Les Fonctions

Abstraction :

* Pratique servant à réduire le code d’un niveau de détail
* Permet de regrouper un certain nombre de classes selon des caractéristiques communes
* Ex dessin vectoriel

***abstract class Shape{***

***public:***

***virtual Area(){};***

***virtual Perimeter(){};***

***protected:***

***int origin; // pixel***

***int height; // pixel***

***int width; // pixel***

***...***

***}***

***class Rectangle:public Form{...}***

***class Cercle:public Form{...}***

* Toutes les figures ont un point d’origine, une hauteur, une largeur, une aire et un périmètre : on peut les regrouper en super-classe
* Grâce au polymorphisme on peut définir le comportement précis de chacun des types de figures simplement en faisant référence à la classe Shape sans se soucier de ce qu’est réellement cette figure

I faire court

* Les fonctions doivent rarement dépasser les 20 lignes
* Blocs/indentation
* **If**/**else**/**while** ne doivent occuper qu’une ligne (appel de fonction)
* Ça ajoute une valeur documentaire : fonction appelée aura un nom descriptif
* Niveau d’indentation d’une fonction doit être <= 2

II Faire une seule chose

* Doit mettre en œuvre des étapes qui se trouvent à un seul niveau sous son nom
* S’il est possible d’extraire de notre fonction une autre fonction dont le nom n’est pas une reformulation de son implémentation
* La fonction fait plusieurs choses
* Sections dans les fonctions
* Une fonction qui ne fait qu’une seule chose ne peut pas être décomposée en sections

III Niveau d’abstraction par fonction

* Pour être certain qu’une fonction ne fait qu’une seule chose les instructions doivent être au même niveau d’abstraction
* Mélanger les niveaux d’abstraction est toujours déroutant
* Difficile de cerner si une expression est un concept essentiel ou un détail
* Le mélange détails/concepts essentiels induit encore plus de détails au sein de la fonction
* Lire le code de haut en bas : règle de décroissance
* Chaque fonction doit être suivie de fonctions de niveau d’abstraction inférieur
* Permet de descendre d’un niveau d’abstraction à la fois en descendant dans le code
* On doit pouvoir lire le programme comme s’il s’agissait d’un ensemble de paragraphes POUR (chacun décrivant le niveau d’abstraction actuel et faisant référence à des paragraphes POUR de niveau inférieur)

IV Switch

* Difficile d’écrire des instructions **switch** courtes
* Même si elle ne comprend que 2 cas elle sera plus longue que la taille adéquate d’un bloc ou d’une fonction
* Difficile d’obtenir une instruction **switch** qui ne fait qu’une seule chose
* Par essence elle effectue **N** choses
* Si on est obligé de l’utiliser on peut faire en sorte que chaque instruction switch soit enfouie dans une classe de bas niveau et qu’elle ne soit jamais répétée (en utilisant le polymorphisme)

***Public Money CalculatePay(Employee e){***

***throws InvalidEmployeeType{***

***switch (e.type){***

***case COMMISSIONED :***

***return calculateCommissionedPay (e);***

***case Hourly :***

***return calculateHourlyPay (e);***

***case SALARIED /***

***return calculateSalariedPay (e);***

***default :***

***throw new InvalidEmployeeType (e.type);***

***}***

***}***

***}***

* Cette fonction présente plusieurs problèmes :
* Trop longue : et continuera de s’allonger lors de l’ajout de nouveaux types
* Prend en charge plusieurs choses
* Ne respecte pas le principe de responsabilité unique (existe plusieurs raisons de la modifier)

→ Ne respecte pas le principe ouvert/fermé (doit être modifiée dès l’ajout de nouveaux types)

→ Un nombre illimité d’autres fonctions auront la même structure

***isPayday(Employee e, Date date);***

***deliveryPay(Employee e, Money pay);***

* Solution : enfouir l’instruction **switch** au plus profond d’une FABRIQUE ABSTRAITE (GOF)
* Utiliser **switch** pour créer les instances adéquates des classes dérivées d’**Employee**
* Les différentes fonctions (**calculatePay**, **isPayday**, **deliveryPay**) seront distribuées par polymorphisme depuis l’interface Employee

***public abstract class Employee {***

***public abstract boolean isPayday();***

***public abstract Money calculatePay();***

***public abstract void deliverPay(Money pay);***

***}***

***-----------------***

***public interface EmployeeFactory {***

***public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;***

***}***

***-----------------***

***public class EmployeeFactoryImpl implements EmployeeFactory {***

***public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {***

***switch (r.type) {***

***case COMMISSIONED: return new CommissionedEmployee(r) ;***

***case HOURLY: return new HourlyEmployee(r);***

***case SALARIED: return new SalariedEmploye(r);***

***default: throw new InvalidEmployeeType(r.type);***

***}***

***}***

***}***

* Instructions switch
* Tolérées quand elles n’apparaissent qu’une seule fois
* Pour créer des objets polymorphes
* Cachées derrière une relation d’héritage (de manière que le reste du système ne puisse pas la voir)

V Choisir des noms descriptifs

* Un nom qui décrit bien l’objectif de la fonction
* Ne pas avoir peur de créer un nom à rallonge (ça vaut mieux qu’un nom vague suivi d’une tonne de commentaires)
* Se servir d’une convention de nommage
* Facilite la lecture des multiples mots composant les noms de fonctions
* Permet à tous ces mots de créer des noms qui décrivent le rôle des fonctions
* Ne pas avoir peur de passer du temps à choisir les noms
* Avec les IDE modernes (Eclipse/IntelliJ) il est très facile de changer les noms
* Choisir des noms descriptifs : clarifier la conception du module
* Sera plus facile à modifier
* Rester cohérent dans le choix des noms
* Employer les mêmes phrases/groupes nominaux/verbes dans les noms de fonctions/modules

VI Arguments d’une fonction

* Idéalement le nombre d’arguments devrait être égal à zéro (niladique)
* Ensuite on a les fonctions à 1 argument (monadiques/unaires) puis à 2 arguments (diadiques)
* EVITER TANT QUE POSSIBLE LES FONCTIONS A 3 ARGUMENTS (triadiques)
* Les fonctions qui prennent plus de 3 arguments exigent une très bonne raison
* EN GENERAL NE DOIVENT JAMAIS ÊTRE EMPLOYEES
* Les arguments sont très pénibles au point de vu des tests
* > 2 arguments : le test de chaque combinaison de valeurs est décourageant

1. Forme Unaire classique

* 2 raisons de n’avoir qu’un seul argument
* Poser une question à propose de cet argument (boolean : fileExists)
* Manipuler l’argument pour le transformer en autre chose et le retourner
* Choisir des noms qui permettent de distinguer clairement ces 2 cas
* Toujours employer les deux formes dans des contextes cohérents
* L’événement est une forme moins fréquente mais utile
* 1 argument d’entrée et aucun argument de sortie
* Le programme global est conçu de manière à interpréter l’appel de fonction comme un événement

→ Utilise l’argument pour modifier l’état du système

→ Soigneusement choisir les noms/contextes

* Il faut éviter les fonctions unaires qui ne correspondent pas à ces formes

1. Arguments indicateurs

* Donner une valeur booléenne à une fonction est une pratique épouvantable
* Complique la signature de la méthode en proclamant que la fonction fait plusieurs choses
* La fonction réalise une chose quand l’indicateur vaut True et une autre quand il vaut False

1. Fonctions diadiques

* Plus compliquée à comprendre que la fonction unaire
* On devrait exploiter tous les mécanismes disponibles pour les convertir en fonctions unaires

1. Fonctions triadiques

* Encore plus complexes à appréhender que les diadiques
* Multiplication des problèmes d’ordre/pause/ignorance

1. Objets en arguments

* Quand une fonction a besoin de >= 2 arguments : certains d’entre eux devraient être enveloppés dans leur propre classe

1. Listes d’arguments

* Parfois on doit passer un nombre variable d’ arguments à une fonction :

***String.format(‘’%s worked %.2f hours.’’, name, hours);***

* Les arguments variables sont tous traités de manière identique : équivalent à un seul argument de type List
* **String.format** est en réalité une fonction diadique (les mêmes règles s’appliquent) :

→ ***public String format (String format, Object…args)***

* Les fonctions qui prennent un nombre variable d’arguments peuvent être unaires/diadiques/triadiques
* Ça serait une erreur de leur donner un plus grand nombre d’arguments

1. Verbes/mots-clés

* Forme unaire : la fonction et l’argument doivent représenter un couple verbe/nom parfaitement associé
* **write (name)** ou encore **writeField (name)**
* Il serait préférable d’écrire **assertEquals** sous la forme **assertExpectedEqualsActual (expected, actual)**
* Allègement du problème de mémorisation de l’ordre des arguments

1. Eviter les effets secondaires

* La fonction doit faire une chose alors qu’elle en fait également d’autres de manière cachée
* Peut modifier de façon inattendue les variables de sa propre classe

***public class UserValidator {***

***private Cryptographer cryptographer;***

***public boolean checkPassword(String userName, String password) {***

***User user = UserGateway.findByName(userName);***

***if (user != User.NULL) {***

***String codedPhrase = user.getPhraseEncodedByPassword();***

***String phrase = cryptographer.decrypt(codedPhrase, password);***

***if ("Valid Password".equals(phrase)) {***

***Session.initialize();***

***return true;***

***}***

***}***

***return false;***

***}***

***}***

* Cette fonction utilise un algorithme classique pour mettre en correspondance **userName** et **password**
* Retourne **True** s’ils correspondent sinon renvoie false cependant elle dissimule un effet secondaire

→ **Session.initialize()** : **checkPassword** ne peut être invoqué que lorsque la session a été initialisée en toute sécurité

→ Si **checkPasword** n’est pas invoquée au bon moment : les données de session peuvent être perdues

→ C’est ce qu’on appelle un couplage temporel

* Les couplages temporels sont déroutants surtout quand ils sont la conséquence d’un effet secondaire
* Si on doit crée un couplage temporel il faut l’indiquer clairement dans le nom de la fonction
* Dans le cas ci-dessus on pourrait l’appeler **checkPasswordAndInitializeSession**
* Mais ça ne respecte pas la règle : ne faire qu’une seule chose

1. Arguments de sortie

* Les arguments sont naturellement interprétés comme les entrées d’une fonction
* **appendFooter(s) ;**
* Cette fonction ajoute **s** à la fin de quelque chose, ou ajoute-t-elle quelque chose à la fin de **s** ?
* L’argument **s** est une entrée ou une sortie ?
* La signature de la fonction **public void appendFooter (StringBuffer report)** clarifie la question
* Tout ce qui oblige à vérifier la signature de la fonction équivaut à une hésitation
* C’est une coupure cognitive qui doit être vérifiée
* Il serait mieux d’invoquer **appendFooter** de la manière suivante : **report.appendfooter () ;**
* En général les arguments de sortie sont à proscrire
* Si la fonction doit modifier quelque chose il faut que ce soit l’état de l’objet auquel elle appartient

VII Séparer commandes et demandes

* Les fonctions doivent soit faire quelque chose soit répondre à quelque chose **PAS LES DEUX**
* Doit modifier l’état d’un objet ou retourner des informations concernant cet objet
* En faisant les 2 elle amène une confusion
* **Public boolean set (String attribute, String value) ;**
* Fixe la valeur d’un attribut nommé et retourne True en cas de succès ou false si l’attribut n’existe pas
* Ça conduit à des instructions étranges : **if (set(‘’nomUtilisateur ‘’, ‘’OncleBob ‘’)**…

→ L’attribut **‘’nomUtilisateur ‘’** était précédemment fixé à **‘’OncleBob’’** ?

→ **‘’nomUtilisateur’’** a pu être fixé à **‘’OncleBob’’** ?

→ Difficile de déduire l’objectif car le terme **set** peut être un verbe comme un adjectif

* Ici **set** (verbe) : « si l’attribut **nomUtilisateur** était précédemment fixé à **OncleBob** »

→ Et pas (adjectif) : « fixer **nomUtilisateur** à OncleBob et si ça réussi alors… »

* La solution consiste à séparer la commande et demande afin d’écarter toute ambiguïté

***if (attibuteExists (‘’nomUtilisateur’’)) {***

***setAttribute (‘’nomUtilisateur’’, ‘’OncleBob’’) ;***

***…***

***}***

VIII Préférer les exceptions au retour de codes d’erreur

* Retour de codes d’erreur à partir de fonctions de commande : violation subtile de la séparation commandes/demandes
* Incite à employer des commandes comme des expressions dans les prédicats des instructions if
* **if (delatePage(page) == E\_OK)**
* pas de confusion verbe/adjectif mais conduit à des structures profondément imbriquées
* Lorsqu’un code d’erreur est retourné ça oblige l’appelant à immédiatement traiter l’erreur