Découpe et contrats en génie logiciel

Olivier Cailloux

LAMSADE, Université Paris-Dauphine

Version du 26 décembre 2016

LAMSADE
UMR CNRS 7243
laboratoire d'analyse et modélisation de systèmes pour l'aide à la décision

DAUPHINE
UNIVERSITÉ PARIS

PSL
RESEARCH
UNIVERSITY

Sous-routines

- Sous-routine : entrée (facultative), ensemble d'instructions, sortie (facultative)
- Fait qqch
- On peut l'appeler sans savoir comment elle procède (boite noire)
- `Math.random();` `Math.sqrt(4);`
- Sous-routine sqrt statique se trouvant dans classe `Math`
- Classes regroupent des variables et sous-routines statiques
- `System.out` : variable statique `out` dans classe `System`
- Permet d'*organiser* et de *nommer*

Exemple

- Dans méthode main
- Manipulation donnée d'une chaine
- À plusieurs endroits

Réusinage

- Pour éviter duplication du code:
- En faire une sous-routine
- Appelée à plusieurs endroits

Intérêt

Pourquoi créer une sous-routine ?

- Éviter la duplication de code. Pourquoi ?

Intérêt

Pourquoi créer une sous-routine ?

- Éviter la duplication de code. Pourquoi ?
  - Bugs : correction à un seul endroit
  - Factorisation : application conçue comme assemblage de blocs élémentaires
  - Clarté du code
  - Réusinage peut être facilité (trouver tous les endroits où routine est appelée)

Intérêt

Pourquoi créer une sous-routine ?

- Éviter la duplication de code. Pourquoi ?
  - Bugs : correction à un seul endroit
  - Factorisation : application conçue comme assemblage de blocs élémentaires
  - Clarté du code
  - Réusinage peut être facilité (trouver tous les endroits où routine est appelée)
- Clarté du code : auto-documentation ; boite noire
- Partage du travail entre développeurs
- Estimation quantité de travail

Factorisation

- Code peut se ressembler sans être identique
- Modifier pour qu'il soit identique mais paramétré
- Exemple : Échecs, dessin du plateau vu du côté noir ou blanc

$\Rightarrow$ Une seule routine de dessin, paramétré selon couleur

Contrat

- Clarification des responsabilités nécessaires
- Sous-routine fonctionnant sous certaines conditions
- Exemple : entier fourni en paramètre > 0
- Appelées *préconditions*
- *Contrat* entre appelant et programmeur de la sous-routine
- Sous ces conditions, méthode fournit un service
- Exemple : renvoie un nombre aléatoire entre 0 et l'entier fourni, exclu
- Si conditions non remplies : pas de garanties offertes !

Contrat à expliciter

- Contrat facilite l'implémentation de la sous-routine
- Contrat facilite la vie de l'utilisateur
- À condition de rendre le contrat explicite
- Documenter les préconditions
- Utilisateur averti : pensera plus probablement à vérifier les préconditions

Programmation défensive

- Aider les utilisateurs imprudents
- Si précondition non satisfaite : le dire
- Principe de l'*échec rapide* (*fail-fast*)
- Mieux vaut une erreur immédiate qu'une action inattendue
- Facilite les corrections de bug
- En pratique : tester les préconditions en entrée de sous-routine (lorsque raisonnable)

Assertions en Java

`assert` : solution *imparfaite* pour tester facilement les préconditions en Java

- `assert i > 0;`
- Voire `assert i > 0 : "Integer must be strictly positive, received " + i + ".";`
- Assertions non exécutées par défaut
- Exécutées ssi JVM reçoit paramètre `-ea`
- Intérêt ?

Assertions en Java

`assert` : solution *imparfaite* pour tester facilement les préconditions en Java

- `assert i > 0;`
- Voire `assert i > 0 : "Integer must be strictly positive, received " + i + ".";`
- Assertions non exécutées par défaut
- Exécutées ssi JVM reçoit paramètre `-ea`
- Intérêt ? Pas de ralentissement en production

Assertions en Java

`assert` : solution *imparfaite* pour tester facilement les préconditions en Java

- `assert i > 0;`
- Voire `assert i > 0 : "Integer must be strictly positive, received " + i + ".";`
- Assertions non exécutées par défaut
- Exécutées ssi JVM reçoit paramètre `-ea`
- Intérêt ? Pas de ralentissement en production
- Si assertion non satisfaite : arrêt de l'exécution + affichage de l'appel fautif (et des appels parents)
- Inconvénients ?

Assertions en Java

`assert` : solution *imparfaite* pour tester facilement les préconditions en Java

- `assert i > 0;`
- Voire `assert i > 0 : "Integer must be strictly positive, received " + i + ".";`
- Assertions non exécutées par défaut
- Exécutées ssi JVM reçoit paramètre `-ea`
- Intérêt ? Pas de ralentissement en production
- Si assertion non satisfaite : arrêt de l'exécution + affichage de l'appel fautif (et des appels parents)
- Inconvénients ?
  - Changement de comportement en production !
  - Pas de distinction préconditions et autres vérifications

\+ Gain de performance souvent négligeable

$\Rightarrow$ Assertions à réserver pour tests couteux en temps

Licence