Références

- https://www.gnu.org/ (système d'exploitation et mouvement du logiciel libre)
- https://producingoss.com (livre sur le développement open source)
- https://openhatch.org (archive: organisation aidant à contribuer dans l'open source)
- http://teachingopensource.org/ (actualité / blog posts sur l'enseignement open source)
- http://open-advice.org/ (conseils de développeurs open source)
- http://oss-watch.ac.uk (conseil sur l'utilisation, le développement et les licences open source)

Ce chapitre va traiter de la façon dont un logiciel libre s'organise autours du packaging et de la publication du logiciel.

Dans ce contexte, qu'elle est la différence majeure avec le développement propriétaire ?

Ce chapitre va traiter de la façon dont un logiciel libre s'organise autours du packaging et de la publication du logiciel.

Dans ce contexte, qu'elle est la différence majeure avec le développement propriétaire ?

Le manque de contrôle centralisé sur l'équipe de développement, i.e. les groupes libre / open source ne sont pas aussi monolithiques !

Exemple frappant lors de la préparation d'une nouvelle version :

- **Entreprise** : contraindre une équipe à se concentrer sur une version à venir
  => Mise en attente des nouvelles fonctionnalités et correction non critiques
- **Libre / open source** : les contributeurs travaillent sur ce qu'ils veulent !
  => Ceux qui ne souhaitent pas aider pour une version veulent continuer de développer pendant le processus de stabilisation & publication de la version !

💭 Les développements ne s'arrêtant pas, les processus de publication d’une version ont tendance à être plus long dans le libre / open source
💭 Dans l'open source le processus de publication d’une nouvelle version est moins perturbateur !

[Illustration] : Comment fait-on pour réparer une autoroute ?

💭 Les développements ne s'arrêtant pas, les processus de publication d’une version ont tendance à être plus long dans le libre / open source
💭 Dans l'open source le processus de publication d’une nouvelle version est moins perturbateur !

[Illustration] : Comment fait-on pour réparer une autoroute ?

1. Fermer complètement la route pour qu'une équipe se consacre à pleine capacité sur sa réparation
   => Approche efficace pour l'équipe de réparation mais pas pour les conducteurs
2. Travailler sur quelques voies à la fois tout en laissant les autres accessibles
   => Approche plus contraignante pour les réparateurs MAIS l’autoroute est encore utilisable !

Les projets libre / open source fonctionnent selon la **seconde manière**

💭 Un projet mature a plusieurs release maintenues simultanément. Il est dans un état de *"réparation routière mineure"* permanent, i.e. il y a toujours :

- quelques voies qui sont fermées;
- des inconvénients tolérables pour l'ensemble des développeurs !

=> Les version sont effectuées selon un calendrier régulier

Le modèle qui rend cela possible consiste à paralléliser les tâches qui ne sont pas interdépendantes !

💭 Pas unique au développement libre / open source mais implanté de manière à ne peut pas trop contraindre ni l'*"équipe de réparation"*, ni *"la circulation quotidienne"*

Que se passerait-il si des activités en excluaient d’autres ?

Que se passerait-il si des activités en excluaient d'autres ?

Des contributeurs resteraient inactifs la plus part du temps et cela est un problème pour un projet libre / open source.

⚠️ Un contributeur ennuyé == un ex-contributeur en puissance !

Les méthodes que nous allons voir dans les prochains slides sont orientées sur la publication de nouvelles versions mais s'appliquent aussi à d'autres tâches parallélisables, e.g. la traduction, la modification importante d'une API (qu'il faut réaliser progressivement), ...

V – Packaging, version et développement quotidien

V.1 – Numérotation des versions

V.2 – Branche de version

V.3 – Stabiliser une version

V.4 - Packaging

V.5 – Tester et publier

V.6 – Gérer plusieurs versions

V.1 – Numérotation des versions

Que veut dire "version" pour un utilisateur ?

Que veut dire "version" pour un utilisateur ?

- D'anciens bugs ont été corrigés
  💭 Seule chose que les utilisateurs peuvent attendre à chaque release !
- De nouveaux bugs ont été ajoutés
  💭 Peut aussi être escompté à chaque release, sauf dans le cas de versions de sécurité / autres événements ponctuels !
- De nouvelles fonctionnalités peuvent avoir été ajoutées
- De nouvelles options de configuration peuvent être ajoutées
- La signification d'anciennes options peut avoir subtilement changé
- Les procédures d'installation peuvent avoir légèrement changées
- Des changements incompatibles peuvent avoir été introduits
  Exemple : formats de données des anciennes versions non utilisable sans une étape de conversion unidirectionnelle

Nommage des versions

Un utilisateur expérimenté aura donc toujours une appréhension lors d'un changement de version

- Surtout lorsque le logiciel faisait déjà ce que l'utilisateur souhaitait !
- L'arrivée de nouvelles fonctionnalité est une bénédiction mitigée
- Peut indiquer que le logiciel se comportera de manière inattendue !

=> Il est important que la numérotation d'une release soit compréhensible

💭 **[Petit aparté]** : cette appréhension est aussi présente lors de l'ajout de nouvelles dépendances à un produit !

Nommage des versions

Quelles informations doit transmettre une numérotation ?

Nommage des versions

Quelles informations doit transmettre une numérotation ?

Objectifs de la numérotation :

- Communiquer sans ambiguïté l'ordre des versions
- Indiquer de manière compacte le degré et la nature des changement d'une version !

Connaissez-vous une stratégie de numérotation ?

Nommage des versions

Quelles informations doit transmettre une numérotation ?

Objectifs de la numérotation :

- Communiquer sans ambiguïté l'ordre des versions
- Indiquer de manière compacte le degré et la nature des changement d'une version !

Connaissez-vous une stratégie de numérotation ?

Plusieurs stratégies existent mais un principe universel est accepté et requis : **être cohérent** !
=> Choisissez une numérotation, documentez-la et respectez-la !

Les composants du numéro de version

Les numéros de versions sont des groupes de chiffres séparées par des points.

Exemple sur ma machine (logiciel --version):

- Git version 2.25.1
- Cmake version 3.24.3
- Docker version 24.0.5, build ced0996
- pdfTeX version 3.14159265-2.6-1.40.18

💭 Convention / Norme : les points sont des séparateurs et pas des décimales ! Mais cela n'a pas toujours été le cas, e.g. le noyau Linux et sa séquence "0.95", "0.96" ... "0.99" menant à Linux 1.0

Les composants du numéro de version

Il n'y a pas de limite au nombre de composants !
💭 La plupart des projets de vont pas au-delà de trois ou quatre. Cette norme permet de comprendre l'ordre entre deux releases, e.g. 0.1.2 est suivi de 0.1.3

En plus des composantes numériques, les projets peuvent utiliser une étiquette descriptive. Connaissez-vous de telles étiquettes ?

Les composants du numéro de version

Il n'y a pas de limite au nombre de composants !
💭 La plupart des projets de vont pas au-delà de trois ou quatre. Cette norme permet de comprendre l'ordre entre deux releases, e.g. 0.1.2 est suivi de 0.1.3

En plus des composantes numériques, les projets peuvent utiliser une étiquette descriptive. Connaissez-vous de telles étiquettes ?

Exemples : MyProject 2.3.0 (Alpha), AnotherProject 5.11.4 (Beta), ...
Les qualificatif *"Alpha"* ou *"Beta"* induisent que cette version précède une future version qui aura le même numéro sans l'étiquette: *2.3.0 (Alpha)* mènera à *2.3.0*

Les composants du numéro de version

💭 Ces versions candidates peuvent induire plusieurs versions candidates à la suite, les étiquettes auront alors des méta-qualificatifs.

Exemple de séquence de versions candidates:
MyProject 2.3.0 (Alpha 1) → MyProject 2.3.0 (Alpha 2) → MyProject 2.3.0 (Beta 1) → MyProject 2.3.0 (Beta 2) → MyProject 2.3.0 (Beta 3) → MyProject 2.3.0

💭 Une version "2.3.0" est écrit "2.3" par soucis de concision sauf lorsqu'il y a des étiquettes en plus

Les composants du numéro de version

Il existe d'autres étiquettes : Stable, Unstable, Development, RC, ...

Les plus utilisées sont : *"Alpha"*, *"Beta"* et *"RC"*
💭 RC inclut toujours un méta-qualificatif numérique, e.g. on ne publie pas MyProject 2.3.0 (RC) mais MyProject 2.3.0.RC1, puis RC2, ...

💭 **[Conseil]** Utilisez ces trois étiquettes et évitez les autres (même si ce sont des mots normaux) ! Ils sont largement connus et familiers des personnes qui installent des logiciels ! Il n'y a pas de raisons de faire gratuitement les choses différemment de tout le monde

Les composants du numéro de version

Connaissez-vous les noms des composants typiques d’une version ?

Les composants du numéro de version

Connaissez-vous les noms des composants typiques d'une version ?

Majeur - Mineur - Micro / Patch

Par exemple : 2.3.12 est la treizième micro-version de la quatrième ligne de version mineure de la deuxième série de version majeur !

💭 *"Série"* est une notion informelle désignant un ensemble partageant un même numéro de majeur / mineur, par exemple :

- 2.3.0 n'est pas dans la même série mineure que 3.3.1
- 2.3.0 est dans la même série mineure que 2.3.10

Les composants du numéro de version

Quelle signification donner à ces nombres ?

Les composants du numéro de version

Quelle signification donner à ces nombres ?

- Majeur : des changements majeurs se sont produits
- Mineur : des changements mineurs se sont produits
- Micro : des changements vraiment insignifiants se sont produits

💭 Certains projets utilisent un composant supplémentaire pour affiner le contrôle des versions. Celui-ci peut être utilisé, par exemple, comme un numéro de build et est modifié à chaque build (sans représenter d'autre changement). Cette approche est utile lorsque le projet est distribué en binaire.

Il existe de nombreuses conventions différentes de numérotation et nous allons voir les plus largement utilisées

Gestion de sémantique de version

Connaissez-vous des règles associées au changement d'un numéro de version ?
Sont-elles les mêmes pour le nombre micro / mineur / majeur ?!

Calendar Versioning (CalVer)

Convention de version basée sur le calendrier de publication de votre projet au lieu de nombres arbitraires.

Examples:

- **Ubuntu**: YY.0M.MICRO, e.g 4.10.LTS (Octobre 2004 Long Time Support)
- **certify**: YYYY.MM.DD, e.g. 2024.8.30
- **????**: 0Y.WW.MICRO, e.g. 09.52.1 (Derniere semaine de 2009)

Semantic Versioning (SemVer)

Convention de version basée sur les notions de compatibilité descendante (rétrocompatibilité) et ascendante.

Politique largement utilisée par la communauté et formalisée !

Lien : https://semver.org/spec/v2.0.0.html

Semantic Versioning (SemVer)

Quels sens donnez-vous à ces mots (compatibilité descendante et ascendante) ?

Semantic Versioning (SemVer)

Quels sens donnez-vous à ces mots (compatibilité descendante et ascendante) ?

[Exemple] Projet qui est une application client / serveur :

- Être **rétrocompatible** (compatibilité descendante) signifie que mettre à jour le serveur en version 2.3.0 ne devrait pas poser de problème aux clients encore en version 2.2.4 :
  - pas de perte de fonctionnalité;
  - pas de comportement qui diffère d'avant (sauf pour les bogues corrigés).

Semantic Versioning (SemVer)

Quels sens donnez-vous à ces mots (compatibilité descendante et ascendante) ?

[Exemple] Projet qui est une application client / serveur :

- Supposons maintenant que le client soit mis à jour comme le serveur (2.3.0). Il pourrait alors avoir accès à de nouvelles fonctionnalités :
  - celles-ci peuvent ne pas être accessible aux clients encore en version 2.2.4

Dans ce cas, la mise à jour **n'assure pas une compatibilité ascendante**. On ne peut pas repasser le client en version 2.2 4 et garder les fonctionnalités de la version 2.3.0 !

Semantic Versioning (SemVer)

Modifications apportées au numéro micro : doivent être rétrocompatible et assurer la compatibilité ascendante.

Connaissez-vous des exemples de telles modifications ?

Semantic Versioning (SemVer)

Modifications apportées au numéro micro : doivent être rétrocompatible et assurer la compatibilité ascendante.

Connaissez-vous des exemples de telles modifications ?

- correction de bogue;
- modifiations sans nouvelles fonctionnalités;

Semantic Versioning (SemVer)

Modification apportée au numéro mineur : doit être rétrocompatible mais pas nécessairement vérifier la compatibilité ascendante.

Connaissez-vous des exemples de telles modification ?

Semantic Versioning (SemVer)

Modification apportée au numéro mineur : doit être rétrocompatible mais pas nécessairement vérifier la compatibilité ascendante.

Connaissez-vous des exemples de telles modification ?

- nouvelles fonctionnalités dans l'API publiques;
- nouvelles fonctionnalités ou amélioration dans le code privé

⚠️ La modification du mineur réinitialise le micro !

Semantic Versioning (SemVer)

Modification apportée au numéro majeur : marque la modification des repères de compatibilité et peut ne pas vérifier la compatibilité ascendante et descendante.

Semantic Versioning (SemVer)

Modification apportée au numéro majeur : marque la modification des repères de compatibilité et peut ne pas vérifier la compatibilité ascendante et descendante.

Connaissez-vous des exemples de telles modification ?

Semantic Versioning (SemVer)

Modification apportée au numéro majeur : marque la modification des repères de compatibilité et peut ne pas vérifier la compatibilité ascendante et descendante.

Connaissez-vous des exemples de telles modification ?

- changements non rétrocompatibles dans l'API publique;
- peut inclure de nouvelles fonctionnalités voir des nouveaux ensembles de fonctionnalités.

Semantic Versioning (SemVer)

[Autre exemple illustratif] Projet qui est une application gérant un format de données écrit sur disque (ou autre sockage permanent) : les formats que lit / écrit le logiciel doivent suivre les directives de compatibilité !

La version 2.6.0 doit pouvoir lire les fichiers écrits par la 2.5.4
💭 Peut améliorer en interne le format en quelque chose que 2.5.4 ne peut lire

⚠️ Si votre projet distribue des bibliothèques appelable depuis d'autres code, l'API est également un domaine de compatibilité :

- Les règles de compatibilité source et binaire sont énoncées pour que vos utilisateurs ne se demandent jamais si c'est *"sûr"* de mettre à jour.
- Regarder les chiffres permet de le savoir instantanément.

Semantic Versioning (SemVer)

Cette approche de gestion des versions ne vient pas sans inconvénient. D'après vous, quel est-il ?

Semantic Versioning (SemVer)

Cette approche de gestion des versions ne vient pas sans inconvénient. D'après vous, quel est-il ?

Inconvénient : pas de nouveau départ possible sans incrémentation du majeur !

Gênant lorsque vous souhaité faire des modifications qui ne peuvent être réalisées avec maintient de la compatibilité, e.g. :

- des nouvelles fonctionnalités que vous souhaitez ajouter;
- des protocoles que vous souhaité reconcevoir.

💭 Pas de solution magique si ce n'est de concevoir votre projet de la manière la plus extensible possible ...

Odd-Even Versioning (OddEven Ver)

Connaissez vous une numérotation qui attache de la sémantique aux numéros pairs et impairs ?

Odd-Even Versioning (OddEven Ver)

Connaissez vous une numérotation qui attache de la sémantique aux numéros pairs et impairs ?

Utiliser la parité du nombre mineur pour indiquer la stabilité du logiciel, i.e. numéro pair == stable, numéro impair == instable.

💭 La sémantique associée aux autres nombres reste inchangée

💭 Convention anciennement utilisée par le noyau Linux

Avantage : permet de fournir une version avec une fonctionnalité à tester:

- Evite d'impacter la production avec du code potentiellement instable
- Tout le monde comprend que :
  - "1.2.2" est "sûre" à installer sur un serveur web en ligne;
  - "1.3.1" devrait rester confinée à des stations de travail expérimentales.

Odd-Even Versioning (OddEven Ver)

Règles autours de cette notation :

- Les développeurs traitent les rapports de bogues des versions impaires
- Quand cela se tasse après plusieurs version micro :
  - incrémentation du nombre mineur;
  - réinitialisation du nombre micro
  - distribution *"présumée"* stable

Cette convention ne rentre pas en conflit avec les lignes directrices de la précédente:

- Surcharge simplement le nombre mineur et tend à doubler son nombre d'incrémentation;
- Convention adaptée aux projets ayant un cycle de release long et qui ont des utilisateurs conservateurs (stabilité >> nouvelles fonctionnalité).

Intended Effort Versioning (EffVer)

Convention de version basée sur la quantité d'effort qu'un utilisateur doit mettre pour adopter la nouvelle version.

Macro.Meso.Micro : Gros efforts . Quelques effort . Pas d'effort

Examples de changements:

- micro : correctif de petit bogue / nouvelle fonctionnalité;
- meso : gros bogue accoutumié des utilisateurs / changement cassant dans une fonctionnalité
- macro : gros changement cassant / gros aspects du projet

Liens: https://news.ycombinator.com/item?id=39400576 et https://jacobtomlinson.dev/effver/

Gestion de sémantique de version

En résumé, **publier une politique de compatibilité des versions et y adhérer est un point incontournable de la distribution de logiciels**

⚠️ Une mauvaise surprise peut vous aliéner de nombreux utilisateurs

[Avis personnel] Utilisez le SemVer car cette sémantique est déjà très répandue tout en étant facile à expliquer et à retenir !

💭 Ces règles ne s'applique généralement pas aux versions antérieures à la 1.0. Il est *"accepté"* de pouvoir passer de 0.1 → 0.2 → 0.3 avec des différences arbitrairement importantes.

V.2 – Branche de version

Comment faire une version dans le libre ?

Comment faire une version dans le libre ?

Du point de vue d'un développeur, un projet est en état de release continue

- Tendance à mettre à jour sa copie du code tous les jours (voir plusieurs fois par jour)
- Besoin de mise à jour pour repérer les bogues
- Connaît suffisamment le projet pour éviter les zones instables

A votre avis, comment le projet doit-il faire une version formelle ? Devrait-il prendre un instantanné, le packager et le distribuer au monde ?

Comment faire une version dans le libre ?

Du point de vue d'un développeur, un projet est en état de release continue

- Tendance à mettre à jour sa copie du code tous les jours (voir plusieurs fois par jour)
- Besoin de mise à jour pour repérer les bogues
- Connaît suffisamment le projet pour éviter les zones instables

A votre avis, comment le projet doit-il faire une version formelle ? Devrait-il prendre un instantanné, le packager et le distribuer au monde ?

Non et pour plusieurs raisons :

- l'arborescence de développement peut ne jamais être propre et prête à être publiée;
- des nouvelles fonctionnalités sont en WIP (divers états d'achèvements);
- les débat autours d'un changement majeur enregistré et controversé.
  💭 Retarder le jusqu'à la fin du débat ne fonctionnerait pas car un autre débat pourrait démarrer en parallèle, puis un autre, ...

Comment faire une version dans le libre ?

Illustration de la réponse précédente :

Context : notons "2.3.0" la version associée au snapshot. Le prochain snapshot devrait être la version "2.3.1" et donc contenir principalement des correctifs de bogues trouvés dans la version "2.3.0"

Que sont censés faire les développeurs entre ces deux versions ?!

💭 Si la sémantique SemVer est utilisée : pas de nouvelles fonctionnalités !
=> Un développeur doit choisir entre rester inactif et travailler sur quelque chose qui ne l'intéresse pas forcément.

⚠️ Il est important de ne pas frustrer les contributeurs (outre mesure) surtout si ce n'est que pour *tranquilliser* l'arborescence de développement.

Utilisation de branche de version

Idée associée : Branche quelconque où le code destiné à cette version peut être isolé du développement principal.

💭 Approche parfois utilisée dans les logiciels propriétaires mais plus rarement car c'est vu comme une sorte de bonne pratique théorique et c'est souvent mis de côté dans le feu de l'action associé à une échéance majeure.

⚠️ Faire sans branche de version converge souvent vers **des développeurs qui deviennent inactifs** et d'autres (une minorité) qui s'efforcent de sortir la version. Alors :

- la **dynamique globale** de développement est **ralentie**
- la version est de **moins bonne qualité** (peu de personne dessus)
- **divise psychologiquement** en créant une situation où des types de travaux différent interfèrent inutilement les uns aux autres

Création d'une branche de version

A votre avis, comment est gérée la création de branche de version ?

Création d'une branche de version

A votre avis, comment est gérée la création de branche de version ?

💭 Les mécanismes de création de branche de version dépendent de l'outil de gestion de versions mais suivent généralement les mêmes concepts : création d'une branche à partir d'une autre branche (tronc principal ou non)

Illustration : une branche jaillissant du tronc et associé à la version 1.0
💭 Nommez la brance "1.0.X" au lieu de 1.0.0 afin de pouvoir utiliser la même branche pour toutes les micro-versions associées à une version mineur!

- Quand la branche est stabilisée et prête : instantanné + label "1.0.0"
- La prochaine micro-version démarre de "1.0.X" et suit le même traitement: stabilisation → préparation → instantanné + label
- La prochaine version mineure ("1.1.x") sera créée à partir du tronc

💭La maintenance des versions 1.0.x et 1.1.x peut être réalisée en parallèle

💭 Les autre développements sont réalisés en parallèle dans le tronc principal ou des branches éphémères qui seront intégrées dans le tronc principal

💭 Il n'est pas rare de publier des versions sur deux lignes de versions différentes en parallèle, e.g. Python 3.3.7 est sortie en Septembre 2017 et Python 2.7.15 est sortie en Mai 2018. Cela permet :

- aux utilisateurs / administrateurs conservateurs de ne pas faire de grand saut sans préparation minutieuse;
- aux aventuriers d'obtenir les dernières fonctionnalités au prix d'une possible instabilité.

Démonstration en direct

Points à aborder :

- Liens entre *main* et *release/2.3*
- Liens entre *tag* et *CICD*
- Rêgles associées aux branches et tags
- Secrets

⚠️ La stratégie présentée **n'est pas l'unique manière de faire** ! Mais s'en est une qui fonctionne bien en pratique.

Toute autre stratégie est valable tant que la branche de version assure :

- l'isolation du travail sur la version sur les développements quotidiens;
- une entité physique autour de laquelle le projet organise son processus de publication.

V.3 – Stabiliser une version

Stabilisation d'une version

Qu’est ce que signifie stabiliser une version ?

Stabilisation d'une version

Qu’est ce que signifie stabiliser une version ?

- choix des changements inclus (ou pas) dans la version;
- façonner le contenu de la branche en conséquence.

💭 A l’approche d’une nouvelle version, il est fréquent que les développeurs se démènent pour terminer leurs modifications en cours.

⚠️ Il vaut mieux éviter d’intégrer du code terminé à la hâte car ils sont une bonne source de bogues

Stabilisation d'une version

Qu’est ce qui peut rythmer la parution de version ?

Stabilisation d'une version

Qu’est ce qui peut rythmer la parution de version ?

- le temps : parution à un rythme régulier et la version sort peu importe les fonctionnalités / corrections de bogue prêtes ou pas
- les fonctionnalités : parution à un rythme irrégulier où les fonctionnalités et le périmètre ont la priorité sur la date

Dans la suite des slides, nous allons voir des conseils qui s’appliquent aux deux approches.

Stabilisation d’une version

Quels critères utiliser pour autoriser ou non des changements dans une version lors de sa stabilisation ?

Stabilisation d’une version

Quels critères utiliser pour autoriser ou non des changements dans une version lors de sa stabilisation ?

Certaines directives formelle peuvent exister, par exemple :

- correctif de bogue grave ou de bogue sans contournement connu;
- mise à jour de la documentation;
- pour certains autres changement à faible risque; …

⚠️ Le besoin d’un jugement humain est inévitable dans certains cas !

Stabilisation d’une version

Emergence d'un **problème** : beaucoup de personnes seront motivées pour ajouter du changement (surtout si ça les concerne / intéresse) et peu seront motivés pour l’interdire !

=> La stabilisation d’une version consiste principalement à créer des mécanismes pour dire *"non"*.

Objectif : trouver un moyen de dire *"non"* tel que :

- les développeurs ne soient pas blessés / déçus;
- les changement méritant soient incorporés dans la version !

Il existe plusieurs méthodes pour cela. Nous allons en voir deux.

1) Dictature par le propriétaire de la version

Le groupe nomme une personne **propriétaire de la version**.

Fonctionnement : celle-ci **a le dernier mot sur les modifications apportées à la version** et **n'empêche pas les discussions et arguments**.

Profil associé :

- aptitudes techniques : comprendre tous les changements et pouvoir les évaluer objectivement;
- aptitudes relationnelles : limiter le fait de *blesser les sentiments*

1) Dictature par le propriétaire de la version

Exemple de bon commentaire :

> “ Je ne pense pas qu'il y ait quoi que ce soit de mal avec ce changement, mais nous n'avons pas encore eu assez de temps pour le tester. Pour cette raison, il ne devrait pas entrer dans cette version. ”

⚠️ Ce commentaire peut mener à des demande de justification sur la décision et/ou sur des remarques sur le risque associé à un changement

💭 Cela aide beaucoup que le propriétaire ait des connaissances techniques !

2) Vote pour les changements

Fonctionnement : le groupe vote pour les modifications à inclure dans la prochaine version

Est ce que l'inclusion d'un changement par obtention de la majorité des votes est suffisant ?

2) Vote pour les changements

Fonctionnement : le groupe vote pour les modifications à inclure dans la prochaine version

Est ce que l'inclusion d'un changement par obtention de la majorité des votes est suffisant ?

Non car :

- Dans ce cas, un vote pour et un autre contre induirait l'inclusion.
- Cela pourrait mener à une mauvaise dynamique !
  💭 Un contributeur aurait tendance à voter pour les contributions des autres par peur de représailles !

2) Vote pour les changements

Solution possible : utilisation de sous-groupes de développeurs pour évaluer si le changement doit être ajouté à la version.

Intérêts :

- Cela rend un développeur moins hésitant à voter contre un changement
  💭 Parmi les votants personne ne prendra son refus comme une injure.
- Plus de personnes examinent les changements.
  💭 Plus le nombre de personnes impliquées est important, plus la discussion porte sur le changement et moins sur les individus !

2) Vote pour les changements

Exemple de système basé sur le vote (dans SVN) : pour qu'un changement soit appliqué, au moins trois développeurs doivent voter en sa faveur, et aucun contre. 💭 le *"Non"* équivaut à un veto.

Remarque: tout vote *"non"* doit être accompagné d'une justification. Le veto peut être annulé si suffisamment de personnes estiment qu'il est déraisonnable. En pratique, l'annulation est rare car les gens ne s'oppose que lorsqu'ils ont une bonne raison

💭 Cette procédure est biaisée vers le *conservatisme* et les vetos peuvent être plus procédurales que techniques. Par exemple : quelqu'un veut poser un veto car le changement est trop gros pour une version micro, ...

2) Vote pour les changements

Comment faire pour mettre en place le système de vote ?

2) Vote pour les changements

Comment faire pour mettre en place le système de vote ?

Exemple de solution simple et pratique : utilisation d'un fichier texte *STATUS* ou *VOTES* dans la branche de la version :

- répertorie tous les changements proposés;
- répertorie tous les votes pour ou contre chaque changement

Illustration slide suivant

2) Vote pour les changements

“ • commit b31910a7180fc (issue #49)

Empêcher la négociation client/serveur de se produire deux fois
Justification: évite les rotations supplémentaires du réseau ; petit changement et facile à revoir.

Remarques: discuté dans https://.../msg-7777.html

Votes :
+1 : jsmith, kimf
-1 : tmartin (casse la compatibilité avec certains serveurs antérieurs à la version 1.0 ; ces serveurs sont bogués, mais pourquoi être incompatible si ce n'est pas nécessaire ?) ”

2) Vote pour les changements

La personne qui propose un changement est responsable de s'assurer qu'il s'applique sans conflits ! S'il y a des conflits alors le texte associé au changement doit pointer vers un correctif ajusté OU une branche temporaire

“

- r13222, r13223, r13232

  Réécriture de l'algorithme de fusion automatique de libsvn_fs_fs
  Justification: performances inacceptables (>50 minutes pour un petit commit) dans un référentiel avec 300 000 révisions

  Branche: 1.1.x-r13222@13517
  Votes :
  +1 : epg, ghudson

”

💭 Ici, utilisation d'une **branche avec version ajustée** combinant trois révision en une seule. Message de log associé *"Ajustement de r13222, r13223 et r13232 pour rétroportage vers la branche 1.1.x"*.

Supposons que des changements ont été approuvés. Qui doit se charger de la fusion associée ?

Responsable de version

En théorie : la fusion peut être réalisée par n'importe quel développeur.
En pratique : la fusion est réalisé par une/deux personne(s) appelées *"reponsable(s) de version*"

Rôle différent du propriétaire de version :

- Il garde une trace du nombre de modifications en cours d'étude, des modifications approuvées ou qui semblent bientôt l'être, ...
- **[Optionnel]** Il relance les développeurs pour qu'ils examinent et votent.
- Il fusionne les modifications dans la branche de version.
- Lors de la publication de la version, il s'occupe de la création du package de version finale; de la mise en ligne du package et de la création des annonces publiques.

Packaging

La forme standard de distribution d'un logiciel open source est le code source :

- Exécution directe des sources (interprété) comme en Python, Perl, ...
- Compilation des sources comme en C, C++, Java, ...

💭 Dans le cas de logiciels compilés, les utilisateurs ne compilent pas (en général) les sources et installent à partir de binaires pré-construits

Objectif du packaging : définir sans ambiguïté une version !
💭 Si je distribue *"monprojet.2.3.1"*, je fournis une arborescence de fichier source qui, une fois compilée (si besoin) et installée, produit *"monProjet 2.3.1"* !

Packaging

Privilégiez des formats usuels pour founir le code source :

- *TAR* pour O.S. Unix (compressable via `gzip` ou `bzip2`)
- *ZIP* pour O.S. Windows (compressable automatiquement)

💭 Pour les projets JavaScript, il est standard de fournir les versions *"minifiées"* avec les fichiers sources lisibles par l'homme.

Nommage

Nommage standard : nom du logiciel + numéro de version + format archive

⚠️ Certain projets dérive de cette norme (non recommandé)

Par exemple dans *Spack* : les packages ont un prefixe « py- » s'ils sont utilisés principalement comme bibliothèque python

- Mercurial (écrit en python) se nomme *"mercurial-..."* car CLI
- PyQt (écrit en C++) se nomme *"py-pyqt"* car binding python de la bibliothèque QT

Agencement

Une fois décompressé, l'archive associée à la version devrait créer une nouvellle arborescence nommée « monprojet.2.3.1 » avec:

- Le code source, qui doit y être organisé pour la compilation et l'installation.
- Un README, qui contient ce que fait le logiciel, sa version, des pointeurs vers d'autres ressources (fichiers / lien Web).
- Un INSTALLING / BUILDING, qui contient des instructions pour build et installer le logiciel.
  💭 Par exemple https://github.com/nodejs/node/blob/master/BUILDING.md
- La LICENCE, qui décrit les conditions de distribution du logiciel.
- *(suite prochain slide)*

Agencement

- Un fichier CHANGELOG / NEW / RELEASE_NOTES, qui liste des changements liés à la nouvelle version.
  💭 Par exemple : https://github.com/python-gitlab/python-gitlab/blob/main/CHANGELOG.md

💭 Ce fichier accumule les modifications de toutes les versions (ordre chronologie inversé) et son objectif est de donner un aperçu de ce que gagnerait un utilisateur à changer de version et de signaler les changement incompatibles

⚠️ Il est important de le mettre à jour lors de la stabilisation d'une nouvelle version (peut être fait au fils de l'eau ou à la fin) !

Agencement

L'agencement du code source d'un package doit être aussi proche que possible de celui qu'on obtiendrait depuis le référentiel

- peut contenir des fichiers générés en plus (ex : pour la configuration et compilation);
- peut incorporer des logiciels tiers;
  💭 (exemple perso : Hercule + répertoire deps : pybind11, lm, ...)
- ne doit pas être une copie de travail (clône) !

Voyez-vous pourquoi ?

Agencement

L'agencement du code source d'un package doit être aussi proche que possible de celui qu'on obtiendrait depuis le référentiel

- peut contenir des fichiers générés en plus (ex : pour la configuration et compilation);
- peut incorporer des logiciels tiers;
  💭 (exemple perso : Hercule + répertoire deps : pybind11, lm, ...)
- ne doit pas être une copie de travail (clône) !

Voyez-vous pourquoi ?

- La version doit représenter un point de référence statique !
- Le risque serait que l'utilisateur mette cette version à jour ...

Agencement

Dernière remarque : l'agencement doit rester le même peu importe le type de package (TAR, ZIP, ...)

💭 Certaines différences mineures peuvent exister :

- fichier texte terminant par CRLF (Carriage Return and Line Feed → "\r\n") pour Windows et LF pour Unix
- arborescence légèrement modifiée pour correspondra au attente d'un O.S. lors d'une compilation

Compilation et installation

Il existe des procédures standards que les utilisateurs expérimentés s'attendent à pouvoir suivre ! Typiquement, pour les programmes C et C++, la *norme* sous Unix veut que l'utilisateur utilise trois commandes :

1. `./configure`
   - Détecte automatiquement autant d'information possibles sur l'environnement et prépare le processus de build
2. `make`
   - Lance le build du logiciel sur place (mais ne l'installe pas)
3. `make install`
   - Lance l'installation sur le système

- Pour plus d'info sur les Autotools : http://sources.redhat.com/autobook/

Compilation et installation

Dans le cas des programmes Python, la *norme* veut qu'un utilisateur puisse installer le package via l'une de ces commandes :

- `pip install mon_package` (mon_package peut être une archive, une wheel ou le nom d'un projet)
- `python setup.py install`

⚠️ Il est important qu'un projet suive des procédures de compilation et d'installation standard !

- Si un admin systeme voit des appels connus dans le fichier INSTALL, cela augmente sa foie en votre projet.
- Si un utilisateur voit des procédures totalement non standard, il s'enfuira confus!
- Si un développeur ne peut pas facilement build et tester votre projet, il n'y participera pas

Package binaire

Même si une version officielle est un package de code source, les utilisateurs utilisent souvent des packages binaires !
💭 Peuvent être obtenus via le mécanisme de distribution de l'O.S., le site web du projet, un tiers, ...

⚠️ Ici *"binaire"* ne signifie pas *"compilé"*. Cela désigne une forme préconfigurée du package !

- Cela permet à l'utilisateur d'installer sans passer par les procédures de build et d'installation basées sur les sources
- Sur RedHat GNU/Linux → système RPM, sur Debian GNU/Linux → système APT (.deb), ...

Package binaire

⚠️ Dans le cas où le binaire est obtenus par le biais d'un tiers (distant du projet) celui-ci sera traité par les utilisateurs comme un équivalent à la version officielle du projet !

💭 Cela peut entraîner des tickets dans le gestionnaire de bogue en fonction du comportement du binaire paquets

=> Il est important de fournir des directives claires aux *"packageurs"* pour que ce qu'ils produisent représente le logiciel de manière juste et précise !

Remarque : un *"packageur"* doit toujours basé son package binaire sur une version officielle des sources !

A votre avis, pourquoi doit-il toujours en être ainsi ?

Package binaire

Explication avec une situation imaginaire : un packageur veut embarquer des modifications validées après publication de la version (correction de bogue ou améliorations)

- Pense faire une faveur aux utilisateurs en fournissant du code plus récent.
- Crée de la confusion entre les utilisateurs et le développeurs du projet !

Un projet est préparé à traiter des issues pour des versions publiées ou le tronc principal :

- si un bogue connu est rapporté : permet de confirmer que celui-ci est connu pour cette version et qu'il est corrigé (ou le sera) dans une version plus récente;
  - *(suite prochain slide)*

Package binaire

- si un bogue inconnu est rapporté : permet de facilement le reproduire et de le catégoriser.

Un projet n’est pas préparé à traiter des issues pour des versions hybrides !!

- un bogue rapporté peut être difficile à reproduire, e.g. résultat d’interactions inattendues entre des changements isolés

[Expérience perso] : Hercule + patch pour fixer un bogue + release qui n’aurait pas du contenir le correctif associé

Package binaire

💭 Dans certaines circonstances, un packageur insistera sur le fait que des modifications à la version source sont nécessaires.

Exemple historique : renommage des applications de Mozilla par Debian

- Le renommage est une solution à une incompatibilité entre les principes du logiciel libre (selon Debian) et la politique de marque de Mozilla
- Chaque renommage consistait en : la reprise du code source de l'application d'origine; la modification du nom et du logo (non libres selon Debian); l'application de patchs spécifiques à Debian; des mises à jour et une évolution suivant de près celles de l'application d'origine

Firefox, Thunderbird, Sunbird (Mozilla) | Iceweasel, Icedove et Iceowl (Debian)

Package binaire

Lorsqu'un packageur insiste, il doit être encourager à en parler aux développeurs et à décrire son plan

- peut potentiellement obtenir l'approbation des développeurs;
- aura au moins notifié le projet qui pourra surveiller les rapports de bogues inhabituels.

Possible contre mesure des développeurs :

- mettre une clause de non responsabilité sur le site du projet
- le packageur place aussi une clause à l'endroit approprié
  => les utilisateurs sont notifiés que le package binaire n'est pas exactement la version officiellement publiée

Package binaire

💭 Les objectifs des packageurs diffèrent légèrement de celui des développeurs

- le **packageur** va cibler la **meilleure expérience** prête à l'emploi pour les **utilisateurs**
- le **développeur** le souhaite aussi mais doit s'assurer qu'il connaît les versions disponible du logiciel afin d'avoir des **issues cohérentes** et de pouvoir assurer des **garanties de compatibilité**

⚠️ Il est important de garder une certaine cohésion entre les deux groupes car les packageurs réalisent un travail *"ingrat"* pour rendre le logiciel plus largement disponible !

V.5 – Tester et publier

Test avant publication

Avant que la version soit rendu publique, elle doit être testée et approuvée par un nombre minimum de développeur.
💭 Cette validation est ensuite signalée au monde par le biais de signatures numériques et de hash d'identification

Savez-vous ce que sont les signatures numériques et hash d'identification ?

Test avant publication

Avant que la version soit rendu publique, elle doit être testée et approuvée par un nombre minimum de développeur.
💭 Cette validation est ensuite signalée au monde par le biais de signatures numériques et de hash d'identification

Savez-vous ce que sont les signatures numériques et hash d'identification ?
Exemple : https://pypi.org/project/numpy/#files

Objectif : fournir un moyen de vérifier que la copie reçu n'a pas été falsifiée
💭 Evite qu'un individu malveillant ait placé une porte dérobée vers les données de l'utilisateur

La création de signatures numériques et de hash de version est un sujet vaste :
https://www.apache.org/dev/release-signing.html

[Optionnel] Illustration via sha26

Test avant publication

Nous avons parlé d'approbation des développeurs, mais qu'est ce que cela veut dire ?

- Ne consiste pas simplement à inspecter la version en cherchant des défauts évidents !
- Idéalement, les développeurs vont :
  - télécharger le package;
  - build et l'installer sur un système propre;
  - lancer la suite de tests (unitaire, non régression, ...);
  - réaliser des tests manuels.

Si ces vérifications (et potentiellement d'autres) sont satisfaites, les développeurs signent numériquement le package avec un programme produisant des signatures compatibles OpenPGP, e.g. GnuPG https://www.gnupg.org/

[Optionnel] Illustration via gpg

Test avant publication

```
                # Signature GPG

# Creation d'une paire de clef avec gpg (author)
1) gpg --gen-key

# Author signe l'archive et crée le fichier data.author.asc
2) gpg -u author -a -o data.author.asc --detach-sig data.txt

# Creation d'une clef paire de clef avec gpg (validator)
3) gpg --gen-key

# Validator récupère les fichier data.author.asc et data.txt, vérifie qu'il a une signature valide de author et signe aussi
4) gpg --verify data.author.asc data.txt
   gpg -u validator -a -o data.validator.asc --detach-sig data.txt

# On a ainsi 3 fichiers et on peut les verifier un à un
5) gpg --verify data.author.asc data.txt
   gpg --verify data.validator.asc data.txt

# Note : il est possible de réduire le nombre de fichier en 1 seul
   cat *.asc > allsigs.asc
   gpg --verify allsigs.asc data.txt
```

Test avant publication

Remarques :

- Les développeurs peuvent utiliser une clé de projet partagée
  💭 En général, ils utilisent plutôt leurs signatures numériques personnelles.
- Autant de développeurs qui le souhaitent peuvent signer numériquement
  💭 Plus il y en a, plus la version est testée et plus un utilisateur aura confiance !

Une fois approuvée la version est placée dans la zone de téléchargement du projet AVEC les signatures numériques et les hash. https://nodejs.org/en/download/ (How to verify)

Test avant publication

Plusieurs normes possible, par exemple vous pouvez accompagner chaque package par :

- un fichier contenant les signatures numériques correspondantes;
- un autre fichier donnant les hashs.

💭 Typiquement, dans le même répertoire que mon_projet-2.3.0.tar.gz :

- mon_projet-2.3.0.tar.gz.asc : contenant les signature numérique;
- mon_projet-2.3.0.tar.gz.md5 : contenant la somme de contrôle MD5;
- mon_projet-2.3.0.tar.gz.sha256 : contenant la somme de contrôle SHA256;
- …

💭 La norme choisie n'a pas d'importance, il faut surtout garder un schéma simple, décrit et cohérent d'une version à l'autre !

Publier des versions candidates

Pour les versions contenant beaucoup de changements, n'hésitez pas à utiliser des versions candidates.

Objectif : soumettre le code à des tests « intensif » avant de le bénir comme version officielle ! Par exemple : mon_projet-2.3.0-rc1

- si bogue découvert, il est corrigé sur la branche de version et une nouvelle version candidate est déployée (mon_projet-2.3.0-rc2)
- converge vers la version officielle lorsqu'il n'y a plus de bogue inacceptable

💭 Une version candidate doit être traitée comme une version réelle : envoie de courriers annonçant la version candidate et indiquant que l'objectif et de récupérer un maximum de retour d'informations !

L'indicateur rc avertira les utilisateurs conservateurs d'attendre !

Publication par éditeur de confiance

Concept utilisé par les écosystèmes de gestion de packages de Python (PyPI), JavaScript (npm), ...

Un éditeur ou une organisation qui a été vérifié et identifié comme digne de confiance par la plateforme de gestion de packages.

Objectif: éliminer le besoin de jetons et mots de passe à longue durée de vie !

💭 Réduction du risque d'attaques de la chaîne d'approvisionnement et de fuites de données d'identification, tout en rationalisant le workflow de publication.

💭 Fonctionnement basé sur OpenID Connect (OIDC), une norme ouverte d'attestation et de vérification d'identité fondée sur OAuth2.

V.6 – Gérer plusieurs versions

Gestion de plusieurs ligne de versions

De nombreux projets maintiennent plusieurs lignes de version en parallèle.

Exemple : un projet évoluant sur la ligne 1.0.x

Pourquoi le fait de publier la version 1.1.0 n'est pas une raison suffisante pour terminer la ligne 1.0.x ?

Gestion de plusieurs ligne de versions

De nombreux projets maintiennent plusieurs lignes de version en parallèle.

Exemple : un projet évoluant sur la ligne 1.0.x

Pourquoi le fait de publier la version 1.1.0 n'est pas une raison suffisante pour terminer la ligne 1.0.x ?

Certains utilisateurs ne veulent pas utiliser une nouvelle version démarrant une nouvelle série mineure / majeure !

=> Si le projet découvre un bogue majeur dans une version 1.0.x il est important que sa correction soit présente dans la nouvelle série et dans une nouvelle micro (1.0.x+1?)

Gestion de plusieurs ligne de versions

Dans ce contexte, comment inciter à la transition vers une version mineure / majeure plus récente ?

Gestion de plusieurs ligne de versions

Dans ce contexte, comment inciter à la transition vers une version mineure / majeure plus récente ?

💭 Déclarez la série 1.0.x en fin de vie et annoncez le officiellement !

Objectif : notifier les utilisateurs pour qu'ils prennent des décisions sur la mise à jour

Fin de vie d’une série

Certains projets fixent une fenêtre de temps pendant laquelle ils s’engagent à supporter la série précédente de versions

- Dans ce contexte, *"support"* signifie accepter des rapports de bogues concernant la série en question et faire des versions de maintenance lorsque des bogues important sont trouvés.
- **Evaluer les rapports de bogue** relatif à une série est un bon moyen de **déterminer *"quand"* déclarer la fin de vie d’une série** !

Pour la fin de support à python 2.7 : apparition d'un message d’obsolescence prochaine (e.g. à l'installation de paquet via `pip` l'utilisation) et site dédié au compte à rebours de fin de support https://pythonclock.org/

Version de sécurité

Une version faite uniquement pour fermer une vulnérabilité de sécurité

Contexte particulier car :

- Le bogue ne doit pas être rendu publique tant que la version n’est pas disponible
- Le correctif ne doit pas être introduit dans le référentiel ET la version ne peut pas être testée publiquement !

=> Il est important de partir d'une version existante et d’ajouter les correctifs sans autres modifications !
💭 Des modifications non testées pourraient être la source de nouveaux bogues

Version de sécurité

Comment est introduit une version de sécurité dans le flot des versions prévue (avec correctif publiquement annoncés) ?

Version de sécurité

Comment est introduit une version de sécurité dans le flot des versions prévue (avec correctif publiquement annoncés) ?

- Solution possible : subterfuge consistant à remplacer une version prévue et à reporter les correctifs qui lui étaient liés à la version suivante
- Autre solution : ajouter un composant supplémentaire au numéro de version, e.g. la version 2.3.0 devient 2.3.0.1
  💭 Plus rarement utilisé car rompt avec la numérotation à trois composants

Version de sécurité (extra slide)

Connaissez vous des méthodes pour auditer votre code ?

Version de sécurité (extra slide)

Connaissez vous des méthodes pour auditer votre code ?

Exemples d'outils d'analyse pour paquet Python:

- Safety : vérifie les dépendances d'un projet pour identifier les paquets vulnérables. Compare les paquets à une base de données de vulnérabilités connues.

  Equivalent dans d'autres langages : npm audit, OWASP Dependency-Check, ...

- Bandit : analyse statique du code pour identifier des failles potentielles, e.g. injections SQL, fonction dangeureuse, gestion incorrecte des fichiers, ...

  Equivalent dans d'autres langages : NodeJSSCan, FlawFinder, ...

Développement quotidien

Maintenir des versions en parallèle nécessite une **certaine discipline**

Voyez vous ce que cela induit ?

- Faire en sorte que chaque commit soit un seul changement logique
- Ne pas mélanger des changements sans rapport dans un même commit

💭 Si un changement est trop important :

- partitionner le changement en N commits bien délimités
- chaque commit ne doit pas inclure quelque chose sans rapport avec le changement global

Exemple de commit mal pensé au prochain slide

Exemple de commit mal pensé

“ Commit 2a01e815
Auteur : S. Morais sebastien.morais@proton.me
Date : Mar 2, 2020 10:33am

FEATURE : Ajout d’une classe dédiée à l’écriture sur disque

utils/writter.py
    ajout de la classe HerculeWritter
    modification du parser de type (sémantique)
utils/constants.py
    centralisation des constantes
setup.cfg
    oublie de mise à jours lors du précédent commit
doc/index.rst
    correction de typos ”

Développement quotidien

Le problème devient apparent si quelqu'un veut seulement récupérer une partie des modifications, e.g. celles sur le fichier de configuration « setup.cfg »

- Il ne veut pas récupérer les autres modifications.
- C'est difficile de récupérer uniquement ce qu'il veut car le système de gestion de version voit cela comme un ensemble de changement logiquement groupés.
- Avant même de récupérer la modification, il est difficile de citer la modification (qui pourrait être soumise à un vote) !
  - Pas de numéro de commit simple
  - Besoin de créer un patch spécial OU une branche pour isoler la portion du code proposée

Développement quotidien

=> Beaucoup de travail dû à un contributeur qui n'a pas pris le temps de séparer sa contribution !

La modification aurait dûe être éclatée en 4 commits :

- un pour la classe introduite et l'effet de bord sur la sémantique;
- un pour le fichier de configuration;
- un pour la correction de typo;
- un pour l'extraction des constantes.

💭 Par ailleurs, il est plus simple de review et d'annuler un commit sémantiquement unifié.

Planification des versions

Dans quelle mesure la planification des versions diffère entre un projet libre / open source et propriétaire ?

Planification des versions

Dans quelle mesure la planification des versions diffère entre un projet libre / open source et propriétaire ?

- Un projet propriétaire a généralement des délais plus fermes
  - promesses à un client
  - coordination avec d'autres efforts à des fins marketing
- Un projet libre / open source visent à maintenir une atmosphère de travail coopérative entre plusieurs parties
  - le maintient des relations de travail prévaut sur les délais de n'importe quelle partie !

💭 Certains projets open source sont financés par des entreprises et ont donc une gestion soucieuse des délais !

Planification des versions

Comment gérer les conflits entre contributeurs payés et bénévoles ?

Planification des versions

Comment gérer les conflits entre contributeurs payés et bénévoles ?

Conflits provenant souvent de la date / planification des versions

- **Salariés** : veulent choisir une date à laquelle la sortie d'une version aura lieu et s'assurer que les activités de chacun s'aligneront
- **Bénévoles** : pensent que la sortie de la version devrait attendre car :
  - souhaitent **terminer des fonctionnalités**;
  - veulent réaliser des **tests**.

⚠️ Pas de panacée pour ces conflits → besoin de discuter et faire des compromis

Planification des versions

Il est possible de minimiser la fréquence et le degrés de friction de tels conflits. Comment feriez-vous ?

Planification des versions

Il est possible de minimiser la fréquence et le degrés de friction de tels conflits. Comment feriez-vous ?

Découpler l'existence d'une version de la date à laquelle elle sera publique :

- Orienter la discussion sur le sujet des versions prochaines
- Parler des fonctionnalités qui y seront
- Ne pas mentionner de date (ou approximative avec grosses marges d'erreurs)

💭 Evite la discussion centrée sur une version et améliore la prévisibilité !

⚠️ Crée aussi un biais d'inertie contre l'élargissement du périmètre d'une version : si le périmètre est bien défini, il y a alors besoin de justifier l'extension de périmètre (même si la date n'a pas été fixée!)

Planification des versions

Stratégie alternative : laisser l'entreprise réaliser des versions intermédiaires séparées pour ses clients

- ✅ Versions publiques et open sources
- ✅ Pas de dégât au projet tant que les versions intermédiaires sont distinguées des versions officielles !
- ❌ Le maintient de telles lignes de versions est coûteux !
  ⚠️ Besoin de suivis de modification et de transfert dans les deux sens !

💭 Technique qui ne fonctionne que si l'entreprise a les ressources nécessaires pour dédier des personnes à la gestion des versions séparées !