L'ART DE LA PROGRAMMATION

Published : 2017-06-21

A PROPOS DE CE GUIDE...

Ce guide fait partie d'une série de guides thématiques sur des sujets spécifiques.

Chacun de ces guides traite d'une facette particulière d'une thématique, ils
se complètent et sont conçus pour ne pas avoir à être abordés de façon linéaire.
Bien entendu, certains couvrent les bases et seront de préférence abordés avant
d'autres.

Leur objectif est de rassembler les informations essentielles sur ces différents thèmes. En effet, l'apprentissage de l'informatique est le
travail de toute une vie. Parfois, pour arriver quelque part, il faut traverser
plusieurs sujets, et chacun d'entre eux peut être l'objet de plusieurs recherches et d'un long apprentissage. Alors que quelquefois, il nous suffirait
d'une ou deux clés pour passer le cap de ce qui nous intéresse sur le moment.

C'est dans ce but-là qu'ont été rédigés ces guides, de tâcher de présenter
l'essentiel pour démarrer dans une thématique donnée, tout en donnant les
aiguillages nécessaires pour un approfondissement. Dans cette optique,
l'essentiel est abordé dans le guide, et des références sont mises à disposition
pour creuser plus loin.

Bien entendu, ils ne sont pas là pour retirer le plaisir de chercher, ils sont
plutôt là pour donnner le point de départ et rendre l'apprentissage d'un abord
plus facile.

L'ART DE LA PROGRAMMATION

La véritable programmation est un art.

Souvent, elle est encadrée par des méthodes et méthodologies qui aiguillent
le développement, qui le canalisent et permettent d'aboutir à une réalisation
qui tient la route.

Cependant, pour réaliser véritablement un "bon programme", la méthode ne suffit
pas. Il faut pouvoir allier plusieurs qualités: connaissances techniques, savoir-faire, sens de l'esthétique et intuition.

La maîtrise de ces qualités nous permet de développer notre propre méthodologie.

Tout l'enjeu réside dans le fait que, quelque soit le support utilisé,
le contenu et le fond soit compréhensible et intelligible pour d'autres.

Car c'est le fondement nécessaire à une création collective et, en même temps,
la preuve de la qualité de notre réalisation.

Dans ce tableau, les connaissances techniques constituent le socle qui permet
de savoir ce qui est possible, ce qui est efficace et robuste.

Le savoir-faire, c'est ce qui nous donne les moyens de construire, d'élaborer
quelque chose d'une certaine ampleur en ayant connaissance de ce qui marche
ou pas.

L'intuition, c'est ce qui nous aide à déterminer quelle est la meilleure approche pour la situtation donnée. L'intuition permet d'avoir un regard
évolutif, notre perception du problème et de sa solution peut changer au fil
du temps. C'est ainsi que la science informatique évolue.

C'est là que la programmation devient art.

LA MODÉLISATION

La modélisation est le processus au coeur de la programmation et du développement logiciel.

En effet, quelque soit le sujet du programme ou du logiciel, ce qui découlera
du développement sera un modèle de celui-ci.

Qu'il s'agisse d'une application informatique telle qu'un éditeur de texte, un
agent de courrier électronique, un logiciel de retouche d'image, une application web, des outils de gestion, des outils d'analyse mathématique,
scientifique ou financière, qu'il s'agisse de pilotes informatiques de bas
niveau pour le clavier, l'affichage graphique, ou pour des protocoles d'échanges réseau, qu'il s'agisse de jeux ou de logiciels industriels, ce qui
est programmé décrit un comportement qui est le reflet du modèle qui a été
défini.

C'est ce modèle qui définit ce que le programme fait: ce qu'il produit comme
sorties à partir de telle ou telle entrée. C'est ce qui fait que pour une action donnée, l'utilisateur s'attend à un résultat obtenu.

Ceci est valable pour les programmes conçus dans une logique déterministe, ce
qui constitue la majorité des cas.

Note: une logique déterministe peut renvoyer un résultat non déterministe,
exemple: renvoyer l'heure courante. La logique est déterministe puisque pour
une action donnée (requête de l'heure), la même fonction est appelée: renvoyer
l'heure courante, c'est simplement la valeur renvoyée qui est fluctuante.

Dans certaines situations par contre, des programmes sont bâtis sur des
automates non déterministes. Ce cas de figure dépasse le sujet de ce guide. A
titre d'informations, voici quelques références du web à ce propos:

- Les automates:
  http://pauillac.inria.fr/~maranget/X/421/poly/automate.html#sec127
- Illustration des automates par un exemple:
  http://benhur.teluq.ca/SPIP/inf6104/spip.php?article91

Le point de départ d'un développement informatique est donc la modélisation.

Il existe plusieurs degrés de modélisation. Ils sont décrits dans la section
suivante.

DIFFÉRENTS NIVEAUX DE MODÉLISATION

La modélisation intervient à différents niveaux du logiciel.

En principe, le premier degré est l'analyse de l'ensemble logiciel.

Ensuite, vient l'architecture, qui décrit comment le projet s'articule en composantes logicielles et matérielles. Elle comporte également la description
des interfaces entre tous les composants.

L'élaboration du projet se fait dans les étapes de conception détaillée (description des éléments de code) et d'implémentation (écriture du code).

Les tests mettent à l'épreuve le modèle conçu par rapport à un modèle de
référence, implémenté en tant que module de test.

Ce guide commence par aborder le niveau de l'implémentation dans sa forme
d'expression la plus simple. Parce que c'est plus aisé de comprendre les
techniques de modélisation sur des petites réalisation, de bas niveau, avant de
monter vers des projets de plus grande envergure. Il est à noter que le rendu
obtenu ne sera pas forcément le plus simple.

Bien entendu, les technologies choisies, telles que le matériel, le système
d'exploitation, les applicatifs en jeu et, évidemment, le langage de programmation choisi auront une influence sur la manière de réaliser la
modélisation.

LE CORPS DE PROGRAMME

Le corps de programme est le niveau d'implémentation le plus direct: c'est
celui du programme qui tient entièrement dans le bloc principal, ou du script
unique.

Il est adapté à toute réalisation simple bien cernée, à tout script utilitaire,
à tout programme qui tient dans un algorithme unique (voir la section
"Algorithmes").

Il correspond au cas de figure où l'on saisit une feuille de papier et un
crayon, que l'on gribouille le schéma général et qu'on se met à le coder
directement. L'étape "papier" est parfois même éludée par le programmeur
expérimenté.

Enfin, dans des situations où les performances sont critiques, et où chaque appel de fonction pourrait compter, un programme dans son corps unique
sera la solution adaptée pour remplir une tâche simple qui doit être effectuée
un très grand nombre de fois.

C'était le cas autrefois, lorsque les machines étaient beaucoup moins puissantes et la complexité des programmes nettement moindre, ou aujourd'hui
dans des systèmes où la complexité et les ressources sont limitées expressément
(applications embarquées pour une seule fonction minimaliste).

Ce cas de figure est donné à titre indicatif, en effet, autjourd'hui, la tendance est d'écrire avant tout de façon lisible et compréhensible. Cet aspect
sera abordé plus loin dans ce guide.

C'est pour son champ d'application que le "corps de programme" est étudié en
premier, c'est l'étape de modélisation qui permet de traduire une fonctionnalité simple en une réalisation simple, qui marche.

Elle permet de voir concrètement à quoi correspond la modélisation, puisque le
code de programme écrit correspond directement à un modèle de la fonctionnalité.

Dans le cas d'un problème mathématique, la correspondance se fait facilement,
puisque la solution mathématique est déjà un modèle. L'informatique étant bâtie
sur des lois mathématiques, le modèle propre au programme sera très proche du
modèle de la solution mathématique.

Certains langages se prêtent plus à certaines expressions mathématiques que
d'autres. Il y en a même quelques-uns spécifiques aux mathématiques, conçus
pour des logiciels de mathématiques à part entière, comme R, ou scilab (en
logiciels libres), qui permettent de manipuler directement des grands jeux de
données, des matrices et même des fonctions mathématiques. Un parcours des
différentes familles de langages de programmation est abordé dans la section
"Le choix du langage" pour donner un aperçu des grandes spécificités.

Pour l'instant, nous allons nous concentrer sur un cas particulier de la programmation impérative, celui de la programmation dite "séquentielle"

(https://fr.wikipedia.org/wiki/Programmation_s%C3%A9quentielle).

Il s'agit de tout programme exécutant une suite d'instructions dans un ordre
donné (des conditions peuvent donner lieu à des chemins différents, mais la
séquence d'instruction sera toujours la même).

Voici un exemple de programme en Python très court qui tient directement dans le
corps de programme (source:
https://groups.google.com/d/msg/comp.lang.python/xLY96-kdUcc/yIlXBtmPlzIJ):

```
#! /usr/bin/python
# -*- coding: utf-8-unix; -*-
#
# Code sample: program as a single body
#
# Functionality: act as the dos2unix utility, it converts every DOS
end of line
#                in a Unix style end-of-line in the file given as
argument.
#
#############################################################################

import sys

filename = sys.argv[1]
text = open(filename, 'rb').read().replace('\r\n', '\n')
open(filename, 'wb').write(text)
```

Il s'agit bien de programmation séquentielle car le programme est constitué
d'une suite d'instructions les unes après les autres, sans même de branches
conditionnelles !

Ce qu'il fait ?

Il ouvre en lecture le fichier donné en argument, lit tout son contenu en
remplaçant la chaîne de caractère '\r\n' par '\n' pour le stocker dans une
variable "**text**". Il écrit ensuite tout le contenu de la variable dans le même
fichier, c'est-à-dire qu'il remplace son contenu par le contenu transformé.

Tout ça en quelques instructions Python !

En fait, il fonctionne exactement comme l'utilitaire **dos2unix** (voir le "guide
sur les systèmes Unix", c'est-à-dire qu'il remplace toutes les fins de ligne qui
suivent la convention DOS (un retour de chariot, "CR" = "Carriage Return",
suivi d'un saut à la ligne, "LF" = "Line Feed). Ces caractères spéciaux (voir
"CR" et "LF" dans http://asciitable.com) sont souvent représentés comme "\r"
(*return*) et "\n" (*newline*) dans les chaînes de caractère des langages de
programmation, les shells et les utilitaires du système.

A noter toutefois que la signification de "\n" peut produire "LF" ou "CR LF"
comme résultat selon l'environnements (ex: en C, il produira "LF" sous Unix et
"CR LF" sous MS-DOS/Windows).

A savoir...

La plupart des programmes et outils d'aujourd'hui sont capables de lire les deux conventions. Cependant, certains éditeurs vous mettrons en
évidence la différence si la convention n'est pas celle du système.

Ainsi, si vous faites passer un programme Python ou une page HTML d'un système
Unix à Windows, ou inversement, l'éditeur de texte pourra vous afficher les
caractères "CR" en plus sous Unix. Par contre, le langage Python comme le
navigateur web étant conçus pour traiter les deux conventions, le programme
Python s'exercera et la page HTML s'affichera correctement.

ANALYSE

L'analyse est l'étape préalable qui conduit au choix d'une bonne modélisation.

L'analyse vise à répondre au moins à ces questions:
- que doit faire mon application ?
- comment la structurer ?

Une bonne approche pour nous aider à mener cette analyse, est de réaliser une découpe en "*fonctionnalités*" et "*entrées/sorties*".

Les grandes fonctionnalités vont ressortir en répondant à la question "Que doit faire mon application ?".

Une fois qu'elles sont connues, il faudra établir les interactions entre ces fonctionnalités et les entrées/sorties.

Cette revient à poser les questions:
- Qui fait quoi ? Pour identifier les principaux éléments en jeu.
- Comment ? Pour décrire la logique, les algorithmes.
- Avec qui ? Pour déterminer de quoi les éléments ont besoin et quels autres éléments peuvent le leur fournir.

C'est là que la structure va commencer à se dégager.

L'analyse papier est un bon support pour concrétiser cette étape et se rendre compte visuellement des éléments identifiés et de leurs interactions. Personnellement, je trouve que ça permet de se poser et de laisser venir à soi les éléments qui entrent en scène.

Si vous préférez travailler directement sur l'ordinateur, je cite quelques outils dans la section suivante sur "la modélisation et l'architecture du projet".

En tous cas, je favorise une approche visuelle et schématique qui identifie les éléments qui interagissent, leurs entrées/sorties et les fonctionnalités en jeu, je ne suis pas fan du pseudo-code. Un minimum en sera utilisé pour faire ressortir les logiques essentielles. Je m'insipire ainsi de la modélisation objet, sans chercher à être fidèle à son formalisme.

Un autre ingrédient qui est important à prendre en compte, c'est que l'analyse se fait dans la mesure du possible, de façon indépendante des technologies utilisées.

Néanmoins, dans certains cas, les technologies sont imposées, elles font alors partie des entrées.

Et dans d'autres, il y a certaines technologies qui se prêtent particulièrement bien au contexte. En principe, l'analyse doit conduire à ce choix. Cependant, une fois la technologie connue, elle peut induire une solution, auquel cas, ce serait un peu une perte de temps de faire l'analyse par deux fois. C'est le bon sens qui doit parler.

L'analyse est un processus important, alors qu'il y a des bouquins entiers sur ce sujet, je le décris en peu de mots car pour moi, ce qui est fondamental,

c'est de développer son intuition, parvenir à sentir ce qui colle bien. Evidemment, cette intuition est renforcée par l'expérience et le savoir-faire. Ce savoir-faire vous le puiserez dans les rencontres, les développements en équipe, ou dans ces livres complets. En tous cas, je suis
persuadé qu'en prenant la peine de la laisser s'exprimer cette intuition au
plus tôt, elle s'invitera progressivement à vous plus efficacement que vous ne
le pensez. Elle enrichira le savoir-faire que vous pourrez acquérir.

J'estime que l'analyse est un processus de maturation, vous allez commencer à
faire une ébauche, des choses vous plairont, d'autres seront floues, et puis ça
va gripper à certains endroits. Peut-être qu'il va falloir revoir un point ou
deux, peut-être qu'il va falloir voir les choses complètement différemment.
Parfois, quand quelque chose coince, c'est qu'il faut aborder la question sous
un angle différent.

C'est pour ça que j'aime bien le papier, je peux facilement reprendre un
nouveau schéma, tout en gardant un oeil sur les schémas précédents. Je n'ai pas
besoin de chercher à faire quelque chose de propre, avec des jolies couleurs,
et les polices qui vont bien. Je cherche juste à plaquer un modèle. Je ne
cherche pas à ce qu'il soit fini dès le départ.

Souvent, une idée me vient quelques jours plus tard, quand j'ai laissé décanter. Simplement, parce que j'aurai lu un article, que je serai tombé sur
un bout de code quelque part, ou parce que j'aurai discuté avec quelqu'un en échangeant toutes
sortes d'idées, qu'il m'aura présenté sa vision des choses.

Bref, comme vous le voyez, l'analyse est un processus.

Alors, pour conclure, les éléments essentiels sont pour moi "s'imprégner du
projet" et "répondre aux questions clés évoquées ci-dessus". Le reste suivra avec
l'expérience de chacun, au fil des intérêts qu'il portera. C'est vous qui guidez la conduite de développement que vous souhaitez voir grandir.

MODÉLISATION ET ARCHITECTURE DU PROJET

Nous avons vu deux premiers cas de modélisation:
- Réalisation d'un script avec corps unique
- Découpe en fonctions

Pour aller plus loin, nous devons pousser l'étude de la modélisation:
- Déterminer les technologies et composants logiciels dont nous aurons besoin:
  cela s'appelle l'architecture et demande une vision d'ensemble du projet à
  réaliser et une bonne compréhension des technologies.

- Utiliser la modélisation dans une approche de développement direct
  en s'appuyant sur ces critères:
  - Séparer interface (affichage et entrée) des fonctions (rendu et fonctionnalités)
  - Modéliser l'application en termes techniques sur la base de l'analyse:
    faire des choix d'implémentation, identifier les principaux objets
    et fonctions, commencer à construire la structure (interface vide)
  - Prévoir les cas d'erreurs et leur traitement
    La bonne pratique est de ne jamais laisser un programme "se vautrer".
    Néanmoins, c'est une problématique complexe et spécifique à chaque
    langage ou *framework* selon les facilités qu'ils mettent à disposition.
    Nous n'allons pas le développer ici. Il s'agit de retenir que c'est
    un critère important !

- Structurer son travail en termes d'organisation sur le disque (répertoires
  et fichiers):
  - de la modélisation obtenue en composantes logicielles, se dessine une
    répartition en sous-projets et/ou modules,
  - à partir de cette répartition, nous pouvons élaborer une arborescence pour
    notre projet ; ce sera une étape nécessaire au développement et au suivi
    de version

Représentation visuelle et notions de "programmation orientée objet"

Pour vous aider à visualiser la modélisation, vous pouvez utiliser un support
graphique. Un bon moyen de représentation est la notation UML.

Le langage UML ("*Unified Modeling Languages*") a été créé pour répondre au
besoin d'exprimer des modèles de façon consistante.

La modélisation UML est un champ d'étude très vaste, un sujet à part entière. Un de ses champs d'application est la modélisation "orientée objet"
qui est celle qui nous intéresse ici. Elle répond au besoin de conception de la
"programmation orientée objet". Nous en verrons quelques concepts de base ici.

Pour la partie analyse vue précédemment, ce qui convient le mieux pour nos
besoins du moment est le diagramme de classes, dont une explication synthétique
est reprise ici:
- http://uml.free.fr/cours/i-p14.html
- https://sourcemaking.com/uml/modeling-it-systems/structural-view/class-diagram

Par rapport aux exemples donnés, sachez qu'outre les attributs (variables
faisant partie d'une classe), s'y réfèrent aussi volontier les méthodes
(fonctions faisant partie d'une classe).

Toute la conception derrière la programmation dite "orientée objet" est que les
variables de notre programme peuvent prendre un type plus complexe que les
types ordinaires (entiers, nombres flottants, chaînes de caractère ou tableaux). Ces types sont personnalisés et se rapportent à plusieurs variables
et fonctions qui leur sont propres.

Ce nouveau type "personnalisé" s'appelle une "classe". Ses variables internes
sont appelés "attributs" de la classe et les fonctions propres à celle-ci sont
appelées "méthodes" de la classe.

Lorsqu'une variable de ce nouveau type personnalisé est créé, il se dit que
nous créons une "instance" de la "classe". Le terme "objet" désigne de façon
interchangeable "classe" ou "instance". La "classe" est en quelque sorte la
définition de l'"objet", et une "instance" désignera un "objet" particularisé.

Les liens représentés horizontalement sont des "relations", ce sont elles qui
représentent quels objets sont en interaction entre eux, et à quel titre (le
nom de la relation exprime la nature de la communication). Les interactions ont
lieu au travers des "méthodes" et des "attributs".

Les méthodes et attributs peuvent être "publics" ou internes (on dit "privés")
à la classe.

Dans certains langages, comme Java, la pratique est de ne jamais laisser un
autre objet consulter ou modifier un attribut. L'objet qui interagit avec l'objet cible doit dans ce cas nécessairement passer par une méthode de l'objet
en question pour y accéder. On parle "d'encapsulation" parce que les attributs

sont cachés de l'extérieur de la classe, seule celle-là a un droit de regard
dessus.

Ce qui signifie que beaucoup d'attributs simples, viennent également avec deux
méthodes, l'une d'accès (*getter*) et l'autre de modification
(*setter*). L'idée est de pouvoir éventuellement protéger les accès en
procédant à d'éventuels contrôles.

En Python, par contre, la pratique est de ne jamais prévoir ces deux méthodes
pour des opérations aussi élémentaires, qui reviennent à utiliser l'opérateur
d'affectation et la méthode de consultation d'une donnée. Si des opérations (de
contrôle par exemple) doivent être faites lors de ces deux actions, la classe
viendra surcharger les fonctions par défaut "**__getattr__()**" et
"**__setattr__()**".

Les méthodes sont alors utilisées pour des manipulations plus complexes.

En C++, les deux approches se rencontrent.

La terminologie "surcharger" désigne le fait de remplacer le fonctionnement
d'une méthode "héritée" d'un objet parent par un autre comportement. Par souci
d'intégrité, la nouvelle fonction fait souvent appel à celle du parent, en
complétant son traitement.

La notion "d'héritage" est une des particularités de la "programmation orientée
objet".

Dans un diagramme UML, les "acteurs" servent à désigner les éléments extérieurs
qui sont en interaction avec le système modélisé. Un utilisateur sera un
acteur, mais un autre système dont nous ne nous intéressons pas au modèle peut
aussi être représenté par un acteur.

Par exemple, si je modélise une voiture sans rentrer dans les détails, mon
acteur sera le conducteur. Il pourra appuyer sur la pédale d'accélérateur qui
a pour effet indirect d'augmenter l'accélération. Mon modèle pourra alors avoir
une méthode "**enfoncer_accelerateur()**" avec un paramètre "**course**" qui ira de 0 à
100% (ou 0 à 1) pour indiquer la course de la pédale d'accélération (jusqu'à
quel point, il a accéléré).

Si je modélise par contre uniquement le moteur, l'acteur sera la pompe à
carburant. Mon moteur pourra proposer une méthode "**ajuster_debit()**" que la
pompe viendra solliciter (dans notre modélisation, elle s'appelera en réalité
"**actualiser()**").

Autrement, ce pourrait être la pompe qui soit consultée. Elle pourra offrir un
attribut "**debit**" qui sera lu par l'objet "**moteur**" quand il en aura besoin (ou
il appellerait "**lire_debit()**" si l'approche Java est utilisée). Puisqu'un
acteur n'est pas modélisé, cette analyse sert juste à exprimer l'interface

entre le moteur et l'acteur.

Dans le contexte d'un système industriel, les acteurs peuvent donc se comporter
comme des actuateurs ou des capteurs, pour utiliser la terminologie de cet
univers-là.

En termes de modélisation, il s'agira par contre d'un choix de conception.
L'élément pourra en effet fournir à la fois l'attribut "**debit**" et déclencher la
méthode "**ajuster_debit()**" de l'objet avec lequel il interagit.

Enfin, dans les questions qui surgissaient sur le niveau de modélisation, à
savoir si c'est "**Voiture**" ou "**Moteur**", il est possible de modéliser à la fois
la voiture et le moteur.

Par exemple, dans la modélisation de la voiture seule, nous aurions:
- Un acteur: "Conducteur"
- Un objet "Voiture":
  - Méthodes:
    - actualiser_course_accelerateur(course) ; appelle calculer_vitesse()
    - actualiser_course_freins(course) ; appelle calculer_vitesse()
    - calculer_vitesse() ; calcule la vitesse en fonction de la course de l'accélérateur et des freins
  - Attributs:
    - course_accelerateur
    - course_freins
    - vitesse

Et dans la modélisation complète, nous aurions:
- Un acteur: "Conducteur"
- Un objet "Voiture": objet "Gestionnaire"
  - Méthodes:
    - actualiser() ; héritée de "Gestionnaire", surchargée pour appeler "calculer_vitesse()"
    - calculer_vitesse() ; calcule la vitesse à partir du nombre de tours/min du moteur
  - Attributs:
    - nom ; nom initialisé par le constructeur
    - moteur ; objet "Moteur"
    - freins ; objet "Freins"
    - reservoir_carburant ; objet "Reservoir"
    - reservoir_huile_freins ; objet "Reservoir"
    - pompe_carburant ; objet "Pompe"
    - pompe_freins ; objet "Pompe"
    - pedale_accelerateur ; objet "Pedale"
    - pedale_freins ; objet "Pedale"
    - vitesse
- Un objet "Gestionnaire": objet générique
  - Méthodes:
    - actualiser()
- Un objet "Pedale":
  - Méthodes:
    - actualiser_course(course) ; a un effet de pression sur "levier" et notifie "gestionnaire" de la mise-à-jour
  - Attributs:
    - nom ; nom initialisé par le constructeur
    - gestionnaire ; sera connecté à "voiture"
    - levier ; sera connecté à pompe_freins ou pompe_carburant
    - course ; mesure la course de la pédale
- Un objet "Pompe":
  - Méthodes:
    - actualiser_pression(niveau) ; calcule la pression en fonction du niveau d'ouverture de la valve et des caractéristiques du réservoir en amont
  - Attributs:

- nom ; nom initialisé par le constructeur
- p_min ; pression minimum
- p_max ; presion maximum
- debit_max ; débit maximum
- amont ; équipement connecté en amont
- aval ; équipement connecté en aval, pour notifier un changement de débit et/ou de pression
- debit ; débit actuel
- pression ; pression actuelle

- Un objet "Reservoir":
  - Méthodes:
    - ajuster_debit_pression(pompe) ; ajuste le débit en fonction du débit tiré par la pompe
  - Attributs:
    - volume ; mesure du volume actuel
    - debit_max ; débit maximum délivré par le réservoir
- Un objet "Moteur":
  - Méthodes:
    - connecter_entree_carburant() ; connecte l'équipement qui fournit le carburant en entrée du moteur
    - ajuster_debit_pression() ; ajuste le débit en fonction du débit fourni en entrée
    - calculer_tours() ; calcule la vitesse du moteur en fonction du débit de carburant
  - Attributs:
    - entree_carburant ; équipement qui fournit le carburant
    - nb_tours ; nombre de tours/minutes du moteur
- Un objet "Freins":
  - Méthodes:
    - ajuster_debit_pression() ; ajuste la pressio en fonction de la pression fournie en entrée
    - calculer_force_freinage() ; calcule la force de freinage en fonction de la pression dans le liquide de freins
  - Attributs:
    - force_freinage

En termes d'instances, il y aura pour les pompes et les réservoirs:
- pompe_carburant
- reservoir_carburant
- pompe_freins
- reservoir_freins

La représentation orienté objet permet donc d'obtenir un niveau de modélisation
très poussé et très en image de la réalité. C'est toute sa force.

Par contre, il y a lieu de bien évaluer le niveau de représentation. En effet,
dans l'exemple ci-dessus, il n'y aura d'intérêt à décomposer en objets le
système "**Voiture**" que si les objets individuels ajoutent quelque chose. Autrement, si les équations du système global suffisent, le modèle, le
développement et les tests en restent quand-même bien plus simples !

C'est exactement cela le travail de modélisation, trouver le niveau de représentativité qui convient à la situation.

Vous allez voir qu'une représentation visuelle est un véritable plus lorsqu'il
s'agit de décrire une modélisation.

Pour mettre la mettre en forme , nous avons plusieurs outils à disposition,
selon votre préférence:

- Umbrello (https://umbrello.kde.org/): outil de modélisation UML de KDE, il
  s'agit d'un véritable outil qui prend en compte les spécificités d'UML pour
  les aplliquer au développement
- Dia (http://dia-installer.de/): outil de tracé de diagrammes qui dispose de
  boîtes de diagrammes spécifiques pour UML (gère uniquement l'aspect visuel)
- Inkscape (https://inkscape.org): outil de dessin vectoriel, toute la partie
  UML est à faire à la main (il existe des tutoriels, mais pas de /plugins/
  tout prêt)
- Draw (https://fr.libreoffice.org/discover/draw/): outil de dessin vectoriel
  des suites LibreOffice/OpenOffice.org, qui permet de tracer des boîtes,
  toute la partie UML est à faire à la main

Les logiciels qui intègrent les diagrammes UML comme **Umbrello** et **dia**
offrent des boîtes respectant la notation UML. Ces boîtes ont des propriétés
éditables qui reprennent les différentes entrées possibles, telle que nom de
classe, attributs et méthodes.

C'est donc une aide substantielle pour la réalisation d'un diagramme. Ils
invitent donc à se conformer plus à la norme, ce qui permet aussi de s'appuyer
sur un langage commun. Par ocontre si vous souhaitez plus de liberté
d'expression, vous êtes un peu plus restreint.

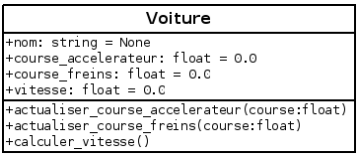
A ce titre, **Umbrello** est un outil de conception informatique, axé sur la
notation UML, alors que **dia** est avant tout un éditeur de diagrammes qui
intègre la notation UML parmi sa panoplie.

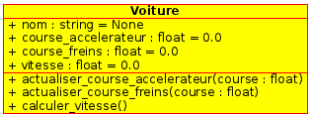
Le premier se prête bien pour se conformer au développement, et même d'établir
un lien avec le code. Le second est plus léger en termes de modélisation,
puisqu'il s'attache exclusivement à l'aspect visuel. En revanche, il offrira
des possibilités graphiques plus variées. Note: la dimension des boîtes
s'aujstent selon les champs renseignés, il faut donc jouer sur le niveau de
zoom pour avoir un rendu visuellement satisfaisant.

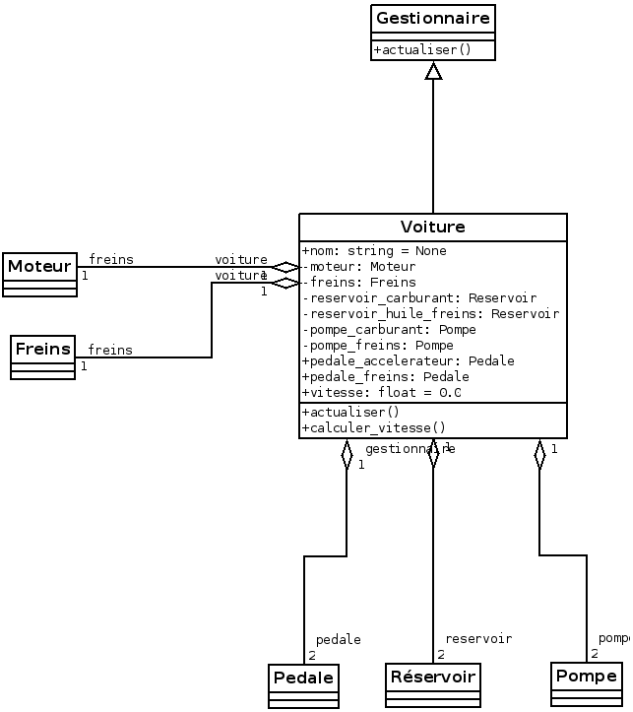
A côté de cela, des outils comme **Inkscape** et **Draw** offrent toutes les
facilités de créativité, vous pouvez donc exprimer la représentation qui vous
convienne. L'essentiel est qu'elle soit parlante, compréhensive et que vous
soyez efficace dans la réalisation.

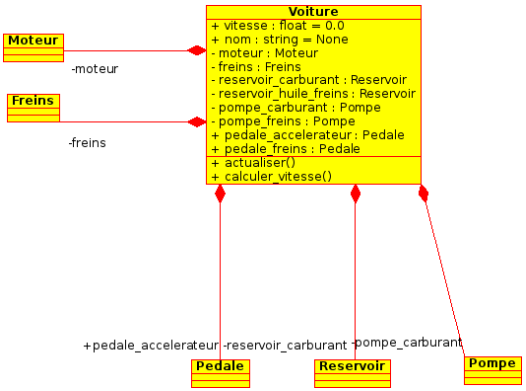
Ci-joint le diagramme de notre système "Voiture-Pompes-Moteur-Freins" modélisé
avec **Umbrello** et **dia**. Avec pour comparaison le modèle "Voiture" seul. Dans
le diagramme détaillé, seule la classe voiture a été explicitée à fin
d'illustrations.

La section qui suit donnera la représentation informatique correspondant à ces
diagrammes.


Voiture
+nom: string = None
+course_accelerateur: float = 0.0
+course_freins: float = 0.0
+vitesse: float = 0.0
+actualiser_course_accelerateur(course:float)
+actualiser_course_freins(course:float)
+calculer_vitesse()


Voiture
+ nom : string = None
+ course_accelerateur : float = 0.0
+ course_freins : float = 0.0
+ vitesse : float = 0.0
+ actualiser_course_accelerateur(course : float)
+ actualiser_course_freins(course : float)
+ calculer_vitesse()


Gestionnaire
+actualiser()
Voiture
+nom: string = None
-moteur: Moteur
-freins: Freins
-reservoir_carburant: Reservoir
-reservoir_huile_freins: Reservoir
-pompe_carburant: Pompe
-pompe_freins: Pompe
+pedale_accelerateur: Pedale
+pedale_freins: Pedale
+vitesse: float = 0.0
+actualiser()
+calculer_vitesse()
Moteur
freins
1
voiture
1
Freins
freins
1
gestionnaire
1
1
1
pedale
2
reservoir
2
Pedale
Réservoir
Pompe



Représentation informatique et "programmation orientée objet"

Nous allons voir comment écrire en termes informatiques les exemples de
modélisations vus dans la section précédente. Le langage choisi sera Python.

Commençons par le modèle "Voiture" seul:

```
#! /usr/bin/python
# -*- coding: utf-8-unix; -*-

class Voiture(object):
    VITESSE_MAX=120

    def __init__(self, nom):
        self.nom = nom
        self.course_accelerateur = 0.0
        self.course_freins = 0.0
        self.vitesse = 0.0
        return

    def actualiser_course_accelerateur(self, course):
        self.course_accelerateur = course
        self.calculer_vitesse()

    def actualiser_course_freins(self, course):
        self.course_freins = course
        self.calculer_vitesse()

    def calculer_vitesse(self):
        # calcule la vitesse en fonction de la course
        # de l'accélérateur et des freins
        self.vitesse = self.course_accelerateur * self.VITESSE_MAX
        # Formule à élaborer pour le freinage...
        return

def main():
    # Création de l'instance "ma_voiture" de l'objet "Voiture"
    ma_voiture = Voiture("Ma Voiture")
    # A la création de l'objet, la voiture est à l'arrêt
    # L'affichage de la vitesse doit donner "0":
    print ma_voiture.vitesse
    # L'acteur conducteur (moi), décide d'appuyer sur l'accélérateur à
    # 15% pour atteindre une petite vitesse de départ (note: une
    # accélération constante est utilisée, l'accélération sur l'appui
    # de la pédale n'est pas prise en compte)
    ma_voiture.actualiser_course_accelerateur(0.15)
    # Affichage de la vitesse après démarrage de la voiture
    print ma_voiture.vitesse

if __name__ == "__main__":
    main()
```

Un brin d'explications...

La ligne "**class Voiture(object)**" définit l'élément "**Voiture**" comme étant
une classe qui dérive de la classe générique "**object**", cela revient à dire
que c'est un objet Python.

Les fonctions définies au sein de la classe (c'est-à-dire, celles indentées par
rapport à celles-ci) sont ses méthodes.

Le première méthode **__init__(self, nom)** est ce qu'on appelle un constructeur,
c'est la méthode qui est appelée à la création de l'objet, ce qui arrive dans
la fonction **main()** du programme principal, à la ligne "**ma_voiture = Voiture("Ma Voiture")**". A ce moment, je crée une instance de la classe
"**Voiture()**", que j'appelle "**Ma Voiture**" et que je choisis de référencer par
la variable "**ma_voiture**".

J'appelle ensuite les méthodes de la classe relativement à l'instance que je
viens de créer. C'est le mot-clé "**self**" qui désigne l'instance à laquelle la
fonction se rapporte.

La variable **VITESSE_MAX** est dite variable de classe car elle est définie au
niveau de la classe et non pas de l'instance. C'est-à-dire que toutes les
instances partageront la même valeur (puisqu'elle n'est pas pas initialisée à
partir de **self.VITESSE_MAX**. Le fait de l'écrire en majuscules est une
convention qui stipule qu'il s'agit d'une constante, c'est donc une constante
de classe plutôt qu'une variable de classe. Il est à noter que cette nuance est
purement indicative, car hormis la rigueur de programmation, rien n'interdit de
modifier cette valeur.

L'appel à la fonction ~main()~ est encadré par un artifice "**if __name__ == "__main__"**". Cela permet d'exécuter le corps principal (ce qui suit le "**if**",
donc l'appel à la fonction **main ()**) uniquement lorsque le script est appelé
en tant que programme et non en tant que module. Dans le premier cas, la
variable **~__name__~** prend la valeur "**__main__**", tandis que dans le second
cas, la variable prendra pour valeur le nom du module importé (c'est-à-dire le
nom du fichier sans l'extension "**.py**", à moins qu'il n'en soit précisé autrement).

Et maintenant, au tour du modèle complet !

```
#! /usr/bin/python
# -*- coding: utf-8-unix; -*-

class Gestionnaire(object):
    def __init__(self):
        return

    def actualiser(self):
        return

class Voiture(Gestionnaire):
    # Constante pour évaluer la vitesse en km/h à partir
    # de la vitesse en tours/min
    VITESSE_KMH_PAR_TOURS_MIN = 120.0/3000.0
```

```
    def __init__(self, nom):
        super(Voiture, self).__init__()
        self.nom = nom
        self.moteur = Moteur()
        self.freins = Freins()
        self.reservoir_carburant = Reservoir("Reservoir Carburant",
40, 1.8)
        self.reservoir_huile_freins = Reservoir("Reservoir Huile de
Freins",
                                                1.0, 0.3)
        self.pompe_carburant = Pompe("Pompe carburant", 0.0, 3.0, 3.0,
                                     self.reservoir_carburant,
self.moteur)
        self.moteur.connecter_entree_carburant(self.pompe_carburant)
        self.pompe_freins = Pompe("Pompe freins", 0.0, 1.0, 0.1,
                                  self.reservoir_huile_freins,
self.freins)
        self.pedale_accelerateur = Pedale("Accélérateur",
                                          self, self.pompe_carburant)
        self.pedale_freins = Pedale("Freins",
                                    self, self.pompe_freins)
        self.vitesse = 0.0
        return

    # Surcharge de la méthode "actualiser" héritée de "Gestionnaire"
    def actualiser(self):
        super(Voiture, self).actualiser()
        self.calculer_vitesse()
        return

    def calculer_vitesse(self):
        # calcule la vitesse en fonction de la vitesse du moteur
        # et de la force de freinage
        self.vitesse = self.moteur.nb_tours *
self.VITESSE_KMH_PAR_TOURS_MIN
        # Formule à élaborer pour le freinage...
        return

class Pedale(object):
    def __init__(self, nom, gestionnaire, levier):
        self.nom = nom
        self.gestionnaire = gestionnaire
        self.levier = levier
        self.course = 0.0
        return

    def actualiser_course(self, course):
        self.course = course
        self.levier.actualiser_pression(self.course)
        self.gestionnaire.actualiser()
        return

class Pompe(object):
    def __init__(self, nom, p_min, p_max, debit_max, amont, aval):
       self.nom = nom
       self.p_min = p_min
       self.p_max = p_max
       self.debit_max = debit_max
       self.amont = amont
       self.aval = aval
       self.pression = 0.0
       self.debit = 0.0
       return

    def actualiser_pression(self, niveau):
       # Calculer le débit et la pression en fonction du niveau
       # d'ouverture de la valve, de p_min et p_max et des
       # caractéristiques du réservoir en amont
       self.pression = self.p_min + niveau * (self.p_max - self.p_min)
       self.debit = (self.pression / self.p_max) * self.debit_max
       self.debit = min(self.debit, self.amont.debit_max)
       self.pression = (self.debit / self.debit_max) * self.p_max
       self.amont.ajuster_debit_pression(self)
       # Faire parvenir l'information à l'équipement en aval
       self.aval.ajuster_debit_pression(self)
       return

class Reservoir(object):
    def __init__(self, nom, volume_init, debit_max):
        self.nom = nom
        self.volume = volume_init
        self.debit_max = debit_max
        return

    def ajuster_debit_pression(self, pompe):
        self.debit = min(self.debit_max, pompe.debit)
        return


class Moteur(object):
    NB_TOURS_DEBIT = 3500 / 2.8
    def __init__(self):
```

```
        __init__
        self.entree_carburant = None
        self.nb_tours = 0
        return

    def connecter_entree_carburant(self, entree_carburant):
        self.entree_carburant = entree_carburant
        return

    def ajuster_debit_pression(self, pompe):
        if pompe == self.entree_carburant:
            self.debit_carburant = pompe.debit
        self.calculer_tours()
        return

    def calculer_tours(self):
        # Calculer le nombre de tours/min en fonction du débit du
carburant
        if (self.entree_carburant == None):
            self.nb_tours = 0
        else:
            self.nb_tours = self.debit_carburant * self.NB_TOURS_DEBIT
        return

class Freins(object):
    def __init__(self):
        self.pompe = None
        self.force_freinage = 0
        return

    def ajuster_debit_pression(self, pompe):
        self.pression_freins = pompe.pression
        self.calculer_force_freinage()
        return

    def calculer_force_freinage(self):
        # Calculer la force de freinage en fonction de la pression
        # dans le liquide de freins
        return


def main():
    # Création de l'instance "ma_voiture" de l'objet "Voiture"
    ma_voiture = Voiture("Ma Voiture")
    # A la création de l'objet, la voiture est à l'arrêt
    # L'affichage de la vitesse doit donner "0":
    print ma_voiture.vitesse
    # L'acteur conducteur (moi), décide d'appuyer sur l'accélérateur à
    # 15% pour atteindre une petite vitesse de départ (note: une
    # accélération constante est utilisée, l'accélération sur l'appui
    # de la pédale n'est pas prise en compte)
    ma_voiture.pedale_accelerateur.actualiser_course(0.15)
    # Affichage de la vitesse après démarrage de la voiture
    print ma_voiture.vitesse

if __name__ == "__main__":
    main()
```

En termes de programmation, ce sont exactement les mêmes notions que dans
l'exemple précédent. Elles sont simplement utilisées plus abondamment.

Il y a tout de même quelques particularités supplémentaires.

L'appel au mot-clé **super(Voiture, self)** sert à appeler la méthode la classe
parente de **Voiture()**, à savoir **Gestionnaire** tel que défini dans "**class
Voiture(Gestionnaire)**".

Enfin, les objets sont associés entre eux en sauvant une référence vers l'un et
l'autre au sein de leurs instances comme dans "**self.pompe_freins = Pompe("Pompe freins", self.freins)**".

Pour parvenir à boucler les références circulaires, il faut enregistrer les
variables en deux temps, puisqu'en effet, il faut que l'objet soit créé avant
de pouvoir être référencé à son tour:
"**self.moteur.connecter_entree_carburant(self.pompe_carburant)**"

Une fois que tous les objets du modèle ont été créés, l'accès au modèle se fait
presque comme sur une véritable voiture. Désormais, pour accélérer, l'acteur
"Conducteur" interagit directement avec la pédale d'accélérateur de la voiture
en tant que l'instance "**pedale_accelerateur**" de l'objet "**Pedale**" faisant
partie de l'instance **ma_voiture**". En termes informatique, l'action d'accélérer s'écrira donc
"**ma_voiture.pedale_accelerateur.actualiser_course(0.15)**".

Nous voyons au travers de cette exemple simple que la programmation orientée
objet donne un caractère très vivant à l'informatique.

C'est un avantage, cependant, cela peut parfois être trompeur. En effet,
construire un modèle adéquat demande parfois du recul et une gymnastique
d'esprit particulière, qui n'est pas forcément ce qui nous saute aux yeux en
première analyse, même si la solution à l'arrivée se retrouve plus intuitive
d'usage.

C'est donc un savoir-faire à développer !

ALGORITHMES

D'après Wikipedia, un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème ou d'obtenir un résultat donné.

L'algorithme correspond donc à la modélisation, souvent mathématique, d'une tâche donnée. Il décrit une manière de répondre à la réalisation d'une spécificité du logiciel en établissant une recette pour obtenir le résultat souhaité.

La qualité d'un algorithme se mesure à sa fiabilité (sa capacité à obtenir le bon résultat), il est alors dit "correct" pour sa capacité à produire la bonne sortie. Elle se mesure aussi à son efficacité (durée de calcul, quantité de ressources utilisées, précision et comportement au facteur d'échelle).

Lors de l'implémentation d'une solution, il sera souhaitable de trouver l'algorithme le plus adapaté à la recherche de celle-ci. Bon nombre de situations courantes ont fait l'objet de recherches et des solutions algorithmiques peuvent être trouvées pour celles-ci. Il existe souvent même des librairies qui implémentent un ou plusieurs algorithmes pour des cas de figures usuels. La librairie offre alors un choix de solutions selon les conditions d'application qui pourraient privilégier l'un ou l'autre, ou le goût du programmeur.

L'algorithmique est à la fois une branche spécialisée et à la fois une compétence que le programmeur sollicite en permanence. En évoluant, le développeur intègre les recettes élémentaires qui façonnent sa démarche de programmation.

Au fur-et-à-mesure de sa progression, il apprendra de nouvelles recettes pour de nouvelles situations, il aiguisera son sens de l'algorithmique. S'il approdondit ce domaine, il pourra se spécialiser dans cette démarche. Dans tous les cas, c'est une matière où il est bon de s'appuyer sur les épaules de ceux qui se sont investis dans la résolution de problèmes spécifiques. C'est la communauté informatique qui évolue dans son ensemble grâce aux apports des uns et des autres.

LE CHOIX DU LANGAGE

La plupart des langages font partie de la programmation de type impérative (https://fr.wikipedia.org/wiki/Programmation_impérative): langage machine ou assembleur, Fortran, Algol, Cobol et Basic, Pascal, C. L'évolution vers l'orienté objet a engendré Ada, Smalltalk, C++. Et la famille des langages interprétés comprend notamment Perl, Tcl, Python, PHP et Java.

Tous ces langages ont la particularité d'être très proche des instructions du processeur et des opérations arithmétiques (qui correspondent d'ailleurs pour bonne part aux instructions du composant "UAL" du processeur, pour "Unité Arithmétique et Logique", s'écrit "ALU" en anglais).

L'apparition des listes et des fonctions sur celles-ci apportent une petite partie du concept de la programmation fonctionnelle.

En effet, dans la programmation fonctionnelle (https://fr.wikipedia.org/wiki/Programmation_fonctionnelle), il n'y a plus d'assignation de variables. Cette programmation se fait comme un appel en cascade de fonctions appelées par la précédente. Les langages faisant partie de cette famille comprennent Lisp et ses dérivés (Scheme, Common Lisp, Emacs Lisp), ML, Haskell, OCaml et Erlang pour les plus populaires.

Cette approche de programmation se traduit souvent par des listes extensibles imbriquées comportant des éléments ou d'autres listes. Les éléments d'une liste peuvent désigner une fonction en premier élément et ses paramètres en suivant. Chaque paramètre peut à son tour être une liste, dont certains implique à leur tour l'évaluation d'une fonction.

Prenons un exemple mathématique simple: "(+ 4 7 100 (- 10 2 3))".

Le premier élément de l'expression est la fonction "+", qui agit sur les éléments "4", "7", "100" et "(- 10 2 3)".

Ce dernier est lui-même une expression à part entière qui s'évalue en soustrayant (opérateur "-") les éléments "2" et "1" à l'élément "10". Sa valeurs est donc "10 - 2 - 3" = "5".

Le quatrième élément de la somme est alors "5", et notre somme est donc "4 + 7 + 100 + 5" = "116".

Ce que nous avons là est la modélisation d'une calculatrice en notation polonaise inverse ("NPI" ou "RPN" en anglais). Comme quoi, chaque langage est adapté à une modélisation particulière.

Bien entendu, n'importe quelle fonction peut trouver sa place en lieu et place d'un opérateur arithmétique. Par exemple, pour utiliser une fonction "sort" plutôt que l'opérateur "+", l'expression remaniée "(sort (list 4 7 100 (- 10 2 3)) '<)" donne comme résultat "(4 5 7 100)" (fonctionne en "Emacs Lisp").

Voilà pourquoi les listes permettent d'apporter un air de programmation fonctionnelle dans les langages impératifs. Il suffit de permettre que certains éléments de la liste soient une fonction. Bien entendu, ils restent des

langages impératifs puisqu'ils ne sont pas conçus dans la logique fonctionnelle.

A côté de ça, il reste la programmation logique (https://fr.wikipedia.org/wiki/Programmation_logique), qui est focalisée sur
les assertions logiques et les prédicats. C'est vraiment une approche particulière, une large part des enchaînements étant pris en charge par le
moteur. Ce type de programmation est très utilisé en intelligence artificielle. Le premier langage de programmation logique est Prolog.

Enfin, il y a des logiciels spécialisés dans certains domaines, comme R et Scilab pour les mathématiques. Ces outils disposent d'un langage de programmation spécifique à leur domaine d'applications. En l'occurrence, il s'agit de programmation impértative, avec une syntaxe
et des fonctions de travail orientée sur les mathématiques. Ces langages travaillent avec des objets mathématiques complexes. Ils manipulent, par exemple, des matrices et des fonctions comme éléments,
avec une aisance qui n'existent pas dans les autres langages.

En conclusion, chaque domaine d'application qui a fait l'objet d'un champ d'études, s'est vu développer un langage de programmation propre. Il convient donc, lors du développement, de porter son choix sur le langage le plus adapté, qu'il soit le plus spécifique pour le domaine d'application, car il répond vraiment au domaine cible, ou plus généraliste, parce qu'il apportera d'autres avangages, comme le fait d'être connu, ou de présenter une flexibiilté plus grande.

LA DOCUMENTATION

La documentation est souvent vue comme ennuyeuse pour une tranche des
développeurs. Pas tous. Certaines aiment à rendre compte d'un bel ensemble avec
leur logiciel.

D'un autre côté, d'autres contributeurs tout à fait bien placés pour ça, aiment
remplir ce rôle.

Et bon nombre d'utilisateurs et développeurs sont contents de pouvoir compter
sur une documentation bien faite à portée de main.

Qui plus est, la documentation peut se retrouver sous diverses formes: dans des
documents à part, en ligne sur le web (ou dans un intranet), en ligne sur le
système (pages **man** ou **info** sur les systèmes Unix), dans l'arborescence du
projet, et même imbriquée directement dans le code !

L'idéal est de satisfaire tout le monde.

Le développeur aimera avoir la documentation attachée à son projet, dans son
système et sur le web.

L'utilisateur final aimera la documentation dans un document qui lui est facile
d'accès, en ligne et dans le gestionnaire d'aide de son système.

Heureusement, il y a des parties qui se recoupent.

Pour arbitrer la question, c'est l'usage des documents qui va parler.
Mais aussi... Qui est susceptible d'en être rédacteur...

LE JUSTE NIVEAU DE DOCUMENTATION

La première question à se poser est peut-être celle-là: "Quel est le juste
niveau de documentation ?".

Cela revient à déterminer:
- Ce qu'il faut documenter
- Où il faut le documenter
- Jusqu'à quel niveau il faut le documenter

Là, il y a différentes pratiques, je vous invite à consulter quelques projets
open source connus, et à vous renseigner sur les usages de votre équipe de
travail, ces questions en tête.

RÉDIGER AU PLUS TÔT

Ensuite, la première chose à faire, c'est de commencer, de poser le contenu,
peu importe la forme. Non pas que la forme ne soit pas importante, qu'elle
n'ajoute pas de la valeur à l'ensemble, mais qu'elle peut être embellie après.

Et l'utilisateur sera content de trouver une information utile, plutôt qu'un
joli tape-l'oeil vide de sens.

Vos idées, une fois qu'elles ne sont plus fraîches, elles, sont plus difficiles
à raviver. Si vous écrivez un jeu, notez de suite les règles dans un manuel,
même de manière synthétique, ce sera toujours plus simple que de relire toutes
vos boucles et conditions dans le code pour déterminer quelle est l'enjeu de la
victoire !

Maintenant, si vous avez une idée de *design*, notez-le, ça fait partie des
choses à se remémorer aussi !

Pour le format, dans le doute, commencez par quelque chose de simple.
Un simple fichier texte peut convenir, il a l'avantage d'être lisible partout,
de pouvoir visualiser d'un coup d'oeil ce qui a changé d'une révision à l'autre
et de ne pas passer trop de temps à trouver le bon formatage.

Oui, bon, c'est un geek qui parle....
N'oubliez pas que c'est la première étape, celle de conserver l'information.

Le formatage, vous pourrez l'enrichir après ; si vous devez changer d'outils,
vous êtes susceptibles de revoir tout ce formatage. Et, rappelez-vous, le
développeur n'aime pas faire 2 fois la même chose...

D'autant qu'il y a des outils qui viennent à son secours.

L'OUTIL DE DOCUMENTATION QUI VOUS CONVIENT

Il y a une multitude d'outils pour parvenir à produire la documentation au
format qui vous plaise, à vous et votre équipe.

Cela va du simple fichier texte écrit avec un éditeur au CMS complet, qui
établit un point entre le code et la documentation. En passant par le
traitement de texte, la documentation rédigée en HTML, la documentation en
ligne dans un Wiki ou autre CMS, et la documentation générée dans différents
formats de publication à partir d'un format d'édition.

Personnellement, je suis à la fois développeur, geek et, récemment
graphiste.

J'attache donc de l'importance à ces critères pour ma documentation:
 - un format ouvert et accessible à tous, sans nécessiter un outillage
   spécialisé
 - une documentation proche du code, pour des raisons d'accès, de
disponibilité
   et de maintenance
 - un sens de l'esthétique
 - une facilité d'édition et de visualisation des changements
 - une diversité de formats de lecture
 - la mise à disposition à travers le réseau (web ou intranet), au moins
par le
   biais du protocole HTTP
 - la mise à disposition sur le système d'exploitation

Ce que j'ai trouvé pour y répondre, ce sont les outils de génération de
documentation.

Ils ont pour principe de partir d'un format source dans lequel la
documentation
est rédigée, pour produire la documentation dans une variété de
formats de
publication.

Voilà leurs caractéristiques:
 - rédaction dans un format source, texte ou LaTeX, ou DocBook:
   - format ouvert accessible à tous, à l'aide d'un simple éditeur de
texte, en
     particulier si la source est de format texte, y compris des formats
axés
     sur une mise-en-page (**reStructuredText**, **markdown**, **org-
mode**, ...)
 - visualisation des changements aisée en l'intégrant au suivi de
version
 - documentation proche du code: dans l'arborescence du projet, et
même jusque
   dans le code source du programme (combinaison des deux
possible)
 - export dans une variété formats
   - diversité de formats de lecture, ouverts et accessibles à tous selon
les
     formats produits (texte, HTML, ePub, DocBook, LaTeX, PDF, Open
Document
     Format, man pages, info pages, ...)
   - sens de l'esthétique, possibilité de rédiger des feuilles de styles
pour
     produire des documents orientés papier (LaTeX/DocBook), ou en
ligne
     (HTML/CSS)
   - accès en ligne via le web pour les pages générées en HTML
   - disponibilité sur le système d'exploitation (HTML, man, info)

Enfin, cet outillage rejoint de très près la philosophie Unix, "un outil
pour
faire chaque chose bien":
 - un compilateur pour produire une application à partir du code
source
 - un outil de génération pour produire un document prêt-à-lire à

partir d'une
  source d'informations pour la documentation
- un éditeur de texte pour rédiger du code et de la documentation,
  c'est-à-dire les entrées des deux outils cités

Parmi les outils qui ont retenu mon attention, il y a:
- Pour Python:
  - les *docstrings* (fonctionnalité de base): **pydoc**
(https://docs.python.org/2/library/pydoc.html)
  - **pdoc** (https://github.com/BurntSushi/pdoc) (amélioration pour les
    doctsrings)
  - **Sphinx** (http://sphinx-doc.org): production de documents
hiérarchisés et
    complets
  - **pydoctor** (https://launchpad.net/pydoctor), une alternative à
Sphinx
- Pour C/C++:
  - **Sphinx** : il est capable de fonctionner en C/C++ et d'autres
langages,
    même s'il n'est pas utilisé jusqu'au bout de ses capacités
  - **asciidoc** (http://asciidoc.org/ et http://asciidoctor.org/)
  - **doxygen** (http://www.doxygen.org/)
- Autres langages: de nombreux langages ont leurs systèmes de
génération de
  documentation, renseignez-vous sur le web sur les meilleurs
pratiques

Et n'oublions pas le fabuleux **pandoc** (http://pandoc.org/) qui se
charge de
réaliser des conversions entre divers formats de représentation de la
documentation.

Ce qu'il manque dans tout ça: la possibilité d'éditer la documentation
avec un
outillage usuel pour les contributeurs. Cependant, cela pourrait être
réfléchi,
proposer une interface WYSIWIG dans un navigateur qui aide à saisir
le contenu
de la documentation source.

LE DÉVELOPPEMENT

Le développement correspond à la réalisation concrète de votre projet.

C'est là que vous allez traduire en termes de programmation votre modèle.

C'est là que vous allez vous confronter à la réalité du code, de la machine et du système.

Pour partir serein, il est opportun de se mettre dans de bonnes conditions.

La marche-à-suivre proposée ici permet d'avancer suivant un parcours
balisé, je vous invite à la découvrir, thème par thème dans les sections qui suivent.

STRUCTURER SON PROJET

Il y a deux excellentes raisons de structurer son projet.

La première, c'est d'y voir clair.

La deuxième, c'est de mettre en place une organisation qui convienne aux
outils qui serviront à l'élaboration de votre logiciel.

Certains d'entre eux exigent une arborescence particulière, d'autres sont
flexibles, il y a pourtant des organisations plus propices à un fonctionnement
ou un autre.

Pour la première finalité, le choix complet vous appartient, c'est à vous de
déterminer l'organisation qui vous convient, en vous laissant guider par ce qui
vous plaît.

La seconde impose ou induit une manière d'organiser les fichiers et les
différentes composantes du logiciel.

A côté de ça, il y a d'autres critères qui interviennent, si je les rassemble
avec les précédents, cela nous donne les ingrédients pour la recette de "la
bonne organisation de votre projet":
- refléter l'architecture telle qu'élaborée pour la "Modélisation et
  l'architecture du projet"
- composer une arborescence adaptée à l'environnement et aux outils avec
  lesquels votre outil ou ses phases de production auront à interagir
- se mettre en phase avec les usages dans la sphère des développeurs
- faire appel à votre bon sens et votre intuition pour mettre en oeuvre une
  organisation qui vous plaise tout en étant compatible avec les critères
  ci-dessus

De l'architecture découle une organisation assez évidente: les briques
logicielles identifiées comme modules seront repérées dans l'arobrescence, soit
comme des fichiers, soit comme des répertoires, soit un mélange des deux. Agir
de la sorte permet d'avoir une consistance entre le modèle d'architecture et le
modèle de développement.

C'est un grand pilier de l'art de la programmation: la consistance ou cohérence.

Parmi les critères qui rentrent dans les bons usages des développeurs, il
s'agit de penser déjà en termes d'empaquetage et fourniture. Dans le monde du
logiciel libre, ce paquetage est composé du code source et de la recette
pour construire le logiciel. C'est la base, qui pourra être déclinée de
plusieurs façons, comme nous le verrons plus bas.

Vous glanerez les autres usages en creusant la documentation sur les outils de
développement, en lisant des articles sur les retours de programmeurs, en
interceptant une curiosité sur une IRC ou une mailing-list, en écoutant les
avis des autres, et bien sûr, juste autour de vous, en discutant avec vos
équipes de travail, où que vous soyez.

C'est là qu'il vous faudra faire le tri, entre ce qui marche bien ou pas, ce
qui est esthétique, ce qui est respectueux des standards et usages, sans s'y
enfermer pour autant, et construire progressivement ce qui correspondra à
votre approche. C'est un savant mélange, parvenir à cela, sera le fruit de
votre expérience, ce joyau inestimable. Ce joyau sera d'autant plus
resplendissant s'il s'imbrique avec harmonie dans son environnement, comme une
pierre sertie dans un bijou !

Un sacré challenge...

Pour commencer, voilà quelques pistes pour placer les divers éléments
constitutifs de votre projet:
- la documentation:
  - un fichier "**README**", "**README.txt**" ou "**Readme.txt**" à la racine du
    projet décrit l'essentiel du projet
  - un fichier "**AUTHORS**" reprend la liste des auteurs du projet
    (cela évite de les stipuler dans chaque fichier)
  - un fichier "**LICENSE**" reprend le texte de la licence du projet (doit
    accompagner le logiciel dans le cas d'un projet GNU)
  - un répertoire "**doc**" ou "**docs**" contiendra la documentation plus détaillée
- le code trouvera sa place dans un de ces répertoires:
  - "**src**" ou "**code**": pour tout "code source" constituant le projet, avec une
    sous-arborescence par modules
  - "**lib**": contiendra le code source des librairies destinées à ce projet ou d'autres projets
  - directement dans un répertoire comportant le nom du module
  - Note: Pour Python
- les tests:
  - un répertoire "**tests**" contiendra les tests accompagnant le logiciel
  - alternativement, les tests pourront se trouver sous chacun des modules du
    projet
- la production (*build*) et la création de paquetage (*packaging*):
  - Les outils "**Makefile**" et "**configure**" pour tout programme compilé
  - Un fichier "**setup.py**" pour une application Python

Les différents éléments seront approfondis dans la section sur "l'empaquetage
et la publication", en particulier les moyens de production et de création de
paquetage.

Pour ce qui est du choix des noms des fichiers, vous êtes invités à consulter
la rubrique "Astuces pour le nommage de fichiers" dans le guide sur les
systèmes Unix.

Concernant la documentation, il y a plusieurs approches.

Par contre, vous avez la liberté de choix: traitement de texte, documentation
en HTML ou pdf, rédigée directement ou générée à partir d'une documentaion en
mode texte. Comme décrit dans la section relative à ce sujet, vous choisissez
"l'outil de documentation qui vous convient".

SUIVI DE VERSION ET FORGES LOGICIELLES

Au cours du développement d'un logiciel, celui-ci est amené à évoluer. Et cela,
depuis sa conception, jusqu'à sa publication et sa maintenance.

Une fois publié, il est vital de pouvoir identifier avec exactitude chaque
version émise du logiciel. En effet, à partir de ce moment, le logiciel se
retrouve entre les mains des utilisateurs. S'ils ont des questions ou s'ils
tombent sur des erreurs de fonctionnement, il vous faut pouvoir savoir quelle
édition du logiciel est concernée.

C'est le rôle du logiciel de suivi de version. Il en existe plusieurs, un "guide
d'utilisation de git" sera bientôt disponible, qui décrit l'un d'entre eux.

Le suivi de version: le logiciel et la documentation

A côté de la version du logiciel proprement dit, c'est tout un ensemble qui est
sujet à cette question: toute la documentation est relative à une version
donnée de l'applicatif. Il est donc nécessaire d'identifier les évolutions dans
la documentation aussi.

Cela peut être fait par le même outil, dans un même dépôt ou non, ou même avec
des outils différents. Ce qui est important, c'est d'établir une correspondance
entre une version de la documentation et une version du logiciel. C'est là
qu'interviennent les étiquettes (*tags*).

Gérer la documentation au sein du projet permet de renforcer la cohésion entre
les deux. Par contre, il est fréquent que ces deux parties n'évoluent pas en
même temps. Dans ce cas, cela peut être troublant que deux parties qui ne sont
pas en phase se retrouvent publiées au même endroit dans des états différents. L'idéal est évidemment de faire l'effort de mise-à-jour avant la
publication d'une nouvelle version. Cela peut être facilité si la documentation
est associée au code (certains apprécient, d'autres pas), tel que présenté dans
la "section sur la documentation".

Les fiches de suivi (*tickets*)

Il y a un autre type d'information qu'il est pertinente associer aux versions
du logiciel: les fiches de suivi (*ticket*), qu'elles concernent des
dysfonctionnements identifiés (*bug report*: rapports de bogues), ou des
demandes d'amélioration (*feature request*). Ces fiches permettent de tracer
toute modification du logiciel, et de faire le lien entre le logiciel et la
documentation (la fiche identifie les documents à mettre-à-jour par rapport à
la modification).

Ces fiches peuvent être gérées dans un outil plus ou moins sophistiqué, qui
permet d'établir une correspondance entre les évolutions d'une modification et
l'état de suivi dans la fiche relative à celle-ci.

Les forges logicielles

Dans le monde du logiciel libre, les outils qui prennent en charge à la fois le
suivi de version, des fiches de modifications ainsi que d'autres aides à la
gestion de projet, portent le nom de "forge logicielle". Parmi celles-ci nous
pouvons citer les grandes forges en ligne "github", "gitorious", "bitbucket",
"Savannah", "launchpad", ou celle de "Google". Mais aussi, des applicatifs open
source prêt à installer sur une instance propre: "Savannah" en fait partie,
mais aussi "launchpad", "CodingTeam", "Redmine", ou "Trac".

Vous pouvez choisir d'en utiliser une en ligne, ou vous pouvez décider d'en
installer une par vous-mêmes. Même si vous utilisez la vôtre, il est opportun
d'apprendre à utiliser les forges les plus courantes, de nombreux logiciels
libres sont disponibles sur celles-ci.

Contribuer et interagir avec les projets est un bon moyen d'apprendre à se
servir des outils existants. Ce sera de toute manière un atout pour vous faire
une idée des caractéristiques et des usages, et fort à parier que votre projet
aura à s'intégrer avec d'autres briques répandues sur des moyens distincts !

Une forge logicielle "fait maison" à "moindre coûts"

S'il y a d'excellentes forges en ligne à disposition, et gratuites pour les
projets open source, vous pouvez en effet préférer disposer de votre propre
infrastructure.

L'installation d'une forge est tout de même une aventure d'une certaine ampleur
puisqu'il faut disposer d'un serveur, configuré convenablement.

Ci-après, vous trouverez une approche pour mettre en place à moindre frais quelques
éléments d'une forge. Puisque les produits envisagés sont open source, "à
moindre frais" est à comprendre dans le sens "à moindre effort".

L'idée est de pouvoir partir "tout de suite", tout en disposant déjà d'un filet
de sécurité et en commençant dès le départ son suivi. Sans être freiné par
l'ampleur d'activités préalables au démarrage.... Qui nous tenteraient de
négliger ce suivi.

Puisque dans les suggestions proposées, tout est en mode texte et versionné de
façon consistante, il sera toujours temps de scripter tout ça et d'importer les
informations dans une forge en temps voulu.

Des fiches de suivi "à la main"

Cette idée est inspirée d'un article sur le site web d'**ikiwiki**, un moteur de
Wiki: https://ikiwiki.info/tips/integrated_issue_tracking_with_ikiwiki/.

La proposition de cet article est d'utiliser **ikiwiki** pour le suivi des incidents (*issue*), en créant une page de Wiki à laquelle sont associés des
tags pour établir des liens entre les pages et créer des rapports en filtrant
sur ces étiquettes.

Eh bien, ce principe pourrait être repris directement dans le dépôt de suivi de
version du logiciel, c'est-à-dire, directement dans l'arborescence du projet.

Evidemment, cette idée a déjà été creusée, comme en témoigne cet excellent
récapitulatif:
http://stackoverflow.com/questions/2186628/textbased-issue-tracker-todo-list-for-git.

Pour notre cas d'étude "fait main", cela consisterait à prévoir un répertoire
"**tickets**" ou "**issues**" dans lequel seraient stockées toutes les fiches de
suivi.

La façon la plus simple est d'avoir un fichier texte structuré de façon à
mettre en évidence les champs clés de chaque fiche.

L'idée n'est pas neuve, puisque c'est ce que fait le projet "GNATS", dont la
version 3.103 date déjà de 1997 (aussi loin que remonte l'historique repris
dans le fichier "**NEWS.old**"): https://www.gnu.org/software/gnats/.

Voici quelques exemples de champs essentiels à prévoir dans une fiche:
 - Numéro: numéro de la fiche
 - Titre: intitulé de la fiche
 - Création: date de création de la fiche
 - Version: version vis-à-vis de laquelle la fiche a été levée
 - Milestone: version pour laquelle la fiche doit être prise en compte
 - Impact: estimation de l'impact sur le logiciel (exemple: anodin, faible,
   moyen, fort, critique)
 - Etat: où en est le suivi de la fiche (exemple: ouvert, analyse,
   réalisation, tests, documentation, suspendu, annulé, clos)
 - Description: Description complète de la thématique rapportée (amélioration
   ou détection d'erreur)
 - Journal: Une entrée datée pour chaque mise-à-jour de la fiche ; dans le cas
   d'un traitement par un outil, une partie de cette entrée est gérée
   automatiquement pour s'assurer qu'aucune modification ne passe inaperçue

Selon les besoins, vous aurez à enrichir ces champs, mais ils constituent déjà
une bonne base.

Pour l'encodage, vous êtes libre de créer votre format, cependant il peut être
judicieux d'utiliser un format déjà connu et existant comme **reStructuredText**
(http://docutils.sourceforge.net/rst.html), **markdown**
(https://daringfireball.net/projects/markdown/syntax) ou le format de la
base-de-données **recutils** (https://www.gnu.org/software/recutils/).

L'avantage d'utiliser un format connu est que vous vous appuyez sur

une
référence, d'autres sont susceptibles de le connaître aussi et vous bénéficiez
des outils qui l'accompagnent. Même si vous n'utilisez pas tout de suite
l'outillage, vous pouvez démarrer comme si c'était un simple fichier texte, et
tirer parti des possibilités offertes par la suite.

Je suggère un fichier par fiche plutôt qu'une seule base-de-données complète,
qui risque de devenir volumineuse et rendrait les recherches à l'aide d'un
éditeur de texte fastidieuses. Et, en se rappelant les "astuces pour le nommage
de fichiers" (voir guide Unix, en cours de rédaction), il sera bon de prévoir
suffisamment de chiffres pour faire face à toutes les fiches susceptibles
d'être émises dans la vie du projet. Avec 5 chiffres, il y a de quoi voir venir !

Prenons pour exemple, la fiche "**ticket_00174.txt**" au format **reStructuredText**, elle pourrait donner ceci:

```
Mettre au point un outil d'un installation
==========================================

  :Numéro: 174
  :Version: 0.4
  :Milestone: 1.0
  :Impact: fort
  :Etat: analyse


Description
-----------

Un outil d'installation serait d'une aide précieuse pour les
utilisateurs
comme pour les développeurs, il facilitera la maintenance et allégera
les
tests de déploiement.


Journal
-------

Réflexion en cours
++++++++++++++++++
  :Date: 2015-10-02

Etude de réalisation en cours, il serait bon qu'il soit prêt pour la
version
1.0. L'outil d'installation sera une aide au développement aussi car
il
allégera les tests de déploiement et facilitera la maintenance.

Champs modifiés:

- Milestone: 1.0
- Etat: analyse

Description à ce jour
~~~~~~~~~~~~~~~~~~~~~

Un outil d'installation serait d'une aide précieuse pour les
utilisateurs
comme pour les développeurs, il facilitera la maintenance et allégera
les
tests de déploiement.


Création de la fiche
++++++++++++++++++++
  :Date: 2015-09-15

Création de la fiche avec l'état suivant:

- Numéro: 174
- Version: 0.4
- Milestone:
- Impact: fort
- Etat: ouvert

Description à ce jour
```

```
~~~~~~~~~~~~~~~~~~~~
```

```
Un outil d'installation serait d'une aide précieuse.
```

Et au format ~recutils~, cela pourrait donner:

```
Titre: Mettre au point un outil d'un installation
Numéro: 174
Version: 0.4
Milestone: 1.0
Impact: fort
Etat: analyse
Description:
+ Un outil d'installation serait d'une aide précieuse pour les
utilisateurs
+ comme pour les développeurs, il facilitera la maintenance et
allégera les
+ tests de déploiement.
Journal:
+ [2015-10-02] Réflexion en cours
+ ------------------------------
+
+ Etude de réalisation en cours, il serait bon qu'il soit prêt pour la
version
+ 1.0. L'outil d'installation sera une aide au développement aussi car
il
+ allégera les tests de déploiement et facilitera la maintenance.
+
+ Champs modifiés:
+   Milestone: 1.0
+   Etat: analyse
+
+ Description à ce jour:
+ Un outil d'installation serait d'une aide précieuse pour les
utilisateurs
+ comme pour les développeurs, il facilitera la maintenance et
allégera les
+ tests de déploiement.
+
+ [2015-09-15] Création de la fiche
+ ---------------------------------
+
+ Création de la fiche avec l'état suivant:
+   Numéro: 174
+   Version: 0.4
+   Milestone:
+   Impact: fort
+   Etat: ouvert
+
+ Description à ce jour:
+ Un outil d'installation serait d'une aide précieuse.
```

Dans ces exemples, la traçabilité a été poussée un peu loin pour refléter ce
qui se rencontre fréquemment dans les forges logicielles bien faites.

Les dates de création et de dernière modification n'ont pas été insérées en
tant que champs puisqu'il s'agit respectivement de la dernière date et de la
première date du journal (classé par ordre chronologique inverse, les entrées
plus récentes d'abord).

Evidemment, ce sera vite fastidieux de maintenir à jour autant d'informations à
recopier à la main, ce qui deviendrait rapidement source d'erreur et de négligence.

Deux solutions s'offrent à nous:
 - créer quelques outils simples pour aider à la modification d'une fiche
 - s'appuyer sur le suivi de version

Pour la première, ce pourrait être une procédure toute simple: garder une copie
de la fiche actuelle, réaliser la mise-à-jour, y décrire la modification pour
le journal et faire passer la nouvelle fiche dans un script qui ajoute au
journal la différence entre l'ancienne et la nouvelle.

Pour la deuxième, c'est tout vu ! Il suffit d'enregistrer chaque changement
au travers du suivi de version. Seule les champs et la notice au journal sont
nécessaires, l'historique des éléments modifiés est directement rendu par
l'historique de version. La fiche s'en retrouve allégée pour une traçabilité
identique !

Au format **recutils**, la fiche allégée serait (à sa dernière version):

```
Titre: Mettre au point un outil d'un installation
Numéro: 174
Version: 0.4
Milestone: 1.0
Impact: fort
Etat: analyse
Description:
+ Un outil d'installation serait d'une aide précieuse pour les
utilisateurs
+ comme pour les développeurs, il facilitera la maintenance et
allégera les
+ tests de déploiement.
Journal:
+ [2015-10-02] Réflexion en cours
+
+ Etude de réalisation en cours, il serait bon qu'il soit prêt pour la
version
+ 1.0. L'outil d'installation sera une aide au développement aussi car
il
+ allégera les tests de déploiement et facilitera la maintenance.
+
+ [2015-09-15] Création de la fiche
```

Cela rend tout de suite beaucoup mieux (c'est moins chargé) et convient mieux
au format de **recutils** qui n'est pas très adéquat pour stocker de longs textes
comme nous l'avons fait précédemment dans le journal.

Pour visualiser les changements entre deux modifications de la fiche, nous
pouvons le faire dans le **shell** en utilisant l'outil **diff**. Pour cela,
supposons que vous en ayez gardé une trace sous les noms
"**ticket_00174.txt.v0**" et "**ticket_00174.txt.v1**":

```
$ diff -u1000 ticket_00174.txt.v0 ticket_00174.txt.v1
--- ticket_00174.txt.v0    2015-09-15 18:10:27.332095652 +0200
```

```
+++ ticket_00174.txt.v1    2015-10-02 16:58:49.180093194 +0200
@@ -1,10 +1,18 @@
 Titre: Mettre au point un outil d'un installation
 Numéro: 174
 Version: 0.4
-Milestone:
+Milestone: 1.0
 Impact: fort
-Etat: ouvert
+Etat: analyse
 Description:
-+ Un outil d'installation serait d'une aide précieuse.
++ Un outil d'installation serait d'une aide précieuse pour les
utilisateurs
++ comme pour les développeurs, il facilitera la maintenance et
allégera les
++ tests de déploiement.
 Journal:
++ [2015-10-02] Réflexion en cours
++
++ Etude de réalisation en cours, il serait bon qu'il soit prêt pour
la version
++ 1.0. L'outil d'installation sera une aide au développement aussi
car il
++ allégera les tests de déploiement et facilitera la maintenance.
++
 + [2015-09-15] Création de la fiche
```

En passant outre les quelques signes ésotériques (qui ont leur signification),
l'historique des changements se comprend tout seul. En effet, les différences
nous apparaissent tout de suite précédées d'un signe "-" pour ce qui a été
supprimé et d'un signe "+" pour ce qui a été ajouté (note: les signes doubles
"-+" et "++" sont dus à l'artifice de syntaxe utilisé pour décrire un champ
recutils sur plusieurs lignes).

Si vous préférez un affichage côte-à-côte, essayez ainsi:

```
r3vlibre@artlab:~/tmp/tickets$ diff -y -W 160 ticket_00174.txt.v0
ticket_00174
$ diff -y -W 160 ticket_00174.txt.v0 ticket_00174.txt.v1
Titre: Mettre au point un outil d'un
installation                              Titre: Mettre au point un
outil d'un installation
Numéro:
174
Numéro: 174
Version:
0.4
Version: 0.4
Milestone:
| Milestone: 1.0
Impact:
fort
Impact: fort
Etat:
ouvert
| Etat: analyse
Description:
Description:
+ Un outil d'installation serait d'une aide
précieuse.                        | + Un outil d'installation serait
d'une aide précieuse pour les utilisateurs

> + comme pour les développeurs, il facilitera la maintenance et
allégera les

> + tests de déploiement.
Journal:
Journal:

> + [2015-10-02] Réflexion en cours

> +

> + Etude de réalisation en cours, il serait bon qu'il soit prêt pour
la versio

> + 1.0. L'outil d'installation sera une aide au développement aussi
car il

> + allégera les tests de déploiement et facilitera la maintenance.

> +
+ [2015-09-15] Création de la
```

```
fiche                                          + [2015-09-15]
Création de la fiche
```

Cette fois, les changements sont repérés par le caractère "**|**" (appelé *pipe*)
et les ajouts par le caractère "**>**" (opérateur de comparaison "strictement
plus grand").

Un avantage du format **recutils** est qu'il est adapté pour faire des recherches
puisqu'il s'agit d'une base-de-données en mode texte.

Cependant, sans devoir l'installer et apprendre son fonctionnement, vous pouvez
déjà vous débrouiller avec "des outils de base Unix" comme **grep** et **awk**.

C'est un réel plus d'apprendre à les utiliser, ils se rencontrent souvent et
peuvent rendre bien des services !

Qui plus est, cette méthode marcherait aussi avec les tickets rédigés au format
reStructuredText au vu de la manière dont nous les avons agencés.

Voyons plutôt:

```
$ grep -e "Etat: ouvert" tickets/ticket_*.txt
tickets/ticket_00021.txt:Etat: ouvert
tickets/ticket_00044.txt:Etat: ouvert
tickets/ticket_00091.txt:Etat: ouvert
```

La commande suppose que vous êtes à la racine de votre projet et que vous y
avez stocké quelques fiches dans un répertoire "**tickets**", en les nommant toujours
bien selon notre exemple.

Pour agir sur plus de critères, il faut passer à un outil un peu plus puissant:

```
$ awk -v RS="" '/Impact: fort.*Etat: (ouvert|analyse)/ { print
FILENAME }' tickets/ticket_*.txt
tickets/ticket_00021.txt
tickets/ticket_00085.txt
tickets/ticket_00174.txt
```

Le résultat nous donne tous les tickets d'impact "fort", qui sont dans l'état
"ouvert" ou "analyse" (l'ordre des critères est important, il doit correspondre
à l'ordre dans lequel les champs apparaissent dans les fiches, la variable
"**RS**" assignée à la chaîne vide **""** fait en sorte que le fichier soit une seule entrée).

Pour un résultat plus poussé, il faut encore aller un peu plus loin:

```
Script shell "tickets.sh"
#! /bin/sh

awk 'BEGIN {
  RS="" ;
  criteres["Impact"] = "(fort|critique)" ;
  criteres["Etat"] =
"(ouvert|analyse|réalisation|tests|documentation|suspendu)" ;
}
{
  fichiers[FILENAME]++;
  for (crit in criteres) {
    crit_recherche = crit ": " criteres[crit] ;
    if (match($0, crit_recherche, result)) {
      criteres_ok[FILENAME,crit] = result[0] ;
    }
    else {
      nb_criteres_nok[FILENAME]++;
    }
  }
}
END {
```

```
  for (fic in fichiers) {
    if (! nb_criteres_nok[fic]) {
      printf "%s",fic "| ";
      for (crit in criteres) {
        printf "%-20s",criteres_ok[fic,crit];
      }
      print "";
    }
  }
}' $*
```

C'est du **awk**, un outil très puissant pour manipuler les fichiers contenant du
texte.

Ainsi, l'appel de ce script **shell** donnera:

```
$ sh tickets.sh tickets/tickets_*.txt
tickets/ticket_00091.txt| Impact: critique    Etat: ouvert
tickets/ticket_00021.txt| Impact: fort        Etat: ouvert
tickets/ticket_00174.txt| Impact: fort        Etat: analyse
tickets/ticket_00085.txt| Impact: fort        Etat: analyse
```

Voilà, le résultat obtenu est l'inventaire de toutes les fiches d'impact "fort"
ou "critique", qui ne sont pas à l'état "clos" ou "annulé" (tous les autres
états sont valides).

En procédant de la sorte, vous voyez que vous pouvez dès le départ démarrer
votre suivi avec un outillage minimal, sans avoir le besoin de mettre une
infrastructure lourde en place. Il suffit de créer un modèle vide du genre
"**ticket_nnnnn.template**" pour vous assurer que toutes les fiches auront le
format correct.

En prenant le soin d'associer leur mise-à-jour à chaque modification de code,
votre système est maintenu en cohérence, simplement en vous occupant du suivi
de version (vous pouvez prévoir une branche à part pour ne pas interférer avec
la branche de développement du projet).

A l'avenir, vous pourrez toujours migrer vers une forge logicielle plus complexe en y important vos données. Ce sera plus ou moins complexe selon les
cas, en tous cas, vous êtes sûr d'avoir toutes les informations depuis la
création du projet.

Un suivi de version "artisanal"

Tout comme pour les fiches de suivi, il est possible de mettre en place un
suivi de version rudimentaire à condition d'un peu de discipline.

Là, ce sera plus court, car le procédé est assez basique: il s'agit d'utiliser
l'outil **tar** et encore et toujours les "astuces pour le nommage de fichiers".

Voilà la procédure, il faut commencer par se positionner dans le répertoire du
projet, puis exécuter la commande suivante:

```
$ tar cfz ~/archives_projets/le-projet_2015-09-16_rev1_tag-
optionnel.tgz .
```

La clé est dans le nommage: les fichiers seront classés par ordre chronologique
grâce au format de la date, et le numéro de révision est donné à titre indicatif (pour visualiser facilement, à quel stade se trouve la révision).

Dans ce cas, il faut veiller à incrémenter chaque fois de "1" la révision.

Il est à noter que c'est un artifice qui n'est pas nécessaire ou qui peut être
inséré automatiquement.

Un mot clé supplémentaire permet d'attribuer un *tag* (étiquette) à la révision. Ce mécanisme est utilisé dans les outils de suivi de version pour
identifier une version particulière, par exemple une révision qui est publiée
officiellement, le numéro d'une fiche de modification qui a été réalisée.

Avec cette méthode, il devient délicat d'attribuer plus d'une étiquette à une
révision, mais c'est déjà pas mal.

Pour construire automatiquement le numéro de révision en considérant qu'une et
une seule sauvegarde a été réalisée pour chaque révision, envisager les
commandes suivantes (à mettre dans un script pour les pérériser):

```
$ rev=$(ls ~/archives_projets/le-projet_*.tgz|wc -l)
$ rev=$(expr $rev + 1)
$ tar cfz ~/archives_projets/le-projet_2015-09-16-rev${rev}.tgz .
```

Et pour s'en passer:

```
$ rev=0 ;
  for archive in $(ls ~/archives_projets/le-projet_*.tgz|sort);
  do
    rev=$(expr $rev + 1) ;
    echo "$archive    Rev $rev" ;
  done
/home/user/archives_projets/le-projet_2015-09-16.tgz   Rev 1
/home/user/archives_projets/le-projet_2015-09-17.tgz   Rev 2
/home/user/archives_projets/le-projet_2015-09-24.tgz   Rev 3
```

Voilà, c'est loin de fournir les fonctionnalités d'un outil de suivi de version, mais cette simple précaution vous permet de conserver un historique complet de votre projet, et de pouvoir revenir en arrière si jamais vous avez chamboulé trop de choses d'un coup, ou si vous faites une gaffe, ce qui arrivera sûrement un jour ou l'autre !

Pour aller un cran plus loin, la commande de sauvegarde pourrait être effectuée par un script qui prendrait un message en argument. Ce message serait stocké dans un fichier "**REVISION**" dont la première ligne serait le numéro de révision, les suivantes le message.

Ce message correspondrait au message de *commit* des outils de suivi de version, et vous voyez venir la suite: le jour où vous mettez en place le suivi de version, vous avez tout l'historique prêt à disposition.

Et quel est le coût ? A peine quelques commandes de **shell** !

Attention, cette méthode ne sera valable que pour un petit projet, bien sûr. Si vous êtes plusieurs à travailler, ou si vous travaillez sur un projet conséquent de plusieurs centaines de fichiers, il est impératif d'utiliser un outil adapté.

Mais pour démarrer, c'est une bonne manière de prendre le pli dès le début, de mettre en route une méthode de travail sûre, sans avoir à apprendre tout le fonctionnement d'un outil.

Et, même le jour où vous aurez un outil spécialisé, sachez que ça peut toujours être bon de faire une sauvegarde du dépôt, notamment au début, et aussi lors d'opérations un peu complexes. C'est toujours délicat de rétablir l'état d'un dépôt, puisque par conception, toutes les étapes sont archivées.

DE L'IMPORTANCE DE LA LISIBILITÉ

La lisiblité dont il est question ici est la lisibilité du code source d'un projet.

Pourquoi faire attention à la lisibilité du code source dans un projet ?

Eh bien, s'il a une certaine ampleur ou si, naturellement, il est partagé avec
d'autres, plusieurs personnes vont le lire, plusieurs vont l'écrire.

Un jour vous allez vouloir vous relire, ou lire ce que quelqu'un d'autre a
fait, ou ce quelqu'un d'autre voudra comprendre ce que vous avez voulu faire.

A ce moment, la lisibilité du code prend toute son importance, pour rentrer
dans la compréhension de ce qu'a fait l'autre.

Si du temps a passé après que vous ayez écrit quelque chose, parfois vous aurez
l'impression que vous lisez le code de quelqu'un d'autre.

Dans certains cas, vous aurez un peu de compassion eu égard votre style un peu
imparfait, vous sentirez alors que vous avez progressé. Dans d'autres circonstances, vous serez surpris: "C'est bien moi qui ai écrit ça ? Comment
j'ai fait ?". Vous étiez inspiré !

Toujours est-il que si ce code est lisible, votre impression en sera meilleure.

Et puis, lorsque vous aller réaliser ou contribuer à un projet, vous allez écrire du code, vous allez passer du temps dessus, beaucoup de temps.

Il est essentiel que ce temps soit agréable, qu'il vous plaise à être dans ce
code. L'esthétique a son importance. L'esthétique est un des ingrédients qui
fait de la programmation un art.

Enfin, sur un plan concret, si votre code est esthétique, il est plus lisible,
les erreurs sont plus faciles à détecter, les changements et les améliorations
sont plus facile à apporter.

Pour avoir une mesure, repérons les qualités d'un code lisible:
- Le résultat doit être agréable à voir au premier coup d'oeil
- La lecture doit être fluide
- La compréhension doit être facile:
  - Idéalement, le code est porteur de sens en lui-même
  - Des commentaires expliquent le rôle des parties complexes

Ce sont les critères qui donnent lieu à un code de qualité.

Comment honorer ces critères et obtenir une bonne lisibilité ?
- Réaliser une indentation et aération équilibrée du code
- Faire des choix pertinents dans les noms de variables/fonctions: le code
  doit se lire comme un texte
- Maintenir une cohérence dans l'apparence du code
  - Eléments syntaxiques: CamelCase, camelCase, under_score, ALL_CAPS
  - Une fonction est souvent associée à une action: elle est mise en valeur
    par l'usage d'un verbe pour la caractériser
  - Mise-en-page des commentaires, des fonctions, de leurs en-têtes
- Aboutir à une modularité réfléchie dont le découpage apporte de la clarté
- Placer des commentaires judicieux (une partie d'entre eux pourra être extraite par des

outils)
- En-tête qui décrit le but du module, du fichier
- En-tête des fonctions: rôle de la fonction et description des entrées/sorties
- Expliciter les parties de code complexes, de sorte que l'on comprenne
de quoi on part avant d'entrer dans cette portion de code et avec quoi
on ressort à la fin. Détailler des étapes intermédiaires si nécessaire.

Avec ça, vous avez de bons points de repères pour rédiger un code esthétique,
de qualité, aisé à parcourir, propice à la maintenance et accueillant pour les
contributeurs.

RÉALISATION ET LISIBILITÉ

Avec toute cette préparation, vous êtes fin prêts pour la réalisation !

Il y a une approche que je privilégie pour le développement (en fait, c'est quelque chose que j'ai appris dans la vie en général), c'est de commencer petit, à vrai dire "voir loin et commencer petit".

Le préalable, "voir loin", c'est ce que nous avons fait dans le travail de modélisation.

Appliquée à la phase développement, "commencer petit" pour évoluer
jusqu'au projet à maturité se traduit en quelques règles simples (ou lignes de conduite pour être plus juste):

- Dès le début, veiller à la lisibilité
- Réaliser à chaque étape une implémentation minimale qui marche
- Enrichir progressivement ce qui existe et a été validé, ajouter le traitement des cas d'erreur

Le secret réside dans "avoir toujours quelque chose qui marche" et qui
se lit intelligiblement ("veiller à la lisibilité").

"Commencer petit en ayant toujours quelque chose qui marche" revient à
implémenter petit-à-petit les éléments de la modélisation, en se focalisant sur quelques premières fonctionnalités, en faisant en sorte qu'elles soient utilisables, même si elles ne remplissent pas complètement leur rôle.

Pour avoir des fonctionnalités qui marchent sans remplir complètement
leur rôle, elles peuvent être écrites en offrant un comportant simplifié, en se limitant au cas nominal sans traiter les erreurs (avec pour condition de validation de ne pas susciter les cas d'erreur), ou même renvoyer une séquence déterministe, qui soit valide
dans son comportement, et notamment pour les autres fonctions.

Ainsi, son comportement, à ce stade d'implémentation sera valide vis-à-vis des éléments qui l'entourent et en termes de tests (voir dans la suite de ce guide). Cela n'implique pas de passer par un *stub* (fonctionnalité de remplacement dans les tests), c'est simplement son stade de maturation qui en est là, à ce moment.

Les tests permettent d'ancrer et de pérériser "ce quelque chose qui marche", c'est la trace qu'à un moment donné, dans son état, l'application a donné les résultats attendus, même à un stade de maturation précoce.

Le suivi de version appliqué aux tests avec leurs résultas de référence en même temps qu'au code source permet d'associer clairement
quel code a donné satisfaction à quels tests, ce qui est tracé par un jeu de résultats de référence.

En matière de lisibilité, il va de soi que cette précaution est à prendre également dans la réalisation des tests.

LES STANDARDS DE CODAGE

Le rôle des standards de codage est d'établir uen série de règles pour donner
une direction au style d'écriture pour le code.

Selon les cas, cela peut conduire à des choix heureux ou non.
Mais le principal objectif est de donner une cohérence au projet, donc il est
essentiel de veiller à s'harmoniser avec l'ensemble.

Voilà une execellente synthèse des règles de style pour le codage en Python:
http://docs.python-guide.org/en/latest/writing/style/.

C'est un exemple d'article qui peut vous aider à rehausser la qualité globale
de votre programmation.

Le fait de respecter un standard permet aux développeurs de s'attendre à
trouver une information donnée sous telle ou telle forme, ou à faciliter sa
localisation. C'est quelque chose de précieux.

Si un projet n'a aucune référence de ce genre, tentez de vous harmonisez avec
le code sous la main.

Certaines de ces références existent, tout en étant plus souples en matière de
règles de conduite. Elles vous permettent dès lors plus de liberté dans la
rédaction de votre code ; veillez toutefois toujours à ce que votre production
soit en phase esthétiquement et techniquement avec les choix faits par le
projet.

Cependant, si vous estimez qu'une règle nuit à la qualité ou à la robustesse
d'une impléemntation, vous pouvez toujours le faire valoir. Ce pourra être le
cas pour de jeunes projets. Pour les projets bien pensés et bien établis, ces
aspects-là ont souvent été traités en profondeur.

Bien entendu, rien n'est jamais figé, mais ce qui est sûr, c'est que c'est
votre implication dans un projet qui déterminera la part de poids que vous
pouvez avoir dans son évolution.

LES TESTS

L'objectif des tests est de s'assurer que le projet réponde au cahier des
charges et se comporte comme attendu.

Le premier test, c'est celui que nous faisons naturellement en écrivant un bout
de code, pour le lancer ensuite et vérifier s'il fait ce que nous souhaitons.

Ca a un côté grisant, car c'est interactif, le résultat est immédiat et la correction s'en suit de suite.

Cependant, dans la pratique, ce n'est pas gérable sur le long-terme:
 - oubli des situations intéressantes (il faut tester les cas aux limites !)
 - cumul des combinatoires au fur-et-à-mesure de l'enrichissement des
   fonctionnalités
 - une fonctionnalité qui a marché à un moment peut ne plus fonctionner par
   après
 - des portions de code peuvent se trouver difficilement atteintes une fois que
   le code s'est enrichi
 - un projet conséquent ne pourra jamais être conçu d'une pièce avant d'être
   exécuté !

La réponse est donc de mettre en place des tests automatisés, il existe des
études complètes sur ce thème et des outils spécialisés à ce sujet.

Ici, nous allons nous concentrer sur une approche en phase avec le développement par "étapes successives qui marchent".

En effet, dans ce modèle, le projet est construit par palliers.

Chaque pallier est conçu de sorte qu'il fonctionne, même si seulement un
très petit sous-ensemble des fonctionnalités prévues est couvert.

Pour s'assurer que "l'étape marche" comme stipulé dans le pré-requis, il va
falloir le vérifier.

La méthodologie de test dans notre cas, sera de construire un test qui valide
que l'étape marche.

Là, où ça se corse un peu, c'est qu'il va falloir péréniser ce test, c'est-à-dire que l'enjeu est qu'il puisse fonctionner plus tard, lorsque le
programme se sera enrichi des fonctionnalités futures.

C'est tout le travail de cette méthode.

La découpe en modules aide grandement à cela, il convient de structurer les
tests par modules et d'identifier comment solliciter chacun d'eux pour vérifier
leur comportement dans les cas de figure les plus essentiels, en particulier
les cas aux limites.

A côté de cela, au fur-et-à-mesure que l'application s'enrichit, une batterie
de tests permet de vérifier le fonctionnement de l'application dans son
ensemble.

Ainsi, les tests module par module, permette de valider les fonctionnalités des
modules individuels dans leur exhaustivité. Et les tests sur l'application construite progressivement permet de valider le fonctionnement des

modules
entre eux, dans les possibilités qui s'offrent à eux depuis les entrées système
et utilisateur. C'est-à-dire qu'un module fournira un éventail de sorties limité par les entrées qui le sollicitent (qu'elles proviennent de l'utilisateur ou du système), il ne sera donc pas forcément à même de stimuler
son voisin selon toutes ses entrées possibles.

La formalisation de ces tests dans un environnement de test donne une synthèse
de toutes les situations envisagées et des résultats obtenus.

Il permet leur exécution à guise et donc de vérifier à tout moment dans
l'évolution du projet que ce qui a marché à un moment donné marche
toujours. C'est ce qui s'appelle vérifier la non-régression.

Tout en conservant ce modèle, il sera bénéfique de s'appuyer sur des outils de
tests, qui nous facilitent la vie.

Je vous invite dès lors à lire la rubrique sur les moyens de tests pour Python
présentés dans cet article très riche:
https://www.jeffknupp.com/blog/2013/08/16/open-sourcing-a-python-project-the-right-way/

L'EMPAQUETAGE ET LA PUBLICATION

L'empaquetage et la publication interviennent à la phase finale d'une étape de développement. Ils consistent à produire un ensemble consistant pour le logiciel, les tests et la documentation, prêt à être publié à un instant donné.

D'autres termes répandus s'emploient pour désigner l'étape de "publication":
"*release*" en anglais, et "livraison" dans l'industrie.

La notion "d'empaquetage" se rapporte à la forme que prendra la publication.

En effet, un vrai programme constitue un tout, il s'intègre dans une environnement complet. Il faut penser à sa compilation, son installation et sa configuration.

Selon les cas, le logiciel sera fourni sous forme de code source ou de binaire, ainsi que quelques documents indispensembles, de fichiers de configuration par défaut et quelques exemples, et accompagné ou non de
la documentation complète, des tests et de leurs résultats.

Parfois, ils seront fournis dans une variété de déclinaisons, avec ou sans la documentation, ou les tests intégrés.

Une structure bien pensée (voir "Structurer son projet") facilite l'empaquetage et la publication.

Enrichie d'un outil d'installation et publication bien fait, cela permet de produire facilement le projet en différentes déclinaisons.

Pour les manières de fournir un logiciel "prêt-à-construire" ou "prêt-à-l'emploi", je vous invite à vous référer à la documentation propre à chacun des langages de programmation.

Sachant que des applications peuvent être constituées de langages multiples et portables sur plusieurs plateformes ("*cross-platforms*"). Pour ce dernier cas, il existe des outillages qui apportent des fondations solides pour une compilation multi-plateformes (par exemple **autoconf** et **automake**).

Pour avoir une guidance sur les manières de construire le contenu du projet à publier, vous pouvez vous référer à:
 - les standards de codage GNU (https://www.gnu.org/prep/standards/standards.html)
 - http://python-packaging.readthedocs.org/en/latest/minimal.html
 - http://stackoverflow.com/questions/193161/what-is-the-best-project-structure-for-a-python-application
 - https://www.jeffknupp.com/blog/2013/08/16/open-sourcing-a-python-project-the-right-way/

Enfin, au-delà d'un livrable pour un projet compatible multi-plateforme, chaque système d'exploitation dispose de ses habitudes et exigences en matière de fourniture d'un logiciel.

Dans les systèmes GNU/Linux, il sera question de "paquet" pour une distribution donnée.

La plupart du temps, ceux-ci sont distribués sous forme de paquetage source et de paquetage binaire. Outre, le programme du projet lui-même, le paquetage prévoit toutes les données de configuration, et
une installation conforme aux attentes du système (en matière de répartition des composantes du projet dans les répertoires cibles et de configuration par défaut).

Je vous renvoie aux différentes distributions pour connaître les modalités spécifiques à la création d'un paquetage.

CONSEILS POUR AFFINER VOTRE ART DE LA PROGRAMMATION

Notre style est vivant, il va évoluer, s'enrichir et mûrir au fil du temps.

Apprenez des "grands", récoltez les pratiques qui vous semblent inspirantes,
retenez les recettes toutes prêtes pour les situations courantes.

Ne soyez pas rigide, parce que vous avez décrété que vous suivriez toujours
telle ou telle convention. Faites-le parce qu'elle a du sens pour vous au
moment présent, tendez toujours vers ce qui va pour un mieux.

La plupart des projets ont une certaine ampleur, ils se font à plusieurs.
Le point de départ, c'est de s'intégrer dans l'existant, de faire oeuvre commune.

Avec l'expérience, votre style s'affirmera, et des bonnes pratiques vous
apparaîtront essentielles. Tout l'art consistera à les appliquer tout en
restant dans l'harmonie de l'ensemble.

Si des nouveaux choix doivent être faits, faites valoir votre avis, sans
l'imposer. Il y a des pratiques différentes qui se valent, il y a des pratiques
qui se font "parce qu'elles ont toujours été appliquées", d'autres juste
parce qu'un individu a envie que ça se fasse ainsi. Si votre avis est
différent, manifestez-le, non pas pour avoir raison, mais parce que vous
estimez que c'est ce qui est mieux pour l'évolution de l'art.

Une approche qui rencontre pas mal de succès, je trouve, est la démonstration
par l'exemple. L'idée est de faire quelque chose qui vous plaise, de le faire
vivre, et de montrer à l'oeuvre, en situation réelle.

Veillez cependant à ne pas forcer la main, faites connaître quelque chose qui
vous rend des services depuis que vous l'avez mis en place, présenter en quoi
vous trouvez cela pertinent, démo à l'appui.

S'il y a un réel apport, cela va faire son effet.

C'est comme ça que l'art se répand à travers le temps.