Séminaire « Calcul numérique et outils informatiques »

Quelques bonnes pratiques de développement logiciel

Sébastien Brisard*

*Laboratoire Navier (UMR 8205), Université Paris-Est, CNRS, ENPC, IFSTTAR

Jeudi 21 novembre 2019

Toute profession a ses règles de l'art

« Il appartient donc de fixer suivant les règles de l'art le rapport des sections d'armatures dans deux directions perpendiculaires »

BAEL 91 révisé 99, A.3.2.5

C'est une notion assez floue et donc sujette à controverse !

Les informaticiens ont développé des « bonnes pratiques »

- Travail collaboratif
- « Prouver » que le code livré au client fonctionne
- Permettre la mise-à-jour (par des développeurs différents)

Elles s'appliquent à nous

Nous travaillons toujours au moins à deux sur le même code : moi(t) **et** moi($t + 6$ mois)**.**

Peyo, *La Schtroumpfette*, Dupuis, 1967

- Cet exposé présente **quelques** pratiques que je mets en œuvre
- Cela ne signifie pas que toutes les autres pratiques sont mauvaises !
- Mettre en œuvre ces règles est (un peu) chronophage
- Mais c'est un temps bien investi
- À défaut de les appliquer, sachez qu'elles existent
- Il existe des outils pour les mettre en œuvre aisément

Dans cet exposé…

Hypothèses

- Vous voulez utiliser un code sans avoir à le lire
- Vous voulez partager un code
- Vous avez adopté une architecture modulaire

Plan de l'exposé

1. Outils de base
2. Style de programmation
3. Tests
4. Documentation

Une référence intéressante

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., … Wilson, P. (2014). Best Practices for Scientific Computing. *PLoS Biology*, 12(1), e1001745. doi:10.1371/journal.pbio.1001745

Outils de base (1/4)
Environnement de développement

Éditeurs généralistes

- VSCode https://code.visualstudio.com/
- Atom https://atom.io/
- Emacs https://www.gnu.org/software/emacs/
- Vim https://www.vim.org/

Avantages et inconvénients

- Outils légers (?)
- Fonctions d'édition généralement puissantes
- Polyglottes : programmation et LaTeX
- Très configurables, parfois trop…
- Notion de projet mal définie

Env. de développement intégrés

- Spyder https://www.spyder-ide.org/
- Visual Studio https://visualstudio.microsoft.com/
- Eclipse https://www.eclipse.org/
- Code::Blocks http://www.codeblocks.org/
- KDevelop https://www.kdevelop.org/

Avantages et inconvénients

- Notion forte de projet
- Installation « clés en main »
- Outils plus lourds
- Un outil par langage (en général)

Faites votre choix…

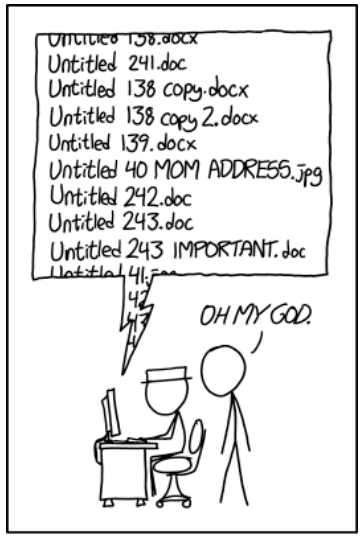



Votre environnement de développement doit vous permettre d'intégrer les outils que je vais présenter.

Passez un peu de temps à le configurer !




Outils de base (2/4)
Logiciel de gestion de versions

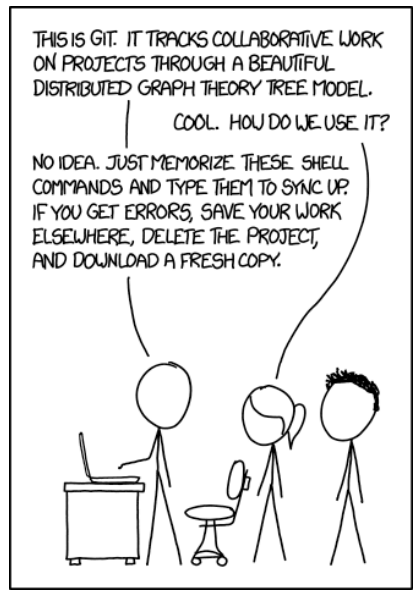


https://xkcd.com/1459

Une impression de déjà-vu ?

Avantages d'un gestionnaire de versions

- Historique des modifications
- Retour en arrière
- Travail à deux mains : gestion simple des conflits
- S'applique à un article, un mémoire de thèse…



https://xkcd.com/1597

https://git-scm.com/

- Documentation : https://git-scm.com/book/en/v2
- Intégration dans l'éditeur, outils graphiques

démo

https://gitlab.enpc.fr

- Visibilité
- Plateforme complète : gestion des droits d'accès, tickets, wiki, …
- Liens entre tickets et révisions : historique du projet
- Voir aussi : https://projets.ifsttar.fr, https://sourcesup.renater.fr

Bonnes pratiques pour git

Quelques conseils

- Enregistrer **souvent**, de **petites** modifications
- Messages courts (mais **obligatoires**)
- Tronquer les lignes des fichiers sources
- Jupyter notebook : `Cell → Clear all outputs`
- `LibreOffice` : privilégier les formats `*.fodt`, `*.fods`, … (*flat file*) (**vérifiez la réversibilité !**)

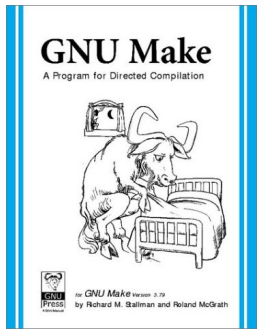
Configuration de votre éditeur

- Format de caractères (`latin1`, `utf8`)
- Tabulations remplacées automatiquement par n espaces
- Supprimer automatiquement les espaces en fin de ligne à la sauvegarde (*remove trailing whitespaces*)

Outils de base (3/4)
Outils d'aide à la compilation

Les classiques

La préhistoire : `make`



- Complexe
- Restreint aux plateformes POSIX (compatibilité limitée avec Windows)
  - R.M. Stallman, R. McGrath, *GNU Make: A Program for Directed Compilation*, GNU Press, 2002

L'usine à gaz : `cmake`

- Très puissant
- Peut très rapidement devenir plus complexe que votre programme
- Multiplateforme
  - *An Introduction to Modern CMake* : `https://cliutils.gitlab.io/modern-cmake/`

MESON

https://mesonbuild.com

```
project('scapin', 'c',
        default_options: ['default_library=both'])
math = meson.get_compiler('c').find_library('m',
                                            required : false)
glib = dependency('glib-2.0')
scapin_lib = library('scapin', ['scapin.c', 'scapin.h'],
                     dependencies: glib,
                     install: true)
scapin_test = executable('test_scapin', 'test_scapin.c',
                         link_with: scapin_lib,
                         dependencies: glib)
test('Tests of scapin', scapin_test)
```

Outils de base (4/4)
Débogueur (pour mémoire)

La programmation est un exercice d'écriture

« Règles de codage » (*coding style guide*)

Des querelles sans intérêt, aussi vieilles que l'informatique

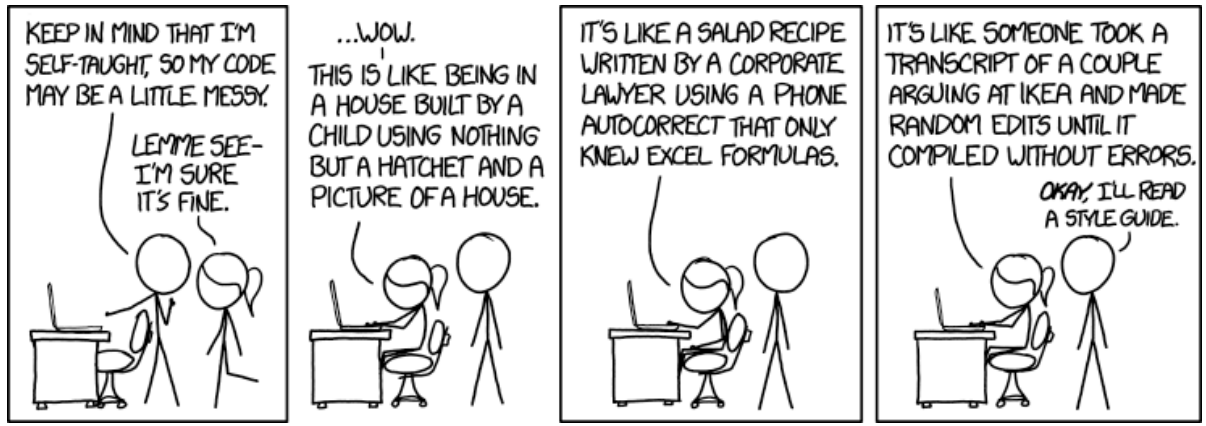
Style K&R

```
void a_function(void)
{
    if (x == y) {
        something1();
        something2();
    } else {
        somethingelse1();
        somethingelse2();
    }
    finalthing();
}
```

https://fr.wikipedia.org/wiki/Style_d%27indentation

Style GNU

```
void
a_function (void)
{
  if (x == y)
    {
      something1 ();
      something2 ();
    }
  else
    {
      somethingelse1 ();
      somethingelse2 ();
    }
  finalthing ();
}
```



https://xkcd.com/1513

- Adopter un style homogène facilite la relecture
- Peu importe le style : on s'habitue à tout
- Définissez (et documentez) des règles stylistique pour votre projet

Ne perdez pas votre temps inutilement

- Utilisez un outil de mise en forme automatique
- Joignez un fichier de configuration à votre projet
- `C`, `C++` : `clang-format` — `Python` : `black` démo

Chameaux et serpents
Conventions pour l'attribution de noms

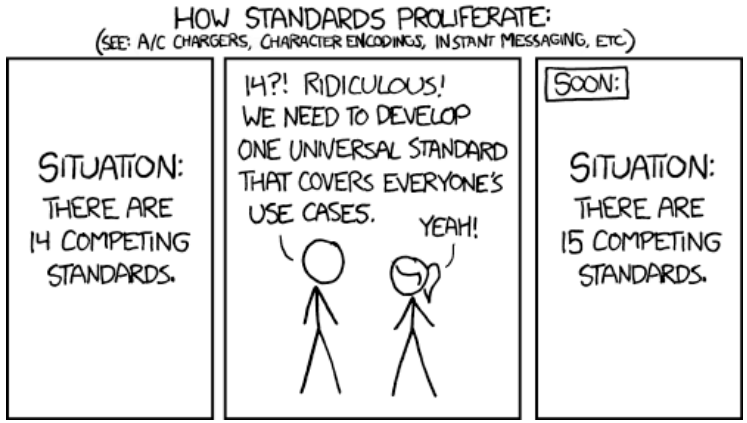
- Sujet très sensible : blessant ou saugrenu
- Très confortable de pouvoir deviner la nature d'un objet (constante, variable, fonction, classe) à la seule lecture de son nom !

Exemple : Python

- `NOM_DE_CONSTANTE`
- `nom_de_variable, nom_de_fonction`
- `NomDeClasse`

Contre-exemple pénible : LaTeX

- `\pageref` mais `\DeclareMathOperator`
- `\setcounter{enumi}{}` mais `{\setlength}{\pagewidth}{}`



https://xkcd.com/927

- Python style guide (PEP 8)

  https://www.python.org/dev/peps/pep-0008/

- Google C++ Style Guide

  https://google.github.io/styleguide/cppguide.html

- Airbnb JavaScript Style Guide

  https://github.com/airbnb/javascript

- …et bien d'autres !

- N'ajoutez pas le vôtre ! Utilisez un format existant !

Constructions idiomatiques (1/4)

“Er sah das Kind, das im Garten spielte.”

$\neq$

“Er sah das im Garten spielende Kind.”

Boucles en `Python` : `C` traduit en `Python`

```
>>> animals = ["cat", "dog", "bird"]
>>> for i in range(len(animals)):
...     print(animals[i])
...
cat
dog
bird
```

Constructions idiomatiques (2/4)

Boucles en Python : version idiomatique

```
>>> for animal in animals:
...     print(animal)
...
cat
dog
bird
```

« C'est très joli, mais j'ai besoin de l'indice »

```
>>> for i, animal in enumerate(animals):
...     print("The {}-th animal is: {}".format(i, animal))
...
The 0-th animal is: cat
The 1-th animal is: dog
The 2-th animal is: bird
```

Constructions idiomatiques (3/4)

Principe de moindre surprise

L'utilisation de constructions idiomatiques facilite la relecture par un tiers

Mais une construction inhabituelle peut vous permettre d'attirer l'attention de votre relecteur sur un point important

"Im Garten sah er das spielende Kind."

Exemple `Python`

```
>>> intg, err = scipy.integrate.quad(f, a, b)
```

Si l'erreur ne vous intéresse pas (ce qui n'est bien sûr jamais le cas)

```
>>> intg, _ = scipy.integrate.quad(f, a, b)
```

Constructions idiomatiques (4/4)

N'en faites pas trop

```
f = lambda x: [[y for j, y in enumerate(set(x))
                  if (i >> j) & 1]
                 for i in range(2**len(set(x)))]
```

(function that returns the set of all subsets of its argument,
https://wiki.python.org/moin/Powerful%20Python%20One-Liners)

N'abusez pas de constructions complexes…

…sauf si vous êtes le seul développeur de votre projet

- Macros en C
- Templates en C++
- Meta-programmation en Python

Autres recommandations stylistiques

- Ne mélangez pas français et anglais (`get_indice`)
  - c'est moche
  - plus difficile de deviner le nom d'une fonction
- Utilisez des noms explicites (vive l'autocomplétion)
  - `n` : d'accord pour une variable de courte durée de vie
  - `num_nodes` : préférable
- Si votre langage de programmation ne supporte pas les espaces de noms, utilisez un préfixe : `M_PI`, `pw85_contact_function`

Architecture (1/2)

Objectifs

- Utilisation intuitive pour un tiers (homogénéité de l'interface)
- Protection des données (**fermeture** du code)
- Faciliter les extensions (**ouverture** du code)

Quelques questions à se poser

- Signature des fonctions ?
- Quelles méthodes pour quels objets ?
- Organisation en modules ?
- « Coller » à l'interface d'une bibliothèque existante ?

Architecture (2/2)

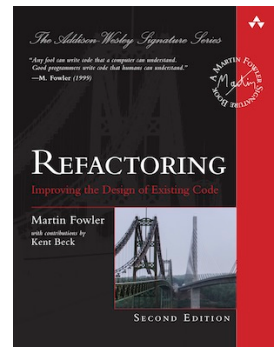
« Patrons de conception » (*design patterns*)

démo

- E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994
- https://sourcemaking.com/design_patterns
- https://www.dofactory.com/net/design-patterns

« Réusinage de code » (*code refactoring*)

- Quand votre code devient un plat de spaghettis
- Revisiter l'architecture (pas l'implémentation)
- Votre éditeur peut vous aider
- Mise en place préalable de tests indispensable !



- M. Fowler et K. Bent, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 2019

Autres recommandations (en vrac)

« Linters »

https://valgrind.org

La base

```
gcc -Wall
```

Autres outils

- `Valgrind` : analyse dynamique de code (fuites de mémoire)
  https://valgrind.org
- `Python` : `pycodestyle`, `pydocstyle`, `flake8`

Évitez les nombres magiques

- Constantes numériques non nommées
- Parfois difficile de comprendre leur sens
- Difficile de modifier leur valeur

Exemple : `FEniCS`

- À éviter

```
element = VectorElement('P', dolfin.triangle, 2, 2)
```

- Mieux

```
element = VectorElement(...  , degree, dim)
```

- Encore mieux (notez l'ordre des paramètres qui n'importe plus !)

```
element = VectorElement(..., dim=dim, degree=degree)
```

Bannissez les variables globales

- Source d'erreur très (très, très) importante
- Code très difficile à comprendre, à déboguer
- Très facile d'introduire de nouveaux bogues
- Solution n°1 : travailler la signature d'une fonction
- Solution n°2 : programmation orientée objets

Quelques mots sur la programmation orientée objets

- Encapsulation : oui
- Polymorphisme : oui
- Héritage : prudence

  (https://en.wikipedia.org/wiki/Composition_over_inheritance)

Évitez le « copier-coller » comme la peste !

Ne réinventez pas la roue (carrée)

Utilisez des bibliothèques tierces

- bien testées
- optimisées
- robustes (cas pathologiques que vous n'auriez sans doute pas traités avec une implémentation maison)

Quelques exemples (à vous de compléter cette liste !)

- PETSc (https://www.mcs.anl.gov/petsc/)
- fftw (http://fftw.org/)
- ITK (https://itk.org/)
- CGAL (https://www.cgal.org/)
- Qhull (http://www.qhull.org/)

RTD

- Lisez la documentation de la bibliothèque
- Ne faites pas d'hypothèse sur son fonctionnement

Exemple 1

"Density functional theory nuclear magnetic resonance calculations established the relative configurations of 1 and 2 and revealed that the calculated shifts ***depended on the operating system*** *when using the "Willoughby–Hoye" Python scripts to streamline the processing of the output files, a previously unrecognized flaw that could lead to incorrect conclusions."* DOI:10.1021/acs.orglett.9b03216

Exemple 2

- Aire d'un triangle dans un maillage

Programmation défensive

Ne faites pas confiance à l'utilisateur

- Validez les entrées d'une fonction exemple
- À moduler si vous avez peur d'une perte d'efficacité **(vérifiez que c'est effectivement le cas)**
- Retournez des messages d'erreur explicites

N'ignorez pas les messages d'erreur

- Traitez les éventuelles exceptions levées par une partie de votre code
- Vérifiez les codes de retour d'une fonction qui ne lève pas d'exception

Tests automatisés

Principe

- *{Unit, integration, functional} tests* : querelles sémantiques sans grand intérêt
- L'important est de tester son programme, et d'automatiser les tests
- Développement piloté par les tests (TDD)

Nous écrivons tous des petits tests rapides

- Conserver ces tests : non régression
- Enrichir au fil du temps la base de tests

Outils pour l'automatisation des tests démo

- Invitation à implémenter plus de tests
- Minimisation de la quantité de code à écrire, **paramétrisation**
- Fonctions permettant de formaliser le succès ou l'échec

Tests automatisés : bibliothèques

Liste non-exhaustive – À compléter sur le Wiki ?

- Python : éviter le module unittest – préférer pytest
  (https://docs.pytest.org)
- C : GLib
  (https://developer.gnome.org/glib/stable/glib-Testing.html)
- C++ :
  - Catch2 : très simple (https://github.com/catchorg/Catch2)
  - Google Test (https://github.com/google/googletest)
  - CTest (même syntaxe – horrible – que CMake)
    (https://gitlab.kitware.com/cmake/community/wikis/doc/ctest)
- Javascript : Jest (https://jestjs.io)

Tester un code numérique

Nous travaillons en virgule flottante !

- Ne pas tester une égalité exacte !

```
abs(act - exp) <= rtol * abs(exp) + atol
```

- Comment fixer la tolérance ? Pas au hasard…
- Comment établir les valeurs attendues ?
  - Valeurs de référence (solutions analytiques exactes)
  - Résultat approché obtenu par une **autre** méthode
- Vérifier également la **vitesse de convergence** démo

Sujet potentiel d'un prochain séminaire !

Autres recommandations

- Éviter les valeurs triviales : $x = 1$ ne permet pas de détecter une coquille x^p à la place de x^q…
- Analyse d'images : tester des images rectangulaires (possible permutation entre lignes et colonnes)
- Jeux de données aléatoires : assurez vous de toujours donner la **même valeur au germe** (répétabilité)

Documentation

Éléments constitutifs d'une doc.

- **Doc. de l'interface (*API doc*) : absolument nécessaire**
- Guide d'installation (y compris dépendances)
- Tutoriels :
  - tests automatisés bien écrits
  - vos propres simulations déjà publiées
- Doc. succincte à l'usage des développeurs
- Réf. théoriques : vos articles (mêmes notations et terminologie)

Une vidéo intéressante

"There isn't one thing called "documentation"… There are four: tutorials, how-to guides, discussion, reference."

D. Procida, PyCon Australia 2017 (`https://youtu.be/t4vKPhjcMZg`)

Ne soyons pas plus royaliste que le Roi

Anne-Louis Girodet — The Yorck Project (2002) 10.000 Meisterwerke der Malerei (DVD-ROM), distributed by DIRECTMEDIA Publishing GmbH. ISBN : 3936122202., Domaine public, https://commons.wikimedia.org/w/index.php?curid=151807

Un bon fichier `README` peut faire l'affaire

exemple

Quelques outils de documentation

Doxygen

http://doxygen.nl/

- Très puissant, formats variés
- Très lourd

Sphinx

https://www.sphinx-doc.org

- Très puissant, formats variés
- Très facile à configurer (script Python)
- Format RestructuredText absolument horrible

MkDocs

https://www.mkdocs.org/

- html seulement : combiner avec pandoc ? https://pandoc.org/
- Très facile à configurer (script Python)
- Format Markdown

Commentaire != documentation

```
if __name__ = "__main__":
    n = 0                                  # Mettre n à 0
    with open("toto.txt", "r") as f: # ouvrir le fichier
        l = f.readline()                   # lire la ligne
        n += 1                             # incrémenter n
```

- Un commentaire noyé dans le code ne sera probablement pas lu (cf. hypothèses)
- Il peut rapidement devenir obsolète
- S'il est important, il doit figurer dans la documentation de l'interface ou dans la documentation des développeurs
- Sinon, il n'est sans doute pas nécessaire si les noms de variables et de fonctions sont bien choisis (*self-describing code*)

Documentation de l'interface (1/2)

Principe

- Chaque fonction et classe accessible devrait être documentée
- Très contraignant, mais vraiment nécessaire (et utile)

Structure d'une *docstring*

1. Une description courte ($\leq$ 80 caractères)
2. Description des paramètres et de leurs éventuelles restrictions
3. Description des valeurs retournées
4. Description des exceptions éventuellement levées

Les détails d'implémentation ne figurent en général pas !

Documentation de l'interface (2/2)

Styles

- `Java` exemple
- `Python` (bibliothèque standard) exemple

Outils

- Chaque fonction est documentée dans le code (commentaires + balises spéciales)
- Commentaires extraits automatiquement et formattés
- `Doxygen` : très bien pour `C++`, moins bien pour `C`, pas adapté pour `Python` exemple
- `Sphinx` (+ `autodoc`) : pas trop mal pour Python exemple

Merci pour votre attention

sebastien.brisard@ifsttar.fr
https://cv.archives-ouvertes.fr/sbrisard
https://sbrisard.github.io