

M1 info



GINF41B2 (Conception et Programmation Orientée Objet)

Cours #7

Contrats et exceptions

Pierre Tchounikine


Plan

- Rappels : encapsulation et délégation
- Notion de fiabilité
- Spécifications et contrats
- Mise en œuvre des contrats (principes)
- Utilisation des assertions
- Le traitement des situations exceptionnelles
- Ce qu'en dit Java

Rappels encapsulation et délégation

3/46
Cours P. Tchounikine

Encapsulation

- Idée :
 - cacher la réalisation effective
 - empêcher d'autres manipulations que celles prévues
 - Principe :
 - lier les données et leurs opérations
 - interdire tout autre accès aux données que par les opérations associées
- 
- « vue externe » = les services offerts aux utilisateurs de l'objet
« vue interne » = comment les services sont réalisés
- Avantages
 - dissimulation de l'information
 - notion d'interface = « contrat », spécification

4/46
Cours P. Tchounikine

Avantages de l'encapsulation



5/46
Cours P. Tchounikine

Classe et encapsulation

- Le « contrat » défini par la classe (les services proposés), c'est :
 - les en-têtes des méthodes publiques
 - le comportement des méthodes
- Remarque 1 :
 - attention à la définition de la portée des attributs
 - attention à la définition de la portée des méthodes
 - attention aux altérateurs
- Remarque 2 :
 - accessibilité des champs/méthodes, en-têtes : vérification syntaxique
 - comportement des méthodes : ...
 - intérêt de pouvoir réifier des contraintes explicites, cf. plus tard

private !

pré-conditions, post-conditions, invariants

6/46
Cours P. Tchounikine

Agrégation / Délégation

- Point de vue « objet = structure »
 - un objet est défini comme une **agrégation** d'autres objets
 - un objet (→ la classe) est définie **à l'aide** d'autres objets (...)
- Point de vue « objet = service »
 - un objet « utilise » un autre objet (→ un « client » et un « serveur »)
 - un objet qui est sensé rendre un service **délègue** à un autre objet une partie de ce qu'il devait faire
- Point de vue « type de relation entre les objets / classes » :
 - relation de « tout » à « partie »

vilain terme indéfini qui peut correspondre à différentes choses

7/46
Cours P. Tchounikine

Le polymorphisme

- Idée
 - une classe définit un comportement
 - les sous-classes redéfinissent (éventuellement) ce comportement pour lui donner une sémantique / une réalité conforme à ce que sont (plus précisément) ses objets
- Conséquences
 - tous les objets de la classe et des sous-classes réalisent le comportement (savent réagir au message)
 - vu de l'extérieur, on peut adresser un message à un objet sans connaître son type (sa classe) précise
- Utilisation
 - demander un service à l'objet sans avoir à connaître son type précis (il suffit de savoir qu'il sait faire)

c'est une autre forme d'encapsulation

c'est une autre forme de délégation

8/46
Cours P. Tchounikine

Résumé

délégation = utilisation d'un service selon un protocole

Une délégation implique

- un serveur (fournisseur)
- un client (demandeur)
- un contrat

« respect du contrat »
fiabilité ...

9/46
Cours P. Tchounikine

Notion de fiabilité

10/46
Cours P. Tchounikine

Fiabilité

Fiabilité = Correction + Robustesse

Correction

capacité du logiciel à s'exécuter conformément à sa spécification

notion relative (cohérence / spec)

Robustesse

capacité du logiciel à réagir aux cas non décrits dans la spécification

assertions

moyen de décrire les spécifications

exceptions

moyen de traiter les situations "exceptionnelles"

11/46
Cours P. Tchounikine

Quelques soucis avec une pile

- Implantation :
 - tableau de 100 éléments
 - sommet de pile

distinguer robustesse et correction

- Soucis possibles :
 - empiler / pile pleine :
 - nature ?
 - origine ?
 - gestion ?
 - dépiler / pile vide
 - nature ?
 - origine ?
 - gestion ?

12/46
Cours P. Tchounikine

Aller vers la fiabilité

- Architectures **simples**
 - Modularité
 - Encapsulation
 - Faible couplage
 - Extensibilité
 - Clarté
 - Lisibilité
 - ...
- Conception / programmation de composants **corrects**

13/46
Cours P. Tchounikine

Spécifications et contrat



B. Meyer

14/46
Cours P. Tchounikine

Spécifications

- A quoi ça sert ?
 - mieux comprendre ce que doit faire le logiciel
 - mieux comprendre les solutions
 - définir ce que doit faire le logiciel
 - **donner les moyens de concevoir un logiciel correct**
 - **donner des moyens de montrer (prouver, argumenter) qu'un logiciel est correct**
 - faciliter la tâche de documentation
 - fournir une base pour les tests et le débogage

15/46
Cours P. Tchounikine

Spécifications et contrat

- Pré-conditions
 - contraintes que doit vérifier une méthode pour fonctionner correctement, i.e., assurer son service
- Post-conditions
 - propriétés qui résultent de l'état de l'exécution d'une méthode
- Contrat
 - engagement stipulant que si la méthode est appelée en respectant les pré-conditions, l'état décrit par les post-conditions sera vérifié après l'exécution de la méthode

Contrat + **Encapsulation** → **Correction**
 (je ne fais les choses que si je peux les faire, et dans ce cas je les fais bien) (pas d'effet exogène) (≠matériel)

16/46
Cours P. Tchounikine

Invariant de classe

spécifique à la POO (≠ pré/post-conditions)

- Invariant de classe
 - assertion dénotant les propriétés intrinsèques des instances de la classe à des « moments stables »
 - méthode publique : préservent les invariants
 - méthodes « outils » : peuvent ne pas préserver un invariant temporairement à condition de revenir dans un état le respectant
- Idée : plus qu'une pré/post-condition
 - caractérise la classe : les services actuels ... et les futurs (héritage)
 - affecte tous les contrats des méthodes

axiomes des types abstrait de donnée

+ « invariants d'implantation » ou « invariants de représentation »

17/46
Cours P. Tchounikine

Invariant de classe : exemples

- Exemples
 - le nombre d'éléments d'une pile est entre 0 et la capacité max de la pile
 - ce n'est pas une pré-condition
 - ce n'est pas une post-condition
 - la somme des points des joueurs de tarot = 0
 - ce n'est pas une pré-condition
 - ce n'est pas une post-condition
 - ce n'est peut être pas vrai à tout moment des manipulations internes, mais c'est vrai aux moments « stables »

18/46
Cours P. Tchounikine

Notion de classe correcte

Une classe *Classe* est correcte si et seulement si

R1 : constructeurs

Pour tout constructeur C de Classe

{Pré-conditions (C)}

Exécution (C)

{Post-conditions (C)} ET {Invariants (Classe)}

pas de vers dans le fruit !

R2 : services fournis

Pour toute méthode M de Classe offerte aux clients

{Pré-conditions (M)} ET {Invariants (Classe)}

Exécution (M)

{Post-conditions (M)} ET {Invariants (Classe)}

19/46
Cours P. Tchounikine

Héritage : une relation sémantique

- Relation sur-classe/sous-classe

- les invariants des surclasses sont des invariants des sous-classes
→ un moyen de « sémantiser » les classes abstraites notamment



Délégation ? Oui, mais délégation contractuelle !

- Redéfinition et implantation des méthodes

- problème : avec le polymorphisme et/ou des méthodes abstraites, le client ne connaît pas nécessairement le fournisseur effectif
→ comment en respecter les pré-conditions ?
- principe : une redéfinition de méthode ne peut remplacer les pré-conditions et/ou les post-conditions de la méthode redéfinie que par des pré-conditions plus faibles et/ou des post-conditions plus fortes

n'attend pas plus et rend au moins le même service

20/46
Cours P. Tchounikine

Mise en œuvre des contrats (principes)

21/46
Cours P. Tchounikine

Contrat : qui fait quoi ?

- Pré-conditions

- contrainte que doit vérifier un bout de code pour fonctionner correctement, i.e., assurer son service

➡ **responsabilité : le code appelant**

*obligation pour le client,
droit pour le fournisseur*

- Post-conditions

- propriétés qui résultent de l'état de l'exécution d'un bout de code

➡ **responsabilité : le code appelé**

*obligation pour le fournisseur,
droit pour le client*

22/46
Cours P. Tchounikine

Contrats vs. programmation défensive

- Contrat

« si tu m'appelles en respectant les pré-conditions, je m'engage à ce que l'état décrit par les post-conditions soit vérifié »

- Corollaire :

le fournisseur **n'a pas** à traiter les cas où le client ne respecte pas le contrat



le code fournisseur n'a pas à tester les pré-conditions

- ❖ éviter la redondance
- ❖ éviter les choses + ou – testées, qui marchent + ou –
- ❖ éviter les vérif + vérif → complexité
- ❖ faciliter la maintenance (pas d'implicites, etc.)
- ❖ etc.



« programmation défensive »
centration sur un bout du code

« programmation par contrats »
prise en compte macroscopique

23/46
Cours P. Tchounikine

Mise en œuvre des contrats

- Différencier

- module = acquisition des données
 - acquiert les entrées
 - les vérifie (tests ; exceptions)
 - fixe une post-condition (données complètes et correctes)
- module = traitement
 - pré-conditions = entrées complètes et correctes
 - post-conditions = traitement réalisé

module « filtre »

- La violation d'une assertion est un bug (point final !)

- violation d'une pré-condition : bug chez le client
- violation d'une post-condition : bug chez le fournisseur
- violation d'un invariant de classe : bug chez le fournisseur

notion d'erreur

24/46
Cours P. Tchounikine

Utilisation des assertions

25/46
Cours P. Tchounikine

Assertions

- Une assertion est une expression booléenne qui exprime les propriétés sémantiques d'une classe
- Une assertion décrit ce qui doit être (mais ne fait rien)

```
x = 3;
assertion : X>0;
```

redondant ?

```
x.f(y);
assertion : X>0;
```

redondant ?

NON !

- ❖ faire ≠ dire
- ❖ quoi ≠ comment
- ❖ on ne s'adresse pas à la même personne (au même rôle)

26/46
Cours P. Tchounikine

Assertions : pour quoi faire ?

- Pour penser en terme de contrats « conception par contrats »
 - construire du code correct / construire du code + debugger
- Pour commenter le code dans le code auto-documentation
 - pré/post-conditions + invariants = description concise du « quoi » (et du « pourquoi ») d'une classe
 - génération de documentation
- Pour vérifier le code ensure ; require ; old
 - description des assertions dans le langage
 - options de compilation pour vérifier ou pas (certaines) assertions

en production

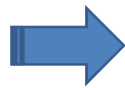
en exploitation

à défaut de pouvoir
« prouver » la correction

27/46
Cours P. Tchounikine

Assertions : pour quoi pas faire ?

assertion = condition de correction



les assertions ne doivent pas servir à gérer
des situations qui peuvent se produire

assertion non vérifiée → arrêt du code !

cf. notion d'exception (...)
cf. doc Java (...)

28/46
Cours P. Tchounikine

Assertions et réutilisation « ouverte »

- Approche « exigeante »
 - le fournisseur a des pré-conditions (point final)
- Approche « tolérante »
 - principe : le fournisseur essaie de gérer différents cas un peu limites
 - justification : une certaine conception de la réutilisation
 - problème : le peut-il ?
(cf. exemple de la pile)



Définir une politique

compromis réfléchi
éviter bretelle + ceinture

29/46
Cours P. Tchounikine

Le traitement des situations exceptionnelles

30/46
Cours P. Tchounikine

Exception

- Une exception est un événement qui *peut* conduire un composant logiciel à ne pas respecter son contrat
- Deux cas :
 - le composant récupère l'exception et respecte son contrat
 ➡ ni vu ni connu
 - le composant est en échec et ne peut respecter son contrat
 ➡ faire remonter le problème (récursivement) jusqu'à un niveau qui sait le traiter (traitements en plusieurs étapes éventuellement)

remontée / appels (et pas / blocs)

il faut définir et se tenir à une politique de gestion des problèmes explicites !



31/46
Cours P. Tchounikine

Gestion des situations exceptionnelles

- Bien distinguer les choses
 - diagnostic = reconnaître qu'une situation exceptionnelle s'est produite (test)
 - signalement = avertir le « client » que le service demandé n'est pas rendu
 - traitement = réagir à la situation

traitement interne

information du client

propagation

utilisation d'un autre fournisseur

entrée en mode dégradé (e.g.,
fermer les fichiers) + propagation

traitement puis relance

consignation des informations disponibles

etc.

32/46
Cours P. Tchounikine

Qu'est-ce qu'en dit Java ?

33/46
Cours P. Tchounikine

Mise en œuvre des assertions en Java

- Pas de gestion de la « programmation par contrat » en Java
- Différentes approches pour s'en inspirer
 - définir les assertions comme des commentaires (et génération de texte spécifique via Javadoc)
 - définir les assertions via des méthodes sans effet de bord
 - utiliser les mécanismes de gestion des exceptions
 - **utiliser l'instruction ASSERT**
 - utiliser des outils (bibliothèques) spécifiques (cf. Jcontractor)

34/46
Cours P. Tchounikine

Assert



“An *assertion* is a statement in the Java™ programming language that enables you to test your assumptions about your program. “

`assert ExpressionBooléenne ;` if false throws an `AssertionError` with no detail message

`assert ExpressionBooléenne : Expression qui a une valeur ;`
the system passes the value of `Expression` to the appropriate `AssertionError` constructor, which uses the string representation of the value as the error's detail message

`assert T1.length == T2.length : "correspondance impossible";`

`assert false : "ben non mon gars on n'est pas sensé venir ici";`
ce message *n'est pas* un bon exemple

On s'adresse

- ❖ à soi (commentaire d'élaboration)
- ❖ aux futurs relecteurs du code (auto-documentation)
- ❖ éventuellement, à l'exécuteur du code

35/46
Cours P. Tchounikine

AssertionError



`java.lang`

Class AssertionError

`java.lang.Object`

└ `java.lang.Throwable`

└ `java.lang.Error`

└ `java.lang.AssertionError`

`assert ExpressionBooléenne ;`

`assert ExpressionBooléenne : Expression qui a une valeur ;`

lève **AssertionError** (avec le résultat de l'expression le cas échéant)

`public class AssertionError extends Error`

Thrown to indicate that an assertion has failed.

36/46
Cours P. Tchounikine

Class Error



public class **Error**
 extends [Throwable](#)

An Error is a subclass of Throwable that indicates serious problems that a **reasonable application should not try to catch**. Most such errors are abnormal conditions.

A method is not required to declare in its throws clause any subclasses of Error that might be thrown during the execution of the method but not caught, since these errors are **abnormal conditions that should never occur**.

37/46
 Cours P. Tchounikine

Options de compilation

les assertions sont « débrayables »



- By default, assertions are disabled at runtime.
 - To enable assertions at various granularities, use the -enableassertions, or -ea, switch
 - To disable assertions at various granularities, use the -disableassertions, or -da, switch
- You specify the granularity with the arguments that you provide to the switch:
 - no arguments
 Enables or disables assertions in all classes except system classes.
 - *packageName...*
 Enables or disables assertions in the named package and any subpackages.
 - ...
 Enables or disables assertions in the unnamed package in the current working directory.
 - *className*
 Enables or disables assertions in the named class.

38/46
 Cours P. Tchounikine

Exemples



“While the assert construct is not a full-blown *design-by-contract* facility, it can help support an informal design-by-contract style of programming.”

- Invariants

- *whenever you would have written a comment that asserts an invariant*
- *Control-Flow Invariants: place an assertion at any location you assume will not be reached*

propriété sur la valeur d'une variable ; cas "default" d'un switch où l'on n'ira jamais

- Pré-conditions

- use an assertion to test a *nonpublic* method's precondition that you believe will be true no matter what a client does with the class

- Post-conditions

- you can test postconditions with assertions in both public and nonpublic methods

39/46
Cours P. Tchaoukine

Exemple (invariant)



- Exemple :

balanced tree data structure of some sort

- Invariant de classe :

the tree is balanced and properly ordered

- Conseil :

combine the expressions that check required constraints into a single internal method that can be called by assertions.

- Mise en oeuvre :

// Returns true if this tree is properly balanced
private boolean balanced() { ... }

- Exploitation :

Because this method checks a constraint that should be true before and after any method completes, each public method and constructor should contain the following line immediately prior to its return:

assert balanced();

40/46
Cours P. Tchaoukine

Contre-exemples



- *Do not use assertions for argument checking in public methods.*
- *Do not use assertions to do any work that your application requires for correct operation.*

problème / principe des assertions

problème / code (dépendance aux options de compilation)

- *By convention, preconditions on public methods are enforced by explicit checks that throw particular, specified exceptions.*

Exemple :

```
IllegalArgumentException();
```

41/46
Cours P. Tchounikine

Exceptions en Java

```
try {
    // Code pouvant lever des xxException standard et/ou des ExceptionMetierx »
}

catch (xxException e) {} // Gestion des xxException (et de ses sous-classes)
catch (ExceptionMetier1 e) {} // Gestion de ExceptionMetier1 (et de ses sous-classes)
catch (Exception e) {} // Gestion de ce qui reste
finally {} // Code toujours exécuté
```

Exceptions standard

- ❖ explicites (sous contrôle) : à traiter obligatoirement
- ❖ implicites (hors contrôle, qui peuvent se produire n'importe où)

Exceptions « métier »

- ❖ à créer (sous-classe de Exception)
- ❖ à lancer (throw)

possibilité de transmettre de l'information pertinente au gestionnaire

```
if (...) throw new ExceptionMetier1 (args du constructeur);
```

42/46
Cours P. Tchounikine

Exceptions



"The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch."

The Throwable class is the superclass of all errors and exceptions in the Java language.

Instances of two subclasses, [Error](#) and [Exception](#), are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information (such as stack trace data).

A throwable contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error. Finally, it can contain a cause: another throwable that caused this throwable to get thrown. The cause facility is new in release 1.4. It is also known as the chained exception facility, as the cause can, itself, have a cause, and so on, leading to a "chain" of exceptions, each caused by another.

43/46
Cours P. Tchounikine

Class Exception

[Overview](#)
[Package](#)
[Class](#)
[Use](#)
[Tree](#)
[Deprecated](#)
[Index](#)
[Help](#)

[PREV CLASS](#)
[NEXT CLASS](#)

[SUMMARY](#)
[NESTED](#)
[FIELD](#)
[CONST](#)
[METHOD](#)

[FRAMES](#)
[NO FRAMES](#)
[All Classes](#)

[DETAIL](#)
[FIELD](#)
[CONST](#)
[METHOD](#)

Java™ 2 Platform
Std. Ed. v1.4.2

java.lang
Class Exception

```

java.lang.Object
├── java.lang.Throwable
│   └── java.lang.Exception
        
```

All Implemented Interfaces:
[Serializable](#)

Direct Known Subclasses:
[AccNotFoundException](#), [ActivationException](#), [AlreadyBoundException](#), [ApplicationException](#), [AWTException](#), [BackingStoreException](#), [BadLocationException](#), [CertificateException](#), [ClassNotFoundException](#), [CloneNotSupportedException](#), [DataFormatException](#), [DestroyFailedException](#), [ExpandVetoException](#), [FontFormatException](#), [GeneralSecurityException](#), [GSSException](#), [IllegalAccessException](#), [InstantiationException](#), [InterruptedException](#), [IntrospectionException](#), [InvalidMidiDataException](#), [InvalidPreferencesFormatException](#), [InvocationTargetException](#), [IOException](#), [LastOwnerException](#), [LineUnavailableException](#), [MidiUnavailableException](#), [MimeTypeParseException](#), [NamingException](#), [NoninvertibleTransformException](#), [NoSuchFieldException](#), [NoSuchMethodException](#), [NotBoundException](#), [NotOwnerException](#), [ParseException](#), [ParserConfigurationException](#), [PrinterException](#), [PrintException](#), [PrivilegedActionException](#), [PropertyVetoException](#), [RefreshFailedException](#), [RemarshalException](#), [RuntimeException](#), [SAXException](#), [ServerNotActiveException](#), [SQLException](#), [TooManyListenersException](#), [TransformerException](#), [UnsupportedAudioFileException](#), [UnsupportedCallbackException](#), [UnsupportedFlavorException](#), [UnsupportedLookAndFeelException](#), [URISyntaxException](#), [UserException](#), [XAException](#)

44/46
Cours P. Tchounikine

Error et Exception



Why is AssertionError a subclass of Error rather than RuntimeException?

This issue was controversial. The expert group discussed it at length, and came to the conclusion that Error was more appropriate to discourage programmers from attempting to recover from assertion failures. It is, in general, difficult or impossible to localize the source of an assertion failure. Such a failure indicates that the program is operating "outside of known space," and attempts to continue execution are likely to be harmful. Further, convention dictates that methods specify most runtime exceptions they may throw (with @throws doc comments). It makes little sense to include in a method's specification the circumstances under which it may generate an assertion failure. Such information may be regarded as an implementation detail, which can change from implementation to implementation and release to release.

45/46

Cours P. Tchounikine

Etats d'âmes



Why not provide a full-fledged *design-by-contract* facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?

We considered providing such a facility, but were unable to convince ourselves that it is possible to graft it onto the Java programming language without massive changes to the Java platform libraries, and massive inconsistencies between old and new libraries. Further, we were not convinced that such a facility would preserve the simplicity that is the hallmark of the Java programming language. On balance, we came to the conclusion that a simple boolean assertion facility was a fairly straight-forward solution and far less risky. It's worth noting that adding a boolean assertion facility to the language doesn't preclude adding a full-fledged design-by-contract facility at some time in the future.

The simple assertion facility does enable a limited form of [design-by-contract style programming](#). The assert statement is appropriate for nonpublic precondition, postcondition and class invariant checking. Public precondition checking should still be performed by checks inside methods that result in particular, documented exceptions, such as `IllegalArgumentException` and `IllegalStateException`.

46/46

Cours P. Tchounikine