Résumé du livre coder proprement avec exemple significatif:

# Nom significatifs

## Choisir des nom révélateurs des intentions

* Important de changer le nom d’un élément lorsque l’on en trouve un plus pertinent et évocateur
* Le nom d’un élément doit répondre à 3 questions, la raison de son existence, son rôle et son utilisation (Ex: int d; // Temps écoulé en jours. **devient** -> int elapsedTimeInDays; )

## Éviter la désinformation

* Éviter les acronymes (hp, aix ) surtout ceux qui pourrait porter à confusion
* Éviter d’utiliser le nom d’une structure ou d’un objet dans le nom si celui pourrait être autre chose (Ex : accountList doit bien être de type List sinon on le renomme accounts ou accountGroup, etc…)
* Éviter les noms qui se ressemble (Ex: XYZControllerForEfficientHandlingOfStrings et XYZControllerForEfficientStorageOfStrings)
* Éviter les lettres l et O pour les noms de variables, car peut-être confondu avec 1 et 0

## Faire des distinctions significatives

* Éviter d’introduire des fautes d’orthographe afin d’utiliser deux objets semblables dans le mêmes scopes (Ex: listes et lisstes ou encore a1 et a2 comme argument d’une fonction, préférable d’utiliser des noms du style origin et destination)
* Ne pas inclure le type dans le noms de la variable (style int[] tab) car si le type change, cela procure de la désinformation Deplus, avec les IDEs mordernes il est facile d’en connaître le type.

## Choisir des noms prononcables

* Choisir des noms prononcables, éviter les noms avec acronyme de type : genymdhms pour generateurYearMonthDayHourMinuteSeconds

## Choisir des noms compatibles avec une recherche

* Plus l’element fait partis d’un grand scope, plus le nom devrait être clair et précis et inclure le context. Les variables i, j, k devrait être réservé au boucle for.

## Interfaces et implémentations

* Éviter le I dans les noms d’interface, cela ne rajoute aucune info pertinente et parfois il est mieux de garder cette info de manière abstraite.
* Les implémentations peuvent avoir le suffix imp pour expliquer de maniere explicit que c’est l’implementation de l’interface

## Éviter les associations mentales

* Éviter d’utiliser des noms qui pourrait être confondu avec un concept qui n’est pas associé avec votre problème (Ex: utiliser Queue pour une variable qui n’est pas une Queue)

## Noms des méthodes

* Getters, setters doit commencer par get et set
* Les méthodes devraient commencer par un verbe suivit d’un context

## Ne pas faire le malin (éviter les jeux de mots, les références culturelles, etc…)

## Choisir un mot par concept

* Ne pas utiliser un verbe différent dans des classes différentes qui représente le même concept (Ex: get, fetch et retrieve représente souvent le même concept, vous devez en choisir un et stick to it)

## Choisir des noms dans le domaine de la solution

* Utiliser des noms qui parle au programmeur, par exemple, tout programmeur comprendra si vous utiliser un nom qui inclut l’algorithme ou la datastructure utilisé. (Ex: JobQueue, AccountVisitor, getBestPathUsingDisktra(Address origin, Address destination))

## Choisir des noms dans le domaine du problème

* Sans terme informatique disponible pour expliquer votre contexte, utiliser le nom du domaine du problème. Aussi, si le code est fortement lié au problème.

## Ajouter un contexte significatif

* Les variable firstName lastName parle d’elle même mais, par exemple, state peut porter a confusion. Parle-t-on de l’état de l’objet ou d’un état d’une adresse. Il peut être judicieux de rajouter un préfix de style addrState pour parler d’état d’adresse (New york etc..)

## Ne pas ajouter de contexte inutile

* Éviter les préfix avec le nom de l’objet dans la définition des attributs lorsque cela n’est pas nécéssaire du style PfirstName, PlastName dans une classe Person.

# Fonctions

## Faire court

* Les fonctions doivent être les plus courtes possible. Si elle peuvent être extraite en plusieurs fonctions qui effectue des tâches différentes, alors cela devrait être fait.

## Bloc et indentation

* Les instructions if, else et while devraient tenir sur une ligne. On appelle une fonction qui as un nom significatif de la tâche réalisé
* Le niveaux d’indentation ne devrait pas être supérieur à un ou deux. Si un if englobe tout votre code, il existe alors un autre if qui tiendra sur une ligne.

## Faire une seule chose

* Une fonction devrait effectuer une seule tâche précise. Sinon, elle doit être extraite en plusieurs fonctions et être inclus dans une fonction avec un nom significatif de la tâche effectuée.
* Un truc est de pouvoir démonter les imbrications des fonctions avec des POUR. (Ex:

Pour inclure les pages de montage et de démontage, nous incluons le montage, puis le contenu de la page de test et enfin le démontage.

Pour inclure le montage, nous incluons le montage d’une suite s’il s’agit d’une suite, puis nous incluons le montage normal.

Pour inclure le montage d’une suite, nous recherchons la page "SuiteSetUp" dans la hiérarchie parente et nous incluons une instruction avec le chemin de cette page. Pour rechercher le parent…)

## Instruction switch

* Référez vous à la page 42, l'explication est un peu compliqué et mérite la lecture

## Choisir des noms descriptifs

* N'ayez pas peur d’avoir des noms longs
* Rester cohérent avec le contexte et la tâche
* Prener le temps de choisir le nom

## Arguments d’une fonction

* Idéalement le nombre d’arguments devrait être égal a zéro ou un. Tolérer deux et maximum trois. Après, les arguments devraient surement être regroupé sous une structure (Liste, objet, etc…)

## Forme unaire (1 argument)

* Dans une fonction avec un argument, soit vous poser une question à l'argument (fileExists(“monfichier”) ) et retourner une valeur booléenne, soit transformer l’argument (Model addBaseInfoToModel(Model)) et devrait retourner la forme transformer de l’argument, soit être un événement comme void print(“monobjet”) mais faire attention à ce dernier.

## Forme diadiques (2 arguments)

* Peut être justifier lors de concept du style Point(0,0) car nous sommes habituer d’avoir deux coordonnées pour un point

## Forme triadique (3 arguments)

* Peut être justifier lors par exemple de précision fournis pour un float .assertEquals(1.0, amount, .001) car il permet de se rapper qu’un point flottant n’est pas précis et absolue par exemple.

## Éviter les effets secondaires

## Éviter les arguments de sortie le plus possible, ou soyez explicit

## Séparer commandes et demandes, pas les deux

## Préférer les exceptions au retour de code d’erreur

## Extraire les try/catch dans une fonction qui ne fait que ca