# Coder proprement

Contenu

[Coder proprement 1](#_Toc394930208)

[1 Code propre 1](#_Toc394930209)

[2 Noms significatifs 1](#_Toc394930210)

[3 Fonctions 2](#_Toc394930211)

[4 Commentaires 3](#_Toc394930212)

[5 Mise en forme 4](#_Toc394930213)

[6 Objet et structures de données 4](#_Toc394930214)

[7 Gestion des erreurs 5](#_Toc394930215)

[8 Limites et frontières 7](#_Toc394930216)

[9 Tests Unitaires 8](#_Toc394930217)

[10 Classes 9](#_Toc394930218)

[11 Systèmes 10](#_Toc394930219)

[12 Emergences 11](#_Toc394930220)

[13 Concurrence 11](#_Toc394930221)

## 1 Code propre

Passer du temps à garder un code propre n’est pas une question de coûts ; il s’agit d’une question de survie professionnelle.

La plupart des directeurs veulent du bon code, même lorsqu’ils sont obsédés par les échéances. Ils peuvent défendre avec passion le planning et les exigences, mais c’est leur travail. Le vôtre consiste à défendre le code avec une passion équivalente.

Vous ne respecterez pas les échéances en travaillant mal. La seule manière de respecter le planning, ou d’aller vite, est de garder en permanence le code aussi propre que possible.

Les boy-scouts ont une règle simple que nous pouvons appliquer à notre métier :

*Laissez le campement plus propre que vous ne l’avez trouvé en arrivant.*

Si nous enregistrons tous un code un peu plus propre que celui que nous avons chargé, le code ne peut tout simplement pas se dégrader.

## 2 Noms significatifs

Puisque les noms sont omniprésents, il est préférable de bien les choisir. **les noms doivent révéler les intentions**. Choisir de bons noms prend du temps, mais permet d’en gagner plus encore

Si un nom exige un commentaire, c’est qu’il ne répond pas à ces questions.

int d; // Temps écoulé en jours.1

Le nom d ne révèle rien ! -> int elapsedTimeInDays;

Il faut éviter les mots dont le sens établi varie du sens voulu -> càd qu’ils peuvent signifier autre chose. Si des noms doivent être différents, alors, ils doivent également représenter des choses différentes. Exemple :

public static void copyChars(char a1[], char a2[]) {

for (int i = 0; i < a1.length; i++) {

a2[i] = a1[i];

}

}

Cette fonction serait beaucoup plus lisible si les arguments se nommaient source et destination.

Les mots parasites représentent une autre distinction dépourvue de sens : ProductInfo ou

ProductData, vous avez choisi des noms différents sans qu’ils représentent quelque

chose de différent.

**Choisir des noms prononçables** pour pouvoir dialoguer entre développeurs.

Choisir des noms compatibles avec une recherche.

Les noms de variables sur une seule lettre introduisent ce problème, a moins que leur portée soit très réduite.

Choisir un mot par concept : il est assez déroutant d’avoir fetch, retrieve et get pour représenter des méthodes équivalentes dans des classes différentes.

## 3 Fonctions

Ecrire des fonctions le plus courtes possible. Il faut aussi que la taille de ces fonctions ne leur permette pas de contenir des structures imbriquées. Le niveau d’indentation d’une fonction ne doit donc pas être supérieur à un ou deux.

Une fonction ne doit faire qu’une seule chose. Une fonction ne doit comporter qu’un seul niveau d’abstraction : haut niveau, intermediaire, bas niveau.

Choisir des noms descriptifs pour les fonctions, n’ayez pas peur si celui-ci estlong.

Plus une fonction est courte et ciblée, plus il est facile de choisir un nom descriptif.

Arguments d’une fonction : idéalement, aucun, 1 ou 2 sont courant, 3 sont à éviter autant que possible, 4 ou plus ne devrait jamais être employés.

Peu d’arguments simplifient les tests de la fonction.

La fonction unaire peut:

* Poser une question via son argument : ex fileExists(“MonFichier”)
* Prendre l’argument en entrée, le transforme en autre chose et le retourne : InputStream fileOpen (“MonFichier”)
* Etre de type évenement, void passwordAttemptFailedNtimes(int attempts)

Les arguments indicateurs compliquent immédiatement la signature de la méthode, en proclamant que cette fonction fait plusieurs choses. Ex render(boolean isSuite) ->Il aurait fallu décomposer la fonction en deux : renderForSuite() et renderForSingleTest().

Fonction diadiques : Plus complexe, nécessite une pause pour les comprendre pour ne pas ignorer un arguement. Attention à l’ordre des arguments, exploiter tous les mécanismes disponibles pour les convertir en fonctions unaires.

Fonctions triadiques : Les problèmes d’ordre, de pause et d’ignorance sont multipliés.

Lorsqu’une fonction semble avoir besoin de plus de deux ou trois arguments, il est probable que certains d’entre eux feraient mieux d’être enveloppés dans leur propre classe.

Si les arguments variables sont tous traités de manière identique, comme dans ce cas, ils sont alors équivalents à un seul argument de type List.

Eviter les effets secondaires çad qu’une fonction en plus de faire une chose, fait également d’autres choses « cachées », les couplage temporels qui oblige à faire appel à la fonction à un endroit précis, et générant des erreurs si on l’appelle à un autre endroit.

Les arguments de sortie sont à proscrire. **Si votre fonction doit modifier l’état de quelque chose, faites en sorte que ce soit l’état de l’objet auquel elle appartient**.

Preferer les exceptions au retour de codes d’erreur, permet de séparer la gestion de l’erreur du code principal et donc de le simplifier. Cela permet aussi d’eviter d’avoir à mettre à jour l’enumeration de code d’erreur à chaque nouveau code d’erreur et d’avoir à tout recompiler/deployer, alors qu’avec les exception il suffit de faire une derivation de la classe Exception

Extraire les blocs try catch dans des fonctions à part, ne contenant qu’un bloc try catch finally et rien derrière.

Eviter le code redondant. Ecrire une fonction proprement revient à l’écrire une première fois de façon imparfaite, puis réfléchir pour l’améliorer encore et encore pour obtenir un code propre, il est normal de ne pas y arriver du premier coup.

## 4 Commentaires

Les commentaires sont au mieux, un mal nécessaire et pallient à notre incapacité à exprimer nos intentions par le code. Ils ne sont parfois pas maintenus et dans ce cas amène confusion et désinformation. Plutôt que dépenser de l’énergie à les écrire juste et à les maintenir, mieux vaut l’utiliser pour écrire un code propre ne nécessitant pas de commentaire.

Quel code préférez-vous voir ? Celui-ci ?

// Vérifier si l’employé peut bénéficier de tous les avantages.

if ((employee.flags & HOURLY\_FLAG) &&

(employee.age > 65))

Ou celui-ci ?

if (employee.isEligibleForFullBenefits())

En général, il ne faut pas plus de quelques secondes de réflexion pour expliquer la

majorité des intentions dans le code. **Dans la plupart des cas, il suffit simplement de**

**créer une fonction qui exprime la même chose que le commentaire imaginé.**

Les bons commentaires sont ceux qui ont pour but :

* ajouter les références légales
* informer
* expliquer les intentions
* clarifier
* explique les raisons d’une décision
* avertir des conséquences
* mettre des TODO (temporaire !)
* amplifier

Les mauvais commentaires :

* marmonner : commentaire obscur/mal placé
* parasite/ redondant : si le code est clair, il n’est pas utile de rajouter en plus un commentaire.
* Trompeurs
* Obligatoire : commenter chaque fonction, chaque variable alourdit inutilement le code. Les commentaires javadoc peuvent être masqués par l’ide, mais c’est des commentaires en plus à maintenir.
* Journalisation, commentaire de modification s’accumulant, et pouvant être remplacé par le contrôle de source ( svn)
* Marqueurs de positions, surtout si ils sont très répandus. Ex // Actions ////////////////
* Code en commentaire : finit par s’accumuler et inutile puisque qu’on a des systèmes de gestion de code source pour retrouver l’historique
* Information non locale
* En tête de fonction : si la fonction est courte, mieux vaut bien choisir son nom, cela est suffisant

## 5 Mise en forme

Le formatage du code est important.

Nous voudrions qu’un fichier source ressemble à un article de journal : un titre, un synopsis, les détails, par ordre d’importance.

Séparer les différents concepts par des lignes vides améliore la visibilité -> espace entre package, import, fonctions…

En revanche, les mêmes concepts doivent être concentrés verticalement : les déclarations de variables d’une classe doivent être rapprochées.

Pour éviter que le lecteur saute de part en part de nos fichiers sources et classes, les concepts étroitement liés doivent être les plus rapprochés possibles.

Les variables des fonctions doivent être déclarés au plus près de leur utilisation. Les variables d’instance doivent être placés au début de la classe.

Lorsqu’une fonction en appelle une autre, l’appelante doit être au-dessus de l’appelé, ou du moins le plus proche possible. De même si plusieurs fonctions ont des similitudes ou partage le même schéma de nommage.

Ranger de haut en bas en plaçant en haut, les fonctions de plus haut niveau puis les fonctions de niveau moyen, et enfin les fonctions de bas niveau.

S’efforcer à écrire des lignes courtes. Respecter la règle de l’indentation.

Respecter les règles de l’équipe en matière de code évite de donner l’impression que le logiciel est le fruit de développeurs en désaccord.

## 6 Objet et structures de données

Une classe ne peut pas se contenter de présenter ses variables au travers d’accesseurs. À la place, elle doit exposer des interfaces abstraites qui permettent à ses utilisateurs de manipuler l’*essence* des données, sans avoir à en connaître l’implémentation.

Il faut une réflexion sérieuse sur la meilleure manière de représenter les données contenues dans un objet. La pire solution consiste à ajouter sans discernement des méthodes d’accès.

Les objets exposent un comportement et masquent les données. Il est ainsi facile d’ajouter de nouvelles sortes d’objets sans changer les comportements existants. Mais il est également plus difficile d’ajouter de nouveaux comportements à des objets existants.

Les structures de données exposent des données et n’ont pas de comportement significatif. Il est ainsi facile d’ajouter de nouveaux comportements à des structures de données existantes, mais difficiles d’ajouter de nouvelles structures de données à des fonctions existantes.

La loi de Déméter stipule qu’une méthode *f* d’une classe *C* ne doit appeler que les méthodes des éléments suivants :

n *C* ;

n un objet créé par *f* ;

n un objet passé en argument à *f* ;

n un objet contenu dans une variable d’instance de *C*.

Décomposer les appels successifs (wagon accrochés les uns aux autres)

final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath(); ->

Options opts = ctxt.getOptions();

File scratchDir = opts.getScratchDir();

final String outputDir = scratchDir.getAbsolutePath();

Eviter les classes hybrides, mi structure de données mi objet. Elles sont impossibles à faire évoluer et signe d’une conception confuse -> Ecrire des classes faisant des choses (objet) et d’autres stockant les données (Objet de transfert de données ou DTO). Ne pas mettre de fonction métier dans un DTO !

Dans les DTO on retrouve :

les beans java : variables privés et accesseurs  
les actives records : données publiques mais offrant des fonctions de navigation tel que save et find.

## 7 Gestion des erreurs

Gérer les erreurs dans des fonctions dédiés utilisant la notion d’exception plutôt qu’utiliser des codes retours et devoir inclure des tests dans la fonction appelante, risquant d’être oubliés.

Ecrire des blocs try catch finally. Le try est vu comme une transaction, le catch permettant de gérer l’erreur, veuillez à bien définir le type de l’exception pouvant survenir.

Les exceptions vérifiés (on renvoie un type d’exception précis à l’appelant) constituent une violation du principe ouvert/ fermé, car il faut déclarer l’exception dans la signature de chaque méthode qui se trouve entre notre code et la clause catch. Si une fonction de très bas niveau est modifiée, elle va impacter toutes celles l’appelant, à l’infini. Sauf le cas d’une bibliothèque critique, le cout de la dépendance dépasse les bénéfices potentiels.

Créer des messages d’erreurs informatifs et les passer avec les exceptions. Mentionnez l’opération qui a échouée et le type de défaillance

Pour traiter les exceptions en évitant la redondance, envelopper des API tierces est une bonne pratique :  
On passe de :

ACMEPort port = new ACMEPort(12);

try {

port.open();

} catch (DeviceResponseException e) {

reportPortError(e);

logger.log("Device response exception", e);

} catch (ATM1212UnlockedException e) {

reportPortError(e);

logger.log("Unlock exception", e);

} catch (GMXError e) {

reportPortError(e);

logger.log("Device response exception");

} finally {

...

}

* Code redondant, qui va revenir à chaque fois qu’on va ouvrir un port dans l’application.

A:

LocalPort port = new LocalPort(12);

try {

port.open();

**} catch (PortDeviceFailure e) {**

reportError(e);

logger.log(e.getMessage(), e);

} finally {

...

}

Avec

public class LocalPort {

private ACMEPort innerPort**; -> On enveloppe**

public LocalPort(int portNumber) {

innerPort = new ACMEPort(portNumber);

}

public void open() {

try {

innerPort.open();

} catch (DeviceResponseException e) {

throw new PortDeviceFailure(e); **-> On a plu qu’un seul type d’exception à gerer**

} catch (ATM1212UnlockedException e) {

throw new PortDeviceFailure(e);

} catch (GMXError e) {

throw new PortDeviceFailure(e);

}

}

...

}

Grace aux enveloppes, nous sommes moins dépendants d’une api en particulier.

Le pattern « Cas particulier » est celui-ci : le code du client n’a pas à s’occuper d’un comportement exceptionnel. Ce comportement est encapsulé dans l’objet dédié au cas particulier.

**Ne pas retourner « null »** : Cette habitude de codage a tendance à attirer les erreurs. Lorsque nous retournons null, nous ne faisons qu’augmenter notre charge de travail et transmettre des problèmes aux appelants qui devront tester si le résultat de la fonction est null. Il suffit d’oublier une vérification de

null pour perdre le contrôle de l’application. Au lieu d’écrire une méthode qui retourne null, vous devez à la place envisager de lancer une exception ou de retourner un objet CAS PARTICULIER. Si vous utiliser une API tierce qui retourne null, envisager d’envelopper cette méthode dans une méthode qui lance une exception ou retourne un objet de cas particulier.  
Exemple avec une méthode renvoyant la liste des employés :  
public List<Employee> getEmployees() {

if( .. il n’y a pas d’employés .. )

return **Collections.emptyList();**

}

Renvoie une liste vide au lieu de null.

**Ne pas passer null** : Dans la plupart des langages de programmation, il n’existe aucun bon moyen de gérer le passage accidentel d’un argument null. Puisque c’est ainsi, l’approche rationnelle consiste à interdire par défaut de passer null.

## 8 Limites et frontières

On utilise du code existant, des logiciels tiers, des api tierces, il convient de garder propre les frontières de nos logiciels.  
Les fournisseurs de paquetages et de frameworks tierce partie s’efforcent d’obtenir une applicabilité étendue afin de travailler dans de nombreux environnements et de répondre aux attentes d’un large public. *A contrario*, les utilisateurs attendent une interface qui se focalise sur leurs besoins précis. Cette tension peut amener des problèmes aux frontières de nos systèmes.

Exemple : on veut utiliser Map, sans que l’utilisateur puisse vider la liste. On va employer map ainsi:

public class Sensors {

private Map sensors = new HashMap();

public Sensor getById(String id) {

return (Sensor) sensors.get(id);

}

//...

}

L’interface qui se situe aux limites (Map) est cachée, et celle-ci peut évoluer sans impacter le reste de l’application, on peut mieux définir l’interface et répondre aux besoins, et le code est plus facile à comprendre, plus difficile à utiliser à mauvais escient.

On évitera les implémentations suivantes :

Map<Sensor> sensors = new HashMap<Sensor>(); -> Sensor s = sensors.get(sensorId );  
ou pire :  
Map sensors = new HashMap(); -> accès via Sensor s = (Sensor)sensors.get(sensorId);  
car le code est peu lisible, et on a tjrs la possibilité de vider la map en faisant Map.clear() ;

Il est difficile de maîtriser le code tiers, tout comme il est difficile de l’intégrer. Il est doublement difficile de faire les deux à la fois. Au lieu d’expérimenter et d’essayer ces nouvelles choses dans notre code de production, **nous pourrions écrire des tests afin de découvrir notre compréhension du code tiers**.

Exemple avec log4j :

@Test

public void testLogAddAppender() {

Logger logger = Logger.getLogger("MyLogger");

logger.removeAllAppenders();

logger.addAppender(new ConsoleAppender(

new PatternLayout("%p %t %m%n"),

ConsoleAppender.SYSTEM\_OUT));

logger.info("hello");

}

Grace à l’écriture de cette classe de test nous connaissons suffisament sur l’api log4j opur encapsuler ces connaissances dans notre propre classe de journalisation, afin que le reste de l’application soit isolé de l’interface limitrophe de log4j.

Les tests d’apprentissage ne sont pas seulement gratuits, ils ont un retour sur investissement

Positif. Si de nouvelles versions du paquetage tiers sont publiées, nous exécutons les tests d’apprentissage pour savoir s’il existe des différences de comportement.

Sans ces *tests aux limites* pour simplifier la migration, nous serions tentés de conserver l’ancienne version plus longtemps que conseillé.

Utiliser du code qui n’existe pas encore : Pour ne pas rester bloquer, nous avons défini notre propre interface du système non existant, l’interface que nous souhaitions avoir. Nous pouvons ainsi garder un code client plus lisible et focalisé sur ce que nous tentons d’accomplir.

Une bonne conception du logiciel s’accommode des modifications sans investissements et efforts importants. Il faut éviter qu’une trop grande partie de notre code connaisse les détails du code

tierce partie : il faut réduire au maximum le nombre d’endroits du code qui y font référence. Nous pouvons les envelopper, comme cela a été le cas avec Map, ou utiliser un ADAPTATEUR pour faire le lien entre notre interface idéale et celle fournie.

## 9 Tests Unitaires

Les mouvements Agile et TDD ont encouragé de nombreux programmeurs à écrire des tests unitaires automatisés et beaucoup rejoignent les rangs quotidiennement. Il ne faut pas oublier cependant certains points et important, d’écrire de bons tests.

**3 lois du TDD**  
- Vous ne devez pas écrire un code de production tant que vous n’avez pas écrit un test unitaire d’échec.

- Vous devez uniquement écrire le test unitaire suffisant pour échouer .l’impossibilité de compiler est un échec.

- Vous devez uniquement écrire le code de production suffisant pour réussir le test d’échec courant.

Les tests et le code de production sont écrits *ensemble*, en commençant par les tests. Si nous travaillons ainsi, ces tests couvrent virtuellement tout le code de production. Mais l’ensemble de ces tests peut présenter des problèmes de gestion.

Garder des tests propres  
**Porter la même qualité de code aux tests et au code de production**. Ne pas écrire des tests jetables, plutôt écrire un ensemble de tests unitaires automatisés.

les tests grossiers sont équivalents, sinon pires, à aucun test. les tests doivent évoluer avec le code de production. Plus les tests sont négligés, plus il est difficile de les modifier.

**Si vous disposez de tests, vous n’avez pas peur de modifier le code.** Sans les tests, chaque modification est un bogue potentiel.

La carateristique d’un test propre est la lisibilité, encore plus qu’un code de production : clarté, simplicité et densité d’une expression.

Pour écrire les tests, utiliser le pattern BUILD OPERATE CHECK en divisant le code des tests en trois parties : la 1ere construit les données du test, la deuxième exploite ces données et la troisième vérifie que l’opération a produit les résultats escomptés.

->ne pas hésiter à écrire des fonctions supplémentaires pour le code redondant. Une série d’assertTrue / false peut etre remplacé par un AssertEquals + pattern :

assertTrue(hw.heaterState());

assertTrue(hw.blowerState());

assertFalse(hw.coolerState());

assertFalse(hw.hiTempAlarm());

assertTrue(hw.loTempAlarm())

deviant

assertEquals("HBchL", hw.getState());

avec

public String getState() {

String state = "";

state += heater ? "H" : "h";

state += blower ? "B" : "b";

state += cooler ? "C" : "c";

state += hiTempAlarm ? "H" : "h";

state += loTempAlarm ? "L" : "l";

return state;

}

* On peut utilizer des strings plutot que String buffer car l’environnement de test n’est pas l’environnement de production, et on priviligie la simplicité plutôt que l’efficacité et l’espace mémoire.
* Pour les noms des fonctions on peut utiliser la convention classique *given-when-then. Exemple :*

public void testGetPageHierarchyAsXml() throws Exception {

givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

whenRequestIsIssued("root", "type:pages");

thenResponseShouldBeXML();

}

**Une assertion par test :** C’est un bon principe mais dans la pratique cela peut conduire à inclure du code redondant. Il faut cependant garder en tête de minimiser le nombre d’assertions dans un test.

**Un seul concept par test** :

Nous ne voulons pas de longues fonctions qui testent diverses choses hétéroclites l’une après l’autre.

Les tests propres sont :

Rapide : pour pouvoir être executé souvent, et pouvoir vérifier l’ état du système rapidement

Independant : les tests ne dépendent pas les uns des autres

Reproductible : quel que soit l’environnement

Auto validant : ils doivent réussir ou échouer d’eux-mêmes, sans qu’on ai besoin d’ouvrir un fichier resultat par exemple

Ecrit au moment opportun : juste avant l’écriture du code de production, pas après.

## 10 Classes

**Organisation d’une classe**

* Liste des variables : statiques publiques, statique privées, variables d’instances privées
* Fonctions publiques, fonctions privés appelés par la fonction publique à sa suite

Petites classes : comme les fonctions, les classes doivent être le plus petite possible, en mesurant non pas leur taille ou leur nombre de fonctions, mais leurs responsabilités.

Le nom de la classe doit décrire ses responsabilités. Si il nous est impossible de trouver un nom concis ou non ambigu, alors la classe est sans doute trop grande. Il faut pouvoir la décrire sans utiliser « si, et ou mais ».

**Principe de responsabilité unique**

Il ne doit exister qu’une seule raison de modifier une classe ou un module. Lorsque nous essayons d’identifier les responsabilités (c’est-à-dire les raisons d’être modifié), il nous est plus facile de reconnaître et de créer de meilleures abstractions dans notre code.

Le principe de responsabilité unique fait partie des concepts les plus importants de la conception orientée objet.

**Cohésion**

Nous souhaitons une cohésion élevée, chaque méthode d’une classe doit manipuler une ou plusieurs des variables d’instances (les méthodes et variables sont interdépendants)

Si on a une prolifération de variables d’instances utilisé par un sous ensemble de méthode, alors cela signifie presque toujours qu’il existe au moins une autre classe qui essaie de sortir de la classe plus grande-> Séparer les variables & méthodes en deux classes , plus cohésives.

Le découpage d’une longue fonction en plusieurs petites fonctions nous donne souvent l’opportunité de créer plusieurs classes plus petites.

Pour arriver à ce résultat il faudra souvent redecouper une classe / un programme trop long. Cela réduit le risque lié aux changements. Toute modification de la classe a la possibilité de casser une autre partie du code de la classe, elle doit être intégralement testée à nouveau.

**Principe ouvert/fermé** (OCP, *Open-Closed Principle*)

les classes doivent être ouvertes à l’extension mais fermées à la modification. Dans un système idéal, nous incorporons de nouvelles fonctionnalités en étendant le système, non en modifiant le code existant.

**Cloisonner le changement**

Une classe cliente qui dépend de détails concrets présente un risque lorsque ces détails changent. Nous pouvons introduire des interfaces et des classes abstraites pour limiter l’influence de ces détails.

## 11 Systèmes

Nous devons modulariser le processus d’initialisation, séparément de la logique d’exécution normale et établir une stratégie globale cohérente pour résoudre nos dépendances majeures. Il faut se méfier de l’initialisation paresseuse, qui est un idiome commode mais qui disperce la logique d’initialisation et reduit la modularité.

1 construire dans la fonction main, le main appelle un constructeur et execute l’application en lui fournissant les objets initialisés dont elle a besoin. L’application n’a pas de code d’initialisation.

2 Utiliser des fabriques abstraites : la responsabilité du moment de création de l’objet est laissé à l’application, mais les détails de cette construction est fait dans une fabrique.

3 Injection de dépendances : l’application de l’inversion de contrôle à la gestion des dépendances. Exemple : la recherche JNDI

MyService myService = (MyService)(jndiContext.lookup(“NameOfMyService”));

L’objet invoquant n’a aucun contrôle sur le type d’objet réellement retourné (tant qu’il implémente l’interface appropriée), mais il résout néanmoins activement la dépendance.

4 Spring / framework : L’infrastructure requise pour l’application, y compris les préoccupations transversales comme la persistance, les transactions, la sécurité, la mise en cache, le basculement, etc.,

est incluse en employant des fichiers de configuration déclaratifs ou des API.

Ainsi le client a l’impression de converser avec le POJO, en fait c’est avec l’element externe d’un ensemble d’objets decorateurs imbriqués qui étendent le comportement de base du POJO.

Spring avec ses annotations permet de garder un code simple et propre.

Une bonne API doit être pratiquement invisible la plupart du temps, afin que l’équipe dépense son énergie créatrice sur les scénarios utilisateurs à implémenter.

## 12 Emergences

Selon Kent, une conception est "simple" lorsqu’elle respecte les quatre règles suivantes :

* réussit tous les tests ;
* ne contient aucune redondance ;
* exprime les intentions du programmeur ;
* réduit le nombre de classes et de méthodes.

**Tests**

Le système doit fonctionner comme prévu et ce fait doit pouvoir être vérifié par des tests. Un système non testable et donc vérifiable ne doit jamais être déployé. Pour qu’un système soit testables, il faut qu’il soit composer de classes petites et ayant des objectifs unique (SRP)

**Suppression de la redondance**

Pour supprimer la redondance, le design pattern « patron de méthode » -> Pour eviter d’écrire deux méthodes dont la majeure partie du code est redondant pour gérer un cas particulier ou un autre. On définit des sous classes overridant la méthode , chaque sous classe ayant sa logique propre..

**Expressivité**

Une grande partie du coût d’un projet logiciel se trouve dans la maintenance à long terme.

Nous pouvons nous exprimer en employant une nomenclature standard. Les motifs de

conception, par exemple, réalisent un important travail de communication et d’expressivité.

En utilisant des noms de motif standard, comme COMMAND ou VISITOR, dans les noms des classes qui implémentent ces motifs, nous pouvons décrire succinctement notre conception aux autres développeurs.

**Réduction du nombre de classes et de méthodes**

En nous efforçant d’obtenir des classes et des méthodes courtes, nous risquons de créer beaucoup

trop de toutes petites classes et méthodes.la règle stipule que nous tentions également de maintenir le nombre de fonctions et de classes aussi faible que possible.

Cette règle est la dernière des quatre règles de conception simple et qu’elle est donc la moins prioritaire. Par conséquent, même s’il est important de réduire au maximum le nombre de classes et de fonctions, il est plus important d’avoir des tests, de supprimer la redondance et d’améliorer l’expressivité.

## 13 Concurrence

La concurrence améliore *parfois* les performances, uniquement lorsqu’il est possible

de partager de longs temps d’attente entre plusieurs threads ou plusieurs processeurs.

la conception d’un algorithme concurrent peut être très différente de celle

d’un système monothread. Le découplage du *quoi* et du *quand* a généralement un

impact très important sur la structure du système.

Si on utilise un conteneur web ou un EJB ( système permettant de gérer la concurrence partiellement) il est préférable de connaître le fonctionnement du conteneur et de savoir comment résoudre les problèmes de mises à jour concurrentes et d’interblocage.

*La concurrence implique un certain surcoût,* à la fois en terme de performances et

d’écriture d’un code supplémentaire

*Une bonne mise en oeuvre de la concurrence est complexe,* même dans le cas de

problèmes simples.

*Les bogues de concurrence ne sont généralement pas reproductibles,* et ils sont généralement considérés comme des événements exceptionnels

*La concurrence implique souvent un changement fondamental dans la stratégie de*

*Conception*

Pour se prémunir des problèmes de concurrence, voici un ensemble de principes et techniques :

* Principe de responsabilité unique : il ne doit exister qu’une seule raison de modifier une méthode, une classe ou un composant.

Le code lié à la concurrence possède son propre cycle de développement, de changement et de réglage, il présente ses propres défis. En conséquence gardez le code lié à la concurrence séparé de tout autre code

* Limiter la portée des données : une solution consiste à utiliser le mot-clé synchronized pour protéger une *section critique* du code qui utilise l’objet partagé. **Il faut limiter le nombre de sections critiques**, afin d’éviter les erreurs tel qu’oublier de proteger un ou plusieurs endroits, devoir redoubler d’efforts, et il sera plus difficile de determiner les sources de dysfonctionnements.
* Utiliser des copies de données au lieu de données partagées et verrouillées. Chaque thread utilise une copie, et un thread fusionne les résultats de chacun d’eux. Le surcout mémoire des copies est compensé par l’économie des mécanismes de verroullage.
* les threads doivent être aussi indépendants que possible : chaque thread traite sa requete cliente, sans partager de donnée avec d’autres threads.

Connaitre la bibliothèque java 5

* Les collections et outils thread safe doivent être employées
  + ConcurrentHashmap du package java.util.concurrent.\*
  + Reentrant lock : verrou obtenu dans une méthode et libéré dans une autre
  + Semaphore : verrou avec compteur
  + CountDownLatch : verrou qui attend un certain nombre d’evenements avec de debloquer tous les threads en attente .

Definitions :

* Ressources bornées : dont la taille ou le nombre est figé, utilisées dans un environnement concurrent : ex connexions a la base de donnée ou tampons de lecture/ecriture.
* Exclusion mutuelle : un seul thread a la fois peut accéder à des données ou des ressources partagées
* Famine : Un thread ou plus est interdit de fonctionnement pendant une durée trop longue ou indéfiniment
* Interblocage : deux threads ou plus attendent chacun la fin de l’autre. Ex : si chaque thread dispose d’une ressource dont l’autre a besoin , et aucun ne peut se terminer tant qu’il n’a pas obtenur la ressource de l’autre
* Interblocage actif : Les threads sont actifs, mais en réalité en situation d’interbloclage

Modèle d’execution employés dans la programmation concurrente

Voici les 3 principaux scénarios de concurrence que l’on rencontre dans la programmation concurrente :

**Producteur – Consommateur**

Les producteurs créent du travail et le placent dans un tampon ou

une file qui constitue une ressource bornées. Un ou plusieurs threads consommateurs récupèrent ce travail à partir de la file et le mènent à terme. Chacun doit attendre que la file se libère pour y ajouter un travail ou pour le récuperer. Les producteurs avertissent les consommateurs que la file est non vide, et les consommateurs avertissent les producteurs que la file n’est plus pleine. Les deux acteurs peuvent attendre d’être avertis pour pouvoir continuer.

**Lecteurs – Rédacteurs**

Il faut trouver le bon équilibre pour qu’aucun des groupes ne tombent en situation de famine, a attendre que les lecteurs aient fini de lire, ou les redacteurs aient fini d’écrire.

**Diner de philosophes**

Les philosophes a table autour d’un plat de spaghetti pensent, puis ont faim et cherche à obtenir deux fourchettes pour manger. Un philosophe ne peut manger sans ces deux ressources. Si une deux ressources, il attend. Le philosophe qui a fini de manger remet les fourchettes au centre de la table.

Le mot clé synchronised permet de protéger l’accès à une partie du code. Cependant si une même classe partagée possède plusieurs méthodes synchronisées, votre système risque alors d’être mal écrit.

Si ce cas est justifié et qu’on a plusieurs méthodes synchronised dans une même classe, alors voici 3 méthodes pour obtenir un code correct :

Verrouillage par le client : le client verrouille le serveur, avant d’appeler la 1ere méthode, et s’assure que le verrou est bien étendu jusqu’à ce qu’il appelle la dernière méthode.

Verrouillage par le serveur : une méthode spécifique verrouille le serveur, appelle toutes les méthodes, puis relâche le verrou. Le client appelle cette méthode.

Serveur adapté : Créer un intermédiaire se chargeant du verrouillage. Equivalent au verrouillage serveur mais ne nécessitant pas de modifier celui-ci.

Les verrous coutent chers en ressources, garder vos sections synchronised le plus courtes possible.

Réfléchir au cas où le processus parent doit attendre le résultat des threads enfants, également quand ce système doit s’arrêter.

Tester vos systèmes multithreads, n’ignorer pas un échec d’un test s’il a réussi une fois auparavant.

Les tentatives pour reproduire les bogues peuvent etre pénibles. Voici comment procéder :

Commencer par rendre le code normal opérationnel : créer des POJO invoqués par des threads , qui pourront être testés hors de l’environnement multithread.

Code multithread enfichable : différentes configurations :

* variation du nombre de threads au cours de l’execution,
* interaction du code avec un composant réel ou une doublure du test,
* tests s’exécutant rapidement, lentement ou de manière variable.
* un nombre variable d’exécution

Code multithread réglable : pour obtenir le bon équilibre au sein des threads, les performances doivent être mesurables. Etudiez comment parametrer le nombre de threads, le fait de pouvoir modifier ce nombre pendant l’execution, voire le débit et l’utilisation du système.

Executer le code avec plus de threads que de processeurs/cœurs dans l’ordinateur.

Executer le code sur différentes plates formes

Instrumenter le code pour essayer et forcer les echecs :

* soit manuellement ajouter dans les tests les appels à des méthodes comme Object.wait(), Object.sleep(), Object.yield() et Object.priority() permet de detecter des failles difficiles à trouver.

Automatisé via des outils comment Aspect-Oriented Framework, CGLIB ou ASM. ConTest d’IBM. L’idée est d’employer un simple aspect qui choisit aléatoirement entre ne rien faire, s’endormir ou relâcher le flux de contrôle. L’association de tests bien écrits et du secouement peut énormément

augmenter les chances de trouver des erreurs.

# 14 Améliorations successives

# 15 JUnit

# 16 Remaniement de SerialDate

# 17 Indicateurs de code

Commentaires

C1 : informations inapropriés :  
ne pas mettre d’informations pouvant être mis ailleurs, à savoir un système de gestion du code source, un outil de suivi de bogues, ni mettre d’historiques de modifications, métadonnées, auteurs, date de dernieres modifications… Les commentaires sont réservés aux notes techniques concernant le code et la conception.

C2 : commentaires obsolètes :  
Ancien, non pertinent et incorrect = obsolète. Il est préférable de ne pas écrire un commentaire qui deviendra obsolète. Si vous en trouvez un, mettez-le à jour ou supprimer le.

C3 : commentaire redondants

Un commentaire est redondant lorsqu’il explique quelque chose qui est déjà suffisamment

descriptif en soi. Les commentaires doivent expliquer des choses que le code est incapable d’exprimer

par lui-même.  
C4  commentaires mal rédigés  
C5  code en commentaire : a supprimer

Environnement  
E1 construction en plusieurs étapes: La construction de l’environnement doit pouvoir se faire de façon simple.  
E2 tests en plusieurs étapes: Le lancement de tout les tests unitaires doit se faire en une seule commande.  
  
Fonctions  
F1 trop grand nombre d’arguments: Le nombre d’arguments d’une fonction doit être aussi réduit que possible.  
F2 arguments de sortie: Les arguments de sortie sont contre intuitifs. Si votre fonction doit modifier un état, ce doit être celui de l’objet sur lequel elle est invoquée.

F3 arguments indicateurs: Les arguments booleens mettent en évidence que la fonction fait plusieurs choses. Il faut les supprimer et créer plusieurs fonctions.  
F4 fonction morte: Fonction mortes : si elle n’est jamais appelé, elle doit être retiré.

Général:  
G1 multiples langage dans un même fichier source: plusieurs langages dans un même fichier source -> a éviter.  
G2 comportement évident non implémenté: comportement évident non implémenté -> il faut coder ce que la fonction est censé intuitivement faire, afin que les autres développeurs puissent s’appuyer dessus sans devoir faire la lecture du code pour savoir ce que la fonction fait réellement.  
G3 comportement incorrect aux limites : la fonction doit fonctionner correctement aux limites -> le vérifier toujours, ne pas se fonder sur son intuition.

G4 sécurité neutralisées: ne pas neutraliser les sécurités par facilité : serialVersionUID, test Junit, avertissement du compilateur…

G5 redondance : chaque fois que vous rencontrez une redondance dans le code, elle représente en réalité une opportunité d’abstraction manquée-> un sous-programme, une autre classe. La redondance peut etre :

un pan de code identique -> méthode,

une chaine if/else ou switch testant les mêmes conditions et présente dans plusieurs modules-> polymorphisme,

module présentant le même algorithme -> pattern Patron de méthode ou Stratégie

G6 code au mauvais niveau d’abstraction :  
Il est important de créer des abstractions qui distinguent les concepts généraux de niveaux supérieurs des concepts détaillés de niveau inférieur. Haut niv -> classe abstraite, bas niveau -> classes dérivées, avec séparation totale des concepts, il ne faut pas que les concepts de haut niveau et bas niveau soient mélangés.

G7 classes de base dépendantes de leurs classes dérivées. Cf G6 les classes de base ne doivent rien connaitre de leurs classes dérivées, sinon on peut supposer un problème de conception. Exception : le nombre de classes dérivées est fini et le code de la classe de base choisit la classe dérivée adéquate.

G8 Trop d’informations : un bon module propose peu d’interface permettant de faire de nombreuses choses, et non de nombreuses interfaces permettant de faire peu de chose. Reduisez le nombre de vos fonctions dans vos classes, et le nombre de variable de vos fonctions. Maintenez un couplage faible en limitant les informations.

G9 Code mort :

Correspond à du code qui n’est jamais execute, donc jamais testé ni maintenu. A supprimer sans hésiter.

G10 Séparation verticale :  
Les variables et fonctions doivent être définies au plus près de leur utilisation. Variable locales -> juste avant leur 1ere utilisation, avec portée verticale réduite.

G11 Incohérence :  
Principe de moindre surprise : il faut se tenir à la convention décidé auparavant. Cela permet d’obtenir un code plus facile à comprendre et à modifier.

G12 : désordre  
Garder le code propre, ordonnée et sans désordre : enlever les constructeurs sans paramètres, les variables non utilisés, les fonctions jamais appelés, les commentaires inutiles…