

Séance 14

The Pragmatic Programmer



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

Rappels

- Définition de **qualité logicielle** et critères d'évaluation
 - Décomposer et organiser un logiciel en sous-systèmes
 - « Slogans » à suivre : WET, KISS, YAGNI, DOA, DAMP...
 - Sept grands principes du développement logiciel
- Critères et méthodes d'**évaluation de la qualité**
 - Principaux critères de qualité logicielle
 - Quantifier les fonctionnalités d'un système avec function point
 - Métriques MOOD pour mesurer la qualité d'un système OO

Objectifs

- Aspects d'une **approche pragmatique** de la programmation
 - Bonnes pratiques de code, gestion erreur et debugging
 - Avoir des bons outils et les maîtriser
- **Comportement du développeur** pendant le codage et après
 - Faire attention à la gestion des erreurs
 - Penser flexible et avoir un programme configurable
 - Ne pas oublier les tests et le refactoring de code



Approche pragmatique

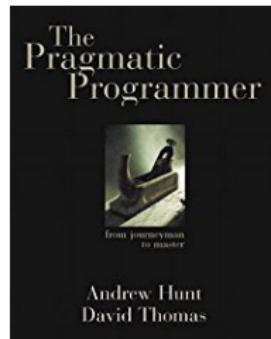
Pragmatisme

- Définition Larousse du **pragmatisme**
 - Doctrine qui prend pour critère de vérité le fait de fonctionner réellement, de réussir pratiquement.
 - Attitude de quelqu'un qui s'adapte à toute situation, qui est orienté vers l'action pratique.
- Bonne pratique d'aller vers une **solution pragmatique**

Ne pas trop réfléchir, mais parvenir à produire quelque chose

Approche pragmatique

- **Trucs et astuces** s'appliquant à tout niveau du développement
 - Des idées qui sont presque des axiomes
 - Des processus qui sont virtuellement universels
- Écrire du **code meilleur** et plus robuste, plus vite



Programmeur pragmatique

- Chaque développeur est unique avec ses **forces et faiblesses**
Construction de son propre environnement avec le temps
- Cinq caractéristiques des **programmeurs pragmatiques**
 - **Early adopter/fast adapter** veille technologique, aime tester
 - **Inquisitive** oser poser des questions
 - **Critical thinker** ne prend pas les choses comme acquises
 - **Realistic** comprend la nature des problèmes à résoudre
 - **Jack of all trades** bouge sur nouveaux domaines et challenges

Démon de la duplication

- La **duplication** rend difficile la maintenance

Constamment en maintenance, pas seulement après une release

- **DRY** (*Don't Repeat Yourself*)

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

- Plusieurs **sources de duplication** possibles

- **Imposée**, le programmeur semble ne pas avoir le choix
- **Inattention**, le développeur ne s'en rend pas compte
- **Impatience**, cela semble plus facile de dupliquer
- Duplication **inter-développeur**

Source de duplication (1)

- Duplication **imposée**, mais il y a des techniques pour modérer
 - Représentation multiple de l'information

Même donnée représentée différemment peut être générée
 - Documentation

Les commentaires sont pour le mauvais code (ou haut niveau)
 - Conséquence du langage de programmation

Par exemple, séparation interface/implémentation
- Duplication par **inattention**

Par exemple lorsque champs mutuellement dépendants

```
1 public class Line {  
2     public Point start, end;  
3     public double length;  
4 }
```

Source de duplication (2)

- Duplication par **impatience**

- Ne pas céder aux pressions temporelles d'un projet
 - “Short cuts make for long delays”

Gagner peu de temps maintenant et en perdre beaucoup plus tard

- Le bug du Y2K, avec années sur deux chiffres

Ne pas avoir paramétré la taille des variables pour les dates

- Duplication **inter-développeur**

- Il faut un design clair et un bon *technical project leader*
 - Division des responsabilités claire pour le design
 - Communication active et fréquente entre développeurs

Orthogonalité

- Orthogonalité, concept important et critique

Systèmes faciles à concevoir, construire, tester et étendre

- Deux lignes sont orthogonales si elles se coupent à angle droit

Se déplacer sur une ligne ne change pas la projection sur l'autre

- En informatique, indépendance et découplage

- Contrôles de l'hélicoptère pas du tout orthogonal
- Élimination d'effets entre des éléments non liés

- Développement de composants en respectant la cohésion

Self-contained with single well-defined purpose

Avantage de l'orthogonalité

- Augmentation de la **productivité**
 - Changements localisés, meilleurs temps de développement/test
 - Force la réutilisation, nouveaux composants par combinaison
 - Subtil gain de productivité en combinant composants
- Réduction des **risques**
 - Isolation des sections de code malades
 - Système moins fragile avec impacts changements localisés
 - Système plus facile à tester et donc mieux testé
 - Moins de dépendance à des vendeurs, produits, plateformes...

Réversibilité

- Ingénieurs préfèrent **solutions simples/uniques** aux problèmes
Et les managers sont souvent du même avis !
 - Minimiser les décisions **critiques non réversibles**
Choix d'un SGBD, d'un framework, d'une plateforme...
 - Prévoir dès le départ une **architecture flexible**
Utiliser des abstractions permettant des modifications
- “*Nothing is more dangerous than an idea if it's the only one you have.*” — *Emil-Auguste Chartier, Propos sur la religion, 1938*

Balle traçante (1)

- Pas facile de **tirer sur une cible** dans le noir

Connaitre position précise de la cible et état de l'environnement

- **Deux approches** possibles au problème

- Calcul très précis de la trajectoire à donner à la balle
- Utilisation de balle traçante émettant lumière phosphorescente

- **Feedback immédiat** dans mêmes conditions que vraies balles

Très pratique dans environnement risquant de changer

Code traçant (1)

- Spécifier son système à mort ou utiliser une balle traçante

Les programmeurs pragmatiques préfèrent la seconde solution

- Ecriture incrémentale d'un système et tracer le code

Du front-end au backend, de la GUI à la base de données...

- Commencer avec version basique avec une fonctionnalité

Permet de tester que les liens entre les composants sont corrects

- Pas un code jetable, il est destiné à rester et évoluer

Pas un prototype, contient vérification erreur, structure...

Code traçant (2)

- **Quatre avantages** principaux à l'approche de code traçant
 - Possibilité de voir le programme et faire une démonstration
 - Construction d'une structure dans laquelle travailler
 - Plateforme d'intégration implicite
 - Meilleur sentiment de progrès et satisfaction personnelle
- Permet de voir **ce qu'on a touché**, pas toujours cible voulue

Le code devra être ajusté suite aux feedback de la simulation



Outils de base

Les outils de base

- Tout artisan commence avec un **ensemble basique d'outils**

Outils de bonne qualité et biens maîtrisés

- Outils choisis **avec amour** selon plusieurs critères

Pour durer, job spécifique non redondant, et qui plait

- Vient ensuite le processus d'**apprentissage et d'adaptation**

Comprendre la personnalité et les caprices de chaque outil

- “*The better your tools, and the better you know how to use them, the more productive you can be.*”

La puissance du texte brut (1)

- Caractères imprimables (éventuellement structurés)

Directement lisibles et compréhensibles par l'humain

Field19=467abe versus DrawingType=UMLActivityDrawing

- Possibilité d'avoir du texte brut **structuré**

Par exemple \LaTeX , HTML, XML, JSON, YAML...

- Beaucoup plus de possibilités qu'avec un **format binaire**

Notamment possibilité de faire du versioning

► “*The best format for storing knowledge persistently is plain text.*”

La puissance du texte brut (2)

■ Avantages

- **Insurance/Obsolescence**, survit au temps et aux applications
- **Leverage**, tous les outils de l'univers savent gérer du texte
- **Easier testing**, création de jeux de tests aisé

■ Désavantages

- Nécessite plus d'espace pour être stocké/transmis
- Complexité temporelle de traitement accrue

La force du shell (1)

- Un shell est un **outil très puissant**

Lancer des applications, débugguer, éditeurs, utilitaires...

- Création de **macros complexes** pour automatiser des tâches

- Créer une archive avec les fichiers sources

```
tar cvf archive.tar *.h *.c
```

- Fichiers Java qui n'ont pas changé depuis la semaine dernière

```
find . -name '*.java' -mtime +7 -print
```

- Versions Windows **inférieures** à celles de Unix

Utilisation du package Cygwin sous Windows

► “*Use the power of command shells*”

La force du shell (2)

- Plusieurs possibilités selon le système d'exploitation
 - Invite de commande, PowerShell, GitBash... sous Windows
 - Ash, Bash, Zsh... sous Linux/Unix
 - Terminal, iTerm2, TotalTerminal... sous MacOS
- Possibilité ou non d'écrire des scripts exécutables

Par exemple des .bat sous Windows, des .sh sous Linux...

Le besoin d'édition (1)

- Mieux vaut **connaitre** un éditeur et savoir bien l'utiliser

Édition de code, documentation, memos...

- Il faut être efficace et **compétent**

Connaitre et savoir utiliser les raccourcis clavier

- **Trois fonctionnalités** de base

- **Configurable**, police, couleur, raccourci clavier...
- **Extensible**, intégrer nouveau compilateur...
- **Programmable**, scripts et macros...

► “*Use a single editor, well*”

Le besoin d'édition (2)

- Éditeur exécuté au sein d'un **terminal**

GNU Emacs, Vi, Vim...

- Éditeur comme programme indépendant

Sublime Text, Atom, Brackets, Microsoft Visual Studio Code...

- Pleins d'**autres fonctionnalités** avancées

- Coloration syntaxique, auto-complétion, auto-indentation
- Templates ou code de références selon le langage
- Fonctionnalité IDE comme compilation, debugging, versioning
- Possibilités d'extensions et modules complémentaires

Contrôler son code source

- Garder trace de tous les **changements effectués** dans un code

Outils de gestion de version

- Permet de répondre à **plusieurs questions** (pour audit)

- Qui a modifié cette ligne de code ?
- Quelles sont les différences d'un fichier depuis la semaine passée ?
- Combien de lignes de code ont été modifiée dans cette release ?
- Quels fichiers ont été modifiés le plus souvent ?

- Stockage dans un **dépôt centralisé**, facile pour l'archivage

► “*Progress, far from consisting in change, depends on retentiveness. Those who cannot remember the past are condemned to repeat it.*”
— Georges Santayana, *Life of Reason*

Bug



Rear Admiral Grace M. Hopper, USN, Ph.D.

Debugging

- Depuis le 14^e siècle, **bug** associé à « *objet de terreur* »
 - Sujet **sensible et émotionnel** pour beaucoup de développeurs
 - Déni, pointage du doigt, excuses boiteuses, apathie...
 - Aborder le debugage comme un problème à résoudre
 - Se mettre le défi de réussir à le résoudre
 - Attention à la **myopie du debugage**

Ne pas se contenter d'éliminer les symptômes observables
- “*The easiest person to deceive is one's self.*” — Edward Bulwer-Lytton, *The Disowned*

Stratégie de debugging

- **Visualiser** les données

Visualisation des données sur lesquelles le programme opère

- **Retracer** l'exécution

Suivre le flux d'exécution du programme, analyser les logs

- L'élément de **surprise**

Ne pas se baser sur des hypothèses, mais faire des preuves

Rubber Ducking



WICKEDCAMPERS.CO.UK

Book online

JUST BECAUSE YOU'RE NOT PARANOID

REGUS

P859
YGJ

GB

DOESN'T MEAN THEY'RE NOT OUT
TO GET YOU!!



Paranoïa

Paranoïa

- Le software **parfait** n'existe pas, il faut l'accepter et vivre avec
 - Tout le monde sait qu'on est le **meilleur conducteur** sur terre
“L'enfer c'est les autres” (Sartre), on conduit donc défensivement
 - Identifier les **problèmes** avant leur apparition
Anticipation l'inattendu, et être prêt à le gérer
 - Programmer **défensivement**
Valider les données de l'extérieur, poser des assertions, tester la consistence et poser des contraintes sur les bases de données
- “*You can't write perfect software.*”

Design by Contract

- Un **contrat** définit les droits et responsabilités des deux parties
Accord sur les répercussions en cas de non-respect
 - **Design by contrat** par Bertrand Meyer pour Eiffel
Technique puissante basée sur la documentation
 - Un programme **correct** fait ni plus ni moins qu'il annonce
Au niveau du programme, classes, routines, blocs d'instructions...
- “Nothing astonishes men so much as common sense and plain dealing.” — Ralph Waldo Emerson, Essays

Spécification

- **Préconditions** d'une méthode

Ce qui doit être vrai avant l'appel à la routine

- **Postconditions** d'une méthode

Garantie sur ce que la routine va faire, si préconditions respectées

- **Invariants** de classe

Conditions toujours vérifiées par rapport à l'extérieur

```
1  /**
2  * @pre contains(n) == false
3  * @post contains(n) == true
4  */
5  public void insertNode (final Node n);
```

- Java Modelling Language pour faire de la DBC en Java

Utilisation d'annotations particulières dans le code source

- Écriture formelle des préconditions et postconditions

À l'aide des mots-clés requires et ensures

- Disponibilité de plusieurs outils utilisant JML

jmlc, escjava2, jml...

```
1 //@ requires x >= 0.0;
2 /*@ ensures JMLDouble
3   @      .approximatelyEqualTo
4   @      (x, \result * \result, eps);
5 */
6 public static double sqrt (double x)
7 {
8     /* ... */
9 }
```

Un programme mort ne ment pas

- Toujours vérifier le retour des appels extérieurs

Ne pas se contenter du « Ça peut arriver »

- Ne pas hésiter à crasher son programme en cas d'erreur

Si malloc fait une erreur, peut-être pas continuer

- ▶ “Crash early, don’t trash”

Assertion

- Certaines **assertions** peuvent être posées
 - Ce programme ne sera pas utilisé plus de trente ans, deux chiffres suffisent pour stocker la date
 - Ce programme est destiné à la France, pas besoin d'i18n
 - La variable count ne peut pas être négative
- Utilisation des assertions en **mode développement**

Elles ne peuvent pas avoir d'effets de bord

► “*If it can't happen, use assertions to ensure that it won't happen.*”

Exception

- Code de réaction à des situations exceptionnelles

Tout en gardant le flux d'exécution normal clair

- Quand faut-il utiliser les exceptions ?

- Si le programme s'arrête pour uncaught exception
- Et qu'on supprime tous les gestionnaires d'exception
- Le programme tournera-t-il toujours ?

- Exemple, ouverture d'un fichier

- Le fichier doit être là, alors exception
- Le fichier peut ou pas être là, alors code d'erreur

A photograph showing silhouettes of several trees against a sky transitioning from deep blue at the top to a warm orange and yellow at the horizon. In the foreground, a small, bare tree stands on the left, while larger, leafy trees dominate the upper right. The overall mood is somber and contemplative.

Plie ou casse

Flexibilité

- Il faut écrire le code le plus **flexible et découplé** possible

Permet notamment de faciliter la réversibilité et l'évolutivité
- Plusieurs principales **techniques de découplage**
 - Loi de Déméter pour séparer les concepts
 - Écrire moins de code en pratiquant la métaprogrammation
 - Couplage temporel
 - Architectures spécifiques comme le MVC ou Blackboard

Loi de Déméter (1)

- **Loi de Déméter** (LoD) ou principe de connaissance minimale

Se résume en « Ne parlez qu'à vos amis immédiats ! »

- Soient les deux objets A et B

- A peut utiliser des services de B (appel de méthodes)
- Pas utiliser B pour accéder aux services d'un troisième objet

Toute méthode M d'un objet O ne peut invoquer que :

- 1 ses propres méthodes ;
- 2 les méthodes de ses paramètres ;
- 3 les méthodes des objets qu'elle instance ;
- 4 les méthodes de ses objets composants ;
- 5 et les méthodes des variables globales accessibles.

Violation de LoD (1)

- Pas appeler une méthode d'un **objet renvoyé**

Violation immédiate de la Loi de Déméter

```
1  public class C
2  {
3      private B b;
4
5      public void m(A a)
6      {
7          hello();
8          a.hello();
9          new Z().hello();
10         b.hello();
11         Singleton.INSTANCE.hello();
12
13         // [...]
14     }
15
16     public void hello(){}
17 }
```

Violation de LoD (2)

- Qu'en est-il pour les deux **exemples d'appel** suivants ?

Peut-on résumer LoD en « un seul opérateur . d'appel max » ?

```
1  public class C
2  {
3      private B b;
4
5      public void m(A a)
6      {
7          // [...]
8
9          a.x.hello();
10         b.x().hello();
11     }
12
13     public void hello(){}
14 }
```

Loi de Déméter (2)

■ Avantages

- Logiciel plus facilement maintenable et adaptable
- Moins de dépendances sur la structure interne des autres objets
- Applicable pour les architectures logicielles en couche

■ Inconvénients

- Augmentation du nombre de méthodes
- Surcharge en temps en mémoire avec les appels de méthode

Métaprogrammation

- Suppression des détails pour les sortir du code

Obtention d'un code configurable et facilement adaptable

- Configuration dynamique décrite avec des métadonnées

Couleur, texte, choix d'algorithme, middleware, style GUI...

- Penser déclaratif sur ce qui doit être fait, pas comment

- Design abstrait plus robuste et programme flexible et adaptable
- Personnalisation de l'application sans devoir la recompiler
- Métadonnées exprimée dans langage plus proche du domaine

► “Put abstractions in code, details in metadata.”

Dodo-code

- Ne pas écrire du **code Dodo** !

Ne s'est pas adapté à la présence d'humains et s'est éteint



Il faut pouvoir s'adapter !

Sass

■ Syntactically Awesome Style Sheets (Sass)

Langage qui étend CSS offrant des constructions de haut niveau

■ Écriture d'un fichier .sass et compilation vers un .css

```
1 @import 'main';
2
3 $font-stack:    Helvetica, sans-serif;
4 $primary-color: #333;
5
6 body {
7   font: 100% $font-stack;
8   color: $primary-color;
9 }
10
11 @mixin border-radius($radius) {
12   -webkit-border-radius: $radius;
13   -moz-border-radius: $radius;
14   -ms-border-radius: $radius;
15   border-radius: $radius;
16 }
17
18 .box { @include border-radius(10px); }
```

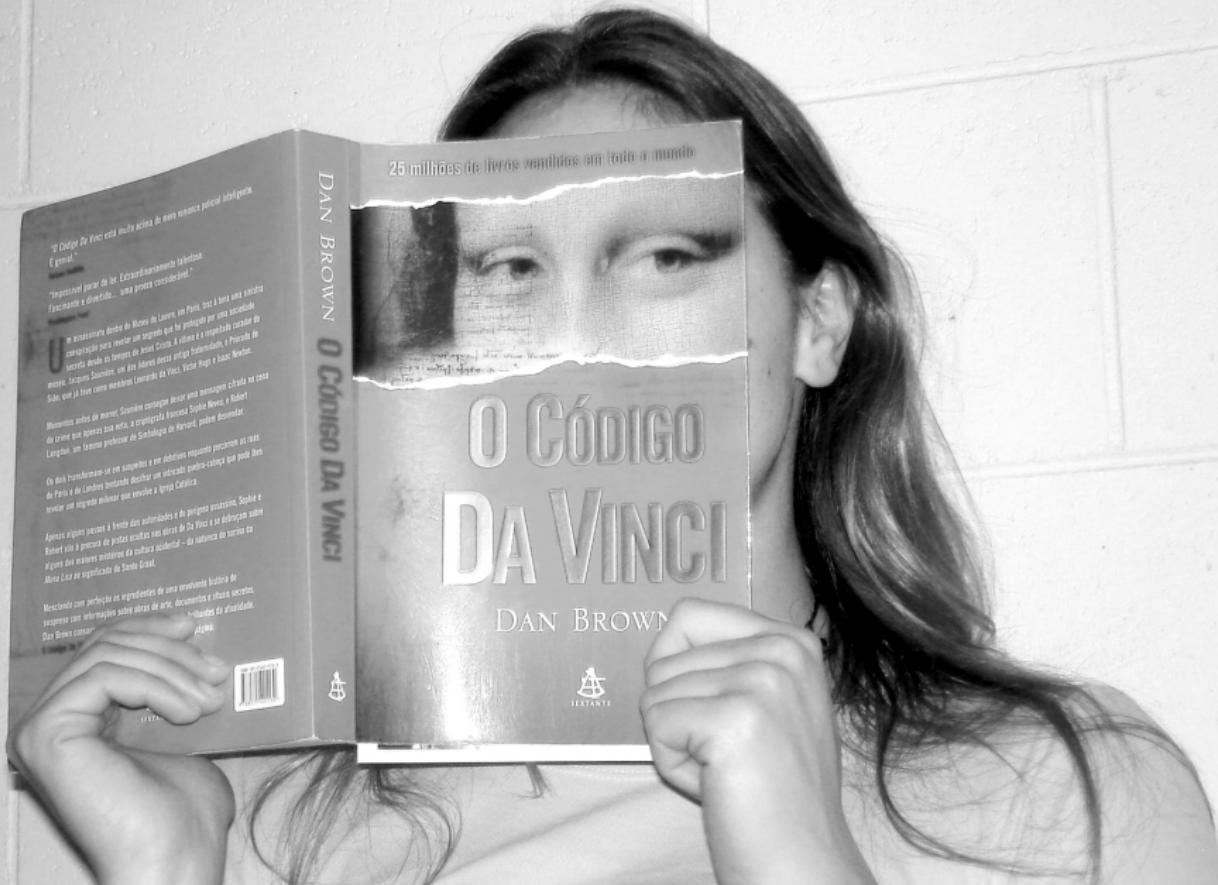
Couplage temporel

- Deux **aspects temporels** à prendre en compte dans un projet
 - Choses se passant au même moment par concurrence
 - Positions relatives des choses dans le temps créent un ordre
 - Analyse du **flux d'exécution** à l'aide d'un diagramme UML

Diagramme d'activité permet d'identifier la concurrence
 - Importance de **designer** avec support de la concurrence

Patterns spécifiques, architectures dédiées...
- “*Tick must happen before tock.*”

Pendant le codage



"O Código Da Vinci está muito ativa e é uma rotina policial intensa.
E geral."

"Impressionante parar de ler. Edificiosamente telefonou.
Funcionava e divertido... uma prova convincente."

Uma necessidade deixa de Mônaco do Louvre, em Paris, traçar para uma sociedade
conspiração para revelar um segredo que há séculos por sua estrutura
verdadeira das lendas de Jesus Cristo. A violência e a morte é a característica central da
missão. Jacques Saunière, um dos Mônacos dessa página fraternamente, o Pároco de
São João que é feio como membros Laureado da Faz. Victor Hugo e Balzac Werthe.

Momentos antes de morrer Saunière categoria deixar uma missão-chave na cena
do crime que apesar sua morte, o consigo Jacques Sophie Monroe e Robert
Langdon, um famoso professor de Simbolologia de Harvard, podem desvendar.

Os dois transformaram-se em suspeitos e os deputados engajados perceberem os rastros

de Paris e de Londres tentando descobrir um intruso quinto-categoria que pode ter

revelado um segredo milenar que envolve a Igreja Católica.

Agora alguns passam à frente das autoridades e o perigoso assassinato, Sophie e
Robert vão à procura de pistas ocultas nas obras de Da Vinci e se deslocam sobre
as grotas das mazelas mestras da cultura ocidental - da natureza de surpresa da
Mona Lisa ao significado de Santa Cris.

Mesclando com perfeição os ingredientes de uma envolvente histórica de
pistolas com informações sobre obras de arte, documentos e rituais secretos,
Dan Brown constrói

DAN BROWN

O CÓDIGO
DA VINCI

DAN BROWN



TESTANTE

Phase de codage

- Codage pas simple traduction mécanique **design en exécutable**
Programme moche, inefficace, mal structuré, pas maintenable...
- Il faut s'**impliquer** par rapport au code que l'on produit
 - Penser aux performances et donc au choix d'algorithme
 - Garder un esprit critique par rapport à son code et le réfactorer
 - Prévoir son code pour faciliter les test qu'il faudra réaliser
- Attention aux **automatismes**, comme avec conduite voiture
Toujours rester alerte et évaluer la situation

Programmer par coïncidence

- Soldat qui tâte le terrain avec un bâton pour **trouver mines**

Fait ensuite l'hypothèse qu'il n'y a aucune mines, mais...

- Attention aux **fausses conclusions** lors de la programmation

- Éviter les implémentations accidentielles

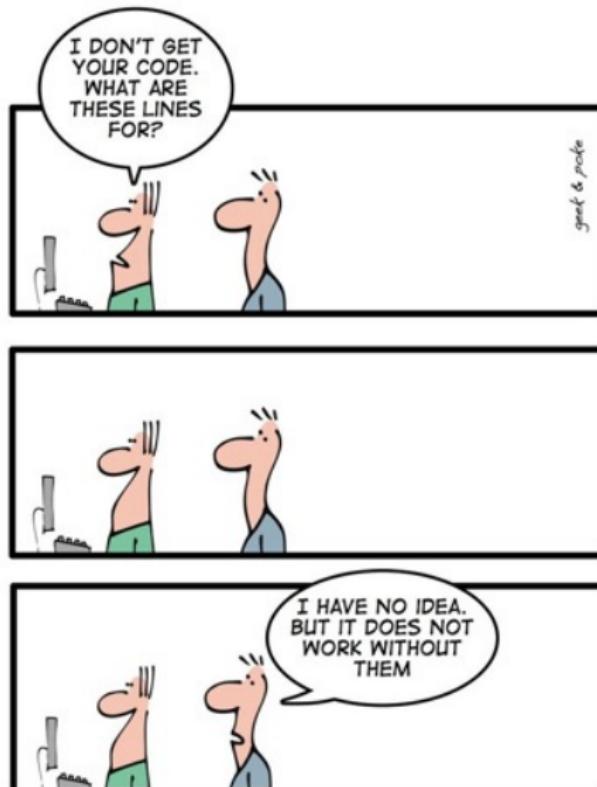
Confiance à erreur ou condition frontière non documentée

- Préférer programmer délibérément

Être conscient de ce que l'on fait, à tout moment

```
1 paint (g);
2 invalidate();
3 validate();
4 revalidate();
5 repaint();
6 paintImmediately (r);
```

Pourquoi ça marche ?



Vitesse des algorithmes

- Importance d'estimer **ressources consommées** par programme
Consommation en temps, processeur, mémoire...
 - **Analyse** précise ou approximative des algorithmes
Sens commun, instructions bas niveau, notation Big-Oh...
 - Algorithme le plus rapide **pas toujours le meilleur** pour le job
Peut dépendre des données effectivement fournies en input
- “*Premature optimization is the root of all evil.*” — Sir Tony Hoare

Refactoring (1)

- Nécessité de **réfactorer son code** alors qu'il évolue
 - Repenser décisions passées et retravailler des portions de code
 - Réécrire, retravailler, ré-architecturer
- Développement software similaire au **jardinage**

Suivre plan et conditions initiales puis modifications
- Plusieurs **indicateurs** de nécessité de réfactorer un code
 - Duplication de code, violation du principe DRY
 - Design non orthogonal
 - Connaissances outdated, meilleure compréhension du problème
 - Déplacement de code pour améliorer performances

Refactoring (2)

- Ne pas laisser un code devenir **trop bordélique**

Dans la vraie vie, pas de temps de refactoring en fin de projet

- Quelques conseils et bonnes pratiques du refactoring
 - Ne pas refactorer et ajouter nouvelle fonction en même temps
 - S'assurer d'avoir des bons tests avant de refactorer
 - Procéder par petites étapes délibérées, changements localisés

► “*Refactor early, refactor often.*”

Faciliter les tests

- Notion de “**Software IC**” par Cox et Novobilski (Objective-C)

Composant software fiable à combiner comme en électronique

- Plusieurs **tests** possibles pour des puces hardware

- Built-In Self-Test (BIST)

Diagnostics de base exécutés en interne, machine self-test

- Test Access Mechanism (TAM)

Test harness pour stimuler la chip et collecter sa réponse

- Tests similaires des **composants software** sous plusieurs formes

Test unitaire ou du contrat avec tout un faisceau de tests

► “*Test your software, or your users will.*”

Crédits

- Photos des livres depuis Amazon
- <https://www.flickr.com/photos/2ni/6779068688>
- <https://www.flickr.com/photos/jlgriffiths/9385903809>
- [https://en.wikipedia.org/wiki/File:Commodore_Grace_M._Hopper._USN_\(covered\).jpg](https://en.wikipedia.org/wiki/File:Commodore_Grace_M._Hopper._USN_(covered).jpg)
- https://www.flickr.com/photos/dyhchan_260/12671254994
- <https://www.flickr.com/photos/66176388@N00/5129088956>
- <https://www.flickr.com/photos/47500379@N08/14178337147>
- <https://www.flickr.com/photos/via/143131659>
- <https://www.flickr.com/photos/rafaelsato/3810845219>
- <http://geek-and-poke.com/geekandpoke/2009/7/25/the-art-of-programming-part-2.html>