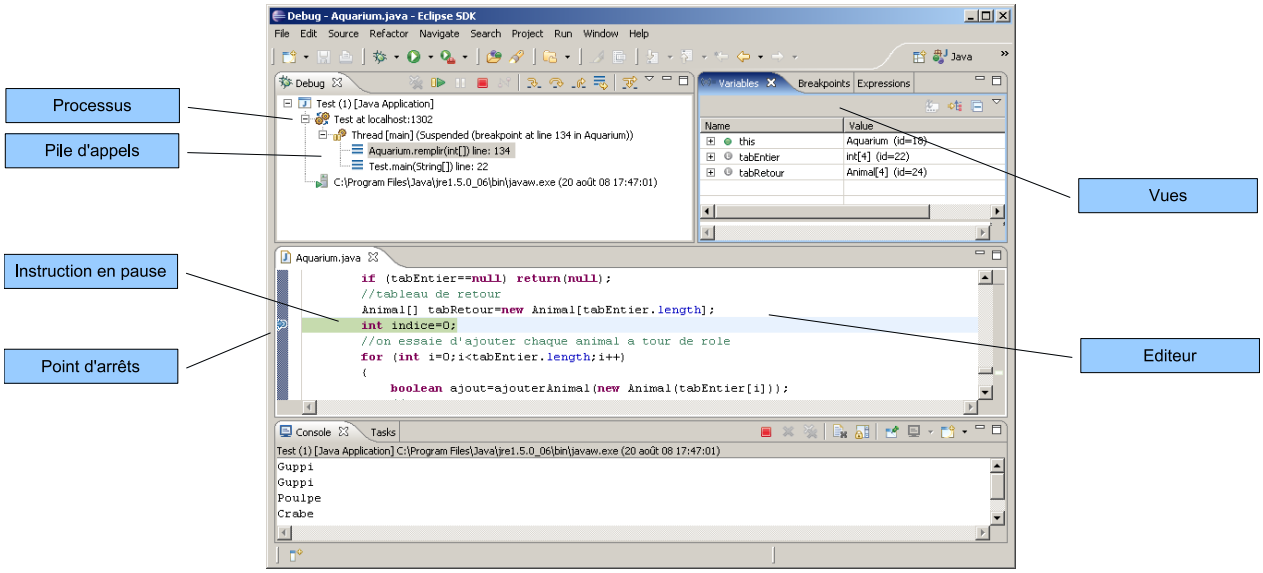


Figure 4: Perspective Debug

- le cadre en haut à droite présente différentes vues : la vue `'Breakpoints'` pour la gestion des points d'arrêts (cf partie 2.3.1), la vue `'Variable'` pour le suivi du contenu de la mémoire (cf partie 2.3.4) et la vue `'Expressions'` pour le suivi des expressions (cf partie 2.3.3)
- le cadre au milieu présente l'éditeur. Les points d'arrêts sont visibles sur le coté gauche (dans la ligne d'icône)
- le cadre en bas présente la vue `'Console'`.

2.2.3 Lancer un programme avec le debugger

Pour lancer un programme java avec le debogger, il suffit de cliquer sur l'icône `'Debug'`. C'est alors le dernier programme JAVA qui a été lancé qui est exécuté.

Pour plus d'options, il faut sélectionner une classe et choisir le menu `'debug as --> debug'` après un clic droit.

2.3 Outils

Le debugger d'Eclipse propose plusieurs outils classiques pour debugger un programme

2.3.1 Les points d'arrêts

Il est possible d'ajouter des points d'arrêts dans le code. Dés que le debbugger arrive sur un point d'arrêt, il met automatiquement le programme en pause.

Cela permet au concepteur du programme d'arriver sur l'endroit du code qui parait problématique pour analyser ce qui s'y passe en détail.

Par défaut Il suffit de double-cliquer sur la ligne d'icône de l'éditeur pour poser un point d'arrêt (représenté par une puce bleue) à la ligne correspondante.

Sous condition Il est possible de rajouter des conditions d'arrêts sur le point d'arrêt (bouton droit sur un point d'arrêt et sélectionner le menu `propriétés`)

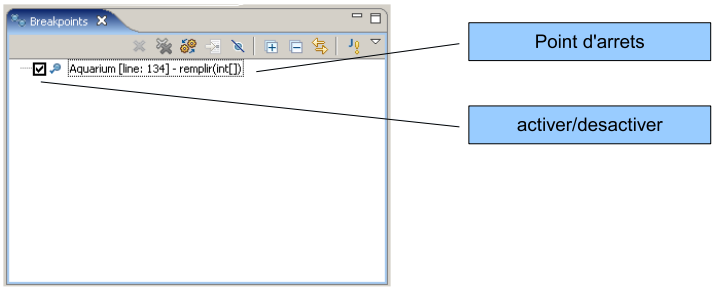


Figure 5: Vue Breakpoints: La liste des points d'arrêts

- soit par rapport à une condition booléenne
- soit par rapport à un nombre de passages sur le point d'arrêt - ce qui peut être très utile pour accéder à une certaine itération d'une boucle

2.3.2 Exécution pas à pas

Quand l'application est en pause (soit à cause d'un point d'arrêt soit manuellement) il est possible d'exécuter l'application pas à pas en utilisant les icônes de la vue `'Debug'`.

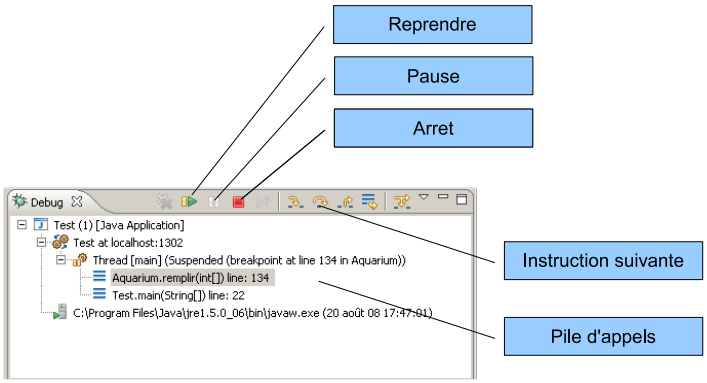


Figure 6: Vue Debug: Contrôle de l'exécution

2.3.3 Le suivi d'expressions

La vue `'Expressions'` permet de suivre l'évolution d'expressions préalablement écrites. Une expression peut être une variable ou une opération impliquant des variables et des constantes.

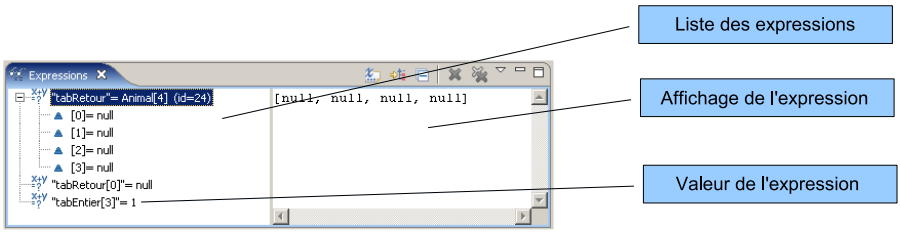


Figure 7: Vue Expression: Suivi d'expressions à l'exécution

Suivi d'expression Le suivi d'expression est automatique. A chaque pause de l'application java à debugger, la valeur des expressions est mise à jour

Ajouter une expression Il suffit de faire un clic droit sur la vue des expressions et de suivre les menus.

2.3.4 L'affichage du contenu de la mémoire

La vue `'Variable'` permet de vérifier le contenu de la mémoire. On accède à la pile et il est possible de suivre les références en cliquant sur le + pour connaître le contenu de la variable.

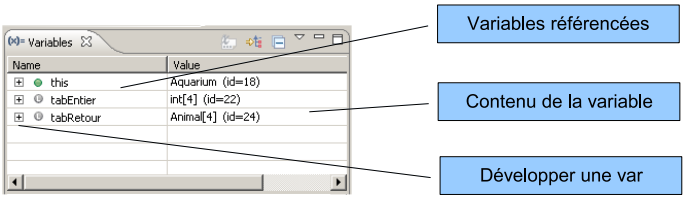


Figure 8: Vue Variable: contenu de la mémoire

Il est par exemple possible d'accéder au contenu d'un tableau ou d'une `ArrayList`.

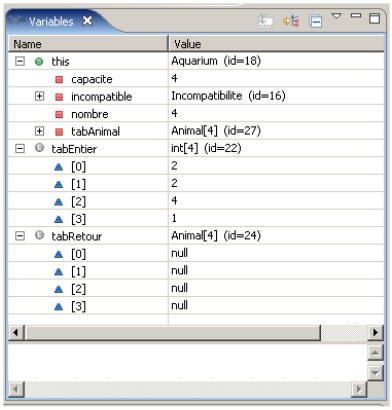


Figure 9: Vue Variable: contenu plus développé

Le contenu de la variable s'exprime de la forme suivante:

- si c'est un **type primitif**, la valeur est affichée directement sous la forme: `valeur` .
- si c'est un **type objet**, le type objet suivi de la référence est affiché sous la forme: `type(id=ref)`
  - où `type` désigne le type de l'objet
  - et `ref` est un nombre désignant la référence de l'objet

2.3.5 Bilan

Le debugger est un outil très puissant qui permet d'éviter d'insérer des `println` dans un code (pratique désastreuse).

Il permet de suivre l'évolution d'un programme, de ses variables et du contenu de la mémoire.

Il est à noter que seule l'utilisation fréquente d'un debugger permet d'en tirer convenablement parti. Il est nécessaire d'expérimenter et de se familiariser avec ces applications pour pouvoir réellement en profiter. Mais cela constitue ensuite un réel plus lorsque vous serez confrontés à un problème.

Objectif pédagogique

A l'issue de cette section vous devrez savoir

- comment exécuter un projet en mode debug sous Eclipse
- comment poser des points d'arrêts
- comment vérifier le contenu de la mémoire et des différentes variables

References

- le document très complet 'développons en JAVA avec Eclipse' de JM Doudoux chapitre 8 `http://www.jmdoudoux.fr/accueil_java.htm#dejae`

3 Packages

Contenu

Dans cette partie, on expliquera

- ce qu'est un package
- comment définir un package dans l'absolu / sous Eclipse
- comment faire un fichier jar

3.1 Présentation

3.1.1 Définition d'un package

Un package est une unité regroupant des fichiers `.class`. Un package est censé constituer un bloc proposant une certaine fonctionnalité - exemple affichage - avec toutes les classes nécessaires pour son bon fonctionnement.

On utilisera au cours de ce chapitre l'exemple consistant à produire une bibliothèque pour faire des mathématiques.

Cette bibliothèque

- aura pour nom `mesMaths`
- possédera une classe `Add` permettant de faire des additions
- possédera une classe `Mult` permettant de faire des multiplications et utilisant la classe `Add`

3.1.2 Objectif des packages

L'objectif des packages est de compartimenter du code pour en faire des bibliothèques utilisables de l'extérieur.

Cela peut avoir plusieurs intérêts

- pour celui qui utilise la bibliothèque
  - lorsqu'un utilisateur utilise une bibliothèque, il n'a pas besoin de savoir comment elle est construite en interne et n'a pas à la modifier.
  - lorsqu'un créateur modifie une bibliothèque, l'utilisateur n'a pas à savoir ce qui a été modifié et n'a pas à modifier son code (sauf en cas de modification majeure lorsque les signatures des méthodes publiques ont été modifiées)
- pour celui qui écrit la bibliothèque
  - un créateur peut modifier sa bibliothèque sans savoir comment les autres utilisateurs l'utilisent.

Il est primordial de savoir convenablement utiliser les packages, d'une part parce qu'ils vous seront nécessaires lorsque vous aborderez la conception et la réalisation d'applications complexes, d'autre part parce que vous devez savoir comment facilement inclure des bibliothèques extérieures aux bibliothèques de base de JAVA.

3.2 Construction d'un package

Cette partie présente succinctement comment construire un package et comment le structurer.

3.2.1 Le mot-clef package

Un package est principalement constitué par plusieurs classes/interfaces. Pour spécifier qu'une classe appartient à un package donné, il faut utiliser le mot clef `package` en entête de la classe suivi du nom du package.

Si la classe ne contient pas de descriptif de package, elle appartient à un package anonyme par défaut (c'est comme cela que vous faites pour le moment).
Une fois que le mot clef package est spécifié en en-tête de la classe, la classe appartient alors exclusivement au package dont le nom a été donné. Cette classe ne fait alors plus partie du package par défaut et n'est plus accessible de la même manière (cf les règles de visibilité qui suivent).

La classe `Add` se déclarera donc ainsi

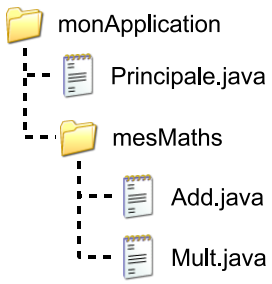
```
package mesMaths;

public class Add
{
  public static int add(int a, int b)
  {
    return(a+b);
  }
}
```

3.2.2 Structure d'un package

Pour rassembler toute l'information d'un package, les classes appartenant au même package sont situées dans le même répertoire possédant le nom du package.

Les classes `Add` et `Mult` seront déclarées dans un sous-répertoire commun nommé `mesMaths`

Pour cet exemple, l'arborescence peut etre représentée par la figure suivant :



3.2.3 Visibilité d'un package

A l'intérieur du package Toutes les classes d'un package voient les autres classes appartenant au même package.

Il est donc possible d'utiliser la classe `Add` dans la classe `Mult` sans rien spécifier.

```
package mesMaths;

public class Mult
{
  public static int mult(int a, int b)
  {
    int result=0;
    for (int i=0;i<a;i++)
    {
      result=Add.add(result,b);
    }
    return(result);
  }
}
```

De l'exterieur Par contre si l'on souhaite utiliser une classe d'un package à partir d'un autre package il faut utiliser le mot clef `import`.

- `import monPackage.*;` importe toutes les classes du package `monPackage`.
- `import monPackage.maClasse;` importe uniquement la classe `maClasse` du package `monPackage`

Une classe extérieure devra utiliser `import` si elle souhaite utiliser des classes du package `mesMaths`.

```
import mesMaths.Mult;

public class UtiliseMath {

  public static void main(String args[])
  {
    System.out.println(Mult.mult(10,10));
  }
}
```

De l'extérieur d'un package, on ne peut avoir accès qu'aux méthodes publiques et éventuellement protected (mais pas aux éléments déclarés avec un accès par défaut).

3.2.4 Localisation des packages

Pour permettre à la JVM de savoir où trouver vos packages, il est nécessaire que votre `classpath` soit correctement mis à jour. Comme la notion de classpath est extrêmement importante, la partie suivante y est entièrement consacrée.

3.3 Notion de classpath

3.3.1 Principe de structuration d'une application JAVA

Il est possible d'avoir plusieurs vision d'une même application JAVA. On distingue ainsi

- la *structure logique* d'une application qui décrit la manière dont les classes sont organisées en terme de package. Cette structure logique est le résultat des déclarations des packages.
- la *structure physique* de l'application qui décrit l'endroit où trouver les fichiers .class (localisation des répertoires).

En Java, la structure logique et la structure physique d'une application sont liées. Un package correspond à un sous-repertoire. Le *classpath* a pour objectif de faire le lien entre la structure logique (déclaration des packages) et la structure physique (localisation des fichiers).

3.3.2 Définition du classpath

Le `classpath` est un paramètre passé au lancement de la JVM définissant les chemins à partir desquels la JVM va chercher vos classes à l'exécution et à la compilation.

Lorsque vous lancez un programme java, à chaque fois qu'une classe est utilisée, la JVM va chercher dans les chemins spécifiés par le classpath le fichier `.class` correspondant. Si le fichier `.class` est introuvable, JAVA lève une Exception `ClassNotFoundException`.

3.3.3 Utilisation du classpath sans package

Jusqu'à présent, vous utilisiez des classes sans spécifier de package. Supposons que nous avons les classes Rectangle et Point dans un package par défaut stocké sous la forme suivante

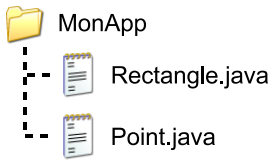


Figure 10: Fichiers Rectangle.java et Point.java

Plusieurs options sont possibles

- lancer la JVM **à partir** du répertoire `MonApp`, auquel cas, le paramètre `classpath` doit contenir la chaîne `"."` pour que la JVM cherche les classes dans le repertoire courant.
- lancer la JVM dans le répertoire parent de `MonApp`, auquel cas, le paramètre `classpath` doit contenir le **chemin relatif** `"MonApp"` pour que la JVM cherche les classes dans le sous-répertoire `MonApp`.
- lancer la JVM dans un répertoire quelconque en précisant le chemin **absolu** vers le répertoire `MonApp` comme `'c:\workspace\MonApp'`.

Il est possible de séparer les fichiers `Rectangle.java` et `Point.java` en les mettant dans des répertoires différents (Rectangle et Point par exemple), auquel cas, le paramètre classpath doit contenir le répertoire de `Rectangle.java` et le répertoire de `Point.java` pour trouver tous les fichiers. Par contre, les fichiers appartiennent toujours au package par défaut puisqu'ils sont déclarés comme tels. En conséquence, ils doivent toujours donc se trouver dans les répertoires pointés directement par le classpath.

3.3.4 Classpath et packages de base JAVA

Si vous affichez le contenu du classpath par défaut (par exemple avec la command `echo $classpath` sous windows), le paramètre classpath est complexe. Il contient en fait les packages dans lesquels

sont stockées toutes les classes de base de JAVA. C'est ce qui permet à la JVM de pouvoir retrouver toutes les bibliothèques habituelles quand vous importez des classes comme `ArrayList`, `JPanel`, etc...

3.3.5 Classpath et vos packages

Lorsque vous utilisez des packages, il est nécessaire de mettre à jour le classpath pour préciser l'endroit où ces package se trouvent. Le classpath doit alors contenir le **répertoire parent** au répertoire du package.

Attention: Si vous ajoutez dans le classpath le répertoire contenant les classes du package, la JVM ne trouvera pas les classes puisqu'elle va chercher les classes du package dans le sous-répertoire et non directement dans le répertoire.

Ainsi, si vous utilisez un package nommé `monPackage` et que ce package se situe dans le répertoire `c:\pack\monPackage`, il est nécessaire que votre classpath contienne le répertoire `c:\pack`. La JVM trouvera alors les classes du package `monPackage` à partir du répertoire `c:\pack` en allant dans le sous-repertoire `monPackage`. Si le classpath contient le chemin `c:\pac\monPackage`, les classes ne seront pas trouvées, puisque la JVM les cherchera dans le sous-répertoire `c:\pack\monPackage\monPackage`.

Attention: Ce sont les déclarations des packages dans les classes qui impliquent les contraintes sur la localisation des fichiers et non l'inverse. Vous devez d'abord raisonner sur l'organisation des packages et leur déclaration, et ensuite seulement les placer dans les bons sous-répertoires en fonction des déclarations.

3.3.6 Classpath sous Eclipse

Le classpath d'un projet Eclipse est sauvegardé dans les données associées au projet et est construit automatiquement en fonction des librairies incluses.

Le classpath est modifiable

- pour la compilation: à partir du menu contextuel du projet `'properties --> java Build Path --> onglet librairies'`
- pour l'exécution: à partir du menu `'run --> run configuration --> onglet classpath'`.

3.4 Fichiers .jar

Les bibliothèques sont amenées à être transmises et échangées. Pour plus de simplicité et pour être sûr que l'ensemble de la bibliothèque est bien transmise, il est possible d'archiver une bibliothèque sous la forme d'un fichier compressé avec une extension `.jar`.

3.4.1 Structure d'un fichier .jar

Un fichier `.jar` contient

- l'ensemble des classes/interfaces du packages
- les ressources nécessaires au package
- un descriptif du contenu du `.jar` appelé `manifest`

3.4.2 Création d'un fichier .jar

Créer un fichier `.jar` en mode console se fait avec la commande `jar`. En lançant la commande `jar` sans option, la liste des options possibles s'affiche.

Pour créer un fichier `.jar`, on retiendra la commande `jar cf nom_archive liste_des_fichiers`.

- l'option `c` signifie que l'on cherche à construire une archive
- l'option `f` signifie que l'archive sera sauvée sous un certain nom
- *nom_archive* désigne le nom de l'archive
- *liste_des_fichiers* désigne la liste des fichiers à inclure dans le `.jar` (souvent plusieurs fichiers)

Pour archiver le package `mesMaths`, il suffit de se mettre dans le répertoire parent et de lancer la commande
`jar cf mesMath.jar mesMath/*.class`

Attention: Si vous lancez la commande jar directement à partir du répertoire du package, les fichiers seront contenus directement dans l'archive sans sous-répertoire intermédiaire et ne seront donc pas trouvés par la JVM car ils seront considérés comme des fichiers appartenant au package par défaut. Il est possible de vérifier la bonne localisation des classes dans le package en ouvrant ou en décompressant l'archive jar obtenue.

3.4.3 Spécification d'un point d'entrée

Il est possible de spécifier le `main` à exécuter dans le `manifest` associé au fichier `.jar`.

En mode console, il faut rajouter `Main-Class: classname` dans le fichier `manifest` du `.jar`, où `classname` désigne la classe contenant le main à exécuter.
Ceci est faisable avec la commande `jar umf ajout-manifest fichier.jar`

- `u` désigne qu'il faut modifier le fichier jar
- `m` que les modifications vont être ajoutées au fichier manifest
- `f` que le fichier jar sera crée
- `ajout-manifest` est le nom d'un fichier texte qui contient les options que l'on souhaite ajouter au MANIFEST
- `fichier.jar` est le nom du fichier d'archive considéré

3.4.4 Utilisation d'un fichier .jar

Pour utiliser un .jar en tant que bibliothèque en mode console, il faut ajouter le fichier `.jar` (et pas uniquement le répertoire où il se trouve) dans le classpath.

- soit en modifiant le classpath
- soit en modifiant localement le classpath avec l'option -cp passé à la JVM

Pour lancer la classe `UtiliseMath`, en supposant que `mesMaths.jar` se trouve dans le même répertoire, il faut faire `java -cp .;mesMaths.jar UtiliseMath`

Avec un point d'accès Si le fichier .jar contient un `main` référencé dans le `manifest`, il est possible de le lancer directement avec la commande
`java - jar nom_package.jar`

3.5 Utilisation d'Eclipse

Création fichier .jar Pour créer un fichier .jar sous Eclipse, il suffit de sélectionner le package ou le projet, de cliquer sur le bouton droit et de choisir le menu `'export --> JAVA --> Jar File'`.

Spécification d'un point d'entrée Lors de l'export en `.jar`, il est possible de spécifier le point d'entrée (troisième page du menu `export`, tout en bas, boite nommée `main class`).

Utilisation d'un fichier .jar Pour utiliser un .jar sous Eclipse, il faut modifier le build path. Pour cela, sélectionner le projet, cliquer droit et choisir le menu `'properties --> java Build Path --> onglet librairies --> Add external Jar'`.

3.6 Dernière remarque

Comme la plupart des bibliothèque sont distribuées sous la forme de fichier jar, il est extrêmement important que vous sachiez inclure des .jar sous Eclipse mais aussi en mode console.

Objectifs pédagogiques

A l'issue de cette section, vous devez savoir

- comment déclarer un package
- comment compiler et exécuter une application constituée de plusieurs packages
- comment construire un fichier jar
- comment inclure des fichiers jar externes

Références

- thinking in JAVA chapitre 5
  `http://penserenjava.free.fr/`

Part II
Tests et vérifications

Une application se doit de présenter

- un fonctionnement correct : elle ne doit pas présenter de bug. C'est l'objectif des tests et des vérifications.
- un fonctionnement robuste : tous les cas pouvant se produire doivent être gérés par l'application. C'est l'objectif des Exceptions.

Dans cette partie, on proposera quelques éléments pour aider les tests et la vérification d'applications.

4 Réflexions sur les erreurs

4.1 Qu'est ce qu'une erreur ?

4.1.1 Définition

De manière naïve, on peut définir l'apparition d'une erreur comme le fonctionnement d'une application non prévu par le concepteur d'un programme. Comme une erreur est par définition non prévue par le concepteur, le concepteur ne peut jamais être sûr de ne pas avoir fait d'erreur. Il existe donc un principe de précaution consistant à vérifier systématiquement une application.

De plus, cette définition s'appuie sur ce que le concepteur peut attendre de son application. Ces attentes peuvent s'exprimer de plusieurs manières

- soit sous la forme d'attentes informelles : le concepteur a des attentes concernant le résultat de l'application. Par exemple, une méthode racine carrée doit retourner la racine du nombre passé en paramètre
- soit sous la forme de spécifications établies au départ : cela peut prendre la forme de *use case* (cf cours UML) et de scénarios décrivant tous les fonctionnements possibles de l'application.

Si on prend l'exemple de la racine carrée, les attentes informelles ne suffisent pas. En effet, aucune information n'est donnée concernant le comportement de la méthode lorsqu'un réel négatif est passé en paramètre[1].

4.1.2 Plusieurs types d'erreurs

Cette réflexion est cependant un peu trop simpliste. Il est possible de distinguer deux types de fonctionnement non prévus

- soit un fonctionnement anormal : lorsque l'exécution d'une méthode ne devrait **en aucun cas** fournir le comportement obtenu.
- soit un fonctionnement exceptionnel : lorsque le programme s'arrête ou présente un comportement étrange à cause **d'un événement possible** mais hautement improbable (comme l'accès à une base de données qui n'existe pas, ...)

Une application doit être sûre et robuste:

- **sûre** : Elle doit fournir le comportement escompté
- **robuste** : Elle doit prendre en compte tous les cas possibles.

4.1.3 Quelles sont les conséquences d'une erreur ?

Les erreurs produites peuvent avoir des effets variés à l'exécution

- Elles peuvent générer des `Exceptions` (comme l'utilisation d'une méthode sur une référence égal à `null`), ce qui fait qu'elles sont facilement détectées[2] même si les corriger peut encore être délicat

[1] Cela dépend du cadre dans lequel on se place, puisque dans le corps des complexes les nombres négatifs ont bien une racibe

[2] A condition que vous ne fassiez pas de *catch* récupérant l'ensemble des exceptions avec un traitement vide `catch(exception e)`, ce qui est une pratique à proscrire définitivement.

  - les exceptions peuvent venir de **comportements anormaux** : une division par une variable ne devant pas être égale à 0, l'accès à une case inexistante d'un tableau parce que l'indice est mal calculé, une référence égale à `null` alors que cela ne devrait jamais être le cas, ...
  - les exceptions peuvent aussi venir de **comportements exceptionnels** : une division par zéro parce que l'utilisateur a mal renseigné le diviseur, une mauvaise entrée clavier sur une demande d'entier, une exception due à l'absence d'une ressource (BDD, fichier, ...)
- les erreurs peuvent conduire à des résultats erronés et sont dans ce cas extrêmement difficiles à déceler. Ce sont les plus dangereuses car les moins facilement détectables.
  - un résultat erroné peut venir d'un **fonctionnement anormal**: mauvaise recopie d'une formule, oubli d'un cast, ...
  - un résultat erroné peut venir d'un **fonctionnement exceptionnel** : base de donnée non initialisée, valeur par défaut utilisée dans des cas particuliers, ...

Comme on peut toujours supposer la présence d'erreurs, il est nécessaire de faire des tests pour vérifier le comportement réel de l'application avec comme principe le fait 'qu'on ne vérifie jamais assez une application'.

4.2 Compromis du traitement des erreurs

Il est impossible de pouvoir prévenir toutes les erreurs d'apparaître. Il va donc falloir trouver des compromis.

4.2.1 A l'exécution

Au cours de l'exécution d'un programme, il est possible de tester les valeurs de différentes variables pour prévenir d'une erreur éventuelle mais

- on ne sait pas forcément quelle devrait être la valeur des variables, puisque c'est justement l'objectif du programme de les calculer.
- ajouter enormement des tests réduit la **vitesse d'exécution** de l'application.

4.2.2 Pendant la phase de test

Après l'écriture d'un programme, une phase de test est nécessaire pour vérifier que tout fonctionne correctement, néanmoins

- il faut pouvoir déterminer les tests à faire et ne pas en oublier
- faire trop de tests prend trop de temps dans le cycle de développement d'une application

Toute la difficulté consiste à déterminer quel est ce 'trop'.

4.2.3 Paradoxe

En outre, il existe un paradoxe lié à la démarche de test. En effet, si le programmeur est un mauvais programmeur, il est fort possible qu'il écrive aussi les mauvais tests. Ainsi, les programmes qui auraient besoin des tests les plus sûrs ont souvent les tests les moins aboutis.

Ceci conduit à plusieurs recommandations (voir ce qu'on appelle l'`extreme programming` pour d'autres bonnes pratiques de développement)

- les tests et le programme sont souvent écrits par des personnes différentes

- les tests sont écrits avant l'implémentation
- l'objectif d'une application est de passer les tests définis au départ et rendant compte de l'ensemble des scénarios possibles

4.3 Comment prévenir correctement des erreurs

On peut prévenir les erreurs à trois niveaux différents

- lors de la conception de l'application
- au cours de phase de test
- lors de son execution

Les sections qui suivent décrivent chacun de ces cas et les outils utilisés.

4.3.1 A la conception

Les méthodes formelles et les preuves de programme permettent de garantir que certaines parties du code fonctionnent correctement dans tous les cas envisagés avant même que le code ne soit écrit

4.3.2 De manière online - pendant l'exécution de l'application

Les Exceptions Les `Exceptions` ont pour but de gérer les fonctionnements exceptionnels à l'exécution du programme. Il s'agit de traiter les scenarii particuliers qui peuvent se produire.

Les `Exceptions` correspondent à des mécanismes complexes car il s'agit de cas qui **peuvent** se produire et qu'il faut donc correctement traiter. (quelle est la conséquence de ce qui a levé l'exception, quelle partie de l'execution du programme est remise en cause, ...). De plus, la déclaration d'une Exception **oblige** le programmeur à prendre en compte ce cas dans la suite du developpement de son code.

Les Exceptions ont été présentées dans le cours Approfondissement du Langage JAVA de première année.

Les assertions Les assertions ont pour objectif de detecter à l'execution les fonctionnements anormaux. Il s'agit de detecter les bugs.

Les assertions correspondent à des mecanismes simples puisqu'il s'agit de detecter des cas qui sont censés ne jamais se produire. Les assertions seront décrites dans le chapitre 5.

4.3.3 De manière offline - au cours d'une phase de test

Une fois qu'une application est écrite, il reste encore à vérifier qu'elle fonctionne. Les Tests ont pour objectifs de vérifier cela de manière offline (c'est à dire en dehors de l'exécution en fonctionnement de l'application).

Les tests unitaires Les tests unitaires ont pour objectif de tester au cours du développement le code écrit.

Ils automatisent toute la phase de tests et permettent de tester à chaque avancée le bon fonctionnement de l'ensemble des classes. Les tests unitaires seront décrits dans le chapitre 6

Les tests d'intégration L'objectif des tests d'intégration est de vérifier que des morceaux d'application fonctionnent et interagissent correctement lorsqu'ils sont mis ensemble.

Les tests d'intégration font suite aux tests unitaires vérifiant le fonctionnement de chacun des morceaux de l'application. Ils ne seront pas abordés.

4.4 Bilan

Le tableau suivant récapitule les différentes techniques permettant de se prémunir d'erreurs.

	Conception	Exécution	Test
Fonctionnement exceptionnel	Scenarii	Exceptions (ADLJ)	Tests unitaires (chap 6)
Fonctionnement anormal	Preuves de Programme	Assertions (chap 5)	Tests unitaires (chap 6)

5 Assertions

Pré-requis

Avant d'aborder ce chapitre, vous devez savoir

- manipuler les exceptions
- comprendre leurs intérêts

Contenu

Cette section présentera

- ce que sont les assertions
- à quoi elles servent
- comment les utiliser
- la différence entre exception et assertion

5.1 Présentation

5.1.1 L'objectif des assertions

En quelques mots, l'objectif des assertions est d'avoir un code sûr à l'abri des erreurs.

Les assertions permettent de vérifier **à l'exécution du programme** que celui-ci n'exhibe pas un **comportement anormal** facilement identifiable.

5.1.2 Principe

En anglais, assert signifie affirmer, un assertion est définie comme *une proposition que l'on avance comme vrai*. Il s'agit donc une condition qui **doit toujours être vérifiée** lorsqu'elle est testée.

A chaque fois qu'on ajoute une assertion, cela signifie qu'on **affirme** en tant que concepteur que cette proposition est vérifiée. Si cela n'est pas le cas, c'est que le programme présente un comportement anormal.

5.1.3 Définition

De manière plus précise, une assertion est décrite comme *une déclaration permettant de tester des exigences vis à vis du programme.*

Il s'agit d'une approche fondée sur la **programmation défensive**: on pare les problèmes en annonçant au sein du programme les propriétés que l'on souhaite vérifier.

Dés qu'une assertion est brisée, un comportement anormal est détecté et le programme s'arrête puisque cela est censé signifier un bug.

5.2 Utilisation

Comme les assertions ont pour objectif de détecter des erreurs qui sont censées ne jamais se produire, une assertion est très facile à mettre en place (ce n'est pas la peine de consacrer un temps important à gérer des événements en théorie non présents).

5.2.1 Déclarer une assertion

Pour déclarer une assertion, il suffit juste d'utiliser le mot-clef `assert` suivi d'une condition. Cette condition doit toujours être **vraie** quand elle est testée.

Si cela n'est pas le cas, l'instruction génère une `AssertionError` qui hérite de `Throwable` et qui remonte la pile d'appels (de la même manière qu'une Exception non récupérée). Par contre, il n'est pas nécessaire de déclarer l'`AssertionError` avec `throws` puisqu'une assertion peut se déclarer n'importe où et est censée arreter l'application dés qu'elle est brisée.

Il est possible d'ajouter un message à l'`AssertionError` en ajoutant `:message` à la suite de la condition, `message` devant être une expression.

L'assertion `assert x>0:"x négatif ou nul"` génère une `AssertionError` ayant pour message 'x est négatif ou nul' si la condition `'x>0'` n'est pas vérifiée.

5.2.2 Fonctionnement d'une assertion

Une assertion fonctionne selon l'algorithme suivant

- dés qu'une assertion est rencontrée, elle est testée
- si elle est vérifiée
  - l'execution se poursuit
- sinon
  - une `AssertionError` est levée
  - l'`AssertionError` remonte la pile d'appels
  - il est possible de la récupérer (mais à éviter)

5.2.3 Exécuter avec les assertions

Par défaut les assertions ne sont pas testées. A chaque fois que la JVM rencontre une assertion, elle n'exécute pas la condition correspondante (y compris s'il s'agit d'un appel de méthode).

Pour tester les assertions à l'exécution, il faut passer l'option `-ea` à la JVM.

```
java -ea testAssertion
```

5.2.4 Exemples

Soit le code suivant

```
public class Test {

  public static void main(String args[])
  {
    int i=2;
    assert i<0:"i est égal à"+i;
  }
}
```

- On compile la classe `Test` normalement `javac Test`
- Lorsqu'on exécute la classe `Test` avec `java Test` rien ne se passe
- Lorsqu'on exécute la classe `Test` avec `java -ea Test`, une `AssertionError` est déclenchée et fournit le message d'erreur suivant :
  ```
  Exception in thread "main" java.lang.AssertionError:  i est égal à 2
  at Test.main(Test.java:7)
  ```

Soit le code suivant

```
public class Test {

  public static void main(String args[])
  {
   assert false:"arret";
  }
}
```

Lorsque la classe `Test` est exécutée avec les assertions, elle génére une `AssertionError`. Ce type d'assertion peut être utile pour contrôler le flux d'exécution et vérifier que le programme n'arrive pas dans une partie du code en théorie inaccessible.

5.3 Où mettre des assertions

5.3.1 Documenter du code

Lorsque vous avez une supposition implicite, utilisez des assertions.

Par exemple, lorsque vous manipulez une variable `x` qui vaut soit 0 soit 1.

```
public class Test {

  public static void main(String args[])
  {
   .....                                          // x vaut 0 ou 1
   if(x==0) { ... }
     else
     {
        assert x==1:"x different de 1";      // x doit valoir 1
     }
  }
}
```

5.3.2 Pre-conditions d'une méthode privée

Définition Une pré-condition est un test sur les paramètres d'entrée d'une méthode privée nécessaire au bon fonctionnement de la méthode.

Comme la méthode est privée, tous les appels se font en interne. Le programmeur a le contrôle sur tous les appels et peut donc faire la supposition que tous les appels doivent respecter la précondition.

Utilisation L'assertion se met en tête du code de la méthode

La méthode proposée ci-dessous a pour objectif d'extraire le sous-tableau constitué des éléments compris entre l'indice `x` et l'indice `y`. La précondition consiste à vérifier que `x` est bien inférieur à `y` pour que l'extraction ait un sens.

```
private static Object[] subArray(Object[]a, int x, int y) {
   // précondition x doit être <= à y
   assert x <= y : "subArray: x > y";
   ...
}
```

Attention Par contre, pour une méthode publique, les arguments doivent être testés avec des conditionnelles (ou lancer des exceptions) puisque le programmeur ne peut pas garantir que la méthode sera toujours correctement appelée.

5.3.3 Post-conditions

Définition Une post-condition est un test sur l'état du système laissé à l'issue de l'exécution d'une méthode.

Utilisation L'assertion se met à la fin de la méthode

La méthode ci-dessous autorise l'insertion d'objets mais doit conserve l'ordre des objets.

```
public void insert(Object o) {
// effectue l'insertion
...
assert isSorted(); // la liste des objets doit rester triée aprés insertion
}
```

5.3.4 Invariants de classe

Définition Un invariant de classe est une contrainte que doivent respecter les instances d'une classe.

Utilisation On ajoute une assertion dés que les attributs concernés peuvent être modifiés. Ces tests sont a priori effectués en fin de méthode puiqu'il est possible qu'au cours de l'execution d'une méthode un objet passe par un état transitoire non cohérent.

La largeur et la longueur d'un rectangle doivent toujours être positifs.

5.3.5 Invariants de flux

Définition Un invariant de flux est une contrainte concernant le déroulement de l'exécution du programme (points de passage interdits, ou passage uniquement sous certaines conditions, ...)

Utilisation On ajoute des assertions dans les endroits du code qui sont censés ne jamais être exécutés. Il suffit la plupart du temps d'ajouter `assert false` qui déclenche directement une `AssertionError`.

On dispose d'un tableau `T` et on sait que l'élément cherché doit s'y trouver

```
int i=0;
while (i<T.length)
{
if (T[i]==e) return(i); \\des que l'element est trouvé, la méthode retourne i
i++;
}
assert false : "élément non trouve"; \\code inaccessible en theorie
```

Nombre de valeurs réduites pour un élément a 0,1 ou 2

```
Switch (a)
{
case 0: ... ; break;
case 1: ... ; break;
case 2: ... ; break;
default : assert false:"mauvaise valeur de a";
}
```

5.3.6 Invariants logiques

Définition Invariants pour lesquels la logique sémantique du code mène à ce qui semble être une tautologie au moment où on l'écrit mais qui pourrait fort bien être détruit par une modification future du code.

Utilisation Assertion lorsqu'une partie du code implique des vérités non explicites.

$Coef = x * x/(1 + x * x)$, Coef entre 0 et 1 (jamais écrit explicitement)
Le rapport longueur sur périmètre est compris entre 0 et 1.

5.4 Bonne pratique des assertions

5.4.1 Pas d'effet de bord

Une assertion **ne doit pas** utiliser une expression avec des effets de bords. Autrement dit, le test du bon fonctionnement d'un programme ne doit pas modifier le programme lui même.

En pratique, si cela n'est pas respecté, le programme aura un fonctionnement différent si les assertions sont testées ou non.

5.4.2 Garder à l'esprit qu'elles sont debrayables

Comme les assertions sont débrayables, il faut que le test soit **entièrement** conditionné par la présence du mot clef `assert`.

Il est par exemple utile (et lisible) de définir des méthodes chargées de faire des tests et retournant des booléens.

Supposons que `t` soit un tableau d'entiers censé être trié. Les programmes 1 et 2 (qui vérifient le tri) ne sont pas bien conçus, puisque si les assertions sont débrayées, les boucles sont encore exécutées.

```
-------------------------
Programme 1
-------------------------
boolean vrai=true;
for (int i=0;i<t.lengh-1;i++)
{
  if (!(t[i]<t[i+1])) vrai=false;
}
assert vrai;

-------------------------
Programme 2
-------------------------
boolean vrai=true;
for (int i=0;i<t.lengh-1;i++)
{
  assert ((t[i]<t[i+1]));
}
```

La bonne solution consiste à utiliser une méthode de test retournant un booléen.

```
------------------------
Programme 3
------------------------
boolean vrai=true;
assert testTri(t);
....

private boolean tresTri(int[] t)
{
for (int i=0;i<t.lengh-1;i++)
  {
    if (!(t[i]<t[i+1])) return(false);
  }
return(true);
}
```

5.4.3 Assertion versus Exception

Il faut garder à l'esprit que les assertions et les exceptions répondent à des besoins différents et ne sont pas interchangeables !!!

- les **assertions**
  - ont pour objectif de **detecter** un fonctionnement anormal
  - conduisent à l'arrêt du programme
  - sont donc simples à mettre en oeuvre pour avoir une application **sûre** (sans bug)
- les **exceptions**
  - ont pour objectif de **gérer** les scenario exceptionnels
  - doivent permettre un fonctionnement dégradé du programme
  - sont relativement complexes à mettre en oeuvre mais permettent d'avoir une application **robuste** (capable de s'adapter aux différentes situations)

5.5 Conclusion

Les Assertions permettent d'ajouter une sûreté au code développé : elles vérifient au cours de l'execution le bon déroulement de l'application et permettent de détecter des erreurs qui n'auraient pas du se produire.

Les assertions présentent un bon compromis

- elles sont faciles à mettre en place
- une condition est souvent plus facile à écrire que le code qui permet de produire la propriété (il est plus facile de tester qu'un tableau est trié que de trier un tableau)
- elles traitent de cas trés rares en théorie
- elles permettent en outre de proposer une forme de documentation (puisque le programmeur sait que les assertions doivent être vérifiées)

Par contre, ce n'est pas parce qu'aucune assertion n'est brisée que votre code fonctionne correctement. Il est possible que votre test soit mal fait ou que l'application soit dans un fonctionnement anormal que vous ne testez pas.

Il reste à vérifier que le code fonctionne bien en le mettant face à toutes les situations qui peuvent se produire. Les tests unitaires ont pour but d'automatiser cette pratique.

Objectifs pédagogiques

A l'issue de cette section, vous devrez savoir

- comment mettre des assertions dans un programme
- à quel endroit placer des assertions pertinentes
- exécuter du code avec ou sans vérification des assertions

Références

- la page JAVA sur les assertions
  `http://java.sun.com/javase/6/docs/technotes/guides/language/assert.html`

6 Tests unitaires

Contenu

Dans cette section, on présentera

- quelle est la philosophie des tests unitaires
- comment fonctionnent les tests unitaires
- comment les construire avec et sans l'aide d'Eclipse
- comment mettre en œuvre correctement des test unitaires

6.1 Présentation des tests unitaires

6.1.1 Objectif

Les tests unitaires ont pour objectif d'automatiser les phases de tests qui vérifient le fonctionnement correct d'un programme.

Contrairement aux assertions, ils sont utilisés offline (hors exploitation de l'application).

6.1.2 Définition

Les tests unitaires sont des tests écrits par les développeurs en même temps que le code lui-même. Ils doivent s'exécuter automatiquement, sans aucune interprétation de la part du développeur. Ils comportent des assertions qui vérifient si le code fonctionne ou non.

- **Écrits par les développeurs:** les développeurs peuvent écrire les tests à partir des squelettes des méthodes sans que le code n'ait été complètement écrit.
- **S'exécuter automatiquement:** Les tests sont écrits indépendamment du code de l'application. Les lancement et la vérification des tests sont séparés du code de l'application.
- **Comportent des assertions** une assertion est une expression booléenne décrivant une condition particulière qui doit être satisfaite pour que le test soit réussi. Ces assertions sont différentes des assertions du chapitre précédent.

6.1.3 Démarche

Les tests unitaires constituent une pratique de l'*Extreme Programming* (`http://fr.wikipedia.org/wiki/Extreme_programming`). Entre autres, l'Extreme programming préconise de développer les tests avant d'écrire l'application elle-même.

La démarche à suivre est la suivante

- déclarer dans un premier lieu l'ensemble des cas d'utilisations et des scénarii possibles (*use cases*)
- définir les profils des méthodes publiques
- écrire les tests permettant de vérifier l'ensemble des scenarii
- développer l'application avec pour but de se limiter à passer tous les tests écrits préalablement.
- lancer les tests et reprendre le code jusqu'à ce que l'ensemble des tests soit validé.

6.1.4 Spécification des tests

Avant d'écrire les tests, il est nécessaire des spécifier ce que l'application doit vérifier. On utilise habituellement pour cela des *uses cases* et des scenarii.

Comme son nom l'indique

- Un test unitaire se veut **unitaire** dans le sens où il a pour objectif de tester une méthode dans des circonstances particulières
- Un test unitaire est un test c'est à dire un cas particulier qui permet de vérifier le bon fonctionnement d'une application.

Determiner les tests à lancer se fait en plusieurs étapes

- déterminer la méthode à tester (ex : une methode qui effectue une recherche dans un tableau)
- penser à tous les modes de fonctionnement possibles pour la méthode à tester
  - mode normal où l'application fonctionne correctement (exemple: recherche d'une valeur contenue dans le tableau)
  - mode normal sur un cas particulier (exemple : la valeur à trouver se trouve en début ou en fin de tableau)
  - mode différent où l'application fonctionne dans un cas dégradé (exemple : recherche d'une valeur absente du tableau)
  - mode exceptionnel où la méthode ne doit pas fonctionner ou doit retourner une erreur (exemple : recherche dans un tableau `null`)
- pour chaque mode de fonctionnement, déterminer des exemples représentatifs
- établir les tests eux mêmes

Enfin, l'écriture du test nécessite plusieurs éléments:

- déterminer les valeurs d'entrée à fournir pour le test
- determiner les instructions à faire
- déterminer les valeurs de sortie attendues sur tous les paramètres qui ont pu varier (ou rester identiques)
- ecrire des tests vérifiant que les valeurs de sortie correspondent bien aux valeurs attendues

Par exemple, si on souhaite tester une classe `Compte` avec une méthode `retrait`, deux scenarii sont à tester :

- le cas où le retrait est permis
  - valeur entree: un compte bancaire avec un solde de 500, et un retrait de 200
  - valeur attendue :
    - le retrait a été effectué (valeur de retour est à true)
    - **ET** le solde est de 300
  - condition : `(cb.retrait(200)==true)&&(cb.solde()==300)`
- le cas où le retrait n'est pas permis
  - valeur entree: un compte bancaire avec un solde de 500, et un retrait de 600
  - valeur attendue :
    - le retrait n'a pas été effectué (valeur de retour est à false)
    - **ET** le solde est de 500
  - condition : `(cb.retrait(600)==true)&&(cb.solde()==500)`

6.1.5 Mise en oeuvre des tests unitaires

Afin de pouvoir automatiser la phase de test (exécuter automatiquement tous les tests, déceler les erreurs et les causes de ces erreurs,...), les tests doivent respecter une certaine structure.

Les tests unitaires associent à chaque classe de l'application une **classe jumelle de test** permettant de tester toutes les méthodes publiques de la **classe applicative**.

L'application est indépendante des classes de test

- l'application elle-même n'utilise pas les classes de test
- les classes de test peuvent donc être stockées dans un package spécifique `test` qu'il n'est pas nécessaire de fournir lorsque l'application est délivrée.
- chaque test est représenté par une méthode de la classe de test.

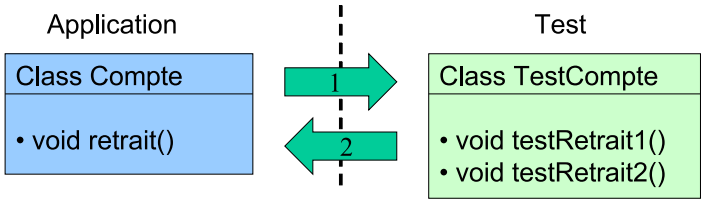


Figure 11: Classe de Test associée à la classe `Compte`

La classe jumelle `TestCompte` chargée de tester la classe `Compte` possède deux méthodes `testRetrait1` et `testRetrait2`, chacune d'entre elles se charge de tester un scenario (retrait possible/retrait impossible).

JUnit est un framework permettant de mettre en oeuvre des tests unitaires en JAVA. Ce framework propose toutes les classes permettant de

- définir des tests et construire les classes de test : chaque classe de test doit hériter de la classe `TestCase`
- référencer tous les tests associés à une application : la classe `TestSuite` permet de rassembler plusieurs tests.
- lancer automatiquement la batterie de tests et analyser les résultats
  - soit de manière graphique avec la classe `junit.awtui.TestRunner`
  - soit de manière textuelle avec la classe `junit.textui.TestRunner`
  - soit dans le code de l'application
  - soit de manière graphique en utilisant eclipse

6.2 Junit

6.2.1 Accés à JUnit

Le framework Junit est disponible sur le site `http://www.junit.org/`. JUnit en est actuellement à la version 4.

Dans ce polycopié nous nous présentons la version 3.8.1, la version 4 est abordée en fin de polycopié.

Dés que vous utilisez JUnit, pensez à inclure junit dans votre classpath.

6.2.2 Principe

Dans la version 3.8.1, les tests se définissent par héritage d'une classe particulière (classe `TestCase`) et doivent respecter certaines contraintes.

Le passage à la version 4 s'est accompagné de nombreux changements. La déclaration des tests dans la classe jumelle se fait en utilisant des annotations spécifiques (comme `@Test`) sans qu'il n'y ait besoin d'utiliser le mécanisme d'héritage.

Les classes permettant d'executer les tests et d'observer les resultats de manière graphique n'existent que dans junit 3.8.1 (junit 4 considère que la plupart des environnements de développement possèdent leur propre interface graphique pour gérer junit et qu'il n'est donc plus utile de maintenir un équivalent dans la bibliotheque junit)

6.3 Mise en oeuvre

Utiliser JUnit se fait en plusieurs étapes

- il faut en premier lieu définir un test unitaire et créer une classe jumelle (cf partie 6.3.2)
- il faut ensuite créer une batterie de test réunissant tous les tests associés à une application (cf partie 6.3.3)
- il faut enfin lancer la batterie de test et analyser les résultats obtenus. (cf partie 6.3.4)

Les parties suivantes se chargent de détailler ces éléments un à un à l'aide de l'exemple Compte bancaire.

6.3.1 Squelette de la classe applicative

Il faut dans un premier temps construire le squelette de la classe à tester. Tant que les profils des différentes méthodes publiques sont inconnus, il est bien entendu impossible d'écrire les tests. Par contre, il n'est pas nécessaire que les corps des méthodes soient complétés.

Dans cet exemple nous cherchons à tester la méthode `retrait` d'une classe `Compte`. On suppose que la classe `Compte` possède une méthode `retrait` prenant en parametre l'argent à retirer et retournant un booleén spécifiant si le retrait a été effectué.

On supposera en outre que la classe `Compte` dispose d'un constructeur prenant en paramètre un entier correspondant au solde du compte.

La classe Compte se déclare donc comme suit

```
class Compte
{
  //peu importe le fonctionnement interne de la classe

  //constructeur à écrire
  public Compte(int solde)
  {
  }

  //methode d'acces à completer
  public int getSolde()
  {
    return(0)
  }

  //méthode à écrire
  public boolean retraitpossible(int retrait)
  {
    return(false);
  }
}
```

6.3.2 Créer un test

Créer un test consiste à définir une classe de test (ayant comme nom le nom de la classe applicative précédé de Test) et à écrire les méthodes de test.

Déclaration de la classe de test La classe de test est une simple classe JAVA héritant de `TestCase`. Il est nécessaire d'importer dans la classe le package `org.junit.*`.

Déclaration d'une méthode de test Une méthode de test est une méthode qui ne retourne aucun résultat et qui ne prend aucun paramètre. Une méthode de test doit débuter par test (comme `testRetrait()`).

Écrire le test Une fois que la méthode de test est déclarée, il ne reste plus qu'à la remplir. Il s'agit simplement d'écrire les instructions correspondant au test à exécuter.

Ajouter les conditions de test Une fois les instructions écrites, il faut rajouter les différentes conditions de test permettant de vérifier que l'application valide effectivement les test.

Il faut utiliser pour cela les méthodes `assertEquals` et `assertTrue` (définies comme méthodes de la classe `TestCase`)

- `assertEquals(Object o1, Object o2)` permet de vérifier l'égalité entre les deux éléments `o1` et `o2`. Par convention, le premier argument désigne l'élément attendu, le second l'élément effectif.
- `assertTrue(boolean cond)` permet de vérifier que la condition passée est vraie

```
import org.junit.*;


class CompteTest extends TestCase     //declaration de la classe de test
{

  public void testretrait1()          //test retrait normal
  {
  Compte c=new Compte(50);            //instruction de tests
  boolean test=c.retraitpossible(5);
  assertTrue(test);                   //condition de test
  }

  public void testretrait2()          //test retrait impossible
  {
  Compte c=new Compte(10);            //instruction de tests
  boolean test=c.retraitpossible(15);
  assertEquals(test,false);           //condition de test
  }
}
```

6.3.3 Créer une batterie de tests

Maintenant que nous savons exécuter les tests `TestCase` par `TestCase`, il peut être utile d'exécuter plusieurs `TestCase` en une seule fois. La classe `TestSuite` permet de rassembler plusieurs tests.

Il faut créer une classe qui retourne un objet `Testsuite` dans une méthode `static Test suite()`

Si l'on dispose de deux classes de test `CompteTest` et `BanqueTest` permettant de tester respectivement la classe `Compte` et la classe `Banque`, on peut créer une classe `AllTests` permettant de lancer tous les tests. Cette classe s'écrira comme suit :

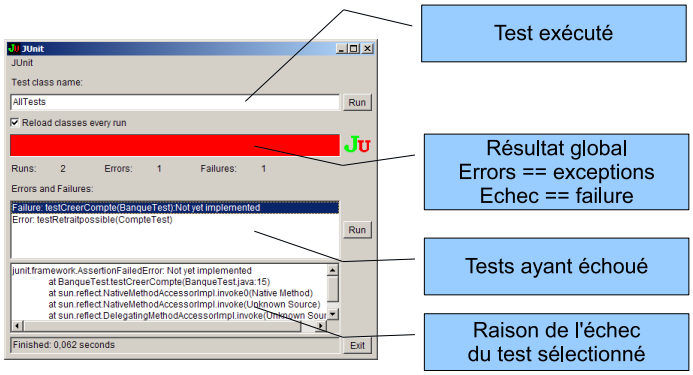
```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {

  public static Test suite()
    {
    TestSuite suite = new TestSuite("tous"); //construction testsuite
    suite.addTestSuite(BanqueTest.class);    //ajout des tests de Banque
    suite.addTestSuite(CompteTest.class);    //ajout des tests de Compte
    return suite;
    }
}
```

Lorsque `AllTests` est executé, tous les tests de `BanqueTest` et de `CompteTest` sont lancés.

6.3.4 Lancer les tests

Une fois qu'une classe `TestCase` a été créée et compilée, il est possible de lancer les tests de la classe grâce à la classe `TestRunner`.

Exemple : `java junit.textui.TestRunner AllTests`



Une fenêtre s'ouvre alors et affiches les résultats du test.

- le nombre de test lancés
- le nombre d'erreurs (errors) : ce sont les tests qui ont généré une exception non récupérée
- le nombre d'echecs (failures) : ce sont les tests pour lesquels une conditon n'est pas vérifiée. Dés qu'une condition est fausse le test s'arrete.

Pour chaque erreur, on peut savoir quelle condition n'a pas été vérifiée et avoir accès à la pile d'exécution.

Il est aussi possible d'avoir les résultat sous forme textuelle.

Exemple : `java junit.awtui.TestRunner AllTests`

```
Time: 0
```

```
There was 1 error:
1) testRetraitpossible(CompteTest)java.lang.IllegalArgumentException
at CompteTest.testRetraitpossible(CompteTest.java:15)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at Principale.main(Principale.java:9)
There was 1 failure:
1) testCreerCompte(BanqueTest)junit.framework.AssertionFailedError: Not yet implemented
at BanqueTest.testCreerCompte(BanqueTest.java:15)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at Principale.main(Principale.java:9)

FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 1
```

- `Tests run` designe le nombre de tests executés
- `failures` désigne le nombre d'échecs (assertEquals ou assertTrue non vérifiés)
- `errors` désigne le nombre de tests qui ont conduit à des erreurs (exceptions levées, ...)
- les failures et errors sont décrites au début du résultat. Chaque echec/error est décrit par
  - le nom du test: la méthode qui est lancée
  - le nom de la classe entre parenthèse: endroit où est écrite la méthode
  - la raison de l'échec ou de l'erreur
  - la pile d'appels au moment de l'erreur.

6.4 Compléments à JUnit

6.4.1 Preparation et liberation des ressources

Dans certains cas, tous les tests d'une classe ont besoin d'une initialisation qui peut être assez longue. Comme par exemple

- la construction d'un objet complexe necessaire à plusieurs tests (comme un graphe compliqué)
- la connection à une base de données
- etc ...

Il est possible de factoriser la préparation des données dans la méthode `public void setUp()` de la classe de test.

De la même manière, la libération commune des ressources (pour fermer une connection ouverte à une base de données) peut se factoriser dans la méthode `public void teardown()`.

Chaque test s'execute alors de la manière suivante:

- Execution de la méthode `setUp()`
- Execution de la méthode de test
- Execution de la méthode `tearDown()`

6.4.2 Tester la levée d'exceptions

Dans certains cas, le comportement normal d'une méthode est de retourner une exception (dans un cas trés particulier, par exemple lorsqu'on calcule la racine d'un nombre négatif). Le test a donc pour objectif de vérifier que la méthode retourne effectivement la bonne exception.

Le test appelle la méthode en passant le parametre menant à la situation perturbatrice et intercepte les exceptions levées. Le test vérifie ensuite qu'il passe effectivement par les instructions dans le `catch` (en mettant un booleen à vrai par exemple).

La classe de Test suivante permet de vérifier que la méthode `racine` de la class `MesMath` lève bien un `IllegalArgumentException` lorsque le paramètre passé est négatif.

```
import org.junit.*;

class RacineTest extends TestCase     //declaration de la classe de test
{

  public void testRacine()            //test racine
  {
  boolean exc=false;                  //boolean pour tester la levée de l'exception
  MesMaths m=new MesMaths();
  try
    {
    m.racine(-1);                     //pour intercepter l'exception
    }
  catch (IllegalArgumentException e)  //pour etre sur de la bonne exception
    {
    exc=true;
    }
  assertTrue(exc);                    //test pour vérifier que l'exception a été levée
  }
}
```

6.4.3 Tester une classe générique

Il n'est pas possible de tester une classe générique dans l'absolu (puisqu'un test est une condition sur un cas particulier). Il faut donc se limiter à tester des classes paramétrées.

6.4.4 Resultats de test

Il est possible de lancer les tests à l'aide de la méthode `run` de la classe `TestSuite` qui stocke les résultats dans un objet de type `TestResult` et d'analyser les résultats dans un programme java (C'est comme cela que TestRunner doit être conçu).

6.5 Tests unitaires sous eclipse

Eclipse simplifie grandement l'écriture et le lancement des tests unitaires en produisant des squelettes de test et en générant automatiquement les `TestSuite`.

6.5.1 Ajouter Junit à Eclipse

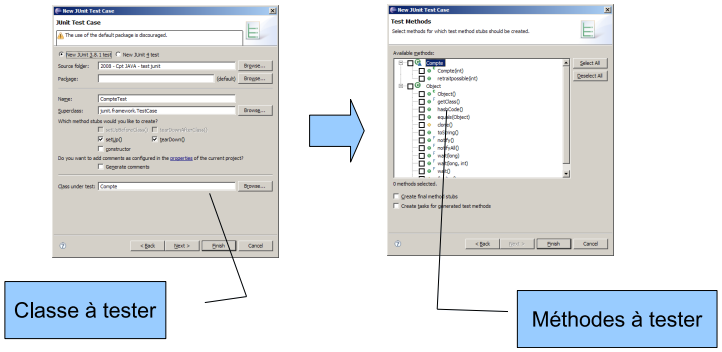
Pour permettre à Eclipse de manipuler des test unitaires, il est nécessaire d'inclure la librairie Junit dans le classpath du projet. Pour cela, il suffit simplement de selectionner `file -->`

`properties --> JAVA build Path --> add Library --> Junit`.

6.5.2 Créer un test

Pour créer un `TestCase`, le plus simple est de faire un click-droit sur la classe à tester et de selectionner le menu `new --> other ... --> Junit --> JUnit Test Case`.

Eclipse ouvre alors une fenetre pour selectionner la classe à tester. Dans un second temps, Eclipse propose de selectionner les méthodes à tester.



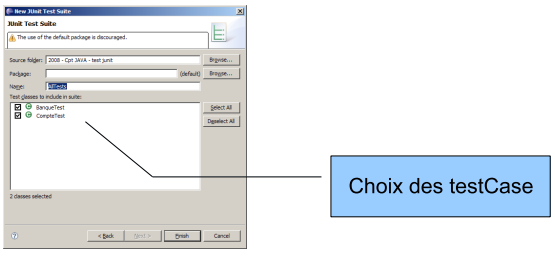
Eclipse crée alors un squelette de test. Par défaut :

- nom de la classe de test est celui de la classe à tester suivi de `Test`
- pour chaque méthode à tester, Eclipse génére une méthode de test qu'il remplit avec `fail("Not yet implemented");`. Cette instruction génére un échec du test lorsqu'elle est executée.

Il ne reste plus qu'à remplir le contenu des méthodes de test.

6.5.3 Creer une batterie de tests

Pour créer une batterie de tests, il suffit de selectionner le menu `File --> new --> other --> junit --> JUnit TestSuite`. Eclipse detecte automatiquement les `TestCase` et vous permet de selectionner ceux que vous souhaitez inclure dans le `TestSuite`.

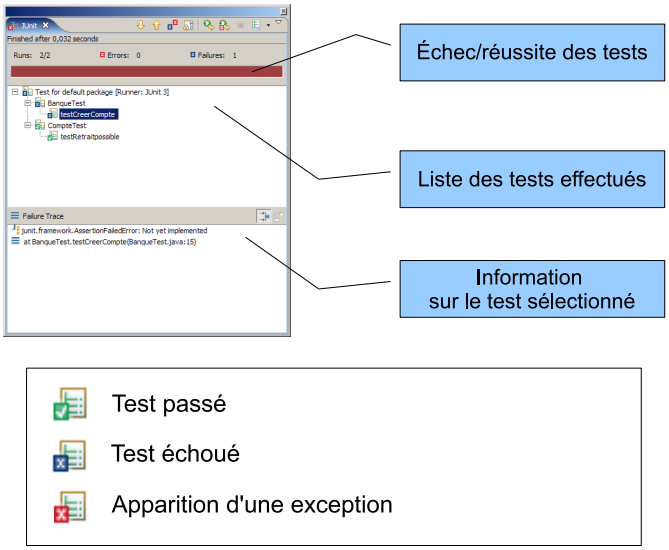


Eclipse génére automatiquement la classe permettant de lancer la suite de tests.

6.5.4 Lancer les tests

Lancer les tests est aussi extrememement simple, il suffit de faire un click-droit sur la classe contenant les tests (soit un TestCase, soit une classe avec une méthode suite retournant des tests) et de selectionner `Run as --> Junit Test`.

Eclipse ouvre alors une nouvelle vue présentant les résultats du test.



6.6 JUnit 4

Même si les concepts de la version 4 sont identiques à ceux de la version 3, JUnit fonctionne légérement différemment.

Il est à noter qu'il n'existe pas de TestRunner dans Junit4.0. Lancer les tests nécessite un environnement de développement adapté (comme Eclipse), se fait manuellement ou utilise des `Adapter` (design pattern appliqué à l'encapsulation de test pour passer de JUnit version 4 à JUnit version 3).

6.6.1 Méthode de test

Les classes de Test n'heritent plus de `TestCase`. Un test se déclare simplement en écrivant l'annotation `@Test` devant la méthode de test.

```
@Test
public void addition() {
    assertEquals(12, simpleMath.add(7, 5));
}

@Test
public void subtraction() {
    assertEquals(9, simpleMath.substract(12, 3));
}
```

6.6.2 Interception d'exception

Lorsqu'un test doit lever une exception, il suffit de le préciser dans le descriptif de test en utilisant le mot clef `expected`.

```
@Test(expected = ArithmeticException.class)
public void divisionWithException() {
    // divide by zero
    simpleMath.divide(1, 0);
}
```

6.6.3 Ignorer des tests

Il est possible d'ignorer temporairement des tests en utilisant l'annotation `@Ignore`.

```
@Ignore("Test non pret")
@Test
public void multiplication() {
    assertEquals(15, simpleMath.multiply(3, 5));
}
```

6.6.4 Nouveaux tests possibles

Il existe de nouvelles méthodes de test permettant de comparer le contenu de deux tableaux. Deux tableaux sont égaux si et seulement s'ils ont la même longueur et si leurs élements correspondent.

```
public static void assertEquals(Object[] expected, Object[] actual);

public static void assertEquals(String message, Object[] expected, Object[] actual);
```

6.7 Bilan

6.7.1 Intérêts

Les test unitaires présentent certaines propriétés intrinsèques

Réutilisabilité Il est possible d'utiliser des tests écrits par un autre développeur ou de réutiliser des tests déja écrits pour une application similaire

Automatisme Lancer une suite de test se fait de manière trés simple comme l'interpretation des résultats. Ils permettent de localiser rapidement et de manière automatique les sources d'une erreur (dans quelle méthode l'erreur intervient)

Code propre Les test unitaires permettent de séparer le code de l'application des tests. Ils sont implémentés dans des classes à part.

Documentation Les tests unitaires peuvent servir de specification de code et de "documentation executable". Un nouveau développeur peut apprehender le fonctionnement des méthodes en lisant les tests unitaires.

6.7.2 Resolution de probleme

L'utilisation de tests unitaires est relativement simple et permet de répondre à un certain nombre de problèmatiques

- Travailler à plusieurs
  - **Probleme: Interdépendance des codes** Une partie utilise du code développée par une autre personne
  - **Objectif: Confiance dans le travail de l'autre** JUnit permet à un développeur de s'assurer que le code sur lequel il s'appuie fonctionne correctement et correspond bien à ses attentes
- Detection de Bug
  - **Probleme: Effets de bord** Ils sont relativement difficiles à detecter
  - **Objectif** En testant méthode par méthode, les tests unitaires permettent de localiser rapidement les erreurs.

- Gérer des modifications
  - **Probleme: Problème de regression** Modifier une partie du code peut conduire à l'apparition de bug
  - **Objectif** En automatisant le lancement de test, Junit permet de lancer simplement tous les tests qui ont pu être écrits et de detecter trés rapidement une erreur qui apparait alors qu'elle n'était pas présente auparavant.
- Programmer efficacement
  - **Probleme: Développer du code inutile** On a souvent tendance à développer des morceaux de code qui ne servent pas directement à l'objectif du projet
  - **Objectif** En se concentrant sur les tests unitaires (comme seul objectif du développement), les tests unitaires permettent de se limiter à la validation du cahier des charges.

6.7.3 Bonne Pratique

Il est recommandé d'écrire les tests avant l'écriture du code car ils permettent de voir tous les cas particulier qui peuvent se produire avant d'écrire le programme plutot que de s'en rendre compte au moment du développement.

Ils ont pour objectif de vérifier à chaque étape du code que l'application fonctionne correctement. Une fois que vos tests unitaires sont écrits, continuez à éxecuter l'ensemble des tests à chaque modification (ce qui ne coute pas grand chose et permet d'éviter des problemes de regression de code).

Dés qu'un bug est détecté, il est trés facile d'ajouter un nouveau test unitaire testant l'apparition du bug. Si le bug vient à réapparaitre, il sera automatiquement détecté.

Lorsque vous écrivez un test, plutôt que modifier un test existant (mauvaise pratique observée en TP), créez un nouveau test. De la même manière, à chaque fois que vous voulez vérifier un élément dans votre application, plutôt que d'ajouter des instructions `println()`, preferrez ajouter un test unitaire.

Objectifs pédagogiques

A l'issue de cette section, vous devrez savoir

- ce qu'est un test unitaire
- comment construire un test unitaire sans outil
- comment construire un test unitaire avec eclipse
- comment automatiser la vérification des tests unitaires

References

- la FAQ de JUnit
  `http://junit.sourceforge.net/doc/faq/faq.htm`

7 Bilan sur les tests

7.1 Distinction assertion, test unitaire

Pour vérifier le bon fonctionnement d'une application, nous avons vu deux moyens les Assertions et les Tests unitaires.

Il est nécessaire de bien distinguer ces deux moyens

- les assertions
  - sont testées au cours du lancement de l'application
  - correspondent à des vérités générales indépendantes des données (une assertion doit toujours être vraie)
- les tests unitaires
  - sont testés indépendamment de l'execution du programme
  - correspondent à des résultats attendus sur des données particulières passées en entrée

7.2 Utilité des tests

Il est à retenir qu'une vérification est souvent beaucoup plus simple à écrire que le programme lui même

- par exemple : une assertion vérifiant qu'un tableau est trié à l'issue d'une méthode de tri est beaucoup plus simple à écrire que le tri lui même
- par exemple : un test unitaire consiste simplement à spécifier ce qui est attendu de l'execution d'une métode sur un cas particulier et est indépendant de la manière dont le code s'execute.

En conséquence, ecrire des tests unitaires et des assertions est un moyen `trés efficace` et `peu couteux` d'augmenter la sureté d'un programme.

N'hesitez pas à en abuser !!!
(y compris dans vos projets)

Part III
Outils pour la conception d'applications

8 Approche Modèle - Vue - Contrôleur

Pre-requis

Pour aborder cette partie, vous devez

- maîtriser le cours sur les interfaces graphiques sous JAVA
- savoir comment fonctionne une JFrame, un JPanel, un Listener, ,...
- bien faire la différence entre tous les constituants d'une interface

Contenu

Dans cette partie, on expliquera

- l'approche modèle vue, contrôleur et le design Observeur/Observé
- comment ce design est implémenté en java
- ce que sont les design patterns

8.1 Problème lié aux Interfaces graphiques

Dans l'absolu, une application n'est pas dépendante de son interface. Il devrait être possible d'écrire un logiciel puis de penser son interface après coup (voire de proposer plusieurs interfaces).

L'objectif de ce chapitre est de vous proposer une solution élégante, l'approche **MVC (Modèle-vue-contrôleur)**, classiquement utilisée et facile à mettre en oeuvre en JAVA pour proposer et développer des logiciels correctement structurés.

A l'issue de ce chapitre, vous devrez avoir une réponse aux questions suivantes :

- comment ajouter ou retirer une interface à l'exécution?
- comment développer des interfaces différentes pour une même application?
- comment ne pas mélanger le code lié à l'interface et le programme chargé de modifier les données?
- comment écrire du code sans se préoccuper de l'interface?

8.2 Modèle Vue Contrôleur

8.2.1 Principe

L'approche MVC cherche à séparer une application en trois blocs

- Le **Modèle** qui se charge de gérer l'ensemble des données et leur évolution (mise à jour des données, dynamique, règles dans un jeu vidéo, ...)
- la **Vue** qui se charge d'afficher les données
- le **Contrôleur** qui se charge de gérer les actions de l'utilisateur et d'appeler les actions correspondantes sur le modèle

8.2.2 Les différents blocs

Le Modèle Le **modèle** gère les données et peut être crée indépendamment du reste. Il doit juste fournir des méthodes publiques correspondant aux actions possibles de l'utilisateur et aux

lois d'évolution du système à modéliser.

Par exemple, si l'objectif est de construire un logiciel de gestion, le modèle doit stocker l'ensemble des données (ou les accès à une BDD) et toutes les méthodes permettant de modifier certaines données et d'effectuer des calculs.

De la même manière, si l'objectif est de construire un jeu de stratégie temps réel (RTS), le modèle doit stocker la position des différentes troupes, gérer leurs positions et les déplacements, gérer les combats, gérer la destruction d'unité, la construction de bâtiments, ...

Par contre, le *Modèle* ne se charge pas d'afficher les données, c'est à la *Vue* de le faire. Le *Modèle* doit simplement connaître la *Vue* pour pouvoir lui demander de mettre à jour l'affichage dès qu'il y a des modifications.

La Vue La **Vue** gère l'affichage des données. En théorie (mais cela peut être plus compliqué), elle ne s'occupe pas de la manière dont les données peuvent évoluer mais a pour objectif de rendre compte sur l'écran des données à un instant t.

Par exemple, si l'objectif est de construire un logiciel de gestion, la vue aura pour objectif de présenter les différentes données dans un `JPanel` à des endroits spécifiques en utilisant certaines couleurs, ...

Si l'objectif est de construire un jeu RTS, la vue se chargera d'afficher la carte du monde, les unités en fonction de leur position et de leur état, les bâtiments visibles, ...

La *Vue* est appelée par le *Modèle* dès que celui-ci estime que l'affichage doit être mis à jour. La *Vue* connaît le modèle puisqu'elle utilise les données du modèle pour construire ce qui sera affiché à l'écran.

Le Contrôleur Le **contrôleur** a pour objectif de gérer l'ensemble des actions de l'utilisateur. Il ne modifie pas directement les données, mais appelle une méthode du modèle avec les paramètres correspondant à ce que souhaite faire l'utilisateur.

Par exemple, si l'objectif est de construire un logiciel de gestion, le contrôleur réagira aux clics de souris de l'utilisateur et appellera les méthodes du modèle permettant d'effectuer des modifications dans les données ou de lancer des calculs.

Si l'objectif est de construire un jeu RTS, le contrôleur se chargera d'analyser les clics de souris et d'appeler les méthodes correspondant aux actions des utilisateurs comme donner un ordre de déplacement à une unité, sélectionner une unité, etc ...

Le contrôleur est appelé par l'utilisateur et connaît le modèle pour pouvoir y appeler des méthodes.

La distinction entre *Vue* et *Contrôleur* est parfois délicate puisque le contrôleur peut s'appuyer sur la vue pour gérer les demandes des utilisateurs (exemple cliquer à un endroit de la Vue pour effectuer un changement). Le plus souvent, le contrôleur correspond à un `Listener` alors que la Vue est le `JComponent` lui-même.

8.2.3 Diagramme d'interaction

Le modèle MVC fonctionne de la manière suivante (cf figure 12):

1. l'utilisateur déclenche une action interprétée par le *contrôleur*
2. le *contrôleur* analyse l'action de l'utilisateur et appelle une méthode spécifique du modèle en fonction des paramètres de l'action de l'utilisateur
3. le *modèle* modifie les données en conséquence et demande ensuite à la vue de se mettre à jour
4. la *vue* accède aux données du modèle en rend compte à l'écran des modifications effectuées

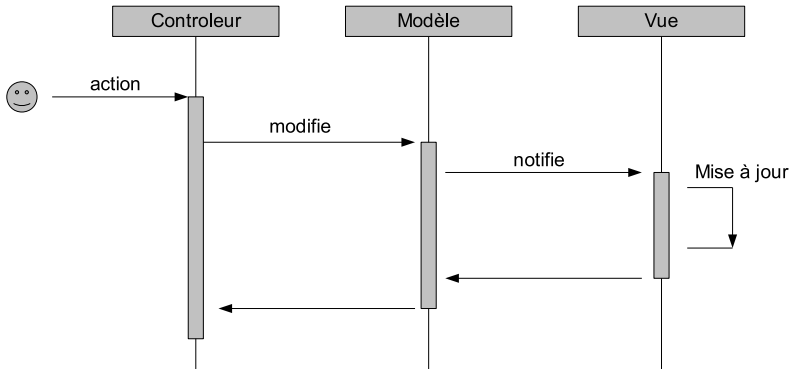


Figure 12: Diagramme d'interaction de l'approche MVC

8.3 Observer / Observable

L'approche MVC est la combinaison de plusieurs '*design patterns*'[3]. Le *design* Observeur/Observé est chargé de gérer le lien entre le modèle et la vue:

- Observé désigne l'objet à afficher et correspond au modèle
- Observeur désigne ce qui va observer l'objet et correspond aux différentes vues

Ce *design pattern* est mis en œuvre en Java par les classes `Observable` et l'interface `Observer` que nous décrirons en détail dans la partie 8.3.2.

8.3.1 le design Observeur-Observé

Le design Observeur/Observé fonctionne de la manière suivante:

Observé Observé désigne l'objet à être affiché. Il correspond au modèle et doit proposer plusieurs méthodes

- une méthode publique `attache` permettant d'associer un Observeur supplémentaire affichant les données
- une méthode privée `notifie` qui demande à tous les Observeurs associés de mettre à jour leur affichage si cela est nécessaire

Observeur Observeur désigne une Vue et doit proposer

- une méthode `miseAJour()` permettant d'afficher correctement l'Observé à qui il est associé

[3]Un **design pattern** est une solution éprouvée à un problème d'architecture logicielle rencontré fréquemment en informatique, cf fin de ce chapitre pour plus d'informations.

Diagramme de Classe simplifié Un *design pattern* est une solution réutilisable. Une telle solution se structure donc en deux parties (cf figure 13)

- la partie supérieure (`Sujet` correspondant à Observé et `Observeur`) correspond aux classes générales permettant de mettre en place le design Observeur-Observé
- la partie inférieure (`Sujet_reel` et `Observeur_reel`) correspond à l'utilisation particulière de ces classes générales pour mettre en œuvre le *design* Observeur-Observe sur un cas spécifique.

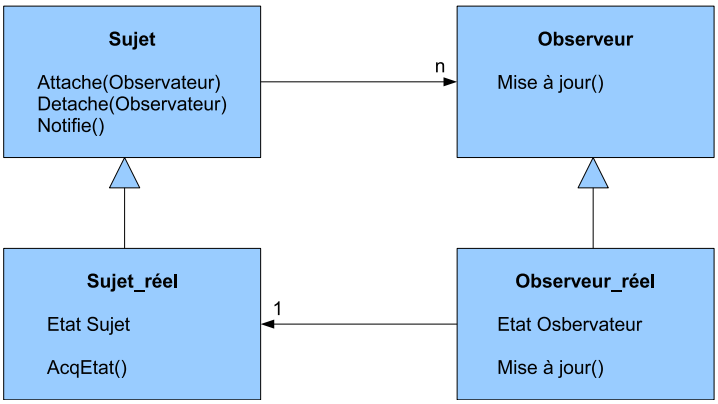


Figure 13: Diagramme classe design Observateur/Observe

Les classes présentées dans la figure 13 sont organisées de la manière suivante :

- **Sujet** désigne la classe de laquelle hérite tous les observés. `Sujet` possède la méthode `Attache` pour ajouter un Observeur et la méthode `notifie` pour prévenir d'une mise à jour
  - la méthode `notifie` effectue une boucle sur tous les observeurs et appelle la méthode `MiseAjour` sur chacun d'entre eux.
- **Observeur** désigne l'interface qu'implémente tous les observeurs. `Observeur` possède la méthode abstraite `miseAJour` chargée de mettre à jour l'affichage quand l'observeur est notifié.
- **Sujet réel** désigne les données spécifiques que l'on souhaite observer. `Sujet Réel` hérite de `Sujet` et dispose d'un état interne correspondant aux données à traiter. `Sujet réel` dispose par héritage des méthodes `notifie` et `attache` définies dans `Sujet`.
- **Observeur réel** désigne la vue que l'on souhaite mettre en oeuvre. Il faut redéfinir la méthode abstraite `miseAjour` de la classe `Observeur` pour gérer l'affichage réel des données du modèle. `Observeur reel` doit en outre possèder un lien vers le modèle `Sujet reel` pour accéder aux données à afficher.

8.3.2 Les classes Observer et Observable

JAVA propose des classes `Observer` et `Observable` qui mettent en œuvre ce *design*.

8.3.3 La classe Observable

La classe `java.util.Observable` correspond à la classe `Sujet`. Elle permet d'obtenir par héritage les différentes classes pouvant être observées. La classe `Observable` définit les méthodes

- `notifyObservers()` permettant de demander aux `Observer` enregistrés de se mettre à jour (par la méthode `update()`)

- `setChanged()` permettant de marquer l'objet comme devant être mis à jour (si la méthode n'est pas appelée auparavant, `notifyObservers()` ne fait rien)
- `addObserver(Observer O)` permettant d'enregistrer un Observer supplémentaire lié à l'Observable.

8.3.4 L'interface Observer

L'interface `Observer` correspond à la classe `Observeur` du design pattern. Elle permet d'obtenir par implémentation les `Observer`. Cette interface ne dispose que d'une méthode à implémenter

- `void update(Observable o, Object arg)` : cette méthode est appelée par la méthode `notifyObservers()` de `Observable` pour la mise à jour de l'affichage. `o` désigne l'observable à afficher et `arg` des arguments qu'il est possible de transmettre lors de l'appel de `notifyObservers()`

8.3.5 Mise en place d'un MVC avec Observer et Observable

Mettre en place le design Observeur-Observé se fait en trois étapes :

a. Le modèle Il faut dans un premier temps créer le modèle par héritage de la classe `Observable` :

- récupérer par héritage les méthodes de `Observable` : `addObserver`, `notifyObservers` et `setChanged`
- ajouter tous les attributs permettant de stocker les données
- ajouter les méthodes permettant de modifier ces données

b. La vue Il faut ensuite créer la vue en implémentant l'interface `Observer`

- Pour ce faire, il est nécessaire de redéfinir la méthode `update`

c. Liaison Modèle-Vue Enfin, il reste à lier le modèle à la vue.

- il faut ajouter les appels à `notifyObservers` et `setChanged` dans le modèle pour demander les mises à jour de la vue.
- Il faut en outre faire un `main` ajoutant la vue au modèle avec la méthode `addObserver` héritée de `Observable`

8.4 Exemple

On souhaite faire une application permettant de modifier la taille d'un disque.

8.4.1 Description des éléments MVC

- le modèle correspond aux données, c'est à dire la description du disque à modifier
  - Le modèle est constitué d'une classe `Disque` de type `Observable`.
  - Cette classe possède un attribut `taille`.
- la vue correspond à l'affichage du disque. On considérera deux vues chacune implémentant `Observer`
  - une vue purement textuelle affichant la taille du disque
  - une vue graphique dessinant le disque sur un `JPanel`
- le contrôleur correspond à l'interaction avec l'utilisateur. on considérera deux contrôleurs
  - un contrôleur textuel avec la classe `Scanner`
  - un contrôleur graphique fondé sur un `JSlider`

8.4.2 Mise en place du modèle

Le modèle correspond au Disque. La classe `Disque` hérite de `Observable` et possède un attribut `taille`.

Pour gérer le lien avec la vue, la classe `Disque` doit pouvoir fournir les informations utiles à l'affichage. Elle dispose ainsi d'une méthode publique `getTaille()` retournant la taille du disque.

Pour gérer le lien avec le contrôleur, la classe doit proposer des méthodes publiques de modification. La classe disque doit donc disposer d'une méthode `setTaille()`

Pour gérer la mise à jour de l'affichage, la classe `Disque` doit appeler la mise à jour de l'affichage à chaque fois que sa taille est modifiée.

La classe `Disque` est donc la suivante

```
import java.util.Observable;

// la classe herite de Observable
public class Disque extends Observable {

    // declaration de l'attribut
    int taille;

    // constructeur
    public Disque() {
      taille = 10;
    }

    // getter pour la vue
    public int getTaille() {
      return (taille);
    }

    //setter pour le controleur
    public void setTaille(int t){
      if (t>0) taille=t;
      //prevenir la modification, methode de Observable
      setChanged();
      //notifier Observer, methode de Observable
      notifyObservers();
    }
}
```

8.4.3 Mise en place de la Vue Textuelle

La vue textuelle se charge simplement d'afficher la valeur de l'attribut `taille`.

Il suffit de créer une classe `VueTexte` qui implémente l'interface `Observer` et de surcharger la méthode `update(Observable o)`.

```
import java.util.Observable;
import java.util.Observer;


public class VueTexte implements Observer {

    //mise à jour de l'affichage
```

```
    public void update(Observable o, Object arg) {
      //consiste simplement à afficher la taille de l'observable passé
      System.out.println("vue texte :"+((Disque)o).getTaille());
    }
}
```

8.4.4 Mise en place de la vue Graphique

La vue Graphique va se charger d'afficher un cercle de la bonne taille lorsque l'affichage est mis à jour. La classe `VueGraph`, hérite de `JPanel` (pour être un composant affichable) et implémente `Observer` (pour pouvoir constituer une Vue). Elle dispose en outre d'un attribut correspondant au modèle à afficher.

Il suffit donc de créer la classe `VueGraph`, de surcharger les méthodes

- `update()` de `Observer` pour demander la mise à jour de l'affichage avec un `repaint()`
- `paint()` de `JPanel` pour afficher les informations que l'on souhaite

```
import java.awt.Graphics;
import java.util.Observable;
import java.util.Observer;

import javax.swing.JPanel;

public class VueGraph extends JPanel implements Observer{

    Disque modele;

    //pour répondre à une demande d'affichage
    public void update(Observable o, Object arg1) {
      //mise à jour du lien vers le modele à afficher
      modele=(Disque)o;
      //appel de la fonction d'affichage paint()
      repaint();
    }

    //surcharge de la méthode paint()
    public void paint(Graphics g) {
      super.paint(g);
      //affichage du cercle si il est reference
      if (modele!=null)
      {
          g.drawOval(0, 0, modele.getTaille(), modele.getTaille());
      }
    }
}
```

8.4.5 Mise en place du contrôleur textuel

L'objectif du contrôleur textuel est de vérifier rapidement que tout fonctionne. Il se limite simplement à un `main` qui crée les vues, fait le lien entre les vues et le modèle et modifie le modèle pour vérifier que la vue se met correctement à jour.

```
import java.awt.Dimension;
import java.util.Scanner;

import javax.swing.JFrame;
```

```
public class ControlText {

    public static void main(String[] args) {
      //création du modèle
      Disque d=new Disque();

      //création de la vue textuelle
      VueTexte vt=new VueTexte();

      //creation de la vue graphique
      VueGraph vg=new VueGraph();
      vg.setPreferredSize(new Dimension(100,100));
      JFrame f=new JFrame();
      //on ajoute le jpanel à la frame
      f.setContentPane(vg);
      f.pack();
      f.setVisible(true);

      //on fait le lien entre modèle et les vues
      d.addObserver(vt);
      d.addObserver(vg);

      //on fait la boucle de modification
      Scanner sc=new Scanner(System.in);
      //demande la nouvelle taille
      int choix=sc.nextInt();
      while(choix>0)
        {
        //modification
        //conduit à la mise à jour des vues
        d.setTaille(choix);
        choix=sc.nextInt();
        }
      System.exit(1);
    }
}
```

La manière dont fonctionne le `main` est décrit par le diagramme de séquence présenté figure 14.

L'appel de `d.setTaille(choix)` dans la boucle déclenche les appels suivants :

- la méthode `setTaille()` de `Disque` qui modifie la taille du disque
  - la méthode `setTaille` appelle `setChanged()` qui specifie que le modèle a été modifié
  - la méthode `setTaille` appelle `notifyObservers()` qui appelle la mise à jour sur les observers référencés (cf `addObserver`)
    - Mise à jour de la vue textuelle
      - appel à la méthode `update` de `VueText` qui affiche la valeur de `taille`
    - mise à jour de la vue graphique
      - appel de la méthode `update` de `VueGraph` qui appelle `repaint` de `VueGraph`
      - la méthode `repaint` de `VueGraph` appelle la méthode `paint` de `VueGraph`
      - la méthode `paint` de `VueGraph` affiche le disque
    - remise de `changed` à false puisque l'affichage correspond au modèle
- sortie de la méthode `setTaille`

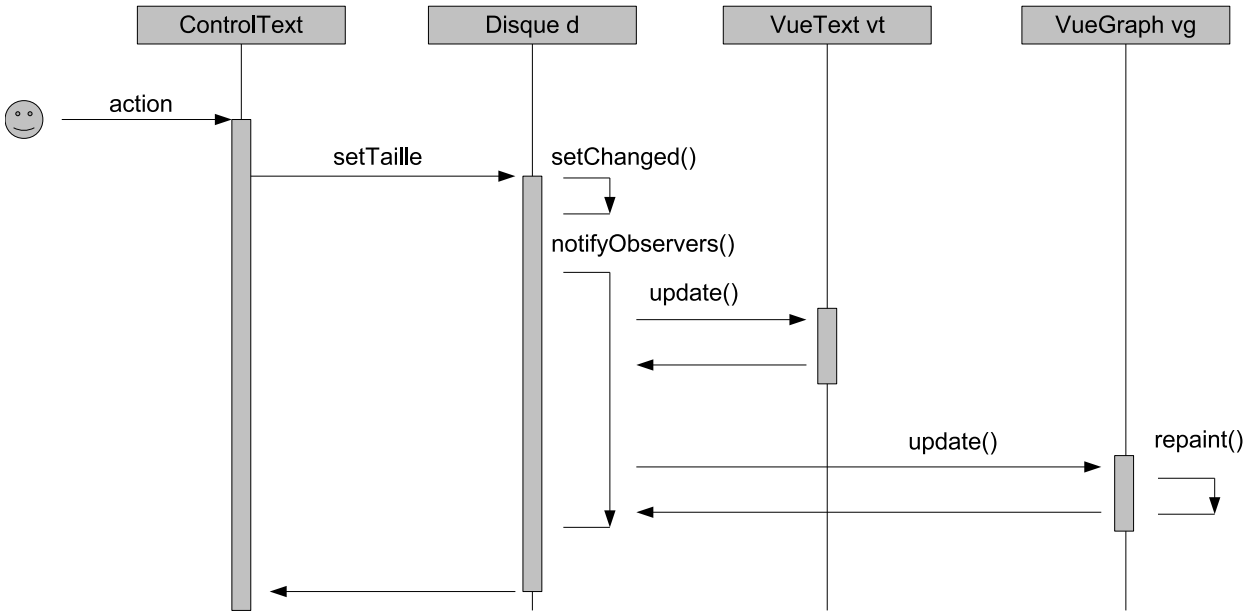


Figure 14: Diagramme de séquence pour l'exécution de la méthode setTaille()

8.4.6 Mise en place du contrôleur graphique

Le contrôleur Graphique sera représenté par un `JSlider`.

L'interaction avec l'utilisateur se fera donc en utilisant un `ActionListener`. Dès que la position associée au Slider change, le `JSlider` appelle la méthode `setTaille` du modèle et le modèle demande automatiquement à l'affichage de se mettre à jour.

Le contrôleur a besoin d'un attribut référençant le modèle pour savoir sur quel objet appeler la méthode.

```
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;


public class Controleur extends JSlider{

  //lien vers le modèle
  Disque modele;

  //constructeur
  public Controleur(Disque d)
  {
    super();
    //lien avec le modèle
    modele=d;

    //initialisation du Slider
    setMaximum(100);
    setMinimum(1);

    //ajout d'un listener pour suivre l'évolution du curseur
    addChangeListener(new ChangeListener(){
        public void stateChanged(ChangeEvent arg0) {
```

```
            modele.setTaille(getValue());
        }
    });
  }
}
```

8.4.7 Agencement des blocs

Il ne reste plus qu'à écrire un `main` pour lier tous les éléments. Le `main` se charge

- de créer le modèle
- de créer les vues
- de lier les vues au modèle
- de créer le contrôleur

Une fois ces éléments mis en relation, le contrôleur appelle les méthodes de modifications du modèle qui appellent automatiquement la mise à jour des vues.

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import javax.swing.JFrame;

public class Test {

    public static void main(String[] args) {
      // creer un modèle
      Disque d = new Disque();

      // creer les vues
      VueTexte vt = new VueTexte();
      VueGraph vg = new VueGraph();

      // attacher la vue au modèle
      d.addObserver(vt);
      d.addObserver(vg);

      // créer le contrôleur
      Controleur c = new Controleur(d);

      // ranger tout dans une frame
      JFrame frame = new JFrame();
      frame.setLayout(new BorderLayout());
      c.setPreferredSize(new Dimension(100, 50));
      vg.setPreferredSize(new Dimension(200, 200));
      frame.getContentPane().add(vg, BorderLayout.NORTH);
      frame.getContentPane().add(c, BorderLayout.SOUTH);
      frame.pack();
      frame.setSize(new Dimension(300, 300));
      frame.setVisible(true);
      d.setTaille(10);
      c.setValue(10);
    }
}
```

8.5 Synthèse de l'approche MVC

L'objectif de l'approche MVC est de séparer les données, de l'affichage et du contrôle.

8.5.1 Démarche

Pour produire un modèle MVC

1. il faut dans un premier temps identifier le modèle
2. il faut développer le modèle en proposant toutes les méthodes utiles
   - méthodes privées internes pour gérer la dynamique du modèle
   - méthodes publiques extérieures correspondant aux actions possibles de l'utilisateur
3. il faut construire les vues
   - déterminer les informations du modèle à afficher et proposer des méthodes pour récupérer l'information pertinente
   - construire les vues différentes en fonction de ce que l'on souhaite
4. il faut construire le contrôleur
   - déterminer l'interface entre le modèle et l'utilisateur
   - écrire le contrôleur qui analyse les actions de l'utilisateur et appelle les méthodes adaptées du modèles
5. créer un `main` qui lie les blocs M, V et C.

8.5.2 Intérêt

Le modèle MVC présente différents intérêts

- la possibilité de développer indépendamment Modèle, Vue et Contrôleur si les méthodes utiles aux uns et aux autres sont bien spécifiées
- la possibilité de développer plusieurs interfaces graphiques (vues) ou plusieurs contrôleurs pour des utilisateurs différents (multi-modalité) ou en changer au cours d'utilisation
- avoir un code correctement structuré et facile à analyser
- pouvoir développer et tester un modèle (qui est souvent le coeur d'un projet) indépendamment de son interface graphique.
- pouvoir facilement développer de nouvelles interfaces graphiques sans remettre en cause le code existant.

8.6 Une autre approche pour MVC

Ce chapitre s'est concentré sur une approche MVC fondée sur le design Observeur/Observé. Cette approche est particulièrement adaptée au langage JAVA pour lequel ce design est implémenté dans les classes fournies avec le langage (cf figure 15).

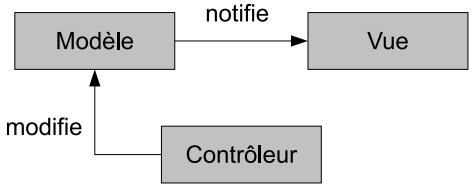


Figure 15: Approche MVC fondée sur le design Observeur/Observé

L'approche MVC peut être abordée différemment dans d'autres contextes[4]. Dans cette approche, le controleur fait office de chef d'orchestre. Il contacte le modele pour effectuer les modifications et récupérer l'information puis transmet le résultat à la vue qui se charge d'afficher les données transmises (cf Figure 16).

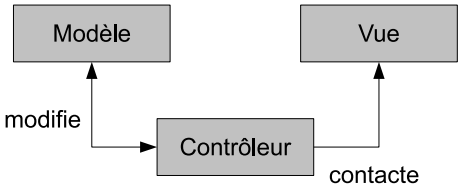


Figure 16: Approche MVC où le controleur est central

8.7 Ouverture : Les patrons de conception

Les patrons de conception ou *design pattern* (comme le design Observeur/Observé) sont des solutions éprouvées destinées à résoudre des problèmes récurrents d'architecture et de conception en informatique.

Un *design* est indépendant d'un langage de programmation. Il correspond à une structure particulière à mettre en place et se présente sous la forme de diagramme de classe.

Il existe de nombreux *design patterns* adaptés à de nombreux problèmes. Il s'agit de solutions élégantes et pratiques. N'hésitez pas à y jeter un coup d'œil de temps en temps, vous trouverez peut être la solution à un problème sur lequel vous bloquez et vous aurez une meilleure vision des bonnes pratiques.

Objectif pédagogique

A l'issue de cette section, vous devrez savoir comment

- construire une interface graphique modulaire
- structurer une application flexible en terme de modèle, contrôle et vue

Références

- pour les design pattern : la page de wikipedia en résume un certain nombre `http://fr.wikipedia.org/wiki/Patron_de_conception`
- le livre de référence sur les design pattern `'Design Patterns - Catalogue de modèles de conceptions' de E. Gamma, R. Helm, R. Johnson, J. Vlissides`
- un livre très facile d'accès sur les design pattern (avec un chapitre dédié au MVC) `'Design patterns, tête la première', chapitre 3, Oreilly, 2005`

[4]Comme par exemple dans le cadre d'applications web (cf cours de pragrammation web) où il est important de séparer l'affichage, la gestion des données et le controleur

9 Généricité

Pre-requis

Pour aborder cette partie, il est nécessaire

- de bien connaître l'utilisation des classes génériques (cf cours ADLJ)
- de savoir manipuler les classes génériques `ArrayList` et `HashMap`

Contenu

Dans cette partie,

- on expliquera ce qu'est la généricité et on donnera quelques définitions (type variable, etc)
- on expliquera comment utiliser la généricité (rappel de ADLJ)
- on expliquera comment définir des types génériques
- on présentera les raw types et les jokers

9.1 Vocabulaire

On distinguera plusieurs notions. Chaque notion sera désignée spécifiquement par une des expressions suivantes

- le **type générique** `ArrayList<E>`
  désigne la classe générique telle qu'elle est déclarée.
- la **variable de type** `E` dans `ArrayList<E>`
  désigne le type formel qui paramétrise la classe générique.
- le **type paramétré** `ArrayList<Integer>`
  désigne une classe générique avec un argument de type spécifié par opposition au type générique.
- l'**argument de type** `Integer` dans `ArrayList<Integer>`
  désigne le type concret à partir duquel la classe paramétrée est construite par opposition à la variable de type par opposition à `E`.

Ces différentes notions seront explicitées dans la suite du document, ce vocabulaire vous permettra de vous y retrouver si vous êtes perdu à un moment.

9.2 Définition

Une **classe générique** est une classe qui est paramétrée par un (ou plusieurs) type formels, appelé **variables de type**.

Au cours de ce chapitre, on utilisera comme exemple la classe générique `Couple` que nous définirons. Cette classe permet de construire des couples d'objets constitués de deux valeurs de même type.

9.3 Utilisation

9.3.1 Déclaration

Déclaration d'une classe générique A la déclaration de la *classe générique*, on utilise une **variable de type** habituellement représentée par une simple lettre majuscule.

Cette variable de type ne correspond à aucune classe, mais permet de référencer le nom de la classe que l'on passera en paramètre (comme les paramètres formels pour la déclaration d'une méthode). On parle de **type formel** puisqu'il ne correspond à aucun type réel.

Le code source suivant permet de déclarer la classe générique `Couple`. `T` correspond à la variable de type.

```
class Couple<T>
{
  T premier;
  T deuxieme;
}
```

Suite à cette déclaration, la classe générique `Couple<T>` possède deux attributs `premier` et `deuxième` tous deux de type `T`. Ce type ne sera connu qu'à l'instanciation d'un objet de type `Couple` (cf chapitre 9.3.2).

Déclaration de constructeur Un constructeur se déclare sans avoir besoin de repréciser la *variable de type* (elle est déclarée dans la classe).

```
class Couple<T>
{
  T premier;
  T deuxieme;

  public Couple()
  {
    premier=null;
    deuxieme=null;
  }
}
```

Déclaration de méthode Il est possible de déclarer une méthode prenant en paramètre un objet de type *variable de type* ou retournant des objets de type *variable de type*. Pour cela, on réutilise simplement la lettre représentant la *variable de type* comme s'il s'agissait d'un type classique.

```
class Couple<T>
{
  T premier;
  T deuxieme;

  public T getPremier()
  {
    return(premier);
  }

  public void setPremier(T v)
  {
    premier=v;
  }
}
```

Par exemple, la méthode `public T getpremier()` retourne l'attribut `premier` dont le type qui correspond à la *variable de type* est pour le moment inconnu.

Contraintes sur la variable de type La *variable de type* peut être utilisée au sein de la classe générique comme un type normal modulo certaines exceptions

- Il n'est pas possible d'appeler un constructeur sur une *variable de type* (ex: `new T()`)
- On ne peut pas utiliser une *variable de type* à droite d'un `instanceof` (ex: `o instanceof T`)
- Il n'est pas possible de déclarer un tableau de *variable de type* p(ex: `new T[5]`). Par contre, il est possible d'utiliser une classe générique comme les `ArrayList` (cf paragraphe suivant)

De la même manière, il est possible de faire des `cast` à partir de la variable de type. C'est d'ailleurs comme cela que la classe `ArrayList<E>` fonctionne en interne. Cette classe possède un tableau d'objets en attribut dans lequel elle peut ajouter les objets de type E. Lorsqu'la méthode `get` est appelée, la classe récupère l'`Object` dans le tableau et le `cast` en `E` pour retourner une valeur correctement typée.

Réutilisation de variable de type Dans une classe générique, il est possible de réutiliser une *variable de type* comme *argument de type* d'une autre classe générique.

Par exemple, on peut souhaiter que la classe générique `Couple` dispose d'une méthode retournant le couple sous la forme d'une `Arraylist`.

```
class Couple<T>
{
  T premier;
  T deuxieme;

  public ArrayList<T> getliste()
  {
    ArrayList<T> l=new ArrayList<T>();
    l.add(premier);
    l.add(deuxieme);
    return(l);
  }
}
```

Ainsi, si `cs` est un objet de type `Couple<String>`, la méthode `getliste` retourne une `ArrayList<String>`. Si `ci` est un objet de type `Couple<Integer>`, la méthode `getListe()` retourne une `ArrayList<Integer>`.

9.3.2 Création d'instance

Instancier un objet d'un type générique correspond à ce qui a été présenté dans le module Approfondissement du langage JAVA.

Il s'agit d'attribuer un type appelé **argument de type** à chaque *variable de type*. La *variable de type* est fixée pour cet objet: c'est comme si toutes les occurrences des *variables de type* étaient remplacées par l'*argument de type* passé lors de l'instanciation.

```
Couple<String> c=new Couple<String>();
```

La variable `c` est donc de **type paramétré** `Couple<String>`.

Un objet ne peut jamais être de type générique `Couple<T>` [5]. Dans certaines circonstances, il peut éventuellement être de type `Couple` (cf chapitre ultérieur sur les `raw-type` 9.6)

9.3.3 Manipulation

Lorsqu'on dispose d'un objet issu d'une classe générique, on le manipule comme un objet classique si ce n'est que toutes les occurrences de la *variable de type* sont remplacées par l'*argument de type* spécifié à l'instanciation de l'objet.

Il est à noter que la vérification des types des classes génériques se fait lors de la la compilation des classes et non pas lors de l'exécution de l'application.

[5] A moins, bien sûr que `T` ne soit défini précédemment, par exemple si `Couple<T>` est l'attribut d'une classe générique ayant `T` en variable de type.

```
Couple<String> c=new Couple<String>();
String s=c.getPremier();
```

- `c` est de type `Couple<String>`
- L'attribut `premier` et l'attribut `second` de `c` sont de type `String`
- La méthode `getPremier` retourne un objet de type `String`
- la méthode `setPremier` prend en paramètre un objet de type `String`

9.4 Variables de type

Maintenant que vous connaissez le fonctionnement élémentaire des types génériques, nous allons aborder quelques points plus subtils concernant les *variables de type*.

9.4.1 Récursivité

Une classe générique peut prendre comme *argument de type* une *classe paramétrée* (c'est à dire possédant elle aussi son propre *argument de type*).

Supposons que la classe générique `Couple<T>` ait été déclarée comme précédemment, il est possible de l'utiliser pour avoir un objet `cc` de type *couple de couple de String*.

```
Couple<Couple<String>> cc=new Couple<Couple<String>>();
```

Ainsi, l'attribut `premier` de l'objet `cc` est de type `Couple<String>` et possède lui-même un attribut `premier` de type `String`.

On peut bien sur réitérer l'opération aussi souvent que l'on souhaite. Il est donc possible de faire des Arraylist de Couple d'Arraylist de Couple de String.

```
ArrayList<Couple<Arraylist<Couple<String>>>> a;
```

9.4.2 Variables de type contraintes

A l'intérieur d'une classe générique, on ne possède aucune information sur la *variable de type* puisqu'il peut s'agir de n'importe quelle classe.

Cela peut poser des problèmes. Imaginons que l'on souhaite faire des listes d'objets affichables. Il pourrait être intéressant d'avoir une méthode `afficherListe()` qui permet d'afficher tous les objets de la liste. Cette méthode consisterait simplement à parcourir la liste pour appeler la méthode `afficher()` sur chacun des objets contenu dans la liste. Or, à la compilation, JAVA ne peut pas garantir que la méthode `afficher()` existe pour toute variable de type `T`. Une erreur de compilation est donc générée.

Pour éviter cela, il est possible d'imposer des contraintes d'héritage sur la *variable de type* en obligeant la *variable de type* à hériter d'une classe ou à implémenter une ou plusieurs interfaces. On utilise pour cela le mot clef `extends` derrière la variable de type lors de la déclaration de la classe générique.

Si on suppose que l'interface `Affichable` existe et définit une méthode `afficher()`, il est possible d'écrire.

```
public class ListeAffiche<T extends Affichable>   //1
{
  //attribut liste d'objets                        //2
  ArrayList<T> l;                                  //3

  //constructeur
  ListeAffiche()                                   //4
  {
  l=new ArrayList<T>();                            //5
  }

  //afficherliste
  public void afficherliste()                      //6
  {
    for (int i=0;i<l.size();i++)                   //7
      {
      l.get(i).afficher();                         //8
      }
  }
}
```

Cette déclaration ne fonctionnerait pas si on oubliait le `extends Affichable` à la ligne 1. Java donnerait une erreur à la compilation ligne 8, la variable `l.get(i)` de type `T` ne possédant pas la méthode `afficher()`.

9.4.3 Classe générique avec plusieurs variables de type

Il est possible de créer des classes génériques paramétrées par plusieurs *variables de type*. Vous connaissez déjà la classe `HashMap` qui nécessite de spécifier le type de la clef et le type des valeurs.

On déclare la classe générique de la même manière en séparant les variables de type par des virgules.

```
public class CoupleBis<A,B>{
  A premier;
  B deuxieme;
}
```

Les attributs `premier` et `deuxième` de la classe générique `CoupleBis<A,B>` peuvent être de types différents (cf exercice construction de `HashMap`).

```
Couple<Integer,String> cis;
```

La variable `cis` a un attribut `premier` de type `Integer` et un attribut `deuxieme` de type `String`.

9.5 Sous-typage

9.5.1 Sous-typage entre classes génériques

Il est possible de définir une classe générique par héritage (ou implémentation) d'une autre classe (interface) générique. Par exemple, parmi les classes JAVA de `java.util`, `ArrayList<E>` implémente `List<E>`.

Cette relation d'héritage est conservée pour tous les types paramétrés construits à partir de ces types génériques. Ainsi,`ArrayList<String>` implémente `List<String>`, `ArrayList<Integer>` implémente `List<Integer>`,...

9.5.2 Sous-typage entre classes paramétrées

Par contre, même si un argument de type `A` hérite d'un argument de type `B`, la classe paramétrée par `A` **n'hérite pas** de la classe paramétrée par `B`.

Par exemple, bien que `Integer` hérite de `Number`, `ArrayList<Integer>` **n'hérite pas** de `ArrayList<Number>`. Dans le cas contraire, on pourrait faire les opérations suivantes

```
ArrayList<Integer> li;
li=new ArrayList<Integer>();
ArrayList<Number> ln=li;      //si la relation d'heritage fonctionnait
Double n;
ln.add(n);                    //on ajouterait un Double dans une liste d'integer
```

Ces opérations poseraient problème puisqu'on pourrait ajouter à une liste d'`Integer` un `Number` qui ne serait pas un `Integer` .

En conséquence, si on dispose d'une méthode prenant en paramètre une liste de `Number`, il n'est pas possible de passer une liste d'`Integer` en paramètre (puisqu'il n'existe pas de relation d'héritage entre ces deux classes).

9.5.3 les jokers

Dans certains cas, il peut néanmoins être utile de traiter de la même manière des types paramétrés avec des arguments de type différents.

Par exemple, on souhaite avoir une méthode permettant de trier les éléments d'une liste (qu'elle soit constituée de `Number`, d'`Integer` ou de `Double` puisque toutes ces classes héritant de `Number` possèdent les mêmes méthodes de comparaison). Or nous avons vu dans le chapitre précédent qu'une méthode `tri(ArrayList<Number> l)` ne pourrait pas prendre en paramètre une `ArrayList<Integer>`.

Il est possible de déclarer une méthode qui prend en paramètre une liste avec des contraintes sur l'argument de type. On utilise pour cela un **Joker** représenté par le caractère `?`.

Exemple : `public void trier(ArrayList<? extends Number>)`
La méthode `trier` prend en paramètre une `ArrayList` ayant pour argument de type un type héritant de `Number`.

```
ArrayList<Number> ln;
ln= new ArrayList<Number>();
classe.afficheSomme(ln);    //on peut passer une liste de number en parametre
ArrayList<Integer> li;
li= new ArrayList<Integer>();
classe.afficheSomme(li);    //on peut passer une liste d'integer en parametre
```

Les jokers servent à beaucoup d'autres choses. Pour plus d'informations, se référer aux références en fin de chapitre.

9.6 Raw types

Afin de maintenir une compatibilité ascendante (c'est à dire permettre aux programmes écrits avant l'apparition de la généricité en Java de continuer à fonctionner sous Java5), les concepteurs de Java ont souhaité garder les types `ArrayList, List, ...` en parallèle aux types génériques.

Ces types sont ce que l'on appelle des `raw type`. Un `raw type` (de l'anglais raw = brut) est un type générique instancié sans préciser d'argument de type.

Exemple: `ArrayList l=new ArrayList();`

`l` se comporte a peu de chose prés comme un `ArrayList<Object>` sauf que les types paramétrés (comme `ArrayList<String>`) sont considérés somme des sous-types du `rawtype`.

Les opérations suivantes sont donc permises par le compilateur mais génèrent des erreurs à l'exécution.

```
ArrayList l;                                    // rawtype
ArrayList<String>ls=new ArrayList<String>(); // type parametre
l=ls;                                           // en passant par le rawtype,
l.add(5);                                       // on elimine les contraintes de type
String s=ls.get(0);                             // erreur non détectée à la compilation
```

Il faut donc faire très attention et éviter l'utilisation de rawtype. Le compilateur java génère un warning dés qu'un rawtype est utilisé et il est possible d'avoir plus de détails en compilant avec l'option `-Xlint`.

9.7 'Template method' pattern

9.7.1 Présentation

Comme vous l'avez vu en approfondissement du langage java, les algorithmes des classes génériques sont souvent fondés sur des méthodes possédées par les arguments de type. Cette structuration particulière est proche du design "template method" (juste pour information) ou 'patron de méthode' en français.

Un patron de méthode définit définit le squelette d'un algorithme à l'aide d'opérations qui seront définies ultérieurement.

9.7.2 Exemple sur ArrayList

Par exemple, la méthode `contains(E elem)` de la classe générique `ArrayList<E>` fonctionne de cette manière. De manière grossière, son algorithme est le suivant

- parcourir la liste des elements
- pour chaque element e
  - si (elem.equals(e)) alors retourner vrai
- retourner faux

Ainsi, le comportement réel de la méthode `contains` dépend de l'implémentation de la méthode `equals` dans l'argument de type.

- **Exemple 1** Si on définit une classe `Point` de la manière suivante

```
public class Point {
  int x;
  int y;

  public boolean equals(Object arg0) {
    Point p=(Point) arg0;
    return ((x==p.x)&&(y==p.y));
  }
}
```

La méthode contains renverra `true` si on cherche un point de mêmes coordonnées qu'un point déjà présent dans la liste.

- **Exemple 2** Si la classe Point est définie de la manière suivante

```
public class Point {
  int x;
  int y;

  public boolean equals(Object arg0) {
    Point p=(Point) arg0;
    return ((x==p.x));
  }
}
```

La méthode `contains` retournera `true` si on cherche un point ayant la même abscisse qu'un point déjà présent dans la liste (puisque les points sont considérés comme égaux - comme le montre la définition de la méthode `équals`)

- **Exemple 3** Si la classe `Point` est définie de la manière suivante

```
public class Point {
  int x;
  int y;
}
```

La méthode `contains` retournera `true` si et seulement si la reference du point que l'on cherche est la même que celle d'un des points présents dans la liste (comme la méthode `equals` n'est pas redéfinie, c'est la méthode de la classe `Object` qui est utilisée)

9.7.3 Conséquence

En redéfinissant certaines méthodes dans les classes passées en argument de type, le comportement de la classe générique est grandement modifié du fait de la liaison dynamique.

C'est grâce à ce mécanisme que les méthodes de tri peuvent être des méthodes de tri génériques (par ex, il suffit juste de définir un ordre sur les classes passées en paramètre d'un `SortedSet`)

Essayez d'utiliser des méthodes redéfinissables lorsque vous écrivez une classe générique (par exemple préférez utiliser la méthode `equals` plutôt que de faire un test `==` sur les références). Gardez cela en tête lorsque vous écrivez ou utilisez des classes génériques sinon vous risquez d'avoir des comportements inexplicables (la méthode `equals` est redéfinie dans la classe `String`, mais pas dans une classe que vous écrirez possédant un attribut de type String).

Si vous avez écrit la classe `Chaine`

```
public class Chaine {
  String s;
}
```

les classes paramétrées `ArrayList<Chaine>` et `ArrayList<String>` se comporteront différemment (puisque `equals` est redéfini dans `String` et non dans `Chaine`)

Objectif pédagogiques

A l'issue de cette section, vous devez

- savoir distinguer classe générique, classe paramétrée, variable de type et argument de type
- savoir déclarer des classes génériques complexes
- savoir utiliser des classes génériques
- comprendre les relations de sous-typage entre classes génériques
- savoir utiliser un joker
- avoir quelques éléments de compréhension sur les rawtype

Références

Voici quelques liens sur le thème de la généricité pour approfondir vos connaissances

- des transparents complets sur la généricité et les problèmes de sous-typage
  `http://deptinfo.unice.fr/~grin/messupports/java/Genericite6.pdf`
- le design pattern 'patron de méthode' sur wikipedia
  `fr.wikipedia.org/wiki/Patron_de_mthode_(patron_de_conception)`
- un livre très accessible sur les design pattern avec 'patron de méthode'
  `'Design patterns, tête la première', chapitre 8, Oreilly Editions, 2005`
  actuellement épuisé mais facilement empruntable en bibliothèque universitaire.

Part IV
Conclusion et Perspectives

Le module complément JAVA avait pour objectif de présenter de nouveaux outils et de nouvelles pratiques pour permettre de développer des applications JAVA plus abouties.

Bien entendu, il ne s'agit que de quelques pistes à creuser et beaucoup d'autres éléments auraient pu être présentés. Cette section contient plusieurs directions qui pourraient intéresser la personne soucieuse de développer ses compétences en JAVA[6].

Je vous conseille néanmoins d'avoir des bases solides en Java sur les concepts fondamentaux (contenus des différents modules de Java de l'IUT: BDLP, PO, ADLJ, IG et Cpt java) avant de creuser ces perspectives. Il ne s'agit pas de vous noyer dans de nombreuses directions sans maîtriser l'essentiel.

De la même manière, certaines de ces notions seront ou ont été abordées dans d'autres modules de votre formation. Utilisez bien entendu avant tout votre polycopié de référence, mais n'hésitez pas à regarder d'autres documents présentant les mêmes notions.

Quelques références sur Java

Les ouvrages suivants présentent le langage Java et peuvent constituer de bons points de départ pour compléter vos connaissances[7]

- *"Java tête la première"* de Kathy Sierra et Bert Bates, Oreilly Edition: livre extrêmement pédagogique qui pose les bases du langage Java.
- *"Penser en java, 2nd edition"*, livre électronique, traduction française de "thinking in java" de Bruce Eckel, la version électronique est disponible à l'adresse `http://penserenjava.free.fr/`.
- *"Développons en Java"* de Jean-Michel Doudoux, extrêmement complet sur le langage Java, la version electronique est disponible à l'adresse `http://jmdoudoux.developpez.com/cours/developpons/java/`.

Conception d'applications

UML

UML est un langage de modélisation graphique permettant de représenter une application et ses composants. Il est fondamental pour communiquer sur une application orientée objet et une bonne maîtrise en est nécessaire.

Un module UML est enseigné à l'IUT, pensez à l'utiliser dans vos projets.

Design pattern

Les *designs patterns* ont été succinctement abordés dans ce module[8]. Un *design pattern* est une solution haut-niveau à un problème fréquemment rencontré en architecture logicielle.

[6] et en programmation objet de manière plus générale

[7] Et regarder ce que vous connaissez déjà sous un autre angle

[8] Ils nécessiteraient un module à part entière pour pouvoir être correctement présentés

Vous en avez déjà rencontré un certain nombre de *design pattern* comme le *pattern singleton*, le *pattern method template*, le *pattern Observeur-Observé* ou encore les *active records*, mais il en existe de nombreux autres.

Plus d'information peut être trouvée dans les ouvrages suivants

- "*Design pattern*" edition tete la premiere, de Eric Freeman, Elisabeth Freeman, Kathy Sierra et Bert Bates qui présente les *design patterns* classique de manière très progressive et pédagogique.
- "*Design Patterns: Elements of Reusable Object-Oriented Software*" de Erich Gamma, Richard Helm, Ralph Johnson et John M. Vlissides, l'ouvrage de référence sur les *design patterns*.
- "*Design Patterns du Gang of Four appliqués à Java*" de Régis Pouiller disponible à l'adresse `http://rpouiller.developpez.com/tutoriel/java/design-patterns-gang-of-four/`

Développement jeu vidéo

Java propose de nombreux outils et approches pour développer des jeux vidéos. Voici quelques pistes.

Java 3D

Java3D est une bibliothèque permettant de créer et d'animer des scènes 3D. La bibliothèque n'est pas incluse dans les bibliothèques de base en java et est disponible gratuitement à l'adresse `https://java3d.dev.java.net/`.

Une scène java3D est constituée par un arbre dont les noeuds sont des transformations ou des objets, permettant une représentation hiérarchique de la scène.

Moteur de jeu

Un jeu est essentiellement constitué par un moteur de jeu chargé de mettre à jour les données et l'affichage. La difficulté est d'avoir un taux de rendu constant pour que le jeu se déroule à la même vitesse.

"*Killer Game Programming in Java*" est un ouvrage de Andrew Davison Editions O'Reilly qui présente différentes approches pour convevoir un jeu vidéo. Une version éléctronique des brouillons du livre est disponible à l'adresse `http://fivedots.coe.psu.ac.th/~ad/jg/`.

Graphisme pour le jeu

De la même manière, l'approche pour le rendu et l'affichage doit être adaptée pour les jeux vidéos. Parmi les techniques, on peut citer entre autres

- le double buffering
- l'active rendering
- le clipping

Ces approches sont elles aussi résentées dans "Killer Game Programming in Java" de Andrew Davison Editions O'Reilly, `http://fivedots.coe.psu.ac.th/~ad/jg/`.

Outils JAVA

Introspection

L'introspection consiste à permettre aux classes d'accéder de manière dynamique à la structure des autres classes. Par exemple, une classe peut ainsi avoir accès à l'exécution aux noms des attributs et des méthodes des autres classes.

Une première présentation de l'introspection est accessible sur développez.com `http://ricky81.developpez.com/tutoriel/java/api/reflection/` et sur "*développons en Java*" de JM Doudoux `http://www.jmdoudoux.fr/java/dej/chap022.htm`.

Applets

Une *applet* java est un logiciel Java qui s'exécute dans un navigateur web. Une *applet* est chargée et exécutée par le navigateur à partir d'une URL. Les *applets* sont construites à partir de la classe `JApplet` de java.

La page tutorial de Java décrit comment concevoir des *applets* à l'adresse `http://download.oracle.com/javase/tutorial/deployment/applet/`

Servlets

Les *servlets* sont des applications java exécutée sur un serveur répondant à des requêtes http, de la même manière que les scripts php. L'intérêt des *servlets* par rapport à php est qu'il est bien sûr possible d'utiliser toutes les bibliothèques et spécificités du langage java.

Il est nécessaire d'installer une application sur le serveur capable de filtrer les requêtes http et d'exécuter des *servlets* installées. `apache Tomcat` est une telle application parmi d'autres et est disponible gratuitement sur `http://tomcat.apache.org/`.

Le tutorial de java.sun décrivant comment concevoir les *servlets* est accessible à l'adresse `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html`.

Développement de projet

Cycle de développement

Il existe de nombreux cycles de développement classiques fournissant une méthodologie pour concevoir une application. Les cycles de développement fournissent un cadre et de bonnes pratiques à suivre pour le développement d'applications en plusieurs phases. Ils peuvent ainsi vous fournir un guide extrêmement pratique lorsque vous serez amenés à développer des applications complexes.

Parmi les cycles classiques, on notera le cycle en V, le cycle en spirale et le cycle itératif.

Extreme Programming

L'*extreme programming* est une méthodologie de gestion de projet informatique caractérisée par un ensemble de bonnes pratiques. L'objectif de l'*extreme programming* est de construire des applications flexibles capables de s'adapter facilement à des besoins changeants.

Les tests unitaires présentés dans ce module font partie de ces bonnes pratiques, mais il en existe de nombreuses autres `http://fr.wikipedia.org/wiki/Extreme_programming`.

Maven

Maven est une application pour la gestion de projets logiciels en Java. L'objectif de *Maven* est de décrire intégralement un projet et ses dépendances pour que la compilation et l'exécution puisse se faire simplement. *Maven* est basé sur la notion de cycle de vie en plusieurs phases d'un projet et cherche à proposer une chaîne automatisée de traitements.

Maven permet ainsi de décrire les éléments à produire à partir des fichiers sources et l'organisation de ces éléments, comme par exemple les fichier jar, les fichiers javadoc, les test unitaires, les bibliothèques nécessaires,... Une fois qu'un projet est décrit, il suffit de lancer la compilation avec *maven* qui va compiler, tester et organiser les différents éléments du projet.

Maven et sa documentation sont disponibles gratuitement sur le site officiel `http://maven.apache.org/`. Un plugin Eclipse est aussi disponible à cette adresse.