



Sensibilisation API

Valeurs Immatérielles Transférées aux Archives pour Mémoire

Planning

- API en général
- API REST
- API Java

Présentation

- Une API est pour les humains, pas pour les machines
 - Elle doit être compréhensible sans lire le code sous-jacent
 - *Une bonne API n'impose pas de savoir ce qui se passe dessous*
 - Elle doit être consistante
 - *Ordre des arguments toujours le même*
 - *Ne retourner jamais « null »*
 - ...
- Une API est un contrat : une fois publiée, elle est engageante
 - Elle doit être sans ambiguïté
 - *Le nom méthode/classe/package, collection/service doit permettre de savoir ce qui est induit, et l'ordre des appels*
 - *N'afficher que ce qui est utile*
 - L'ajout est possible, le retrait ne l'est pas
 - *Ajouter une méthode/collection : ok*
 - *Ajouter un argument : ko sauf si celui-ci est masqué (package/Json)*
 - *Retrait : ko*

Présentation

- Une API est un contrat : une fois publiée, elle est engageante
 - Il faut prévoir les évolutions des API : la compatibilité ascendante est bien mais l'évolutivité va au delà
 - *Par exemple : les arguments ou les réponses peuvent être « packagés » pour permettre une évolution via le package et non la signature pure*
 - *Exemple : Contexte ou Json objet*
 - *Vie d'une API*
 - *Private => Friend => Stable => Official => Deprecated / Under development => Stable => Official => Deprecated*
 - Les API publiques sont évidemment un enjeu majeur (REST et SPI Stockage)
 - Mais les API internes le sont aussi, dans une moindre mesure
 - *Plusieurs équipes internes partageant les API internes*
 - *Greffons utilisant « nos » API en provenance de développeurs externes*
- Less is more
 - Design caché, ne pas supposer plus que ce qui est écrit

Présentation

- Dès la 1ère version
 - Se poser les questions car le changement à un coût
 - *Pour soi : maintenance de l'ancienne version*
 - *Pour les autres : incompatibilité et donc évolution non maîtrisée*
 - Tout est API
 - *REST, Java*
 - *Répertoire, Fichier*
 - *Variables d'environnement, Options de ligne de commande*
 - *Message (log) (exemple : parsing de log)*
 - *Comportement (exemple : return null ou liste vide)*
 - Pour autant : Osez ! Et Partagez !
 - *L'évolution doit être préparée*
- API Review et API Design
 - Le design d'API est comme la sécurité : tant qu'on a aucun incident, c'est que c'est bien fait ; le jour où on a un incident : il est trop tard
 - *Le coût est malheureusement invisible sauf si on compte les API Review dans nos Points de complexité*
 - L'API Review est l'assurance de l'autonomie
 - *Team Design*

Présentation

- Affirmations excessives
 - Une API doit être belle
 - *La beauté n'a rien à voir, l'objectif est que ce soit compréhensible*
 - *Il ne s'agit pas de philosophie mais de concret : de maintenance, de compatibilité, de coût, d'évolutivité*
 - Une API doit être correcte
 - *La facilité d'usage prime sur la « correction »*
 - Une API doit être simple
 - *Si la simplicité empêche de faire des choses compliquées, alors l'API est inutile*
 - *La facilité ne veut pas dire « trop » simpliste*
 - Une API doit être 100 % compatible
 - *C'est bien sûr l'objectif principal, mais il faut parfois admettre de réduire cet objectif à 99 % quand ce '1 %' est bloquant*
 - *Dans certains cas (incompatibilité) => deprecated et encouragement à utiliser la nouvelle API en raison de nouvelles fonctionnalités/performances*
 - Une API doit être symétrique
 - *Cela se rapproche de l'affirmation « belle »*
 - *Un Coder (Json >) et un Decoder (Json <) n'ont pas besoin d'être parfaitement symétrique*
 - *Json > : read(File) ; toObject(Class) ; readStreamSax(File) ;*
 - *Json < : fromObject(Object) ; write(File) ; mais pas de writeStreamSax*

Présentation

- Possibilité d'apprendre collectivement
 - API Design Fest
 - *Le principe du jeu : pour chaque équipe*
 - 1) *Une API est créée sur la base d'un scénario de tests (API + implémentation)*
 - 2) *Une première relecture par un « conseil » neutre*
 - 3) *Une évolution de l'API est demandée (scénario 2 de tests) (API + implémentation)*
 - *Phase de conquête*
 - 1) *Toutes les équipes ont accès aux 2 étapes de la vie de l'API des autres équipes*
 - 2) *Toutes les équipes essayent de proposer des Junit qui démontrent que le comportement de la première API n'est plus respectée dans la seconde*
 - 3) *Pour chaque Junit démontrant une faille unique : +1 point pour l'équipe qui a écrit le Junit*
 - 4) *Pour chaque API sans faille : +5 points pour l'équipe qui l'a écrite*
 - Etudes de cas
 - *Exemple : Extensible visitor pattern*
- Maître mot : une API doit permettre d'être « Cluelessness »
 - Il ne doit pas être nécessaire d'en savoir plus que ce qu'elle indique pour pouvoir l'utiliser, la comprendre

http://wiki.apidesign.org/wiki/Main_Page Jaroslav Tulach (NetBeans)

Organisation d'un API Review

- Voir différentes pages
 - Exemple chez Netbean
 - <http://wiki.netbeans.org/APIReviews>
 - <http://wiki.netbeans.org/APIReviewSteps>
 - <http://wiki.apidesign.org/wiki/APIDesignPatterns>
 - <http://wiki.netbeans.org/APIDevelopment>
 - *Exemple de questions*
 - <https://openide.netbeans.org/tutorial/questions.html>

http://wiki.apidesign.org/wiki/Main_Page Jaroslav Tulach (NetBeans)

Planning

- API en général
- API REST
- API Java

Modèle général : End-Points

- **Versionning des API**
 - Domaine = nom DNS du service
 - Application = domaine de l'application (ingest, access, management, ...)
 - Version = v1, v2, ... versions majeures uniquement (version fine dans le header)

<https://domaine/application/version/>
- **Accès à une ressource/collection**

<https://domaine/application/version/ressources>
- **Accès à un élément de cette ressource**

https://domaine/application/version/ressources/identifiant_ressource
- **Accès à une deuxième ressource issue d'un élément**

https://domaine/application/version/ressources/identifiant_ressource/ressources2
- **Idem interne** : <https://interne/application/version/ressources>
- **Commandes** :
 - GET : Lecture
 - POST : Création
 - PUT : Mise à jour complète
 - PATCH : Mise à jour partielle, modification d'un état (nécessité mais PUT possible)
 - DELETE : Effacement, Annulation
 - HEAD : Informations succinctes (a priori non nécessaire)
 - OPTIONS : Opérations permises (a priori non nécessaire)

Modèle général : End-Points

- Services transverses offerts par chaque « application »
 - Pour les API externes
 - *API de vérification du statut du module*
 - *API permettant de consulter la version du service et la version de l'application déployée*
 - Pour les API internes (non visibles), en plus
 - *API de vérification de l'état des dépendances directes du module*
 - *API d'activation/désactivation d'un service*
 - *API de récupération des métriques du module*
 - *Métriques système et JVM*
 - *Métriques d'utilisation des API (par version)*
 - » *Cette fonctionnalité pourrait remonter pour chaque « endpoint » :*
 - » *le nombre d'appels,*
 - » *les temps d'exécution maximum/minimum/moyen/etc.*
 - » *le nombre d'erreurs rencontrées*
 - » *etc.*
 - » *Le tout sur différentes fenêtres temporelles.*
 - *API de sauvegarde (export) et de restauration (import) (interne)*

Conseils généraux

- Prévoir les cas où les retours sont nombreux
 - Next : pour les cas n'utilisant pas le DSL Vitam
 - Offset, Limit : pour les cas utilisant le DSL Vitam
- Prévoir les résultats asynchrones
 - Async_List : pour les réponses internes par lot itératif
 - Async_tasks ou Callback : pour des réponses externes/internes asynchrones
- L'API REST ne traduit pas l'implémentation mais le métier
 - Une collection n'est pas forcément une table
 - Exemple : *ingests* => *sous-ensemble des opérations*
 - Une table n'est pas forcément une collection
 - Exemple : *objects* (si implémentés) => dans *ObjectGroup*

Planning

- API en général
- API REST
- API Java

Définition d'une API de qualité

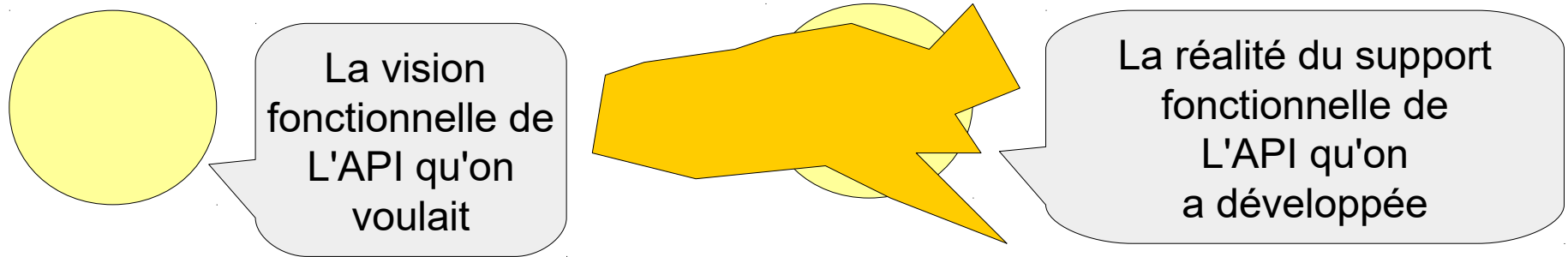
- Compréhensible
 - Une bonne API n'impose pas de savoir ce qui se passe dessous
 - *Une série d'éléments : List*
 - *Un ensemble d'éléments : Set*
 - *Un flux d'éléments : Stream*
- Constante
 - Toujours le même Ordre des arguments
 - *read(byte[]) ; read(byte[], int, int) ;*
 - Ne retourner jamais « null »
 - Modèle de threads
- Découvrable (Discoverability)
 - Facile de trouver les exceptions, les factory, les helpers, les services, les modèles de données, ...
 - Découpage en modules/sous-modules/packaging cohérent et clair
- Tâches simples doivent être faciles
 - Pour les tâches simples, les moyens de le faire doivent être simple
 - *Helpers, méthodes avec des valeurs par défaut pour les 80 % des cas*
 - La complexité est possible mais avec d'autres méthodes/classes
 - *Les options sont des arguments additionnels (autres méthodes)*
 - *L'accès à des constructeurs plus fins sont hors Helper*

Définition d'une API de qualité

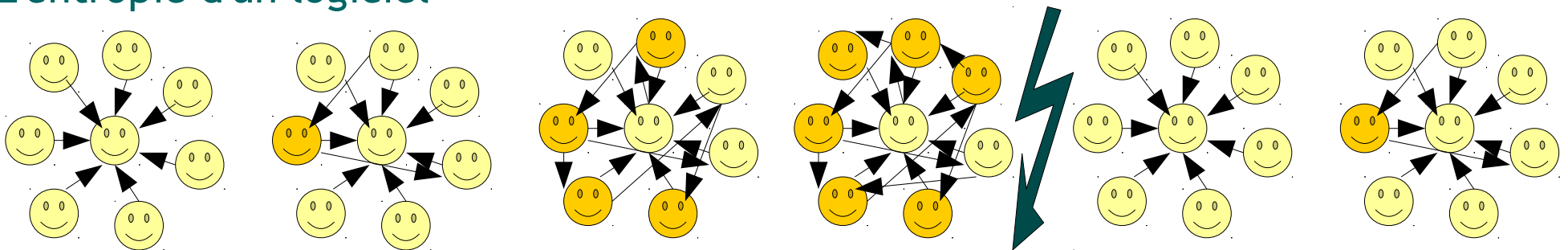
- **Préserver l'investissement**
 - Une fois l'API créée, veiller à ne pas « perdre » ses clients
 - *Junit qui assurent que la montée de version ne crée pas de régression*
 - *Style de programmation homogène pour faciliter la reprise*
- **Compatibilité ascendante**
 - Au niveau du code
 - *Ne pas casser des signatures (méthodes, classes, packages)*
 - Au niveau binaire
 - *Ne pas retirer des classes si elles ont été un jour « publiques »*
 - Au niveau fonctionnel
 - *Le comportement fonctionnel doit être stable (modèle AMOEBA)*
 - *Exemple : return null ; vs return empty ;*
 - L'ajout est possible, le retrait ne l'est pas
 - *Ajouter une méthode : ok*
 - *Ajouter un argument : ko sauf si celui-ci est masqué (dans un package)*
 - *Retrait : ko*
 - Attention : SPI
 - *L'ajout est un problème, le retrait l'est moins*

Maîtriser l'évolution

- Modèle AMOEBA



- L'entropie d'un logiciel



- L'entropie augmente au fur et à mesure et rend impossible la maintenance
- On réécrit tout (refactoring) mais hélas à l'itération suivante l'entropie reprend
 - *Anticipation mais pas trop*
 - *Découpage propre (contrats) pour éviter l'entropie*
 - *Contrats matérialisés par des Junits/Tests de non régression*

Conseils

- 1) Commencer par analyser le besoin métier (Use Case Oriented)
 - 1) Proposer une logique applicative (algorithmie)
- 2) Définir l'interface (API publique)
 - 1) La confronter à différents avis (API Review)
- 3) Documenter l'API publique
 - 1) A quoi ça sert ?
 - 2) Quelle est la logique d'ensemble ?
 - 3) FAQ (mode d'usage, cas d'usage)
 - 4) Documentation de l'API (JavaDoc) : cas particuliers, erreurs et arguments
- 4) Ecrire les Tests d'acceptance et les Junit principaux
- 5) Implémenter l'API publique puis ses implémentations spécifiques
 - 1) Mettre à jour la documentation en fonction
 - 2) Ajouter les Junits nécessaires
- 6) Internationalisation dès le début
 - 1) Commentaires et codes : EN
 - 2) Log : EN (tout texte en static final String XXX)
 - 3) Journaux : FR / EN

Éléments de API Review

- Il faut prévoir les évolutions des API : la compatibilité ascendante est bien mais l'évolutivité va au delà
 - Par exemple : les arguments ou les réponses peuvent être « packagées » pour permettre une évolution via le package et non l'API pure
 - *Exemple : Contexte (contenant beaucoup d'info) et non chaque info*
 - Vie d'une API
 - *Private => Friend => Stable => Official => Deprecated / Under development => Stable => Official => Deprecated*
 - Préférer coder depuis une interface et non depuis l'implémentation
 - Les API publiques sont évidemment un enjeu majeur (REST et Stockage)
 - Mais les API internes le sont aussi, dans une moindre mesure
 - *Plusieurs équipes*
 - *Greffons utilisant « nos » API*
- Less is more
 - Design caché, ne pas supposer plus que ce qui est écrit
 - Pas « tout » public : savoir utiliser les private/protected/package protected
- Séparer client et provider dans les interfaces et les packages
- Le packaging doit indiquer les usages et ne pas mélanger implémentation et définition

Éléments de API Review

- Une méthode est meilleure qu'un champ
- Un Factory est meilleur qu'un constructeur
- Autant que possible tout doit être « final »
 - Classes, méthodes, attributs
 - Les exceptions sont : on sait que l'on va l'étendre (abstract) ou la surcharger (extends)
 - *Uniquement pour les codes « amis »*
- Garder le code là où il doit être
 - Un Setter doit appartenir à la classe qui déclare l'attribut
- Laisser au créateur d'un Object plus de droits
 - Exemple : un argument Class Configuration permet d'avoir plusieurs implémentations de celui-ci et donc de laisser son comportement libre au fournisseur
- Ne pas exposer des hiérarchies trop profondes
 - Contre exemple : JButton < AbstractButton < JComponent < Container
 - *On pourrait croire qu'il est possible d'ajouter des composants à un bouton, ce qui n'était pas le sens premier de cette hiérarchie*

Éléments de API Review

- Être prêt pour l'ajout de paramètres (en argument et en résultat)

```
public interface Compute {  
    public void computeData(Request request, Response response) ;  
    public final class Request{}  
    public final class Response{}  
}
```
- Eviter le « spaghetti » design
 - Les méthodes liées dans la même classe
 - Les méthodes non liées dans des classes différentes
 - Les classes liées dans un même package
 - Les classes non liées ou pour des cas particuliers dans des packages différents
 - Idem pour les modules
 - Aucune référence à un SPI dans l'implémentation d'une API et packages différents
 - *Core API : concentre les opérations essentielles*
 - *Support API : utility methods, helpers*
 - *Core SPI : les opérations pour réaliser un plug-in*
 - *Support SPI : utility methods, helpers*
- Prévenir les mauvais usages de l'API
 - Exemple : `Connection.rollback(Savepoint)` // Laisse croire que l'on peut créer un Savepoint
=> `SavePoint Connection.setSavePoint()` // Oblige le Savepoint à être généré par Connection

Éléments de Code Review

- Points d'attention sur les synchronisation et Deadlock
 - Documenter le modèle de Threads
 - Deadlock à prévenir
 - *Mutual exclusion condition*
 - *Le plus connu : une seule thread peut détenir un lock, une ressource*
 - *Non pre-emptive scheduling condition*
 - *Une ressource est acquise et ne peut être libérée que par le propriétaire*
 - *Hold & Wait condition*
 - *Une ressource est acquise par une thread indéfiniment*
 - *Ressources acquises de manière incrémentale*
 - *Une ressource est acquise alors qu'une autre est déjà acquise (et non libérée)*
- Des éléments similaires extra-JVM : parallélisme inter-process
 - Des conditions similaires peuvent intervenir (deadlock)
 - *Exemple : une demande à un tiers mais la demande n'aboutit jamais => deadlock ou timeout ?*
- Gestion de la mémoire : est-ce que la logique algorithmique utilisée maintient le programme dans des conditions d'usage de la mémoire contrôlables et contrôlées
 - *Exemple : Listing d'un répertoire : combien de résultats ? Tous les résultats vont ils tenir en mémoire ? (valable partout : Shell, Java, HTML, ...)*
 - Utiliser des « singletons vides » pour retourner une absence de résultat (pas NULL)
 - Vider tous les « Objets » inutiles (ne pas faire confiance au Garbage Collector)

Éléments de Test Review

- Test compatibility Kit
 - Utilitaires pour faciliter l'implémentation de tests transverses entre plusieurs implémentations d'une même interface
 - *Exemple : permet de s'assurer de la cohérence fonctionnelle entre une implémentation filesystem / database / cloud d'un système de fichiers abstrait*
- Tests de non régression
 - A chaque évolution, ajouter des « gardes » qui testent la non régression des fonctionnalités qui viennent d'être rajoutées
 - Les précédents gardes assurent que le modèle AMOBIA est maîtrisé et que l'évolution va bien toujours en ajout de fonctionnalités
- Tests pour le modèle de thread et de parallélisme
 - Les 4 cas de deadlock
 - Les cas de « race condition » dus au parallélisme externe (autres programmes)
- Chaque bug rencontré doit voir un Junit vérifiant la non régression

API : cas particulier du Multi-tenants

- Les aspects multi-tenants pris en compte selon 2 modalités
 - 1) Implicitement : API REST externes
 - L'application Front-office n'est utilisée que pour un seul tenant
 - Vitam ajoute systématiquement les informations à chaque requête du tenant concerné
 - 2) Explicitement
 - API REST Externes
 - *L'application Front-office utilise plusieurs tenants (fonction de l'utilisateur connecté)*
 - *L'application doit intégrer un code dans le Header pour indiquer le code du tenant concerné*
 - *X-TenantId : valeur*
 - API REST Internes (et Java)
 - *X-TenantId doit être présent d'une manière ou d'une autre*
- La séparation est logique
 - Données : séparation par filtre
 - Stockage : séparation par container