MINES PARIS | PSL

Introduction au développement logiciel

Génie logiciel

Georges-André Silber, Centre de recherche en informatique
Mines Paris — PSL, janvier 2026

Prélude

In November 1988, a computer virus attacked computers connected to the still-nascent Internet. The virus exploited a programmer error : assuming that another computer could be trusted to send the right amount of data. It was a simple mistake, and the fix was trivial, but the programming language used was vulnerable to this type of mistake, and there was not a standard methodology for detecting that sort of problem.

– A. Barr, *The Problem with Software* (2018)

In April 2014, a computer virus attacked computers connected to the now-ubiquitous Internet. The virus exploited a programmer error : assuming that another computer could be trusted to send the right amount of data. It was a simple mistake, and the fix was trivial, but the programming language used was vulnerable to this type of mistake, and there was not a standard methodology for detecting that sort of problem.

– A. Barr, *The Problem with Software* (2018)

Programmation logicielle : pas (encore) de l'ingénierie

- Marqueurs de l'ingénierie : solidité, durabilité, fiabilité, performance, efficacité, réparabilité, évolutivité...
- Après plus de 60 ans :
- « *vulnerable programming language* »
- « *no way to detect mistakes* »
- D'autres disciplines ont su apprendre de leurs erreurs (aviation)
- ... et produire un corpus de connaissances et de méthodes.
- Résultat : quantité de bugs visibles par les utilisateurs, la réinvention de la roue en permanence, des délais et budgets peu prévisibles.

Savoir académique contre savoir industriel

- L'éducation d'un programmeur : écriture de petits programmes
- Peu de préparation à la création de logiciels à grande échelle
- Mythe du héros, épreuves de codage pour l'embauche
- Peu de réponse à : « *qu'est-ce qu'un bon logiciel ?* »
- Un diplôme en informatique ne garanti pas un savoir précis
- Très différent d'autres disciplines (médecine)
- Adam Barr : « *Is software development really hard, or are software developers not that good at it ?* »

Marie Shaw, CMU, 1990

- IEEE Software magazine : « *Prospects for an Engineering Discipline of Software* »

« Engineering relies on codifying scientific knowledge about a technological problem domain in a form that is directly useful to the practitioner, thereby providing answers for questions that commonly occur in practice. Engineers of ordinary talent can then apply this knowledge to solve problems far faster than they otherwise could. In this way, engineering shares prior solutions rather than relying always on virtuoso problem solving. »

- Beautiful Code (2007)
- Two Solitudes (ACM SPLASH 2013)
- « *How come I didn't know we knew stuff about things ?* »
- The architecture of open source applications (2012-2016)
- It will never work in theory (2011-)

Définitions

Software **: John Wilder Tukey**

- ***Binary digit*** → **Bit** (1948)
- ***Software*** (1958)
- *Logiciel* (1969)

- *Byte* (W. Buchholz, 1956)
- *Nibble* (D. Benson, 1958)
- Resp. *octet* et *quartet*

- Margaret Hamilton (1965) : *Software engineering*
- Conférences de l'OTAN (1968)
- **Méthodes de travail et bonnes pratiques des ingénieurs qui développent des logiciels**

Hardware vs Software

L'ingénierie logicielle est victime de cette fausse idée :

« *Hardware is so-termed because it is **hard** or rigid with respect to changes, whereas software is **soft** because it is easy to change.* »

— Wikipedia, article « *Computer hardware* »

Software is NOT easy to change

En réalité :

- Concevoir et mettre en œuvre du logiciel **de qualité** est **au moins aussi difficile** que pour le matériel ;
- Prévoir la **modification du logiciel** est une dimension spécifique à ce domaine ;
- La composition logicielle génère des systèmes **plus complexes** que le matériel.

Quelques critères de qualité

- Répondre aux besoins
- Produire un résultat juste
- Produire du code correct
- Produire du code maintenable et évolutif
- Produire du code efficace
- Produire du code performant
- Produire du code léger
- Prédire et respecter un délai
- Prédire et respecter un budget

Bugs

Qu'est-ce qu'un bug ?

« *A software bug is an error, flaw or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.* »

- Vol 501 d'Ariane 5 en 1996 : le bit manquant. Voir également le rapport d'enquête.
- *The Friendship That Made Google Huge*, James Somers, The New Yorker, 2018.

« *L'erreur est humaine, mais pour provoquer une vraie catastrophe, il faut un ordinateur.* »

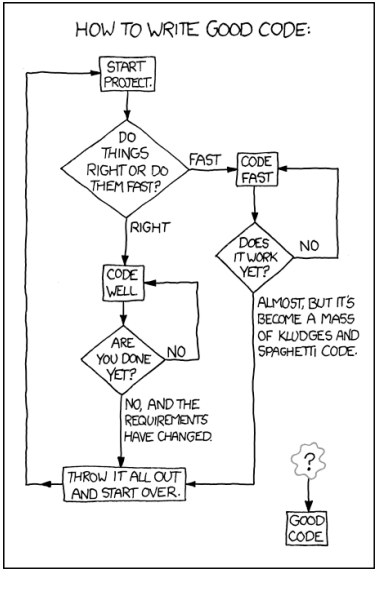
– Auteur inconnu

« *Beware of bugs in the above code; I have only proved it correct, not tried it.* »

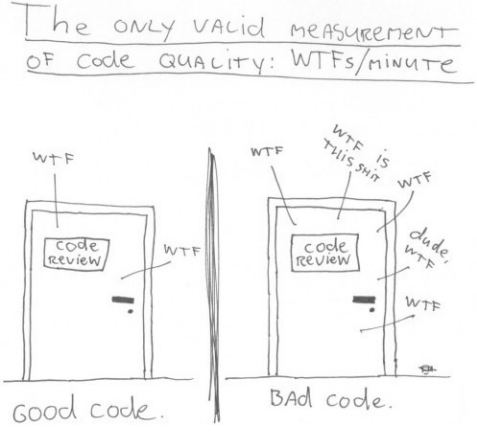
— Donald Knuth

Qu’est-ce qu’un bon code ?

Good Code ?



WTFs / minute

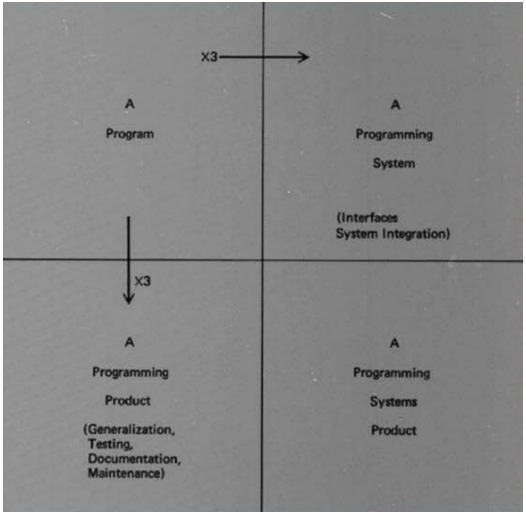

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.

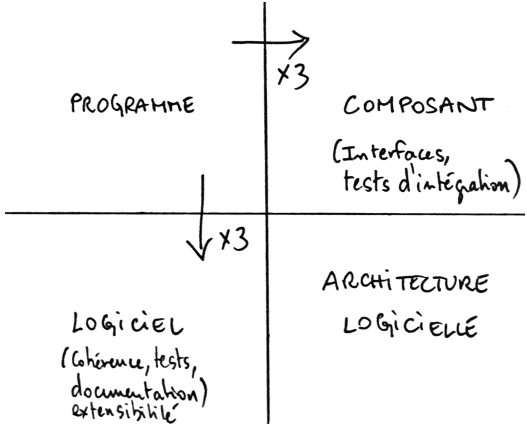
Il n’y a pas de balle magique

Frederick Brooks : The Mythical Man-Month (1975)

- Essai sur l'organisation et les méthodes de travail des équipes de développement ;
- Loi de brooks : ajouter des programmeurs sur un projet en retard accroît le retard ;
- No *silver bullet* : difficultés accidentelles vs essentielles ;
- Différences de productivité entre programmeurs (*chief surgeon*) ;
- Difficultés de la planification.

F.P. Brooks : programme vs logiciel





F.P. Brooks (suite)

« Of all the monsters who fill the nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, we seek bullets of silver that can magically lay them to rest. »

« The familiar software project has something of this character (at least as seen by the nontechnical manager), usually innocent and straightforward, but capable of becoming a monster of missed schedules, blown budgets, and flawed products. So we hear desperate cries for a silver bullet, something to make software costs drop as rapidly as computer hardware costs do. »

F.P. Brooks (suite)

« Not only are there no silver bullets now in view, the very nature of software makes it unlikely that there will be any inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware. We cannot expect ever to see twofold gains every two years. »

F.P. Brooks (suite)

« Most of these attack the central argument that there is no magical solution, and my clear opinion that there cannot be one. Most agree with most of the arguments in "NSB," but then go on to assert that there is indeed a silver bullet for the software beast, which the author has invented. As I reread the early responses today, I can't help noticing that the nostrums pushed so vigorously in 1986 and 1987 have not had the dramatic effects claimed. »

Quelques balles magiques

- Programmation structurée
- Preuves formelles
- Programmation orientée objet
- *Design patterns*
- Méthodes « *agiles* »
- Programmation fonctionnelle
- DevOps
- DevSecOps
- ...

Dijkstra 1972

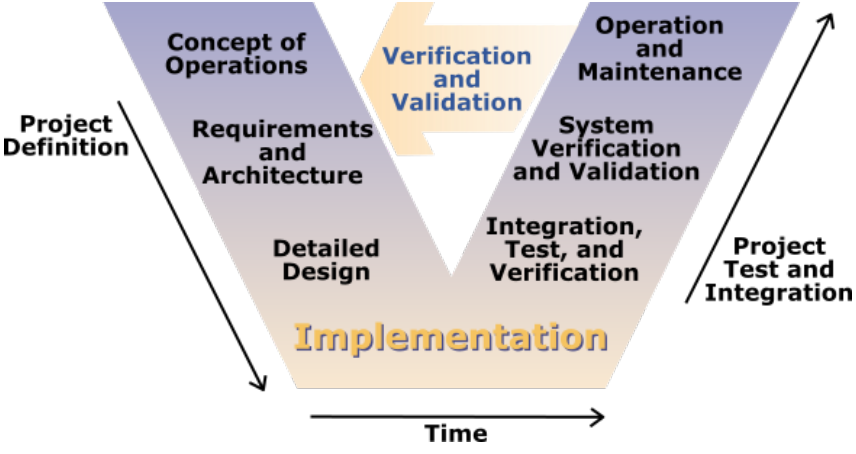
« It would be very nice if I could illustrate the various techniques with small demonstration programs and could conclude with "... and when faced with a program thousand times as large, you compose it in the same way." This common educational device, however, would be self defeating as one of my central theme will be that any of two things that differ in some respect by a factor of already a hundred or more, are utterly incomparable. »

Harlan D. Mills 1980

« It is characteristic in software engineering that the problems to be solved by advanced practitioners require sustained efforts over months or years from many people, often in the tens or hundreds. This kind of mass problem-solving effort requires a radically different kind of precision and scope in techniques than are required for individual problem solvers. If the precision and scope are not gained in university education, it is difficult to acquire them later, no matter how well motivated or adept a person might be at individual, intuitive approaches to problem solving. »

Méthodologies

Cycle en V


Concept of Operations
Requirements and Architecture
Detailed Design
Implementation
Integration, Test, and Verification
System Verification and Validation
Operation and Maintenance
Verification and Validation
Project Definition
Project Test and Integration
Time

Problèmes

- Faire travailler ensemble utilisateurs, architectes et développeurs ;
- Les spécifications peuvent changer au cours du temps ;
- Difficile d'anticiper les problèmes ;
- Difficile d'évaluer le temps nécessaire l'implémentation.


Planning/feedback loops
Release plan
Months
Iteration plan
Weeks
Acceptance test
Days
Stand-up meeting
One day
Pair negotiation
Hours
Unit test
Minutes
Pair programming
Seconds
Code

Manifesto for Agile Software Development

- https://agilemanifesto.org
- Ingénierie logicielle pragmatique
- Travail plus **incrémental**, **adaptatif**
- **Interactions** et **Communication**
- Not a *silver bullet*.

Méthode pragmatique et simple

- Que faire ?
  1. Déterminer où vous êtes
  2. Faire un petit pas vers votre objectif
  3. Ajuster votre compréhension avec ce que vous avez appris
  4. Aller en 1.
- Comment ?
  - Face à au moins deux alternatives apportant le même gain, choisir celle maximisant les possibilités de modifications ultérieures

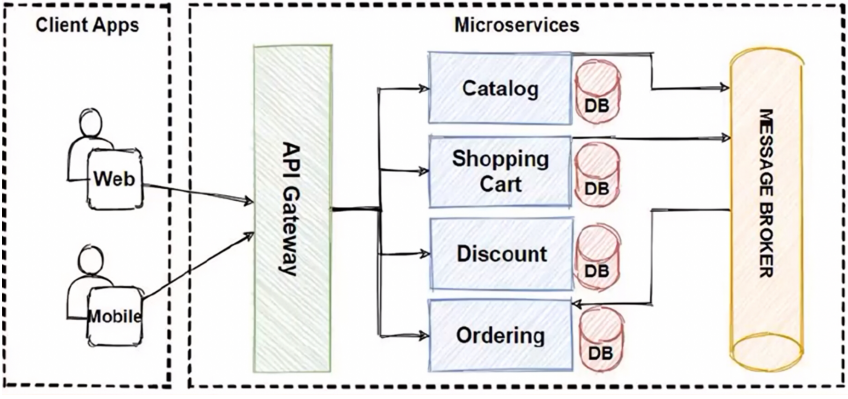
1. Récolter correctement les besoins

- Comprendre les besoins
- Commencer à proposer des solutions
- Proposer des alternatives
- Laisser la porte ouverte à des changements

2. Prendre le temps de concevoir l’architecture

« *Si vous ne savez pas ce que votre programme est censé faire, vous feriez bien de ne pas commencer à l'écrire.* »

- Bien réfléchir aux interfaces
- Tendre vers un couplage minimal
- Réfléchir en services
- Indispensable : **spécifications**


Client Apps
Web
Mobile
API Gateway
Microservices
Catalog
DB
Shopping Cart
DB
Discount
DB
Ordering
DB
MESSAGE BROKER

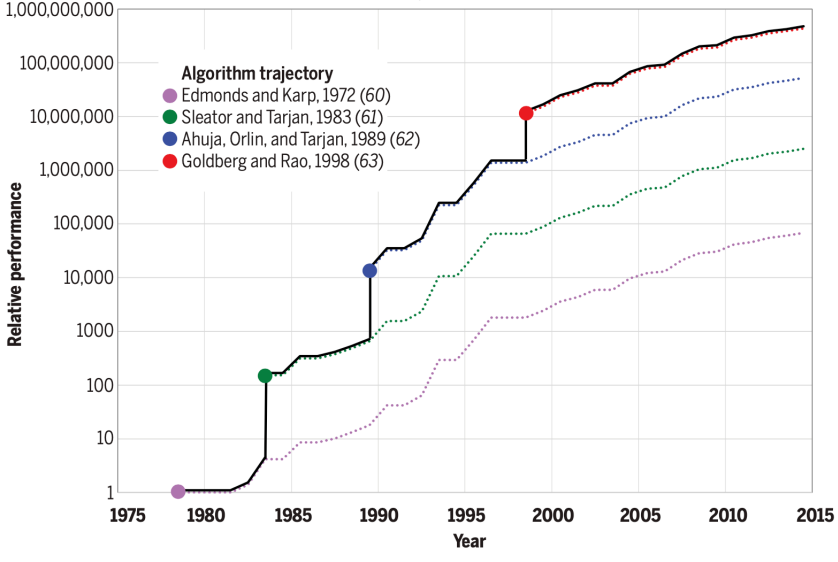
3. Choisir ou créer les bons algorithmes

Qu'est-ce qu'un algorithme ?

- Une formalisation d'une **suite finie d'opérations de calcul** élémentaires,
- résolvant un **type de problème** en se **terminant toujours**,
- exécutable par un **humain** avec du papier et des crayons en une **durée finie**,
- dont on peut **prouver la correction**, la **complexité**,
- l'**optimalité** (le meilleur algorithme possible).

Il existe des problèmes **non calculables**, pour lesquels on ne peut pas trouver d'algorithme.

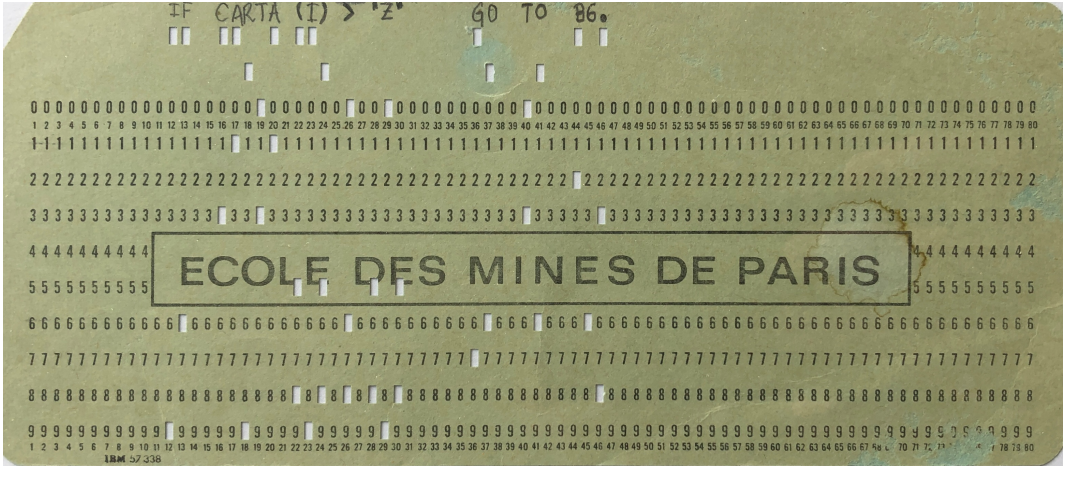
Avancées algorithmiques (ex. Max. Flow Problem)



4. Choisir ou créer le langage approprié

Langages

- « *Il n'y a pas de langage informatique dans lequel vous ne puissiez écrire de mauvais logiciels.* »
- « *Il n'y a pas de langage informatique dans lequel vous ne puissiez écrire de bons logiciels.* »
- Différences : effort à fournir, gardes-fous
- Choisir le bon outil en fonction du problème à résoudre
- Choisir le bon écosystème (bibliothèques, environnements de développement)
- Metaprogramming, DSL
- Approfondir plusieurs langages
- Ne pas suivre la mode (TIOBE)
- Pourquoi un langage a-t-il du succès ?


IF CARTA (I) > 'Z' GO TO 86.
ECOLE DES MINES DE PARIS
IBM 57 338

5. Produire le moins de code possible

« *Software expands to fill the available memory. (Parkinson)* »

« *Software is getting slower more rapidly than hardware is becoming faster. (Reiser)* »

— Niklaus Wirth, *A Plea for Lean Software*, 1995.

« *Software : it's a gas !* »

— Nathan P. Myhrvold, *The Next Fifty Years of Software*, 1997.

6. Produire du code simple

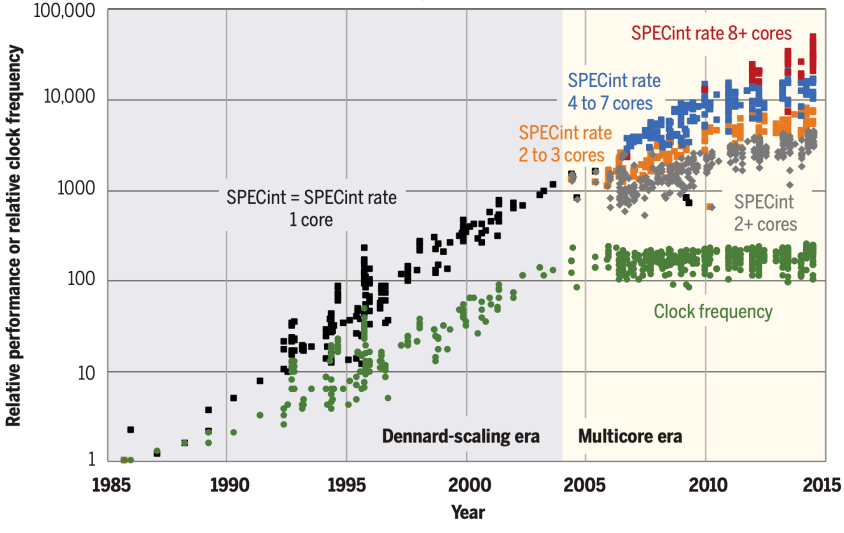
- Nous codons dans un monde qui change
- Un code simple et bien conçu est plus facile à changer
- Des solutions bien structurées et découplées induisent un code plus facile à changer, déployer, maintenir et réutiliser
- Un code plus simple est plus fiable

7. Produire du code efficace

1959 : *There's Plenty of Room at the Bottom*

- Phrase de Richard Feynman
- **Miniaturisation** des **semi-conducteurs** : moteur de l'accroissement des performances pendant 50 ans
- Loi de **Moore** : **doublement** du nombre de transistors par puce tous les **deux ans**
- Loi de **Dennard** (MOSFET) : **même niveau** de consommation d'**énergie** à **surface constante**
- Ces deux **lois empiriques** sont aujourd'hui **fausses**

Fin de la loi de Dennard (2004)



There's plenty of room at the Top : What will drive computer's performance after Moore's law ?
C.E. Leiserson et al., Science 2020.

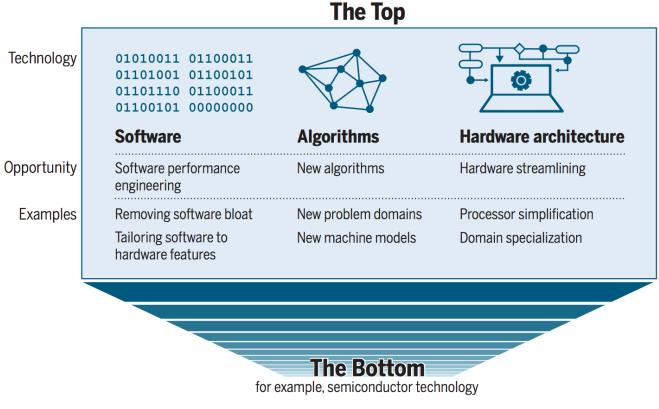


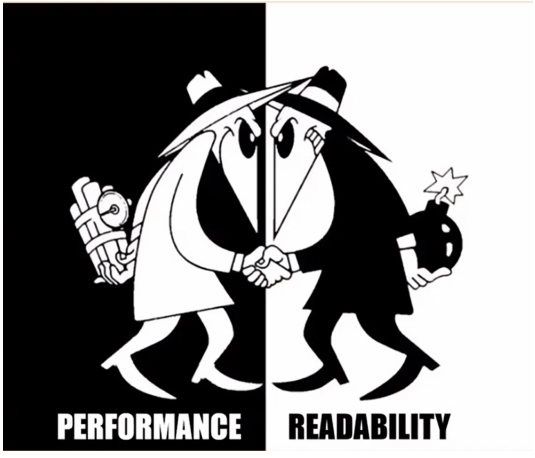
Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

Ne pas être obnubilé par les performances

« There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amount of their time thinking about, or worrying about, the speed of non critical parts of their programs, ant these attemps at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time : premature optimization is the root of all evil. »

– D. Knuth en 1974


PERFORMANCE
READABILITY

7. Prouver ou tester automatiquement

« *Program testing can be used to show the presence of bugs, but never to show their absence.* »

— Edsger W. Dijkstra

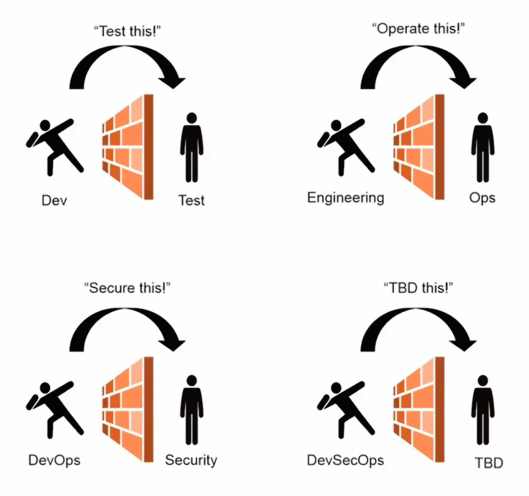
Familles de tests

- **Tests unitaires** : vérification du bon fonctionnement d'une partie précise d'un programme.
- **Tests d'intégration** : vérification de l'interaction de différents composants entre eux. Par exemple, l'interaction entre un composant et une base de données, ou entre l'API de deux composants.
- **Tests fonctionnels** : vérification du comportement de des fonctions de l'application vis-à-vis des utilisateurs.
- **Tests de bout en bout** : tests fonctionnels complexes testant des suites de fonctions, par exemple le parcours complet d'un utilisateur sur une application, de sa connexion à sa déconnexion.
- **Tests d'acceptation** : tests fonctionnels avec des critères complémentaires, comme le temps de réponse.
- **Tests de montée en charge** : permettent de *stresser* l'application pour vérifier son comportement, identifier les goulots d'étranglement.

Automatiser les tests

- Automatiser les tests le plus possible
- Indispensable lors des évolutions : non régression
- `https://docs.pytest.org/en/6.2.x/`
- Stress tests : exemple de Locust

Throw over the wall



Preuves formelles

- Interprétation Abstraite
- Sémantique axiomatique (C.A.R. Hoare)
- Compilateur C prouvé : CompCert `https://compcert.org`

8. Mettre en place des règles de codage homogènes

- Aide à la lecture du code
- https://www.gnu.org/prep/standards/
- https://google.github.io/styleguide/pyguide.html

10. Commenter et documenter

Commentaires

- Commenter le moins possible
- Rendre le code le plus évident possible
- Exemple de ce qu'il ne faut pas faire

Documentation

- Documenter en codant
- En Python : docstrings, Sphinx
- `https://readthedocs.org`
- Exemple de pandas
- Documentation, théorie et pratique : diataxis

Programmation littéraire

- Concept introduit par Donald Knuth
- Le même fichier source contient de manière entrelacée le code et sa description
- cweave, ctangle
- Exemple de la Stanford GraphBase
- Exemple de ”TeX : The Program”, livre et code source
- `https://www-cs-faculty.stanford.edu/~knuth/cweb.html`
- Code source : `https://github.com/ascherer/cweb`

11. Produire du code lisible

Exemple : choisir des noms signifiants

```
class Data(object):
  """ Duration.
  """

  def __init__(self):
    # elapsed time in days
    self.d = 0
```

```
class Duration(object):

  def __init__(self):
    self.elapsed_days = 0
```

12. Utiliser les bons outils

- Git : gestion de versions distribuée
- Github, Gitlab : serveurs Git avec outils de développement
  - Gestions de tâches, *merge/pull requests*, CI
- Plusieurs flux de développement possibles avec Git
- Lire le Git book
- Code source : `https://git.kernel.org/`

- https://datatracker.ietf.org/wg/secsh/about/
- Spécifications du protocole SSH : https://www.openssh.com/specs.html
- OpenSSH, implémentation la plus connue du protocole SSH : https://www.openssh.com/
- Dépôt Git de OpenSSH : https://anongit.mindrot.org/openssh.git
- Nouveau chiffrement post-quantique de SSH : https://nakedsecurity.sophos.com/2022/04/11/openssh-goes-post-quantum-switches-to-qubit-busting-crypto-by
- SSH ControlMaster : http://www.qanuq.com/2017/09/09/diminuer-temps-connexion-ssh/
- MOSH https://cat.pdx.edu/platforms/linux/remote-access/mosh/

Outils de paquetage logiciel

- GUIX
- Paquetages Debian/Ubuntu
- pacman
- rpm
- exherbo
- Dockerfile
- Vagrantfile

Virtualisation et conteneurisation

- Isolateurs : Docker, Linux Containers, Kubernetes
- Hyperviseurs de type 2 : Virtualbox, QEMU
- Hyperviseurs de type 1 : KVM
- https://xania.org/201609/how-compiler-explorer-runs-on-amazon

Automatisation de l'installation

- Pet vs Cattle
- Ansible, Puppet, Chef, CFEngine
- Metal as a Service (MAAS) https://maas.io
- https://serverfault.com/questions/433653/how-do-i-automate-os-installation-on-500-machines

13. Surveiller et mesurer

- Mesurer les performances (Ex : `https://www.datadoghq.com/`)
- Faire remonter les erreurs (exceptions, logging)
- Surveiller en permanence

14. Mettre en place un processus de revue du code

Revue de code

- Principes de l'*extreme programming*
- Soumettre des *patchs*
- Utiliser les fonctions de *pull* ou de *merge requests* de Github ou Gitlab
- Attention au *coding wars*
- Tabs vs spaces

15. Lire, apprendre, être humble

Benjamin Franklin : « *experience is a dear teacher, but fools will learn at no other.* »

Gerald Weinberg en 1971 : « *another essential personality factor in programming is at least a small dose of humility. Without humility, a programmer is foredoomed to the classic pattern of Greek drama : success leading to overconfidence (hubris) leading to blind self-destruction. Sophocles himself could not have invented a better plot (to reveal the inadequacy of our powers) than that of the programmer learning a few simple techniques, feeling that he is an expert, and then beaing crushed by the irresistible power of the computer.* »

Conseils d’A. Barr, 2022

Conseils aux managers

- Laissez du temps aux employés pour suivre les méthodes modernes de développement ;
- Récompensez cette attitude, plutôt que pour l'attitude du héro qui consiste à corriger des problèmes qui avaient été négligés ;
- Si vous pensez que vous n'avez pas suffisamment de maîtrise de ces nouveaux sujets pour échanger avec votre équipe, augmentez votre maîtrise de ces sujets.

Conseils aux développeurs séniors et aux architectes

- établissez des processus obligatoires pour l'équipe, comme les tests unitaires, l'analyse statique du code et la documentation ;
- ne croyez pas aux balles en argent mais ne les négligez pas : elles ne sont pas magiques mais ont de la valeur ;
- lisez des études empiriques, pas des livres tendances sur de nouveaux principes ou processus ;
- ne soyez pas le vieux développeur grincheux obnubilé uniquement par la performance.

Conseils aux développeurs cherchant à s'améliorer

- contribuez aux projets open-source, idéalement ceux avec de nombreux contributeurs qui ont des règles établies et des processus ;
- si vous n'avez pas le temps de contribuer, lisez le code, en essayant de comprendre notamment pourquoi les règles et processus existent.

Faire passer un entretien à un développeur

- ne vous focalisez pas sur le quadrant nord-ouest. Une question de programmation est intéressante, mais ne vous focalisez pas forcément sur l'algorithme trouvé ;
- même si quelqu'un arrive avec un algorithme malin, il est plus important que le code soit lisible et maintenable. Si il est trop "malin", la prochaine personne devant le modifier a de fortes chances de le casser ;
- idéalement, le candidat devrait avoir de l'expérience de travail sur le code d'autres personnes à présenter, ou au moins de l'expérience concernant les tests et la maintenabilité.

Si vous êtes enseignant

- faites lire plus de code à vos élèves, il y a énormément de code open source disponible ;
- la majeure partie du temps devrait être consacrée à la lecture de code, et pas à l'étude des algorithmes et de la syntaxe des langages. Pour produire du code il faut avoir lu du code. Du bon, du mauvais, du code qui a eu du succès, du code qui a échoué ;
- regardez les études empiriques des développeurs dans l'industrie ; idéalement, conduisez-en vous même ;
- n'ignorez pas les "coding camps", il seront bientôt vos concurrents.

Livres à lire

- Harlan D. Mills. ***Software Productivity.*** Dorset House, 1988. URL : https://trace.tennessee.edu/utk_harlan/11
- Frederick P. Brooks. ***The Mythical Man-Month. Essays on Software Engineering.*** Addison-Wesley, 1995
- Gerald M. Weinberg. ***The Psychology of Computer Programming. Silver Anniversary Edition.*** Dorset House, 1998 (version révisée d'un livre paru en 1971)
- Adam Barr. ***The Problem with Software. Why Smart Engineers Write Bad Code.*** The MIT Press, 2018

Ressources diverses

- WAT : https://www.destroyallsoftware.com/talks/wat
- DONKEY.BAS : https://web.archive.org/web/20130918210121/http://www.codinghorror.com/blog/files/donkey.bas.txt Voir également : https://www.retrogames.cz/play_1385-DOS.php
- Tabs vs Space (Silicon Valley) : https://www.youtube.com/watch?v=SsoOG6ZeyUI