Apprendre à coder correctement

Un guide concis pour écrire un meilleur code

Par Robat Williams - vavavoum74 (traducteur)

Date de publication : 6 février 2019

Pour réagir au contenu de ce guide, un espace de dialogue vous est proposé sur le forum.
Commentez

I - Préambule

Un guide/une checklist pour aider les utilisateurs à améliorer leur codage et leur contrôle de code.

Une grande partie est assez générale, mais reflète mon intérêt pour les applications Web « frontend ».

Ndt : il s'agit finalement des éléments du site que l'on voit à l'écran et avec lesquels on peut interagir.

Aller au contenu

Motivation

Par ou commencer lorsque l'on cherche à écrire un meilleur code ? Et comment maintenons-nous la qualité et la cohérence du code lorsque nous travaillons en équipe ?

La révision de code est un excellent moyen de procéder, par le biais de retours d'information et de partage des connaissances. Cependant, il s'agit d'une approche réactive, qui a tendance à ne pas être systématique et difficile à faire évoluer. En instaurant et en créant/adaptant une référence partagée pour les méthodes et la qualité, une équipe peut surmonter ces problèmes.

Il est rarement pratique pour tout le monde ou pour une personne de contrôler chaque ligne de code, tout en poursuivant son travail et ses responsabilités. En tant que relecteurs, nous avons tous notre propre expérience, nos connaissances techniques/de projet et notre niveau de compétence. Nous pouvons en tirer davantage en les partageant avec les autres plutôt qu'en appliquant les nôtres individuellement.

Ceci est un guide personnel, basé sur certaines des choses que je cherche. Tout le monde ne sera pas d'accord avec tout cela, mais je pense que c'est majoritairement raisonnable.

J'espère que cela vous aidera à rédiger un meilleur code, à être un contrôleur plus efficace et à susciter votre intérêt dans un apprentissage ultérieur.

Contrôle du code

Le contrôle du code est plus qu'un processus de contrôle, mais beaucoup de choses que nous recherchons sont simples et il y en a trop à retenir. Les check-lists nous aident à nous souvenir et informent les contributeurs dès le départ des attentes. Cela aide les contrôleurs à avoir plus de temps à consacrer aux choses les plus importantes et les moins simples, et à réduire le temps consacré aux révisions.

Ces éléments pouvant être automatisés devraient l'être (à l'aide d'outils tels qu'ESLint et ses plugins, ou checkstyle). Les détections et les corrections manuelles ne sont pas fiables, ne s'adaptent pas bien et consomment de l'attention qui serait mieux dépensée ailleurs.

Qu'est-ce que ce n'est pas

Ce n'est pas un guide de style, une déclaration de droit chemin, un guide complet ou définitif pour écrire un bon code et être un bon contributeur, ou un raccourci pour devenir un bon développeur. Je ne m'attends pas à ce que tout le monde soit d'accord avec tout cela. Les choses qui devraient être traitées par des outils automatisés configurés de manière appropriée ne sont généralement pas mentionnées. Pour en savoir plus sur le processus de contrôle, la conduite, la collaboration et les outils, consultez **la liste impressionnante de contrôle de code**.

Lectures complémentaires

Les livres suivants sur l'écriture de bons codes sont populaires et appréciés.

- **97 Things Every Programmer Should Know** *(free to* **read online**).
- **Clean Code**.
- **Code Complete**.

Licence

Voir **ici**.

II - Contenu

III - Nommer les choses

Il n'y a que deux choses difficiles en informatique : l'invalidation du cache et nommer les choses

- *Phil Karlton.*

Un nom pour chaque chose et chaque nom utilisé pour une seule chose. Utiliser plusieurs mots pour signifier la même chose, ou le même mot pour signifier des choses différentes gaspille de la concentration et risque de créer de la confusion. Cela s'applique non seulement au code, mais à tout ce qui concerne notre travail, par exemple : noms de variables et de classes, noms de méthodes, données, concepts de domaine, dépôts, configuration et autorisations.

Utilisez des noms bien connus du monde des domaines commerciaux et techniques. Cela permet d'avoir un vocabulaire commun qui rend la communication plus efficace. Évitez d'utiliser ces noms avec des significations différentes.

Utilisez des noms qui ont une signification. La signification est essentielle pour comprendre le code. Les noms génériques tels que « données » ou « valeur » sont trop vagues.

Évitez de laisser des noms pauvres/incohérents s'infiltrer depuis d'autres contextes dans le code. Ces contextes incluent le code hérité, les bibliothèques et les autres applications avec lesquelles nous interagissons. Établissez une limite à l'intérieur de laquelle ces directives de nommage sont respectées et défendez-la de la pollution extérieure.

Il est normal que les noms présentés à l'utilisateur soient différents. Les noms affichés sur l'interface utilisateur/ sur la sortie répondent souvent à des exigences et apparaissent dans leur propre contexte. Lorsque ces noms sont médiocres ou inappropriés du point de vue du code, utilisez-en un autre si possible.

Utilisez des acronymes et des abréviations avec parcimonie. Le code est généralement plus clair et plus facile à lire sans ceux-ci. Ceux qui sont universellement acceptés comme *id* font exception à cette règle, ainsi que ceux du domaine professionnel.

Évitez les noms inutilement longs. Essayez de ne pas être plus long que nécessaire pour transmettre un message. Les noms longs augmentent la nécessité pour les instructions et les appels de méthodes de s'étendre sur plusieurs lignes, ce qui rend le code moins lisible. Cette règle implique d'éviter l'utilisation du nom complet d'un type en tant que nom de variable/paramètre lorsque cela n'est pas nécessaire.

Évitez les noms trop qualifiés. Lorsque le contexte d'un nom a déjà été établi, par exemple en étant dans une classe ou une méthode, il n'est pas nécessaire de reformuler ce contexte. Une « *task* » (tâche), par exemple, n'a pas besoin d'une « taskCompleted» (tâche accomplie).

Utilisez des noms positifs pour les booléens et évitez la négation dans le nom. Les noms négatifs entraînent des doubles négatifs, ce qui rend les expressions plus difficiles à saisir

(par exemple : enabled: true (activé :vrai) vs disabled: false(désactivé: faux)). Utiliser le mot «not» (Pas) pose le même problème.

Évitez l'usage intensif de « get » comme préfixe du nom de la méthode. Les accesseurs (getters) renvoient des valeurs. Préférez des alternatives plus informatives telles que "request" (requête), "fetch" (cherche), "find" (trouve), "query" (demande), "build" (construit) ou "create" (crée), le cas échéant.

Recherchez des noms distinctifs dans le même contexte. Des noms similaires sont faciles à confondre et rendent le code difficile à lire.

Envisagez de qualifier chaque nom lorsqu'il devient nécessaire de le faire. Cela donne souvent plus de clarté et de précision que la seule qualification pour le distinguer, et il est plus probable que le bon sera choisi pour être utilisé aux endroits appropriés. Exemple : previousFoo & nextFoo , au lieu de previousFoo (précédentFoo) & foo.

Pensez à nommer les choses liées afin qu'elles apparaissent ensemble dans l'ordre alphabétique. Cela facilite la découvrabilité lorsque vous utilisez une convention d'ordre alphabétique. Exemple : data , dataError , dataLoaded .

Les méthodes doivent toujours faire ce que leur nom promet. Les noms de méthodes traduisent les attentes et les utilisateurs seront surpris si elles ne sont pas satisfaites. Évitez de ne pas faire ce que le nom promet, et **créez une exception si vous ne pouvez pas le faire**.

Utilisez des noms de paramètres de type significatifs lorsqu'une seule lettre n'est pas claire. Cela améliore la lisibilité. Utilisez une convention établie (par exemple, le suffixe «T») pour les distinguer des noms de types.

Évitez les mots composés avec CamelCasing (Ndt : *Camel case* de l'anglais, littéralement « casse de chameau » est une pratique qui consiste à écrire un ensemble de mots en les liant sans espace ni ponctuation, et en mettant en capitale la première lettre de chaque mot.). Ce sont des mots simples à part entière, alors vous n'en avez pas besoin. Exemple : *callback, password*.

Orthographiez correctement et systématiquement. Cela permet d'éviter les erreurs et améliore la possibilité de recherche dans le code. S'il existe plusieurs orthographes correctes (par exemple, clef/clé), suivez les conventions de la plate-forme.

Suivez les conventions existantes autour de vous. Celles-ci peuvent se trouver dans une méthode particulière, dans l'ensemble du code de base ou la technologie/plate-forme avec laquelle vous travaillez. Efforcez-vous de conserver cette cohérence et évitez de faire les choses différemment sans bonne raison.

Ce résumé peut être utile : **List of names that tend to be useful in programming**.

IV - Flux de code

IV-A - Général (flux de code)

Évitez de faire un travail important dans les constructeurs. Cela limite souvent la flexibilité lors de la construction et de l'initialisation de la classe, ce qui a généralement pour conséquence de rendre difficile les tests.

Méfiez-vous des conditions similaires trop nombreuses. Elles peuvent être le signe d'une abstraction manquante ou mal ajustée, ou un besoin général d'améliorer la conception.

Garde Fou. Faites attention en ne considérant que le scénario le plus courant. Par exemple : pas de données, données non valides, conditions aux limites, erreurs et valeurs null (le cas échéant).

Évitez d'exécuter du code inutile. Si le travail est terminé et que la valeur de retour est prête, retournez-la. L'exécution de code supplémentaire fait perdre du temps et masque l'intention pour ceux qui le lisent.

IV-B - Méthodes (flux de code)

Les méthodes doivent être autonomes, en général. Évitez les méthodes qui nécessitent toujours une « méthode associée » pour être appelées d'abord/après, par exemple pour vérifier les conditions préalables ou pour récupérer un résultat/une erreur.

Protégez de l'extérieur contre les appels non valides prévus. Dans les cas attendus où l'appel de la méthode n'a pas de sens, évitez de l'appeler - au lieu de l'implémenter pour gérer les cas non valides/sans signification. Par exemple, un validateur de formulaire de saisie pour le nombre de décimales ne devrait pas être appelé avec autre chose qu'un nombre.

IV-C - Valeur par défaut

Évitez les valeurs par défaut inappropriées. Les possibilités d'utilisation des valeurs par défaut incluent les variables locales/les valeurs de membres, les arguments de méthodes et les valeurs de configuration. Avant de créer une valeur par défaut, demandez-vous si elle est utile, et pesez le pour et le contre face à son potentiel de cacher des erreurs ou d'être utilisée accidentellement et inconsciemment.

Évitez que les collections par défaut soient vides pendant le chargement des données. Cela supprime/masque la distinction entre les états de chargement et les états sans données, ce qui entraîne souvent une gestion inappropriée/manquante.

Pour en savoir plus sur les valeurs par défaut : mon article sur les **Default values in code and configuration**.

IV-D - Contrôles

Évitez les contrôles de null inutiles. Il y a certaines choses dans les domaines de la logique technique et commerciale sur lesquelles nous pouvons nous fier en toute confiance pour ne jamais être nulles. L'ajout de vérifications explicites peut masquer des problèmes et alourdir le code.

Évitez les vérifications null/d'état inappropriées (ou celles situées dans des emplacements inappropriés) qui corrigent les problèmes sous-jacents. Ce sont généralement des vérifications sautant la logique de branche dans des scénarios non valides ou dans des cas extrêmes, faites à un endroit donné pour éviter/dissimuler un problème qui est apparu ailleurs. Au lieu de cela, préférez traiter/résoudre le problème à la source. Sachez également que les contrôles dans les appelants doivent être aussi dans les appelés, et inversement.

IV-E - Gérer les mauvais raisonnements

Échouer tôt et fort, plutôt que plus tard/tranquillement. Détectez les problèmes dès que possible et rendez-les visibles, par exemple en générant une erreur. Évitez de détecter, puis de masquer les problèmes en sautant du code ou en utilisant une valeur par défaut. Les exemples incluent les contrôles d'initialisation, les contrôles de validité, les conditions préalables de méthode et les contrôles d'état de réponse de l'API. Voir **Functional** > fail fast.

Code défensif contre attentes strictes. L'écriture de code défensif pour prendre en compte les problèmes environnants est une approche qui peut être utilisée pour rendre les applications plus stables - contourner les problèmes au lieu de leur permettre de provoquer une défaillance. Cependant, un inconvénient majeur est que ces problèmes sont moins visibles, ce qui les rend moins susceptibles d'être remarqués et traités. Cela complique également le code avec une manipulation éventuellement inutile « au cas où ». Pour un domaine donné de votre application, pesez les avantages et les inconvénients pour vous aider à décider si une approche défensive ou stricte est la plus appropriée. À une frontière de système où une application interagit avec quelque chose d'extérieur en dehors de son contrôle, par exemple, il est plus que probable que la méthode défensive soit plus appropriée.

V - Structure du code

V-A - Lisibilité générale et maintenabilité

Diviser les choses longues/grandes. Cela inclut les fichiers, les dossiers, les méthodes et les modèles. Avoir beaucoup de lignes/fichiers/parties indique que quelque chose en fait trop, ce qui rend difficile la compréhension, la maintenance et l'adaptation. Cherchez des abstractions et des fonctions pour lesquelles une scission pourrait être faite.

Évitez d'ajouter du code a la volée. Pensez à l'endroit où le nouveau code devrait aller. Il peut parfois sembler pratique d'accomplir une tâche en ajoutant quelques lignes/méthodes ici et là aux méthodes/classes existantes. Peut-être ajouter quelques drapeaux « de marquage » pour ignorer certaines parties de certaines méthodes. Cependant, ce n'est pas parce qu'un endroit est pratique pour mettre quelque chose que c'est le bon, évaluez si de nouvelles méthodes/classes doivent être créées. Pour en savoir plus : **Detecting Refactoring Diligence**, par Michael Feathers.

Évitez les fonctions imbriquées/lambdas non triviales. Le code peut souvent être clarifié en l'extrayant vers des fonctions de niveau supérieur ou des fonctions membres.

Efforcez-vous de ne dépendre que de choses proches/reliées. Changer un code/comportement lointain ne devrait pas casser des choses. Si inévitable, documentez ces relations. Voir **comments**.

V-B - Séparation et encapsulation des responsabilités

Évitez de mélanger différents types de responsabilités dans la même classe/méthode. Chacune devrait généralement être impliquée dans un type de responsabilité. Évitez de mélanger la manipulation de la structure de

données avec la logique métier ou la logique de comportement de l'interface utilisateur avec les opérations DOM qui la réalisent, par exemple.

Éviter de rompre la séparation des couches. Il existe presque toujours un moyen de réaliser ce qui est nécessaire, sans recourir à cela. Les concepts au sein de chaque couche doivent rester abstraits (c'est-à-dire pas du tout visibles) aux couches inférieures ou supérieures. Par exemple, une couche de données n'a aucune connaissance des demandes HTTP ni des codes de réponse, et une couche de service n'a aucune connaissance de JDBC. Voir **gestion des erreurs (error handling)**.

Évitez de violer ou de contourner les modèles de conception appliqués. Lorsque des motifs ou d'autres caractéristiques de conception ont été mis en place, évitez de les contourner, de les renverser ou de les démanteler.

Encapsuler des conventions d'application. La répétition de conventions, même apparemment triviales, les rend difficiles à identifier, à appliquer de manière cohérente et à modifier. Les conventions peuvent être liées à la fonctionnalité (par exemple, l'affichage de nombres/dates) ou au code (par exemple, notre modèle d'implémentation de X). Certaines techniques peuvent être appliquées à un seul endroit (par exemple, la configuration d'un sérialiseur), tandis que d'autres sont définies à un seul endroit mais doivent être utilisées/appliquées par le programmeur chaque fois que cela est pertinent (par exemple, des composants, des fonctions utilitaires).

V-C - Ordonnancement

Pensez à l'ordre des choses en général. Ordonner des objets (par exemple, des méthodes, des fichiers de configuration) de manière bien pensée permet de trouver plus rapidement et plus facilement ce que vous cherchez. Cela inclut de déterminer si ce que vous cherchez existe tout court. Cela vous aide également à remarquer les choses connexes que vous ne recherchiez pas mais que vous devriez probablement savoir. Le meilleur ordre pour la plupart des choses n'est pas au moment où elles ont été ajoutées. Certaines choses nécessitent un jugement individuel, tandis que d'autres sont partiellement/complètement bien définies (par exemple, alphabétique, **lifecycle methods**).

Ordonnez les méthodes de cycle de vie dans l'ordre dans lequel elles sont appelées. Il est intuitif d'utiliser cet ordre déjà établi et familier.

Mettez les méthodes connexes ensemble. Il est plus facile de suivre ce que fait le code lorsque la navigation dans le fichier est réduite.

Mettez le code important en haut et le code moins important en bas. Ceci est connu sous le nom de « structure de code journal » ; commencez par l'important (le titre), puis poursuivez votre lecture pour plus de détails par ordre décroissant d'importance (paragraphe de résumé, texte de l'article complet). Cela permet au lecteur de comprendre rapidement les éléments clés du fichier sans avoir à tout scanner. En prenant cela en exemple, les constructeurs iraient près du sommet, et les méthodes auxiliaires et les accesseurs/mutateurs (getters/setters) iraient au bas.

VI - Programmation générale

VI-A - Écrire, lire et travailler avec du code

Le code devrait être facile à suivre. Après avoir écrit un morceau de code (et le tester, le cas échéant), réfléchissez s'il pourrait être réécrit pour le rendre plus clair. Un code clair consomme moins d'énergie mentale pour le comprendre, ce qui laisse plus de place pour de meilleures utilisations.

Le code doit être facile à utiliser correctement et difficile à utiliser incorrectement. Il devrait guider naturellement les gens sur la bonne voie. Il devrait gêner et rendre les choses difficiles quand quelqu'un essaie de mal l'utiliser.

Le code doit être facile à modifier correctement et difficile à modifier incorrectement. Il devrait aider les gens à décider quoi changer et où le faire. Il devrait résister lorsque quelqu'un essaie de le changer à tort (le casser) et provoquer des échecs de test.

Minimisez le bizarre, l'inhabituel et l'ingéniosité. Préférez la simplicité et la clarté. Écrivez un code ennuyeux et écrivez-le pour aider les autres (et votre futur moi) plutôt que d'essayer de les impressionner.

VI-B - Général (programmation générale)

Préférez une approche fonctionnelle à une approche impérative. Des états et effets secondaires moins mutables rendent le code moins sujet aux erreurs et le raisonnement plus facile.

Réduire l'état de maintien. L'état de tout formulaire (variables, caches, par exemple) a tendance à être une source de complexité et de problèmes, il est donc préférable d'en garder le moins possible. Préférez calculer des informations en cas de besoin plutôt que de les stocker, si les contraintes le permettent.

Aidez les compilateurs/transcompilateurs et les outils d'analyse statique pour qu'ils puissent vous aider. Ces outils peuvent détecter les problèmes dans le code avant même son exécution, ce qui permet une boucle de retour d'information plus courte. Soyez conscient des problèmes que vos outils peuvent détecter et écrivez du code pour les aider. Par exemple, utilisez des annotations telles que @Override, utilisez des variables constantes, évitez d'initialiser des variables inutilement, évitez de contourner le système de sécurité de type, et évitez de ne pas tenir compte des avertissements.

Réfléchissez bien avant de supprimer un avertissement d'analyse statique ou de désactiver complètement la règle qui l'a provoqué. Cherchez pourquoi la règle a été activée et configurée en premier lieu. Les pages de documentation des règles décrivent généralement la motivation de la règle. Vous pouvez aller à l'encontre des meilleures pratiques (ou pratiques de projet) ou utiliser une technique sujette aux erreurs ou déconseillée. *Lectures supplémentaires :* **Chesterton's fence**.

VI-C - Objets et classes

Favoriser l'immuabilité. Les objets immuables sont généralement plus simples et moins sujets aux erreurs que les objets mutables. Cela est particulièrement vrai dans un contexte multitâche.

Empêcher la construction d'objets non valides. Les objets non valides causeront généralement un problème tôt ou tard. Il vaut mieux échouer tôt (voir : **code flow**) en les faisant valider au préalable par leur constructeur d'origine.

Évitez d'utiliser des objets partiellement initialisés/remplis. Cela rend difficile le suivi du processus de création et il y a un risque de les utiliser avant la fin de leur initialisation.

Évitez les classes de données polyvalentes. Il s'agit généralement de classes de données ou DTO (data transfer object) - objet de transfert de données), partiellement renseignées de différentes manières et utilisées à des fins différentes, lorsque plusieurs éléments à représenter sont assez similaires, à l'exception de quelques champs. Ils masquent ce que devrait être le domaine courant, sont sujets aux erreurs et sont difficiles à modifier.

VI-D - Données et valeurs

Créez et utilisez des types de données qui modélisent le domaine. Évitez d'utiliser des chaînes de caractères pour tout, simplement parce que c'est possible et semble être pratique. Les types de données basés sur un domaine sont essentiels à la programmation orientée objet, fournissent une base naturelle à de nombreuses méthodes et au compilateur des informations qu'il peut utiliser pour détecter une utilisation erronée.

Incluez des unités de mesure/grandeur dans les types de données de domaine. Cela permet au modèle de domaine et au compilateur de se prémunir contre les calculs invalides. Cela s'apparente à l'utilisation de l'analyse dimensionnelle des unités d'entrée et de sortie d'une équation de physique dérivée afin de vérifier qu'elle a été correctement dérivée. Cela supprime également la nécessité d'inclure l'unité (par exemple, MWh) dans les noms de propriété/variable.

Extraire des constantes pour les nombres magiques, en général. Il existe cependant des cas où cela ajoute du bruit, comme pour zéro.

VI-E - Méthodes (programmation générale)

Évitez de faire des choses auxquelles l'appelant ne s'attendrait pas raisonnablement. Le nom de la méthode traduit les attentes - évitez les effets de bord surprenants.

Retour rapide des conditions préalables échouées. Éviter que la plupart/toutes les méthodes aient besoin d'être dans une déclaration if (ou dans plusieurs déclarations imbriquées) facilite la lecture.

Évitez les longues listes de paramètres. Elles ont tendance à laisser les appelants passer des paramètres dans le mauvais ordre, même avec l'aide des EDI. C'est particulièrement le cas dans les langages non typés ou lorsque les paramètres sont du même type. Utilisez plutôt un objet de paramètres ou un générateur.

Ordonnez des paramètres intuitivement. Placez les paramètres les plus importants le plus près du premier. Placez ceux qui sont proches les uns à côté des autres. Une astuce pour aider à choisir un ordre intuitif consiste à construire une phrase décrivant un appel à la méthode ; les paramètres tombent souvent naturellement dans le bon ordre dans la phrase.

Ordonnez des paramètres régulièrement. Lorsqu'il y a des surcharges ou de nombreuses méthodes utilisant des paramètres similaires, toutes les méthodes devraient prendre d'abord les paramètres communs.

Évitez les paramètres booléens. Lors de la lecture du code d'appel, les paramètres booléens rendent difficile la compréhension de l'intention des appels. Préférez une énumération à deux éléments, un objet de paramètres ou une méthode distincte pour chaque cas. Pour en savoir plus : **The Pitfalls of Boolean Trap**, par Ariya Hidayat.

VI-F - Logique

Basez la logique sur des identifiants, pas des noms. Les noms ne sont généralement pas garantis comme étant uniques et sont susceptibles de changer. Identifiez les objets en utilisant leurs identifiants pour diriger la logique.

Évitez d'inverser les conditionnelles en utilisant l'expression négative. La gestion du cas false dans le bloc else rend la construction plus facile à suivre, surtout quand il y a plusieurs cas else if.

VII - Fonctionnel

La fonctionnalité fonctionne dans tous les navigateurs pris en charge. Cela inclut des aspects du chemin d'utilisation principal de la fonctionnalité. Internet Explorer en particulier doit être surveillé. Voir **Tech: JavaScript** > feature support.

La fonctionnalité fonctionne dans la version « mode production » (le cas échéant). Les modes de développement, souvent utilisés pour améliorer l'expérience du développeur (productivité), rendent l'application en cours d'exécution moins semblable à celle de la version de production. Tester en utilisant de tels modes peut entraîner des problèmes non détectés et, en règle générale, réduit les performances.

Les cas d'erreur sont traités. Selon la nature de l'erreur, cela inclut : la détection, la consignation, la récupération/ l'abandon et la notification à l'utilisateur. Voir **la gestion des erreurs**.

Échouez vite et de manière proactive. Détecter les problèmes à la première opportunité et les rendre immédiatement et clairement visibles (ex. : log, message d'interface utilisateur, arrêt d'application). Évitez de laisser l'application se poursuivre dans des états non définis, dans l'espoir d'aucune conséquence. La stratégie « rapide et proactive » permet de rechercher et de diagnostiquer facilement les problèmes de nombreuses manières (par exemple, le code, la configuration, l'infrastructure), et de les détecter plus rapidement, tout en contribuant à éviter les

dommages collatéraux. Par exemple, une application s'arrêtera au démarrage si une connexion de base de données/API ne peut pas être établie - plutôt que de le faire lors du premier appel de son API. Autre exemple : une application s'arrêtera au démarrage si la configuration (ou la permutation des autorisations utilisateur) est invalide, plutôt que de retomber sur une valeur par défaut ou de continuer quand même l'exécution.

La date, l'heure, les fuseaux horaires et les transitions heure d'été/hiver sont gérés correctement. Comprenez les concepts de temps et développez une stratégie cohérente à l'échelle de l'application. Utilisez une bibliothèque/SDK pour manipuler le temps et considérez la manipulation manuelle comme un signe d'avertissement (c'est avec raison que les bibliothèques ne prennent pas en charge certaines opérations : elles n'ont aucun sens). Veillez à ne pas rompre le « cas contraire » lors de la résolution des problèmes d'heure d'été/d'hiver (ou liés à la transition). Les applications ne doivent généralement pas inclure de tests unitaires pour les bibliothèques qu'elles utilisent, mais envisagez de faire une exception dans ce domaine pour les cas non triviaux - ne testez pas la bibliothèque, mais testez que vous l'utilisez correctement.

Les demandes/appels de données sont bien conçus. Ils sont créés aux moments appropriés, ne sont pas répétés inutilement et ne génèrent pas de volume excessif au-delà des besoins de l'application (utilisez la pagination/les limites). La mise en cache est utilisée le cas échéant et désactivée si ce n'est pas le cas. Les exemples incluent les demandes aux API et les appels de base de données.

Les états secondaires de l'interface utilisateur sont gérés. Ceux-ci incluent: chargement, erreur, pas de données et trop de données à afficher en même temps.

VIII - La gestion des erreurs

VIII-A - Général (gestion des erreurs)

Rejetez les exceptions dans le domaine de l'interface de la méthode. Cela évite les ruptures d'abstractions ou de couches d'application. Par exemple, les DAO ne devraient pas lancer d'exceptions HTTP ni propager d'exceptions JDBC. Des exceptions peuvent être appréhendées et intégrées à d'autres plus appropriées pour faciliter cela.

Consignez les exceptions ou rejetez-les - pas les deux en général. Les exceptions levées seront interceptées à un certain niveau ; si le lanceur bas niveau ne peut pas gérer l'exception, il n'est probablement pas le mieux placé pour décider de la journaliser et de le faire de manière informative avec le contexte. Les opérations généralisées de journalisation et de rejet aboutissent également à la consignation en double, car une exception remonte dans la hiérarchie des appels. Une exception à la règle est lorsque ce niveau supérieur échappe à notre contrôle (par exemple, une structure) - et ne se connecte pas, se connecte à un niveau non désiré ou n'inclut pas suffisamment de détails.

VIII-B - Contrôle de flux

Annoncez une exception pour une méthode qui ne peut pas faire ce que son nom promet. Le nom **traduit les attentes**, et l'appelant doit être informé si elles ne peuvent être satisfaites.

Évitez les erreurs de journalisation et la poursuite de l'exécution malgré tout. Une telle pratique ne constitue pas un traitement significatif des erreurs et est susceptible de causer des erreurs et des dommages ultérieurs. L'exécution dans le contexte actuel doit cesser ou utiliser un autre chemin de récupération.

VIII-C - Gestion et capture de l'information

Incluez des informations pertinentes et contextuelles dans les journaux et les messages d'exception. Ces informations facilitent le diagnostic des problèmes. Les exemples incluent les valeurs problématiques, l'état et les identificateurs. Pour les exceptions personnalisées, certaines de ces informations peuvent être rendues obligatoires dans le constructeur, contrairement à la pratique courante consistant à accepter un message de chaîne de caractères unique.

Envisagez le attraper/renvoyer pour ajouter un message plus utile et/ou des informations contextuelles. Le lanceur d'origine n'a peut-être pas eu beaucoup de contextes ou de données pour construire un message particulièrement utile. Les appelants de tels lanceurs peuvent intercepter ces exceptions et les inclure dans des exceptions plus informatives avant de les transmettre.

Journalisez les exceptions capturées dans leur intégralité, en général. La journalisation uniquement d'un message générique ou du message de l'exception interceptée supprime les informations potentiellement utiles, à savoir les messages des exceptions de cause encapsulées et la trace de la pile. Au niveau des limites du système (par exemple, lors de l'exposition d'une API), il est généralement souhaitable d'omettre (ou de consigner au bas-niveau) les détails des erreurs du client (par exemple, la validation de la demande) afin d'éviter un bruit de journalisation excessif.

VIII-D - API Web

Répondez avec un code de réponse d'erreur approprié et un organe informatif incluant plus de détails. Cela permet aux clients d'identifier rapidement la nature du problème sans avoir à consulter les journaux de service. Par exemple, une réponse peut indiquer une mauvaise demande accompagnée d'une explication de ce qui ne va pas.

Évitez de révéler (ou de rendre déductible) des informations sensibles ou des détails de mise en œuvre dans les réponses d'erreur. Les informations sensibles sont des informations que le client ne devrait pas connaître (même si elles ne sont pas affichées dans l'interface utilisateur employée), telles que l'existence de données auxquelles il n'a pas accès, ou les restrictions/limitations en place, sur leur compte. Révéler les détails de la mise en œuvre pourrait aider les attaquants de l'application ou d'autres personnes de l'organisation. De nombreux frameworks Web ont des fonctionnalités et/ou des modèles pour traiter ce problème de manière centralisée et cohérente.

VIII-E - Interface utilisateur

Afficher des informations appropriées et profitables à l'utilisateur. Le corps principal doit communiquer en termes non techniques le problème, l'état actuel de ce que faisait l'utilisateur et un moyen de remédier au problème. Des détails techniques peuvent être inclus dans une zone pouvant être révélée pour être incluse dans les rapports de bogues.

Ne demandez à l'utilisateur de réessayer que si le problème est transitoire. Réessayer après un problème de connexion peut très bien fonctionner. Réessayer avec la même entrée de formulaire non valide ne le fera certainement pas.

Voir aussi : **UI text**.

IX - Journalisation

IX-A - Général (journalisation)

Gardez à l'esprit le but de la journalisation. Les principaux sont : déterminer si l'application fonctionne correctement et diagnostiquer le problème si ce n'est pas le cas. Avoir cela à l'esprit aide à décider si nous devons journaliser et, le cas échéant, quelles informations doivent être incluses. Une journalisation informative « à la volée » devrait avoir lieu pendant l'utilisation de l'application, pour indiquer que tout va bien (comme des « voyants verts »). Lorsque les choses ne vont pas bien, apparaîtrait une journalisation « Avertissement » et « Erreurs » pour attirer l'attention sur ce problème et le détailler (comme un « voyant rouge »).

Suivez les conventions d'application pour quand, quoi et comment. Cela garantit que toutes les zones du journal de l'application sont cohérentes.

Évitez toute journalisation triviale, non pertinente ou en double. Une telle journalisation n'est que du bruit, ce qui nuit au « signal » d'une journalisation réellement importante.

Lire la sortie de journalisation pour assurer le « flux ». Il devrait être lisible de manière cohérente, comme une histoire de ce qui se passe. Essayez de le lire tout en effectuant des tâches spécifiques de l'application et pendant un test de charge.

Évitez les collaborations et dépendances de journalisation étroitement liées. Les messages de journalisation doivent pouvoir être autonomes. Ils doivent rester significatifs si d'autres appels de journal distants sont modifiés ou supprimés. C'est-à-dire que la sortie complète du journal ne doit pas être vulnérable. Évitez de faire référence à, de définir des attentes, de créer des « phrases » multi-messages en collaboration avec, ou de vous fier à une journalisation lointaine pour « finir ce que vous avez commencé ».

IX-B - Contenu (journalisation)

Rédigez **des messages clairs, concis et non ambigus.** Les messages qui suivent ces principes sont plus rapides et plus faciles à comprendre et permettent d'éviter les erreurs d'interprétation ou la confusion.

Inclure des informations pertinentes et contextuelles. Ces informations facilitent le diagnostic des problèmes. Les exemples incluent les valeurs clés, l'état et les identificateurs. Pour faciliter la recherche et permettre l'analyse par les outils d'affichage des journaux, envisagez d'utiliser un modèle tel que cle=valeur | autre=valeur.

Distinguer les valeurs du texte du message à l'aide de délimiteurs. Il peut être difficile de distinguer certains types de données du modèle de message lui-même, lorsqu'elles sont incorporées dans celui-ci. Utilisez des délimiteurs tels que des guillemets, des accolades ou des crochets pour clarifier les limites, le cas échéant.

Utilisez le contexte de diagnostic mappé (MDC - Mapped diagnostic Context) pour distinguer la journalisation de plusieurs threads[C3]**.** Il est presque impossible de comprendre ce qui se passe lorsque la journalisation de plusieurs *threads* simultanée est entrelacée. Inclure des informations de contexte telles que les identifiants utilisateur/requête dans des instructions de journal individuelles est fastidieux et répétitif. Au lieu de cela, configurez le modèle de votre enregistreur pour inclure ces informations sur chaque ligne, comme cela se fait avec l'horodatage.

IX-C - Niveaux

Utilisez les niveaux de manière appropriée et cohérente. Établissez (ou obtenez) des directives et suivez-les. Par exemple, lorsque l'application fonctionne correctement, elle ne devrait pas fournir un flot d'erreurs et d'avertissements.

Les erreurs client ne sont pas des erreurs d'application. Si un client envoie une demande non valide ou fait quelque chose d'illégal/incorrect, c'est le client qui se trompe - et non l'application qui gère la demande. La consignation d'erreurs telles que des erreurs d'application génère du bruit lorsqu'un client mal intentionné ou mal implémenté appelle notre application.

Lectures complémentaires

Voir aussi : **Error handling** .

Ressources externes :

- **The Art of Logging**, *de Colin Eberhardt ;*
- **Log Level Inflation**, *par Dustin Marx ;*
- **Verbose Logging Will Disturb Your Sleep**, *par Johannes Brodwall.*

X - Tests

X-A - Général (tests)

Les tests sont aussi du code. Les éléments des autres rubriques de ce guide s'appliquent. Ils requièrent une correction du code et sont soumis aux mêmes contrôles et règles de qualité automatisés et manuels que le code principal. Un code de test médiocre est moins fiable et peut rendre difficile la modification ou la réécriture du code principal.

Gardez le code de test à proximité des lignes de code qu'ils testent. Lire : **files**.

Mise au point et objectif par test. Les tests à large spectre sont moins clairs et rendent plus difficile l'identification de la cause des défaillances. Pointez les échecs uniquement liés au cas décrit dans le nom du test et faites les assertions requises.

X-B - Que tester

Comparez l'apport par rapport aux coûts. Les tests nécessitent des efforts pour l'écriture et la maintenance. Pensez à un morceau de code donné (même le plus petit), si vous pensez que ce que vous gagnez en exactitude et évitement de plantages futurs, en vaut la peine par rapport à la longueur et la difficulté du test. Cet élément n'est pas une excuse pour les contre-exemples qui rendent le code difficile à tester.

La couverture de code est un guide; cela n'implique ni l'adéquation ni l'inadéquation. La couverture indique quels chemins du code ont été empruntés lors de l'exécution des tests. Elle ne dit rien sur la justesse des cas de test, ni sur les affirmations appropriées. Les tests de code à couverture élevée peuvent nécessiter des améliorations, et les tests de code à couverture inférieure peuvent être parfaitement adéquats. Évitez d'écrire des tests de base pour augmenter les statistiques de couverture.

Évitez les tests qui vérifient que le code "fait ce qu'il fait". De tels tests vérifient généralement que certaines commandes sont appelées et ne testent aucune logique de l'unité testée (si l'unité en a même une). Ils sont fragiles, une surcharge pour la recompilation, du parasitage et n'apportent rien.

Évitez tester les objets simulés. Habituellement accidentels, de tels tests n'affirment rien sur notre code et à la place, par exemple, vérifient que le bouchon fonctionne.

Inclure des tests pour les cas extrêmes et les mauvais choix. Nous devons avoir la certitude que notre code fonctionne dans tous les cas, pas seulement dans les cas « normaux » qui peuvent se produire la plupart du temps. Les exemples incluent les erreurs, les délais d'expiration (timeout), les données non valides, les données inexistantes et les valeurs limites.

X-C - Structure et isolement

Évitez les états partagés entre les tests. L'état partagé rompt l'isolement du test, ce qui entraîne généralement un désordre impliquant de faux succès ou de faux échecs selon les tests exécutés ensemble et selon l' ordre. Les exemples incluent les variables partagées (généralement pour éviter les redéclarations), les instances de test et les simulations réutilisées. Recommencez toujours de zéro ou, en cas de simulation, réinitialisez-les.

Évitez la logique d'installation partagée dans les hooks[C5] **.** Cela signifie un ou plusieurs blocs de code que le framework de test exécute avant une collection de tests individuels. Pris à l'extrême, il implique des groupes de tests imbriqués, avec un bloc d'installation partagé à chaque niveau. Il encourage l'utilisation d'état partagé, permet leur utilisation accidentelle, rend la configuration difficile à suivre, la rend difficile pour des tests individuels et rend difficile la modification de tests et l'ajout de nouveaux tests.

Logique de configuration commune abstraite en fonctions utilitaires. Cela évite à peu près tous les problèmes liés à l'utilisation de points d'ancrage. Extrayez la logique de configuration partagée dans les fonctions utilitaires et utilisez-les directement et au besoin à partir de tests individuels. Les fonctions renvoient des objets prêts à être utilisés dans le test et acceptent souvent des paramètres (données ou options) permettant de personnaliser la configuration pour des tests individuels. Ces derniers permettent de voir facilement la différence de configuration entre chaque test.

Évitez de vous fier à l'heure et au fuseau horaire actuels du système. De tels tests ne testent pas la même chose à chaque fois qu'ils sont exécutés et sont susceptibles de poser des problèmes à l'avenir, ou lorsqu'ils sont exécutés par des personnes ou des serveurs CI dans des différents lieux géographiques. Injectez plutôt l'heure actuelle dans votre code, ce qui permet d'injecter des heures fixes spécifiques dans les tests.

X-D - Assertions

Utilisez des assertions strictes. Celles-ci renforcent le test et le rendent plus susceptible de détecter les régressions futures. Les exemples incluent l'égalité stricte et les erreurs/exceptions de types et de messages spécifiques.

Utilisez les assertions les plus appropriées. Les frameworks de test incluent des assertions autres que l'égalité. Ils clarifient le code de test et produisent de meilleurs messages d'échec.

Assurez-vous que les assertions dans le code asynchrone sont réellement exécutées. Utilisez des techniques spécifiques au framework de test pour vous assurer qu'elles sont exécutées. Certains frameworks fournissent des **assertions** synchrones qui affirment que des assertions ultérieures sont bien faites.

X-E - Autre

Simulez des dépendances immédiates plutôt que transitives. Cela permet au test de rester indépendant et concentré sur une seule unité.

Évitez de réutiliser les données d'exécution simulées en tant que données de test simulées. Ces données sont conçues dans un but différent (démonstration, généralement) et doivent rester modifiables sans affecter les tests.

Évitez d'attendre le temps écoulé. Les tests qui attendent de cette façon le code asynchrone et les minuteries sont lents à exécuter ; cela devient un gros problème quand vous en avez des centaines. Au lieu de cela, utilisez la fonction de gestion du temps de votre framework de test qui permet d'avancer immédiatement le temps.

XI - Les fichiers

Bien nommés et selon les conventions. Reportez-vous à **nommer les choses**. Suivez les conventions du projet (suffixes, par exemple) et les conventions typographiques.

Dans un dossier approprié. En évaluant le but et le domaine fonctionnel.

Organisé par séparation fonctionnelle plutôt que par type de fichier. Cela signifie par domaine fonctionnel d'application/module ou composant - et non par type de fichier (par exemple, contrôleurs, styles, DAO). Cette structuration permet de garder les codes associés ensembles, ce qui facilite la recherche et la navigation.

Les fichiers de test doivent être proches du code qu'ils testent. La convention Java standard de séparer src et les dossiers test est le contraire de cela. Les avoir à proximité les rend plus faciles à trouver, permet de les garder à l'esprit et rend plus facile de repérer ceux qui manquent. En pratique, cela signifie les placer dans un dossier proche des fichiers qu'ils testent.

XII - Mise en forme

Presque toutes les résolutions des problèmes de mise en forme doivent être automatisées à l'aide d'outils tels que ESLint ou Checkstyle. Les erreurs devraient empêcher la compilation, de peur qu'elles ne s'accumulent et ne rendent les nouvelles difficiles à remarquer.

Les « builds » de développement local (surveillance de fichiers) ne doivent pas échouer en cas de violation, sous peine de devenir un inconvénient gênant. Les paramètres Editor/IDE correspondant aux exigences doivent être enregistrés **et partagés, par exemple à l'aide de fichiers EditorConfig. Pensez également à exécuter les vérifications automatisées à l'aide d'un « *pre-commit hook* » de code source.** *Ndt : u*n hook (littéralement crochet) permet de lancer un programme personnalisé au moment précis où le programme principal a la tâche de l'exécuter. Un pré-commit Hook est exécuté avant la transaction de commit.

Ce qui suit ne comprend que les éléments clés pour bien faire les choses ; ce n'est pas un ensemble complet de paramètres pour les outils automatisés.

Utilisez des fins de ligne cohérentes. Les outils de contrôle de source considèrent par ailleurs que les lignes identiques avec des fins de lignes différentes sont différentes. Le mélange au sein d'un projet provoquera un bruit inutile (rendant le code plus difficile à contrôler) et fusionnera les problèmes, en particulier lorsque le code est déplacé pendant la recompilation.

Utilisez une indentation cohérente. Une autre cause de "fausses différences" : le mélange des styles causera les problèmes susmentionnés. (Ndt : l'indentation consiste en l'ajout de tabulations ou d'espaces dans un fichier, pour une meilleure lecture et compréhension du code.)

Évitez de laisser des espaces répétés ou des séries de tirets. Ils sont une source de « fausses différences » et il n'y a aucune raison de les avoir.

Terminer les fichiers avec un saut de ligne. Cela évite que la ligne finale existante soit considérée comme modifiée lorsque vous ajoutez une nouvelle ligne après celle-ci.

Utilisez judicieusement des lignes vierges pour séparer les fonctions et les groupes de déclaration. Les lignes vides créent une séparation visuelle entre les parties logiques, ce qui facilite la lecture du code. Le code dense est plus difficile à lire. À utiliser après un groupe de déclarations de variables, autour d'une instruction if else, autour des parties logiques d'une méthode et avant une instruction return.

Évitez toutes modifications inutiles ou uniques de la mise en forme, de l'indentation, etc. Elles nuisent au but du travail à exécuter, provoquent du bruit de fond (et les problèmes associés susmentionnés) et rendent l'historique des modifications ligne par ligne moins utile. Ce problème se pose généralement dans les bases de code où ces instructions ne sont pas respectées (ou appliquées), en particulier lorsque les fonctionnalités de formatage automatique éditeur/IDE sont activées.

En cas de doute, regardez le code existant. Le code existant dans le projet illustrera les conventions établies.

Évitez les marques personnelles. En lisant le code, il ne devrait pas être possible de dire qui l'a écrit. En ce qui concerne le formatage, cela ne devrait pas être un problème si les outils automatisés sont configurés avec des règles adéquates.

XIII - Les dépendances

Déterminez s'il est vraiment nécessaire d'utiliser une bibliothèque/un outil. Si la tâche à accomplir peut être effectuée avec quelques lignes de code simple, il ne vaut peut-être pas la peine d'ajouter une nouvelle dépendance. Familiarisez-vous avec la dernière plate-forme/SDK et les bibliothèques déjà présentes. Ce dont vous avez besoin est peut-être déjà disponible.

Choisissez les bibliothèques/outils avec soin. Tenez compte de facteurs tels que la qualité, la popularité, la documentation, la « vitalité », la dépréciation et la force de la communauté. Examinez les alternatives. En fonction de la nature d'une bibliothèque, son utilisation peut être localisée dans une zone d'un projet - ou généralisée et liée au code de l'application. Ces derniers nécessitent un soin particulier.

Les bibliothèques/outils/structures internes à l'entreprise ne bénéficient pas d'un traitement de faveur. Celles-ci sont tout aussi sensibles aux problèmes que les logiciels libres (peut-être même davantage). Elles peuvent, par exemple, être de mauvaise qualité, sans documentation ou abandonnées. Les bibliothèques internes ont rarement une communauté et ne sont pas soumises à la sélection des consommateurs, contrairement aux bibliothèques libres.

Assurez-vous que la licence est compatible avec votre utilisation. Certaines licences ne sont pas compatibles avec une utilisation commerciale. Le site web **Choose a Licence** contient un résumé pratique des licences les plus populaires.

Utilisez les dernières versions non-bêta, en général. Celles-ci doivent être stables, inclure les dernières fonctionnalités et corrections de bogues, sont les plus susceptibles de recevoir les corrections de bogues et disposent d'une documentation facilement accessible. Des versions majeures des dépendances nouvellement publiées qui interagissent étroitement avec les autres, ou qui ont un écosystème de plug-ins, peuvent valoir la peine d'être différées jusqu'à ce que l'environnement complet rattrape son retard et se stabilise.

Vérifiez que les bibliothèques ne plantent pas votre application. Cela s'applique si votre application est livrée aux utilisateurs) à travers Internet, sur le Web ou sous forme d'application mobile. Utilisez les **outils** de l'écosystème pour analyser les applications.

Suivez les modèles et les idiomes établis pour une dépendance donnée. Les principales bibliothèques, les frameworks et les outils ont le plus souvent une manière établie de les utiliser efficacement. Cela peut être une combinaison entre manière officielle et celle fournie par une communauté. Familiarisez-vous avec cela pour vous aider à en tirer le meilleur parti et éviter d'être à contre-courant. Lisez plus que l'introduction.

Évitez de réimplémenter les fonctionnalités de la bibliothèque/structure en raison de votre ignorance. Essayez de vous familiariser avec les caractéristiques principales de ce qui est déjà disponible. Il est impossible de tout savoir et de rester à jour, vous pouvez donc suivre simplement les règles empiriques suivantes pour savoir si quelque chose est déjà disponible :

1. Suis-je susceptible d'être la première personne utilisant la bibliothèque-X à avoir besoin de cette fonctionnalité ?

2. Si j'écrivais ma propre bibliothèque-X, est-ce que j'inclurais cette fonctionnalité ?

XIV - Spécificités techniques

XIV-A - Spécificité techniques : CSS

XIV-A-1 - Général (CSS)

CSS est aussi du code. Les éléments des autres rubriques de ce guide s'appliquent. Il nécessite un contrôle du code et est soumis à des contrôles de qualité automatisés et manuels, comme tout autre code d'application. Un mauvais CSS est difficile à modifier et à étendre, et peut rendre difficile le changement ou le remaniement de l'application.

Gardez le CSS proche du code du composant dont il définit le style. Lire : **fichiers**.

Résolvez les incohérences de conception. Les apparences des interfaces utilisateur sont parfois incohérentes dans leur style, par exemple les espacements ou les couleurs. Évitez de les intégrer au code de style de l'application ; non seulement les incohérences nuisent au produit, mais elles ont aussi tendance à conduire à un mauvais CSS. Certaines incohérences sont triviales, tandis que d'autres nécessitent une discussion avec les concepteurs.

XIV-A-2 - Règles

Utilisez des noms de classe sémantique. Décrivez le but de la règle (par exemple, product(produit)), pas son contenu (par exemple, redBackground(fond rouge)). Il existe cependant un petit nombre d'exceptions où les classes par présentation/utilité peuvent être utiles.

Placez les règles connexes proches dans le fichier. Cela permet aux responsables de la maintenance de les trouver plus facilement et de les prendre en compte lors de la modification.

Un sélecteur par ligne. Plusieurs par ligne nuisent à la lisibilité.

Évitez de coupler étroitement les sélecteurs à la structure DOM. Cela rend les styles et DOM difficiles à modifier sans modifier les deux. Faites attention aux sélecteurs qui spécifient de nombreuses classes (reflétant l'arborescence DOM) ou utilisant le « > » « Child combinator ».

Placez les déclarations dans un ordre réfléchi. Les règles contenant des déclarations dans un ordre aléatoire (ou par ordre chronologique) sont plus difficiles à lire et à visualiser rapidement mentalement. Elles sont également sujettes à plusieurs déclarations contradictoires (par exemple, remplacées accidentellement). Un ordre réfléchi où les déclarations les plus importantes viennent en premier, et où les déclarations associées sont regroupées, évite ces problèmes. Utilisez un « linter » pour appliquer la commande. Les commandes établies et populaires (pour lesquelles les "linters" auront des préréglages) incluent **Concentric**, **RECESS** et **SMACSS**.

Utilisez des variables pour les couleurs standards, les espacements, etc. Cela évite les répétitions et contribue à assurer la cohérence.

XIV-A-3 - Caractéristiques et techniques

Évitez les dimensions fixes lorsque cela ne convient pas. Les dimensions fixes dissocient les éléments des capacités de dimensionnement du navigateur. Il en résulte une mise en page qui ne s'adapte pas à la taille du navigateur et qui nécessite des ajustements manuels lorsque le contenu est modifié. Ils ne sont pas souvent nécessaires avec les fonctionnalités de mise en page modernes.

Évitez le positionnement absolu lorsque cela n'est pas approprié. Le positionnement absolu extrait les éléments du flux de présentation en les positionnant à des coordonnées particulières. Celles-ci nécessitent un ajustement manuel pour s'ajuster aux éléments environnants et ont tendance à proliférer par nécessité une fois introduites dans un schéma. Il n'est donc pas souvent nécessaire avec les fonctionnalités de disposition modernes.

Évitez d'utiliser des positions flottantes pour positionner les éléments. Les positions flottantes ont pour but de permettre au contenu de circuler autour d'un élément. Cependant, elles ont été utilisées historiquement pour des raisons pour lesquelles ils n'étaient pas conçus, pour réaliser des mises en page qui n'étaient pas possibles autrement en CSS à l'époque. Les fonctionnalités de mise en page modernes telles que *flexbox* sont de bien meilleures alternatives, il est donc rarement nécessaire ou approprié d'utiliser des positions flottantes de nos jours.

Évitez d'utiliser text-align pour aligner des éléments. Il s'agit d'une propriété héritée. Par conséquent, les enfants d'un élément aligné à l'aide de cette dernière doivent neutraliser l'alignement de texte non souhaité en définissant leur propre text-align. Utilisez *flexbox* pour aligner les éléments.

Évitez d'utiliser "!important". Ajouter ceci à une déclaration pour forcer la victoire d'un concours de spécificité de sélecteur doit être évité. Il sort des règles normales de spécificité (être au niveau de la déclaration) et ne peut pas être outrepassé sans en utiliser un autre "!important"- ce qui tend à entraîner une prolifération de son utilisation.

Utilisez la marge ou le remplissage en fonction de la situation. La marge crée un espace autour d'un élément. Utilisez-la pour séparer un élément des autres. Les verticales adjacentes se touchent. Le remplissage crée un espace dans un élément (entre la bordure et le contenu). Utilisez-le pour séparer le contenu d'un élément de ses bords.

Réfléchissez à ce que vous devez accomplir et réfléchissez à nouveau si vous vous trouvez obligé de vous répéter afin de maintenir des espacements constants.

Évitez d'utiliser des techniques obsolètes. Les améliorations apportées aux fonctionnalités CSS continuent d'apporter de nouvelles et meilleures alternatives à de nombreuses techniques habituelles, et les navigateurs les supportent largement, utilisez-les. Avec l'avènement de la "flexbox" en particulier, de nombreux "hacks" et solutions de contournement ne sont plus nécessaires.

Faites attention aux fonctionnalités supportées dans différents navigateurs. Utilisez soit uniquement celles prises en charge par tous les navigateurs que votre application supporte ou effectuez une amélioration progressive. Reportez-vous aux tableaux de référence **Can I use** et aux tables de compatibilité de navigateur dans les **MDN documentation pages**. Utilisez un **linter** pour automatiser les vérifications de compatibilité.

XIV-A-4 - Isolement

Nommez les sélecteurs d'espace pour éviter d'affecter involontairement des éléments non souhaités. Les noms de classe et les sélecteurs étant définis et opérant dans un espace global (la page), une stratégie est nécessaire pour éviter de manière cohérente et simple les règles affectant des éléments indésirables. Les exemples incluent l'utilisation de la classe racine d'un composant comme préfixe de tous les sélecteurs, règles BEM et CSS de ses règles de style. Même avec une telle stratégie, un soin particulier est nécessaire pour éviter que les composants parents n'affectent involontairement d'autres composants imbriqués dans ceux-ci.

Évitez d'avoir accès aux composants imbriqués pour ajouter/remplacer leur style. Ces styles sont couplés à la mise en œuvre interne du composant imbriqué (styles DOM et propres) et il est peu probable qu'ils soient pris en compte si cela change- ce qui les rend fragiles et difficiles à maintenir. Cela revient à utiliser/modifier un état privé dans la programmation orientée objet. Au lieu de cela, le composant imbriqué prend en charge les «classes d'option/mode» sur sa racine, que les composants parents peuvent appliquer sur autorisation.

Rendez les composants insensibles concernant quand/comment ils sont utilisés. Les styles affectant les extérieurs de l'élément d'un composant, tels que le positionnement/la disposition/l'espacement, doivent être laissés à la spécification du code parent utilisant le composant.

Pour en savoir plus sur l'écriture de CSS évolutive, je recommande ce guide « 8 simple rules for a robust, scalable CSS architecture »de Jarno Rantanen.

XIV-A-5 - Commentaires (CSS)

Documentez les solutions de contournement des bogues de navigateur. Cela met en évidence et explique les styles inhabituels ou déroutants, pour le bénéfice de votre futur, à vous et aux autres. Pour les problèmes connus, cela peut être fait de manière concise avec une phrase courte et un lien vers, par exemple, **flexbugs**.

Expliquez le choix de «valeurs magiques» non évidentes telles que les largeurs et les espacements. La documentation de ces derniers permet d'apporter des modifications futures avec plus de confiance et d'éviter les régressions. Les largeurs choisies pour s'adapter aux valeurs les plus longues attendues en sont un exemple.

Voir aussi : **Comments**.

XIV-B - Spécificités techniques : HTML

Évitez d'utiliser l'attribut *id* en général. Il n'est pas valide d'utiliser la même valeur d'identifiant plusieurs fois sur une page. Dans une application comportant plusieurs composants, dont certains ayant plusieurs instances, il est facile de violer cette exigence. Cet attribut est toutefois utile pour les liens qui font défiler à des emplacements spécifiques de la page.

Utilisez le balisage sémantique. Il y a beaucoup d'autres éléments que et qui donnent un sens. Utilisez-les le cas échéant, par exemple : en-têtes, sections, formulaires et paragraphes.

Évitez d'utiliser des classes pour autre chose que CSS. Les sélecteurs utilisant des classes sont souvent employés pour sélectionner par programmation des éléments particuliers, par exemple dans le code JavaScript ou les tests d'intégration. Lors de la modification du balisage et du code CSS, il est facile d'oublier ces utilisations moins visibles, ce qui entraîne des ruptures. Utilisez des attributs personnalisés dédiés (par exemple, data-id, test-id) à la place.

Gardez le code et les expressions complexes en dehors des modèles HTML. Les modèles ont souvent besoin de recourir à de tels éléments pour le rendu conditionnel et la liaison des données. Le fait de mélanger des éléments complexes avec le modèle HTML rend à la fois eux-mêmes et le balisage plus difficiles à suivre. Extrayez à la place des variables et/ou des méthodes pour elles.

XIV-C - Spécificités techniques : Java

C'est un sujet plutôt court, car j'ai principalement travaillé avec d'autres technologies alors que je rassemblais des notes pour ce guide (2018).

Je considère que la lecture d'Effective Java est obligatoire pour tous les développeurs Java et recommande vivement Java Concurrency in Practice et Java 8 in Action.

Utilisez ==and .equals()comme approprié. Utilisez ce dernier lorsque vous en avez besoin, mais pas lorsque le précédent est tout ce dont vous avez besoin.

Évitez d'utiliser inutilement les types primitifs enveloppés (« boxed primitive »). Les vrais types primitifs sont plus simples, évitent les erreurs d'"auto (dé) boxe" (« Auto-Boxing et Auto-unboxing ») et ne peuvent pas être null.

Utilisez valueOf pour obtenir les constantes BigDecimal. La constructeur double-accepting en construira une qui reflète avec précision la valeur donnée, pour que new BigDecimal(0.1)n'aie pas la valeur 0.1.

Pensez à l'échelle lorsque vous comparez l'égalité BigDecimal. La méthode equals()considère l'échelle, alors que compareTo()non. 1 est seulement égal à 1,0 lorsque vous utilisez ce dernier.

Minimiser la visibilité. Tout ne doit pas nécessairement être public : préférez la visibilité par défaut lorsque la confidentialité est trop restrictive. Alors que la visibilité des variables est souvent oubliée et minimisée, les classes et leurs méthodes sont souvent laissées inutilement publiques - probablement en raison de réglages par défaut de l'EDI.

Utilisez des hiérarchies d'exceptions. Cela permet aux appelants concernés de gérer une sous-exception précise en particulier, tout en permettant aux appelants qui ne la détectent pas facilement de prendre celle qui se trouve à la racine de la hiérarchie.

XIV-D - Spécificités techniques : JavaScript

Évitez de vous fier à la différence entre null et undefined. Cela tend à créer un code fragile. Utilisez l'égalité non forcée (==) plutôt que null dans les contrôles d'égalité pour l'un ou l'autre. Évitez d'utiliser les deux pour indiquer un type différent de situation « aucune valeur », comme « champ inconnu » par opposition à « aucune valeur pour aujourd'hui ».

Faites attention au cas zéro avec des contrôles vrai/faux. Si un nombre est présent, il doit souvent être traité de la même manière que tout autre nombre - et non de la même manière que d'autres valeurs fausses.

Arithmétique en virgule flottante, avec ses pièges habituels. Pensez à en faire le moins possible côté client et/ou à utiliser une **bibliothèque de nombres pour JavaScript**.

Méfiez-vous du support des fonctionnalités dans différents navigateurs. Utilisez uniquement ceux pris en charge par tous les navigateurs eux-mêmes pris en charge par l'application ou effectuez une amélioration progressive. Reportez-vous à la référence **Can I use**, les tableaux de compatibilité de navigateur dans les **MDN documentation pages** et les **ECMAScript compatibility tables**. Utilisez un **plugin linter** pour automatiser les vérifications de compatibilité. Certaines fonctionnalités peuvent être « transcompilées » par votre configuration de compilation ou « pluri-remplies », ce qui les rend disponibles dans les navigateurs plus anciens.

Évitez d'utiliser setTimeout()ou similaire d'une manière fragile ou incomprise. Il arrive qu'il soit tentant d'implémenter/de corriger des scénarios en exécutant un morceau de code après un délai soigneusement choisi (généralement par essais et erreurs). La raison pour laquelle de telles approches fonctionnent (au moins de temps en temps) est rarement connue de l'auteur et conduit à un comportement fragile et imprévisible, en particulier lorsque le délai est non nul.

Méfiez-vous des méthodes d'instance qui mutent lorsque cela n'est pas attendu. Celles-ci incluent le rangement in-situ reverse()et de nombreuses méthodes Moment.js (envisagez sérieusement une alternative telle que date-fns ou Luxon).

Array sort()convertit les éléments en chaînes et les trie par défaut dans l'ordre alphabétique. Pour tout sauf les chaînes de caractères, ce n'est pas ce qui est recherché - une fonction de comparaison doit être utilisée.

String replace()remplace uniquement la première occurrence lors de l'utilisation d'une chaîne de caractères en tant que modèle. Le remplacement global nécessite l'utilisation d'une expression régulière.

Évitez les contrôles de type "canard" dans la mesure du possible. Ils sont facilement faussés si des changements sont apportés au "canard" et ne traduisent pas bien l'intention.

Évitez de provoquer la mise en page à travers des lectures et écritures entrelacées. Groupez lectures et écritures dans les propriétés qui causent la mise en page/redistribution autant que possible.

Évitez le travail excessif en réponse aux actions de l'utilisateur. Cela inclut le calcul, les mises à jour d'écran et les demandes adressées au serveur. Envisagez de limiter les entrées au clavier et certains types d'événements de souris.

Évitez d'utiliser des techniques obsolètes. Les progrès linguistiques continuent de proposer de nouvelles et meilleures alternatives à de nombreuses techniques héritées du passé. Utilisez-les. Les exemples incluent async/wait et les modèles de chaînes de caractères.

XV - Commentaires

Évitez les commentaires inutiles qui n'ajoutent aucune valeur. Si quelque chose est clair à la lecture du code, un commentaire ajoute uniquement du bruit.

Déterminez si le code pourrait être amélioré de sorte que le commentaire ne soit plus nécessaire. Les commentaires qui expliquent ce que fait le code, et parfois pourquoi, peuvent souvent être rendus inutiles en améliorant la dénomination, la réécriture (par exemple, l'extraction d'une fonction) ou l'introduction de variables explicatives.

Déterminez si un test unitaire améliorerait la communication. Des tests unitaires bien construits et nommés peuvent expliquer le raisonnement derrière le code, ainsi que démontrer et vérifier son comportement dans différents cas.

Expliquez le raisonnement quand ce n'est pas clair dans le code. Anticipez ce qui risque de laisser les futurs mainteneurs perplexes à propos du code. Les exemples incluent la gestion des cas extrêmes, les contraintes à contourner et les optimisations de performances.

Attirez l'attention sur les surprises et les «pièges». Si quelque chose n'est pas intuitif ou vous a interpellé, il peut être utile de le noter pour aider les autres. Les exemples incluent la « logique » apparemment illogique des entreprises et le comportement surprenant de la bibliothèque de code.

Expliquez le choix de « valeurs magiques » particulières. Cela inclut les paramètres de frawework/serveur, les délais, les limites, la taille des lots/pools, les priorités, la configuration du cache et les commandes. Nous sommes habitués à extraire de telles valeurs de notre code dans des constantes ou des configurations, mais les raisons du choix de la valeur réelle sont souvent omises. Dans certains cas, cela s'applique après des efforts importants sur des activités telles que le test de charge afin de choisir les valeurs appropriées. La documentation de ces derniers permet d'apporter des modifications futures avec plus de confiance.

Mettez en surbrillance les solutions de contournement des bogues, en vous connectant à un rapport de problème. Cela leur permet d'être facilement identifiés et supprimés ultérieurement lorsque le bogue sous-jacent (dans une bibliothèque, par exemple) a été corrigé. Voir les **corrections de bogues**.

Documentez les relations entre les parties distantes et déconnectées du code. Il est assez rare que ceux-ci ne puissent pas être explicités par le code. Ce sont souvent des choses qui sont d'une certaine manière parce que quelque chose de très lointain (qui pourrait même être dans un système en aval) est d'une certaine manière - telle que des dépendances ou des comportements correspondants. Changer l'un pourrait briser l'autre de manière surprenante ou introduire une incohérence dans l'expérience utilisateur.

Écrivez clairement, de manière concise et sans ambiguïté. Les commentaires qui suivent ces principes sont plus rapides et plus faciles à comprendre, et permettent d'éviter les erreurs d'interprétation ou les confusions. Le processus d'écriture peut souvent déclencher des idées ou des solutions aux problèmes, de la même manière que simplement expliquer un problème à quelqu'un peut vous aider à trouver une solution. La réécriture n'est pas seulement pour le code ; après avoir écrit un commentaire, lisez-le et réfléchissez s'il pourrait être amélioré.

Assurez-vous que les commentaires sont **synchronisés et corrects par rapport à la version actuelle du code.** Les commentaires périmés ou obsolètes peuvent semer la confusion.

Suivez la **stratégie** du projet **pour gérer les commentaires** TODO. L'accumulation de tels commentaires dans le code indique souvent une accumulation de retard technique et de travail nécessaire. Ces tâches restent invisibles pour le suivi du travail sur les fonctionnalités/bogues du projet, ce qui les rend susceptibles d'être négligées et oubliées, avec diverses conséquences. L'une de ces stratégies consiste à demander à tous les TODO de référencer un ticket de suivi des problèmes avant qu'une demande d'extraction ne puisse être fusionnée.

XVI - Enregistrement de transaction (Commits)

XVI-A - Contenu (enregistrements de transaction)

Un travail logique par enregistrement de transactions. Séparez chaque fonctionnalité, bogue et réécriture des autres. Cela crée une histoire plus utile et encourage également une approche organisée pour la réalisation des travaux. Si vous devez corriger un bogue pendant que vous travaillez sur une fonctionnalité, envisagez de mettre temporairement de côté vos modifications de fonctionnalité à l'aide d'une fonctionnalité de contrôle de source telle que **Git's stash**.

Préférez des enregistrements de transactions plus petits et plus réguliers. Les enregistrements de transaction trop importants sont plus difficiles à comprendre et à réviser efficacement. Enregistrez de plus gros travaux à mesure que vous atteignez de petits enregistrements.

Séparez la réécriture si possible. La réécriture a tendance à être plus difficile (et donc moins efficace) à analyser, en incluant souvent de nombreux changements et modifications du code existant. Le mélanger avec le travail sur les fonctionnalités/bogues dans un enregistrement de transaction le rend encore plus difficile. Lorsqu'une tâche commence par une réécriture initiale non triviale, envisagez de faire une demande d'extraction distincte plus tôt pour

obtenir les modifications apportées à la branche principale plus tôt. Cela permet d'éviter les conflits avec le travail en cours des autres contributeurs.

Supprimez le code existant rendu inutilisé par vos modifications. Le code en cours de suppression peut avoir été le seul utilisateur d'autres parties du code. Le code désormais inutilisé peut se trouver dans le même fichier, ailleurs dans la base de code, ou tout au long de la pile, de l'interface utilisateur à la base de données.

Adressez ou supprimez les commentaires TODO pertinents existants et nouveaux. Ceux qui existent déjà peuvent renvoyer aux travaux achevés. Les nouveaux peuvent être temporaires et doivent être supprimés ou faire référence à des tickets de suivi pour des travaux ultérieurs.

Évitez d'inclure des modifications isolées et inexpliquées. Une extension du premier élément. De tels changements sont souvent des problèmes de **formatage** ou des erreurs commises lors de la résolution de conflits de fusion/« rebasage ».

Évitez d'inclure le code déchet inutilisé. Un tel code provient souvent d'expérimentations, d'essais de différentes approches ou d'essais et erreurs. Cela n'aura peut-être pas d'effet, voire ne fonctionnera pas du tout. Le CSS, en particulier, a tendance à être très sensibles à cela.

Évitez d'inclure accidentellement des fichiers indésirables tels que la configuration personnelle et les fichiers journaux. Utilisez la fonction **ignore-file** de votre outil de contrôle de code source pour empêcher l'inclusion accidentelle de tels fichiers. Enregistrez le fichier "ignore". Une configuration qui doit être cohérente dans l'ensemble de l'équipe doit cependant être enregistrée.

XVI-B - Messages

Résumez le changement dans les ~ 70 premiers caractères. Cela permet une compréhension immédiate lors de la lecture du journal. De nombreux outils de contrôle de source cachent le reste du message par défaut.

Inclure le « pourquoi » ainsi que le « quoi ». Expliquez brièvement ce qui est fait. Ceci est dans le contexte de l'enregistrement de transaction - il doit être plus précis que juste le titre d'une grande fonctionnalité. Gardez à l'esprit que le détail du « quoi » se trouve dans le diff. Cependant, le « pourquoi » n'est pas toujours apparent ou clair, il faut donc l'inclure dans le message. Pour des réécritures ou des modifications plus complexes, une brève description du « quoi » peut être utile.

Référencez les ID de suivi des problèmes pertinents, conformément aux conventions du projet. Ces identifiants permettent de trouver facilement des informations supplémentaires sur un changement. Le plus souvent, cela concerne le problème sur lequel on travaille actuellement, mais il pourrait aussi s'agir d'autres problèmes utiles et pertinents. Le premier cas peut être automatisé à l'aide d'un crochet de message de validation de détection de branche, tel que **celui-ci**.

Pour en savoir plus sur l'importance des bons messages de validation, je recommande la section d'introduction, en particulier de cet article « How to Write a Git Commit Message »de Chris Beams.

XVI-C - Gestion de versions

Préserver l'historique de ligne. Les outils de contrôle de code source peuvent afficher une vue ligne par ligne de la dernière modification apportée à chaque ligne (souvent appelée « annotation » ou « blâme »). Valider les modifications, puis les gérer plus tard pour remettre le code tel qu'il était, rend cette fonctionnalité moins utile. La modification de tels enregistrements, ou la combinaison (« l'écrasement ») de groupes d'enregistrements présentant un taux de changement important, peut aider dans de telles situations.

Regardez ce qui est enregistré. Il est trop facile d'ajouter toutes les modifications présentes dans votre dossier de travail, prêt pour une validation rapide. Cependant, en ajoutant chaque fichier individuellement, en examinant rapidement les modifications apportées au fur et à mesure, vous aurez l'occasion de constater tout ce qui ne va pas.

XVII - Pull request

Le processus de pull request **et de révision de code est très spécifique aux circonstances et préférences de chaque équipe. Les points ici reflètent inévitablement mes expériences, mais beaucoup d'entre eux doivent être suffisamment généraux/adaptables pour être pertinents pour d'autres équipes. Faites ce qui fonctionne pour votre équipe afin de tirer le meilleur parti du temps passé par les auteurs et les relecteurs.**

Pour en savoir plus sur les pull request **et la révision de code - y compris la culture, la conduite, les pratiques et les outils - commencez par the code review awesome list.**

XVII-A - Général (pull request)

Efforcez-vous de faire des petits pull request. Les pull request importants sont plus difficiles à analyser, ce qui engendre des commentaires plus médiocres, des problèmes inaperçus, une « révision rapide »(Ndt :contrôle superficiel) et une lenteur des progrès en vue de l'approbation. Si vous ne parvenez pas à fractionner des éléments de travail pour des fonctions volumineuses (en raison de processus ou de considérations politiques), envisagez une technique incrémentielle consistant en une série de demandes plus petites (par exemple, 200-400 lignes) dans une "branche collecteur" aboutissant à une seule grosse pull request de cette branche au tronc principal.

Efforcez-vous de faire des pull request de courte durée. Tout travail en cours ou en révision sur une branche ne fait pas encore partie du fichier maître/tronc sur lequel les autres basent leur travail. Cela peut entraîner des conflits qui prennent beaucoup de temps à résoudre et sont générateurs d'erreurs. Être ouvert pendant une longue période augmente les chances qu'une pull request le soit encore plus longtemps, car elle nécessite des mises à jour après la fusion préalable d'autres pull request (impliquant possiblement la résolution d'un conflit). Envisagez de définir des règles de base générales et de type « accords de niveau de service » pour promouvoir des relations publiques de courte durée, par exemple : passez en revue les réponses dans les **n** heures qui suivent leur soumission et passez en revue les réponses ou adressez les commentaires avant de commencer un nouveau travail. Faire de petites demandes pull request les aide à rester éphémères.

Automatiser les choses fastidieuses. Configurez la construction pour exiger le contrôle du style de code et l'analyse statique. Ces problèmes sont plus rapides, moins coûteux et plus systématiquement détectés par les outils que par les utilisateurs.

Définissez les attentes dès le départ. Facilitez la tâche des contributeurs pour soumettre des pull request qui seront approuvées dès la première fois, ou du moins sans nécessiter de modifications majeures. Les éléments qui aident à cela incluent des exigences claires, une compréhension commune de l'architecture d'application et des modèles/pratiques de programmation, et le partage d'idées de conception avant le début de la mise en œuvre.

XVII-B - Avant la soumission

La compilation réussit. Cela inclut les tests de fonctionnement et les contrôles automatisés de la qualité du code.

La branche est à jour avec la branche cible. C'est le seul moyen de s'assurer que les modifications à fusionner sont compatibles et intégrées à la dernière version de la branche cible. Pour ce faire, vous fusionnez lcelle-ci dans la branche source (ou redéfinissez la source sur la cible).

Contrôlez votre code. Lisez les exigences et passez en revue votre propre code. C'est l'occasion de repérer tout problème ou élément manquant et de le résoudre immédiatement, ce qui augmente les chances d'obtenir une première approbation.

Remplissez la liste de contrôle de pré-soumission. Les listes de contrôle aident tout le monde à se rappeler des tâches à accomplir. Ce qui doit figurer sur la liste de contrôle de pré-soumission dépend de l'équipe et du projet, et devrait évoluer avec le temps - par exemple, ce qui est toujours considéré comme un retour d'information. Pour éviter de diluer son importance, il ne devrait pas être trop long ou inclure des choses triviales. Si le dépôt prend en charge les modèles pour les descriptions de pull request, créez-en un contenant la liste de contrôle, sinon, un marque-page scripté permettant de renseigner rapidement le champ de description de la demande d'extraction, peut aider.

Mettez à jour le ticket de suivi des problèmes (si applicable). S'il y a eu des clarifications, des corrections ou des modifications à l'exigence apportées par d'autres moyens (courrier électronique, messagerie instantanée, en personne), documentez-les. Cela permet de s'assurer que tout le monde (réviseurs, testeurs, propriétaire du produit) utilise les informations correctes au fur et à mesure que la fonctionnalité/le correctif est diffusé.

Ajoutez « des idées/conseils d'initiés » pour les testeurs. Selon votre expérience de la mise en œuvre/de la correction de l'exigence/du bogue, il s'agit d'informations utiles qui, selon vous, pourraient les aider, sans que les modifications apportées au code ne soient connues. Ajoutez-les au ticket de suivi des problèmes. En voici des exemples : éléments difficiles à mettre en œuvre, zones/caractéristiques apparemment sans rapport qui sont affectées (par exemple, modification d'un composant commun) et portée des modifications de réécriture ou des tâches techniques.

XVII-C - Soumission

Fournir un titre informatif. Il est utile de pouvoir identifier rapidement unpull request parmi les nombreuses demandes ouvertes dans le dépôt. Cela n'est pas possible lorsque l'identifiant de suivi de problème seul est utilisé comme titre.

Fournissez des informations pour aider les contrôleurs à essayer les changements. Les exemples incluent l'emplacement des données de test appropriées, le type d'utilisateur spécifique/la configuration requise et les détails de service du endpoint. Ajoutez également les parties pertinentes de celle-ci au ticket de suivi des problèmes.

Ajoutez des commentaires avec des astuces permettant de gagner du temps. Pensez en tant que contrôleur et ajoutez des commentaires qui les aideront à bien utiliser leur temps. Par exemple, indiquez le code qui a simplement été déplacé, expliquez une zone criarde de diff où il y a peu de changements, ou expliquez une zone susceptible de susciter des demandes d'explication.

Attirez l'attention sur les zones de code qui la requièrent. Évitez de vous fier aux critiques (ou espérez qu'ils ne le fassent pas). Après avoir effectué le travail, il est probable que vous le compreniez mieux que les contrôleurs. Faites des commentaires sur des choses que vous ne sentez pas bien (nouvelles ou existantes), sur lesquelles vous n'êtes pas sûr (exigences ou techniques), ou sur des décisions d'intérêt que vous avez prises.

XVII-D - Contrôle et commentaires

Passez en revue l'ensemble des changements. Tout dans le code de base contribue directement ou indirectement aux fonctionnalités et à la qualité de l'application. Évitez de considérer certaines zones moins dignes de contrôles et de les ignorer, par exemple des tests, des configurations d'application ou des feuilles de style.

Exécutez le code. Ce n'est pas parce que ça a l'air d'aller, que ça va bien. Si les outils du projet empêchent les utilisateurs de basculer entre leur propre travail et les contrôles de branche, corrigez le problème.

Attention… à tout, vraiment. Tirez parti de votre expérience, de vos connaissances, de vos guides et de tout le reste. Fait-il la bonne chose, a-t-il un sens, est-il bon, est-il maintenable, est-il testé, les hypothèses/interprétations sont-elles claires et valides, etc.

Posez des questions. Le contrôle de code permet de partager les connaissances de la base de code au sein de l'équipe et aide les contributeurs à apprendre les uns des autres. Si quelque chose n'est pas clair ou n'est pas compris,

posez des questions. Si le changement dans son ensemble est difficile à analyser et à comprendre, des améliorations peuvent être nécessaires.

Faites des commentaires clairs, expliquant le raisonnement là où ce n'est pas évident. Les avantages immédiats sont notamment d'éviter les demandes de clarification (ce qui retarde l'avancement) et d'éviter les révisions inattendues dues à des commentaires mal interprétés. L'avantage à long terme est la mise à niveau des autres participants en tant qu'auteurs et critiques. Cela contribue à améliorer la qualité globale, car il est généralement impossible pour une seule personne d'examiner tous les changements.

Rechercher des pièces manquantes du puzzle. Le code actuel, plus les modifications en cours de révision, ainsi que d'autres éléments de travail en attente - constitueront à un moment donné une version distribuable du logiciel. Indiquez tout ce qui vous semble manquant mais nécessaire - un suivi de tâches à accomplir doit éventuellement être créé pour cela.

XVII-E - Adressage des commentaires

Adresser toutes les instances où un commentaire s'applique. Les contrôleurs peuvent ne pas avoir remarqué tous les endroits où un commentaire s'applique, ou s'abstenir de les commenter tous pour éviter de causer du bruit (une bonne pratique). Déterminez si le point soulevé par un commentaire s'applique à d'autres endroits que la ligne sur laquelle il a été fait.

Facilitez le contrôle des révisions par les contrôleurs. En fonction du dépôt/de l'outil utilisé, la modification des versions existantes peut rendre difficile la visualisation de ce qui a été modifié. Les gestions de version détaillées qui utilisent des commentaires individuels (ou des groupes logiques de ces commentaires) sont pratiques à examiner et à corréler avec les commentaires d'origine - permettant aux contrôleurs de voir les modifications apportées en réponse à leurs commentaires. Ils créent également un meilleur historique de contrôle de version. Les gestions de versions uniques de grande ampleur intitulées « Adresse de commentaires » ne présentent aucun de ces avantages.

Minimiser les allers retours répétés. Certaines choses sont plus faciles à résoudre avec une conversation, plutôt que des cycles répétés de modifications et de commentaires, ou de longues discussions de fil de commentaires. Lorsque vous repérez l'une de ces situations de va-et-vient, parlez-en de vive voix ou suggérez une courte session de programmation en binôme.

XVII-F - Avant de fusionner

La construction réussie. Configurez les outils (référentiel et serveur CI) pour exiger que la version la plus récente de la branche fonctionne avant que la fusion ne soit autorisée.

La branche est à jour par rapport à la branche cible. Configurez le dépôt pour exiger que la branche source ne soit pas derrière la cible avant que la fusion ne soit autorisée.

Tous les commentaires ont été résolus. Cela signifie normalement, faire des révisions, répondre à la question d'un contrôleur (à sa satisfaction) ou conclure une discussion. Il peut être difficile de suivre les commentaires en suspens - les fonctionnalités du dépôt telles que les réactions (levée de pouce, etc.) ou les cases à cocher des tâches peuvent aider. Configurez le dépôt (si pris en charge) pour exiger que toutes les tâches soient terminées avant que la fusion ne soit autorisée.

Tous les éléments de la liste de contrôle sont vérifiés. Une liste de contrôle de pull request peut inclure des éléments de pré-fusion en plus de ceux de pré-soumission.

Une approbation adéquate a été donnée par les contrôleurs. Ce qui est adéquat dépend de l'équipe et du projet. Un nombre minimum d'approbations est généralement requis. Il peut s'avérer nécessaire de demander une/quelques autorisations supplémentaires à un groupe de personnes suffisamment responsables. Configurez le dépôt (dans la mesure où cela est pris en charge) pour requérir ces approbations avant d'autoriser la fusion. En outre, faites preuve

de discernement pour décider si l'approbation doit être sollicitée auprès de personnes particulières telles que des spécialistes de zone/de domaine/de technologie.

Les approbations sont toujours valables si des modifications post-approbation ont été apportées. Des révisions importantes peuvent parfois résulter de la rétroaction donnée par un contrôleur après l'approbation d'un autre. En pareil cas, le contrôleur qui a déjà approuvé peut souhaiter réexaminer. Les référentiels peuvent être configurés pour révoquer les approbations lorsque des modifications ultérieures sont apportées, mais cela a tendance à devenir fastidieux, car des modifications de fusion ou des modifications triviales le déclenchent.

XVIII - Caractéristiques

Les ambiguïtés et les questions ont été résolues. Documentez les exigences sur le suivi des problèmes.

Les exigences **ne contredisent pas et n'entrent pas en conflit logique avec les fonctionnalités existantes.** Viser une expérience utilisateur cohérente.

L'implémentation **répond au cahier des charges,** en tenant compte des clarifications et des améliorations. Lisez à nouveau le cahier des charges pour vous assurer que rien n'a été oublié.

L'implémentation n'introduit pas de nouveau comportement indésirable.

L'implémentation **répond aux exigences non fonctionnelles**. Si elles ne sont pas définies, faites un effort pour découvrir et aider à définir celles qui conviennent. Les exemples incluent les temps de réponse, les navigateurs et versions prises en charge, ainsi que les périphériques/facteurs de forme cibles.

L'implémentation **répond aux exigences générales de l'application et aux conventions** qui ne sont pas explicitement mentionnées dans cet ouvrage ou peut-être ailleurs. Familiarisez-vous avec les fonctionnalités existantes et veillez à la cohérence.

L'implémentation **n'ajoute pas de difficultés à l'expérience/au flux de travail du développeur,** ce qui nuirait à la productivité. Il devrait exister un moyen simple de désactiver ou de contourner le problème pour résoudre cela.

Anticipez le travail futur. Nous voulons garder les choses simples et éviter de bâtir des choses dont nous n'aurons jamais besoin. Nous pouvons toutefois éviter de rendre les choses difficiles à l'avenir en étant conscients des besoins à venir, à court et à moyen terme, et en les gardant à l'esprit aujourd'hui.

XIX - Texte de l'interface utilisateur

Ces éléments s'appliquent à *tous les textes, quel que soit leur auteur*. Le texte fourni par l'expérience de l'utilisateur ou les parties prenantes de l'entreprise doit être examiné et tout problème résolu avec l'auteur.

Soyez **précis et sans ambiguïté**.

Évitez de contredire un autre texte ou le comportement réel d'une application. Le texte existant dans d'autres zones de l'application peut nécessiter une mise à jour.

Utilisez **l'orthographe et la grammaire correctes**.

Utilisez **une capitalisation cohérente** des termes.

Utilisez **un style et un ton cohérents**.

Utilisez le temps **correct (actuel/passé).** Cela s'applique également à la journalisation des messages.

Utilisez **une terminologie cohérente** lorsque vous vous référez, par exemple, aux fonctionnalités des applications et aux concepts métier.

Évitez de vous fier à l'ordre des membres enum pour l'ordre d'affichage de l'interface utilisateur. L'ordre d'affichage logique ou alphabétique est souvent différent de l'ordre des membres dans le code. L'ordre des membres est également susceptible de changer, sans en attendre d'effets secondaires.

Utilisez des données factices réalistes et professionnelles. Des données réalistes aident à détecter les problèmes pouvant survenir ultérieurement à l'aide de données réelles. Les données non professionnelles ou les données avec des plaisanteries peuvent offenser ou donner une mauvaise impression de l'équipe - on ne sait jamais qui va les voir et dans quel état d'esprit ces personnes seront.

XX - Correction de bogues

Trouvez la cause avant d'essayer un correctif. Toute solution basée sur une analyse à première vue, sans connaître ni comprendre la cause, est peu susceptible d'être une solution bonne/correcte.

Existe-t-il d'autres occurrences de ce bogue ou des anomalies similaires qui nécessitent une correction ? Elles pourraient être dans des fonctionnalités similaires ou utiliser du code utilisant des techniques/modèles analogues susceptibles de générer de même erreurs de programmation.

Corrigez la cause première. Résoudre les symptômes seuls, ou les causes intermédiaires, est une solution de contournement plutôt qu'une solution complète. Cela implique souvent l'ajout de code pour contrer ce que font d'autres codes.

Comprenez pourquoi le correctif fonctionne. Un correctif qui semble fonctionner sans que l'on puisse expliquer pourquoi, peut être peu fiable ou incomplet.

Ajoutez des tests qui ne sont pas passés sans le correctif. Les tests aideront à prévenir la régression - le bogue étant réintroduit lors de modifications ultérieures. Choisissez des tests unitaires et/ou d'intégration selon le cas. Validez le test en vérifiant qu'il échoue par rapport au code non corrigé.

Décrivez brièvement le problème et le problème sous-jacent dans le message de gestion de version. Cela aide les contrôleurs à comprendre le correctif et contribue à avoir un historique utile. Il est peu probable que les tickets de rapport de bogue liés contiennent des informations ou analyses techniques adéquates.

Signaler des bogues dans les bibliothèques open source. En les réparant à la source, nous pourrons supprimer plus tard toutes les solutions que nous aurions dû mettre en œuvre et améliorer la qualité pour tous. S'il est déjà signalé, faites un « +1 » et ajoutez des informations utiles aux responsables de la maintenance ou à d'autres consommateurs (solutions de contournement, par exemple). Commentez toutes les solutions de contournement dans votre code, y compris un lien vers le rapport de bogue. Abonnez-vous, pour être averti quand il est corrigé.

Que pouvons-nous faire pour éviter ou détecter automatiquement des bogues similaires à l'avenir ? Il peut y avoir des modèles que nous devrions utiliser (ou éviter), ou des connaissances que nous devrions partager. Il peut y avoir des **règles linter**, des **plugins** ou **d'autres** outils que nous pourrions utiliser pour détecter automatiquement les éventuels problèmes à la première occasion.

XXI - Méta

Y a-t-il **quelque chose à ajouter** à ce guide ?

La **détection des problèmes éventuels peut-elle être automatisée ? Oui** que ce soit pendant le contrôle ou pendant le travail/pré-contrôle. L'automatisation fait gagner du temps et assure la cohérence.

XXII - Note de la Rédaction de Developpez.com

Ce guide est une traduction de **decent-code - A concise guide to writing better code**.

Nous tenons à remercier **vavavoum74** pour la traduction, **Christophe** pour la relecture technique et **Jacques Jean** pour la correction orthographique.