Conception logicielle avancée

Grégory Bonnet

Université de Caen Normandie – GREYC

Plan du cours

Présentation

Principe

- Groupe de 4
- Un projet conséquent à réaliser en TP

Évaluation

- 1 note de groupe de travail (avancement sur le projet)
- 1 note de groupe pour un rapport
- 1 note de groupe pour un oral (et démonstration)
- ... modulées individuellement par votre investissement dans le groupe

Remarque

Un ou deux cours seront réservés à vos questions. Vous pouvez me suggérer (et il me semble important d'y participer) des questions à développer.

Projets

CoreWar

Éditeur de livres dont vous êtes le héros

Générateurs de flores vidéos-ludiques

Interpréteur de programme chimique

Le Castor Affairé

Rendu 3D par lancer de rayons

Simulateur pour N corps

Solveur de Ricochet Robots

La Forge et Subversion

→ `https://forge.info.unicaen.fr/` ←

Forge permet de faire circuler l'information autour du projet, de recenser les anomalies et les tâches à effectuer, de disposer d'un dépôt de sources du projet géré par Subversion (SVN).

`http://svnbook.red-bean.com/nightly/fr/svn-book.pdf`

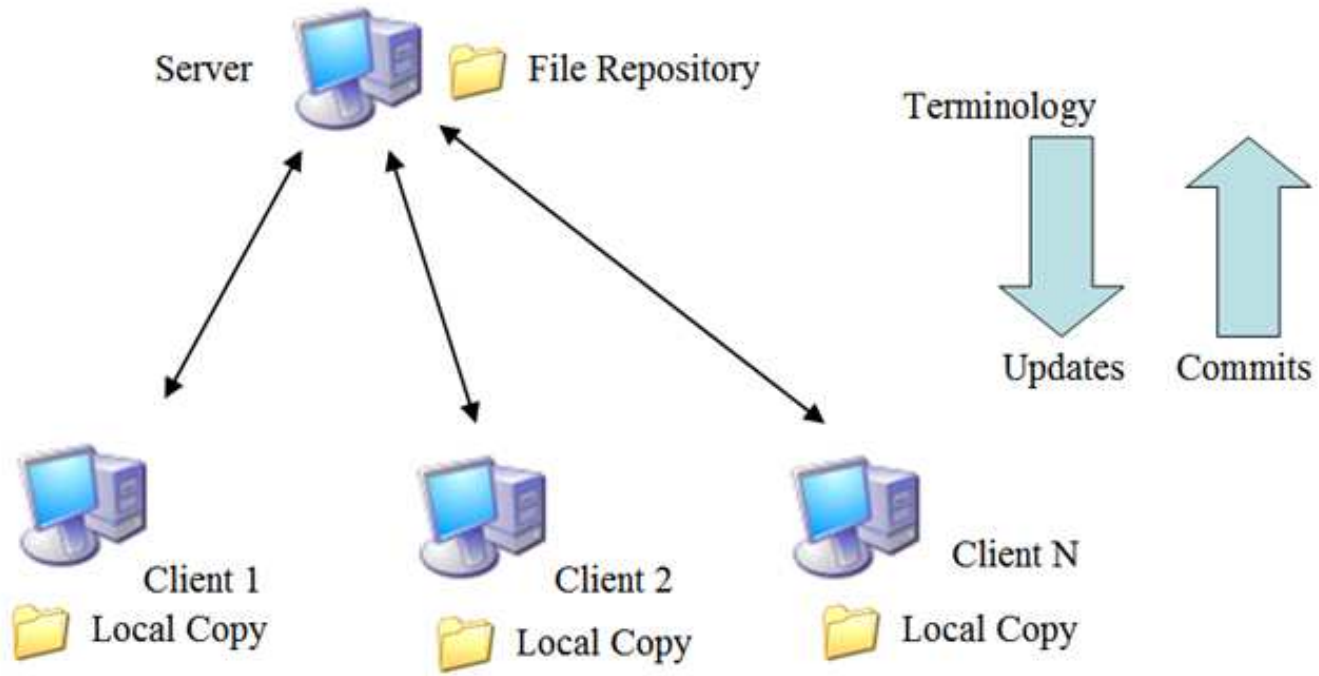
SVN est un gestionnaire d'arborescence de fichiers qui :

- garde en mémoire toutes les modifications effectuées
- permet de revenir à une version antérieure
- gère les conflits d'écriture sur un même fichier

Principes

Connection avec votre EtuP@ss

Pour Windows : `https://tortoisesvn.net/`



Commandes

Création du dépôt local (synchronisation initiale)

```
svn checkout [-username login] url
```

Mise à jour du dépôt local

```
svn update [-username login] répertoire
```

Ajout ou suppression de fichiers locaux

```
svn add nom-de-fichier
svn del nom-de-fichier
```

Mise à jour du dépôt distant

```
svn commit [-username login] [-m "commentaire"]
```

Gestion des conflits

Verrouillage d'un fichier

```
svn lock [-username login] fichier
```

Déverrouillage d'un fichier

```
svn unlock [-username login] fichier
```

Consultation du statut des fichiers

```
svn status
```

Arborescence du dépôt

```
project/
    branches/
    tags/
    trunk/
       src/
       docs/
       resources/
       ...
```

Paquetage et structure d'un projet

Qu'est-ce qu'un paquetage ?

Définition

Un paquetage est une collection de classes dans un (ou des) répertoire(s).

```
myPackage/
    mySubPackageOne/
        Classone.class
        Classtow.class
        ...
    mySubPackageTwo/
        Classthree.class
        Classfour.class
```

Importation

```
import myPackage.mySubPackageTwo.*;
```

Créer un paquetage

Préalable

Un paquetage est une collection de classes dans un (ou des) répertoire(s).

X.java

```
package myPackage;
class X{
    ...
}
```

Y.java

```
package myPackage.mySubPackage;
class Y{
    ...
}
```

Importation

Si deux paquetages proposent une classe portant le même nom, il faut nommer complètement la classe dans le fichier important les paquetages. Exemple :

```
import java.util.*;
import java.sql.*;
java.util.Date d;
```

Archives JAVA

Fichier JAR

Une archive Java (JAR) est une archive ZIP contenant les classes et les ressources nécessaires à l'exécution d'une application.

Manifeste

```
Manifest-Version: 1.0
Created-By: Gregory Bonnet
Main-Class: ExempleJar
```

Opérations

- Création : `jar cmf manifest-file jar-file input-file(s)`
- Visualiser : `jar tf jar-file`
- Extraire : `jar xf jar-file [archived-file(s)]`
- Exécuter : `java -jar jar-file`
- Réviser : `jar uf jar-file new-files(s)`

ANT

Qu'est-ce que ANT ?

C'est un logiciel de construction automatisée d'application qui se sert d'un fichier XML, décrivant des associations entre des commandes (ou cibles ou *target*) et des tâches à réaliser.

`http://ant.apache.org/`

Execution en ligne de commande

ant [option] [target]

Fichier de configuration

Par défaut, ANT cherche le fichier XML `build.xml` dans le répertoire courant et éxecute la cible indiquée dans le fichier. L'option `-buildfile [file.xml]` permet de spécifier quel est le fichier XML de configuration à lire.

Squelette ANT à télécharger

Bientôt en ligne sur eCampus.

Exemple

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="exemple" default="compile" basedir=".">

<!- Definition des proprietes du projet ->
<property name="projet.sources.dir" value="src"/>
<property name="projet.bin.dir" value="bin"/>
<property name="projet.lib.dir" value="lib"/>

<!- Definition du classpath du projet ->
<path id="projet.classpath">
     <fileset dir="$projet.lib.dir">
         <include name="*.jar"/>
     </fileset>
     <pathelement location="$projet.bin.dir" />
</path>

<!- Compilation des classes du projet ->
<target name="compile" description="Compilation des classes">
     <javac srcdir="$projet.sources.dir"
            destdir="$projet.bin.dir"
            debug="on"
            optimize="off"
            deprecation="on">
         <classpath refid="projet.classpath"/>
     </javac>
</target>

</project>
```

Exemple

```
<!- Un peu d'initialisation ->
<target name="init">
     <echo message="Initialisation de $ant.project.name"/>
     <delete dir="$basedir/gen"/>
     <mkdir dir="$basedir/gen"/>
</target>
<!- Execution de la classe EntryPoint ->
<target name="run" description="Execution" >
     <java classname="EntryPoint" fork="true">
         <classpath refid="projet.classpath"/>
     </java>
</target>

<!- Génération de la Javadoc ->
<target name="javadoc">
     <javadoc sourcepath="src" destdir="doc" >
         <fileset dir="src" defaultexcludes="yes">
             <include name="**" />
         </fileset>
     </javadoc>
</target>

<!- Création d'une archive JAR ->
<target name="packaging">
     <jar jarfile="test.jar" basedir="src" />
</target>
```

Structures de données

Collection et Map : un bestaire de structures de données

Deux paquetages importants

- `java.util.Collection` : pour gérer un groupe d'objets
- `java.util.Map` : pour gérer des éléments de type paires de clé/valeur

	Usage général	**Usage spécifique**	**Accès concurrent**
List	ArrayList LinkedList	CopyOnWriteArrayList	Vector Stack CopyOnWriteArrayList
Set	HashSet TreeSet LinkedHashSet	CopyOnWriteArraySet EnumSet	CopyOnWriteArraySet ConcurrentSkipListSet
Map	HashMap TreeMap LinkedHashMap	WeakHashMap IdentityHashMap EnumMap	Hashtable ConcurrentHashMap ConcurrentSkipListMap
Queue	LinkedList ArrayDeque PriorityQueue		ConcurrentLinkedQueue LinkedBlockingQueue ArrayBlockingQueue PriorityBlockingQueue DelayQueue SynchronousQueue LinkedBlockingDeque

Vue de haut niveau des classes et interfaces

Interface de haut niveau

- `Collection` : interface implémentée par la plupart des objets qui gèrent des collections
- `Map` : interface définissant des méthodes pour gèrer des collections sous la forme clé/valeur
- `Set` : interface pour des objets qui n'autorisent pas de doublons dans l'ensemble
- `List` : interface pour des objets qui autorisent des doublons
- `SortedSet` : interface qui étend l'interface `Set` et permet d'ordonner l'ensemble
- `SortedMap` : interface qui étend l'interface `Map` et permet d'ordonner l'ensemble

Interfaces utilitaires

- `Iterator` : interface pour le parcours des collections
- `ListIterator` : interface pour modifier les éléments lors du parcours
- `Comparable` : interface pour définir un ordre de tri naturel pour un objet
- `Comparator` : interface pour définir un ordre de tri quelconque

Classes principales

- `HashSet` : `Hashtable` qui implémente l'interface `Set`
- `TreeSet` : arbre qui implémente l'interface `SortedSet`
- `ArrayList` : tableau dynamique qui implémente l'interface `List`
- `LinkedList` : liste doublement chaînée qui implémente l'interface `List`
- `HashMap` : `Hashtable` qui implémente l'interface `Map`
- `TreeMap` : arbre qui implémente l'interface `SortedMap`

Type générique

Un type générique est une classe paramétrée par un type. Les collections sont toutes des types génériques : il faut spécifier quel type d'objet elles vont pouvoir contenir.

Exemple avec `ArrayList`

```
import java.util.ArrayList;
import java.util.List;

public static void main(final String[] args) {
     List<String> list = new ArrayList<String>();
     list.add("1");
     list.add("2");
     list.add("3");
     for (String element : list) {
         System.out.format("%s", element);
     }
}
```

Complexité des opérations

Classe	get	add	contains	next	remove
ArrayList	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$
LinkedList	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
HashMap	$O(1)$	$O(1)$	$O(1)$	$O(h \div n)$	$O(h \div n)$
LinkedHashMap	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
HashSet	–	$O(1)$	$O(1)$	$O(h \div n)$	$O(h \div n)$
LinkedHashSet	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$
TreeSet	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
PriorityQueue	$O(1)$	$O(\log n)$	–	–	$O(\log n)$

Cas particuliers

Pour la `PriorityQueue`, il faut utiliser la méthode `offer(E o)` à la place de `add(E o)` et `peek()` ou `poll()` à la place de `get()`. Pour les `Map`, il faut utiliser la méthode `put(K a, V b))` à la place de `add`. La méthode `contains` se décline en `containsKey(Object k)` et `containsValue(Object v)`.

Exemple : une HashMap pour la bande d'une machine de Turing

```
import java.util.TreeMap;
import java.util.Map;
import java.util.Random;

public static void main(final String[] args) {
     Map<Integer, Boolean> band = new HashMap<Integer, Boolean>();
     band.put(0,true);
     head = new Integer(0);
     random = Random();

     // avance sur la bande jusqu'à trouver un false
     while (band.get(head)) {
         head = new Integer(head.intValue() + 1);
         if (!band.containsKey(head) {
             band.put(head, new Boolean(random.nextBoolean()));
         }
     }
}
```

Tester un logiciel

Tout cycle de développement passe par une ou plusieurs étapes de tests

Un projet type peut comporter

- 10000 lignes de code
- 15 personnes
- 10 mois de travail

Coût d'un bogue

- coût économique : 64 milliards de $ par an rien qu'aux US (2002)
- coût humains ou environnementaux (Ariane 5, panne de réseau, etc.)

Difficile de tester sans perdre trop de temps tout étant sûr de bien tester

- Que tester ?
- Comment s'assurer que les tests pratiqués seront les bons ?
- Comment s'assurer que l'on a rien oublié de tester ?

Qu'est-ce qu'un test ?

Entité à tester	fonction classe module logiciel
Cond. initiales	valeurs des variables globales valeurs des attributs fichiers ouverts fichiers fermés
Stimuli	arguments d'une fonction saisie au clavier exception générée
Effets attendus	valeurs retournées affichages valeurs non modifiées

Un ensemble de tests sur une même entité est un jeu de tests

Qu'attend-on d'un test et que faire ?

Propriétés désirables

1. chercher à écrire des jeux de tests automatisables
2. définir des jeux de tests rapides à exécuter
3. les résultats doivent être facilement interprétables
4. tester l'entité dans l'environnement réel d'utilisation

Que faut-il tester ?

1. ce que le programme doit faire (évidemment)
2. ce que le programme ne doit pas faire
3. les spécifications implicites
4. le comportement en cas d'erreurs ou d'exception

Limites des tests exhaustifs et des preuves pour la vérification et la validation

Idéalement : prouver que l'objet est sans défaut et conforme aux spécifications

- la vérification consiste à vérifier que l'on a bien fait le logiciel (absence de défauts)
- la validation consiste à vérifier que l'on a fait le bon logiciel (réponse aux attentes)

Preuves formelles

Les preuves formelles sont restreintes à certaines classes de programmes car il est prouvé qu'il n'existe pas de méthode automatique de preuve pour tous les programmes. De plus, elles ne peuvent comparer un programme qu'à des spécifications formelles

Test exhaustifs

Les tests exhaustifs sont des jeux de tests qui portent sur toutes les conditions initiales et tous les stimulis possibles. En pratique, ils sont impossibles car, par exemple, il faut 2^{80} tests pour une entrée de 10 octets

Différents types de tests selon leur granularité

Tests unitaires

Tests des objets indépendamment les uns des autres

Tests d'intégration

Tests de la bonne coopération d'objets préalablement testés unitairement. On peut les réaliser de manière incrémentale, en ajoutant les objets un à un et en simulant les objets non encore ajoutés, ou de manière globale.

Tests du système

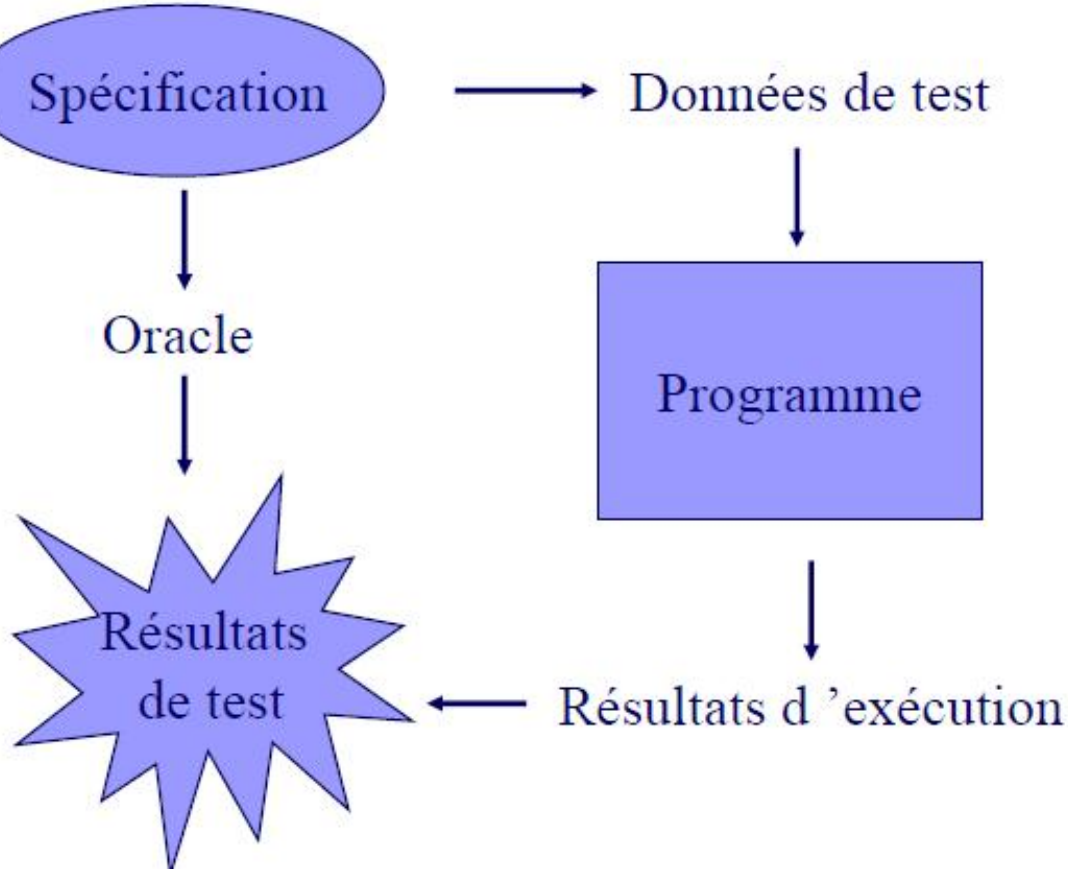
Tests du logiciel dans sa globalité (tests de recettes, tests de performance, de sécurité, d'installation)

Tests de non-régression

Tests destinés à vérifier qu'une correction ou modification n'a pas fait apparaître de nouveaux défauts. Cela consiste à refaire passer les tests précédents pour tous les objets en relation de dépendance en plus des nouveaux tests pour les objets ou fonctionnalités ajoutés ou modifiés

Conformité par rapport à la spécification sans tenir compte de l'implantation

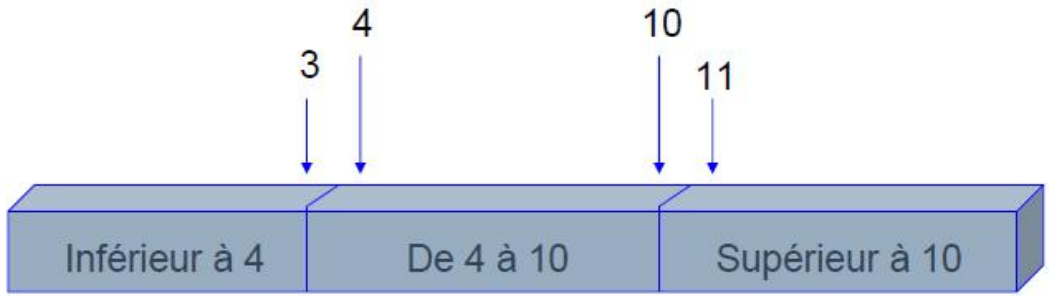
- tests de types « boîte noire » ou « black-box testing »
- ils peuvent être employés pour la vérification comme pour la validation



Tests aux limites

Définition

Les *tests aux limites* étudient le comportement des entités logicielles lorsque les entrées sont situées aux bornes de leurs intervalles de définition.



Exemple

Pour une boucle, il faut tester les valeurs provoquant aucun passage, un ou deux passages, le nombre maximum de passages ainsi que les cas limites avec un nombre de passage négatif et au-delà du maximum.

Avantages et limites

- produit des tests nominaux (à l'intérieur) et de robustesse (à l'extérieur)
- ne couvre pas nécessairement l'ensemble des comportements

Règles de construction de jeu de tests

1. l'entrée appartient à un intervalle

- un test pour les 2 valeurs correspondant aux 2 limites de l'intervalle
- un test pour les 4 valeurs correspondant aux valeurs des limites $\pm\ \epsilon$
- **exemple :** si $n \in [0 \ldots 10]$ il faut tester $\{0, 10, -1, 1, 9, 11\}$

2. l'entrée est un ensemble ordonné de valeurs

- un test pour le 1^{e} et le 2^{e} élément
- un test pour le dernier et l'avant-dernier élément
- **exemple :** si $n \in \{-1, 2, 4, 8, 16\}$ il faut tester $\{-1, 2, 8, 16\}$

3. l'entrée spécifie un nombre de valeurs

- un test pour les nombres minimum et maximum
- un test pour des nombres invalides qui sont hors limites
- **exemple :** cf. transparent précédent

Tests partitionnels

Analyse partitionnelle

L'*analyse partitionnelle* est une méthode qui vise à diminuer le nombre de cas de tests par calcul de classes d'équivalence. Une *classe d'équivalence* regroupe des ensembles de stimuli supposés produire le même comportement.

Conception du jeu de test

1. calculer les classes d'équivalence pour chaque entrée
2. choisir un représentant pour chaque classe d'équivalence
3. établir les stimuli en faisant le produit cartésien des représentants

Exemple

Soit une fonction qui prend en entrée trois entiers représentant les longueurs des côtés d'un triangle et retourne s'il s'agit d'un triangle équilatéral, isocèle ou scalène et son angle le plus grand est aigu, obtus ou droit. Nous avons donc 8 classes d'équivalence : {scalène aigu}, {scalène obtus}, {scalène droit}, {isocèle aigu}, {isocèle obtus}, {isocèle droit}, {équilatéral} et {non triangle}.

Règles de partitionnement des domaines

1. l'entrée appartient à un intervalle
   - une classe pour les valeurs inférieures
   - une classe pour les valeurs supérieures
   - n classes valides
2. l'entrée est un ensemble de valeurs
   - une classe avec l'ensemble vide
   - une classe avec trop de valeurs
   - n classes valides
3. l'entrée est une obligation ou une contrainte (forme, sens, syntaxe)
   - une classe avec la contrainte respectée
   - une classe avec la contrainte non-respectée

Tests aléatoires

Définition

Un *test aléatoire* est une méthode qui consiste à tester un grand nombre de valeurs, choisies aléatoirement, pour les stimuli.

1. une **loi statistique** représentative du domaine des entrées
2. un **oracle** permettant de vérifier l'exactitude du résultat

Exemple

- loi normale pour la taille d'une personne
- loi de Poisson pour la taille d'un tampon

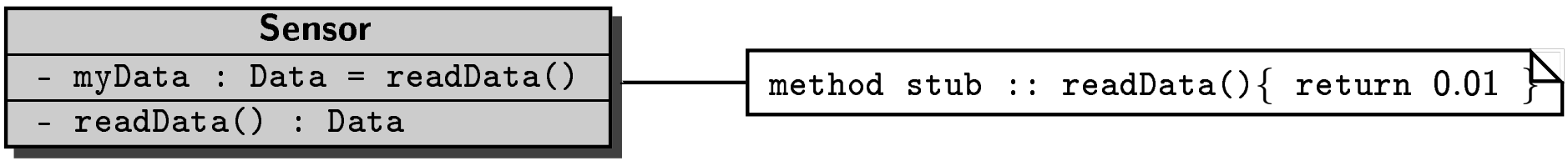
Avantages et limites

- facilement automatisable pour la sélection d'un jeu de test
- fonctionnement aveugle mais représentatif d'un comportement réel
- difficulté à produire des comportements spécifiques

Remplacement de code (*method stub*)

Pour chaque variable de contexte de la classe à tester

- définir une méthode qui permet de l'obtenir
- remplacer le contenu de cette méthode par du code qui génère le jeu de test



Limites

- nécessite des pré-compilations ou des recompilations
- pas très objet tout ça...

Objets blancs (*mock objects*)

Pour chaque variable de contexte de la classe à tester

- définir un objet qui permet de l'obtenir
- définir un objet qui simule son obtention (objet blanc)
- unifier ces objets avec une interface ou une classe abstraite
- l'objet à tester ne « voit » que l'interface

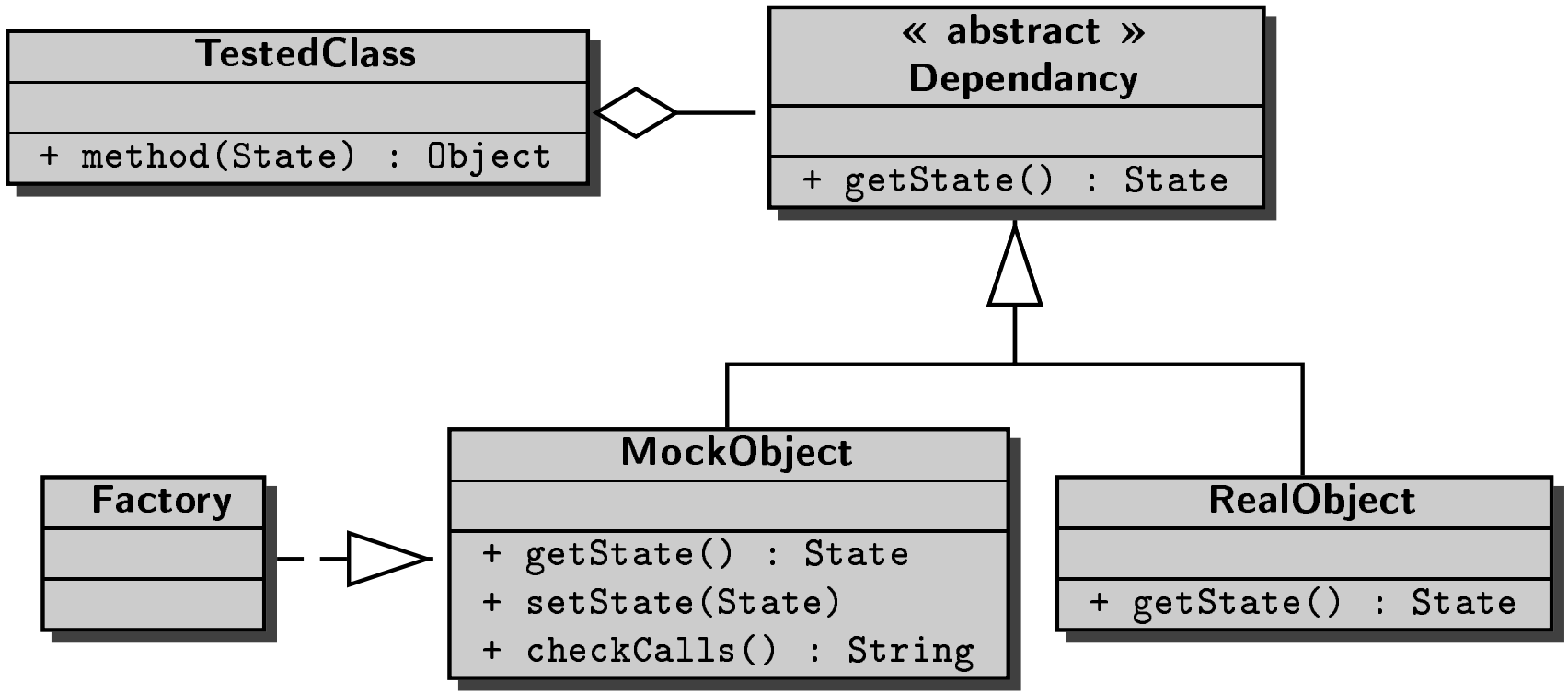
Utilité

- pour les tests aléatoires ou difficiles à produire
- pour ne pas avoir à initialiser de vrais objets (base de données)
- pour modifier dynamiquement son comportement
- pour inclure des informations supplémentaires (log)

Limites

- faire attention aux renommages de l'interface ou de la classe abstraite
- difficile à utiliser si l'objet réel provient d'un autre projet

Patron de conception d'un objet blanc



Programmation par contrat

Programmation par contrats

Qu'est-ce que qui est attendu d'une fonction (autre que sa correction) ?

1. préconditions
2. postconditions
3. invariants

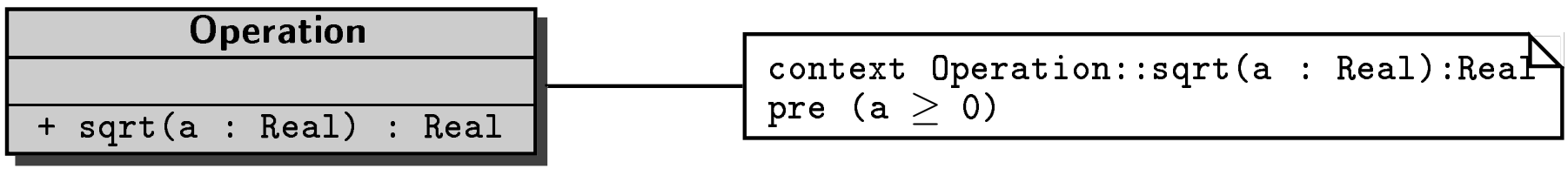
Ces contraintes (contrats) doivent être documentées

- ajout de notes dans le diagramme de classes
- ajout de commentaires dans le code
- spécification formelle dans certains langages (Effeil, OCL)

Préconditions

Définition

Les préconditions d'une méthode sont les conditions d'utilisation de la fonction spécifiées par son concepteur et que les utilisateurs de la méthode doivent respecter. Il s'agit généralement de conditions sur les valeurs des arguments.



Tester la satisfaction de la précondition

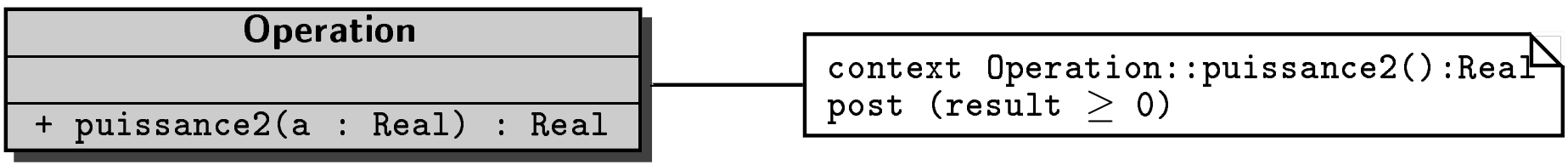
1. avant l'appel de la méthode $\longmapsto$ confiance dans l'utilisateur
2. dans le corps de la méthode et lever une exception $\longmapsto$ traitement supplémentaire

Dans tous les cas, il faut documenter le corps de la méthode

Postconditions

Définition

Les postconditions d'une méthode sont des propriétés toujours vraies une fois la méthode exécutée. Ceci concerne en général les effets de bord comme la modification d'un argument, d'un autre membre de la classe ou d'un code de retour.

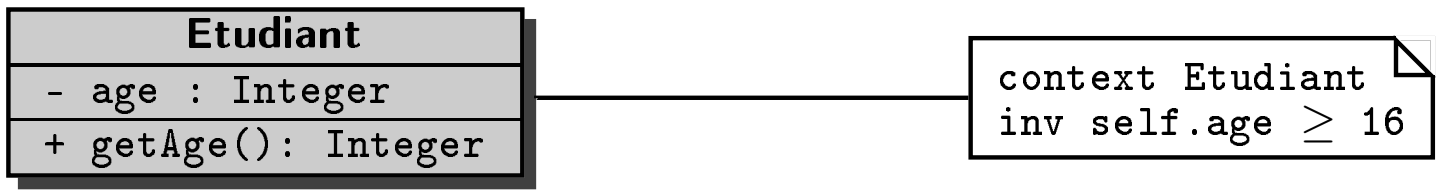


La satisfaction de la postcondition doit toujours être testée dans la méthode

Invariants

Définition

Les invariants d'une classe sont des propriétés toujours vraies que toute instance de cette classe doit respecter.



Comment vérifier la satisfaction des invariants ?

- utilisation de la précompilation (C++)
- utiliser une méthode privée appelée par toutes les méthodes publiques de la classe
- certains langages possèdent une fonction `assert` (Effeil, Java 1.5)

Le mot-clé assert

Syntaxe

```
assert condition [: objet] ;
```

Exécution

```
java -ea:myPackage.myClass -da:myPackage.myOtherClass EntryPoint
```

Que faire avec ce mot-clé ?

- Préconditions et post-conditions
- Invariants logiques
  $\mapsto$ vérifier qu'une expression ne devient pas fausse si le programme était modifié
- Invariants de contrôles de flux
  $\mapsto$ vérifier que le programme ne passe jamais pas un point donné
- Invariants fonctionnels
  $\mapsto$ vérifier que les attributs d'une classe sont bien initialisés

La classe Logger

Initialisation

```
private static final Logger logger = Logger.getLogger("Nom du logger");
```

Remarque : le nom du `logger` l'identifie dans toute l'application (objet statique)

Utilisations simples

- `Logger.info(String msg)`
- `Logger.severe(String msg)`
- `Logger.warning(String msg)`

Redirection vers un fichier

```
FileHandler filename = new FileHandler("file.log");
logger.addHandler(filename);
```

Un peu de parallélisme

Threads

Qu'est-ce que c'est ?

- Traitements parallèles qui s'exécutent au sein d'un même processus
- Partagent une même mémoire
- S'exécutent en temps partagé (pseudo-parallélisme)

Usage

- faire des traitements en tâche de fond
- accélérer des traitements longs qui n'utilisent pas les mêmes ressources
- éviter les appels bloquants dans les interfaces graphiques

La classe Thread

```
import java.lang.*;

class myThread extends Thread {
       public void run() {
         // code à exécuter
     }
```

Méthodes principales

- `start()` : lance le thread qui exécute son `run`
- `setDaemon(boolean)` : l'arrêt du programme principal arrête le thread
- `setPriority(int)` : modifie la priorité (sans garantie de contrôle)
- `sleep(long)` : arrête le thread pendant un temps en millisecondes

L'interface Runnable

```
import java.lang.*;

class myTreatment implement Runnable {

    public void run() {
        // code à exécuter
    }

public static void main(final String[] args) {
    Thread myThread = new Thread(new myTreatment());
    myThread.start();
```

Intérêt

- permet d'hériter au besoin d'une classe
- meilleure séparation des rôles
- évite les erreurs de polymorphisme
- permet d'utiliser une unique instance `Runnable` pour plusieurs `Thread`

Rapports et soutenances

Soutenances

Calendrier

- remise des rapports :
  - le mardi 07/04/20 pour les groupes 1A, 1B, 3A, 3B, 4A
  - le vendredi 10/04/20 pour les groupes 2A, 2B et 4B
- soutenances : entre le 27/04/20 et le 30/04/20

Organisation

- Une demi-journée par groupe de TP
- L'ordre de passage est à déterminer avec l'encadrant de TP
- 15 minutes de présentation, 10 minutes de questions, 5 minutes de délibération

Principe

- Votre code, votre diaporama, votre rapport doivent être sur la Forge
- **Le rapport est à déposer lors de votre dernière séance de TP**
- Venez avec votre machine pour la présentation et la démonstration

Plan-type d'un rapport (15 à 20 pages)

1. Objectifs du projet
   - Description du concept derrière l'application
   - Ce qu'il fallait faire
   - Ce qui existe déjà
2. Fonctionnalités implémentées
   - Description des fonctionalités
   - Organisation du projet
3. Élements techniques
   - Algorithmes
   - Structures de données
   - Bibliothèques
4. Architecture du projet
   - Diagramme de classes
   - Cas d'utilisation
   - Chaînes de traitement
5. Expérimentations et usages
   - Capture d'écrans
   - Mesures de performance
6. Conclusion
   - Récapitulatif des fonctionnalités principales
   - Propositions d'améliorations

Conseils

1. Soignez votre syntaxe et orthographe
2. Ne sautez pas des pages inutilement
3. Rédaction multi-modale (diagramme et explications en langue naturelle)
4. Un algorithme doit être accompagné d'un exemple
5. Pas de code mais du pseudo-code ou des schémas
6. Le code doit être réservé pour des spécificités du langage
7. Un diagramme de classe seul ne suffit pas : il faut le justifier
8. Argumentez vos choix
9. Une capture d'écran doit être lisible mais ne doit pas faire de remplissage
10. Attention au plagiat : pas de copié-collé, citation des sources

Soutenance (15 minutes)

1. Préparez un diaporama
2. Pas de diapositives trop chargées, pas de diapositives vides
3. Numérotez les diapositives
4. Répartissez-vous équitablement la parole
5. Ne lisez pas votre texte
6. Tout ce qui est dans le rapport n'a pas vocation à être à l'oral
7. Mettez en valeur votre réalisation
8. Assurez-vous de la lisibilité des diagrammes et images
9. Préparez votre démonstration à l'avance (faites une vidéo)