Audit qualité

Système PoGo

Ce document reporte les résultats d'un audit succinct de la qualité du système logiciel PoGo développé par l'établissement publique français « *Voies navigables de France* » (VNF). Cet audit concerne la qualité du code contenu dans le système. Cette audit est réalisé sur base d'une analyse dite statique (sans exécution) du code source du système en se focalisation sur les risques liés au manque de qualité.

Ce document est organisé comme suit : La section 1 décrit le contenu et la découpe du système analysé, la section 2 présente les résultats de l'analyse organisés selon la découpe de la section 1 et la section 3 prend du recul sur les résultats pour donner une conclusion globale à l'analyse.

1. Description du contenu analysé

Cette section décrit le contenu concerné par cet audit. Le contenu est identifié selon le code source reçu dans l'archive fournie le **17 juin 2015**. Nous décrivons sommairement l'archive reçue puis définissions le périmètre de l'analyse.

1.1.1 Description de l'archive

L'archive reçue le 17 juin 2015 contient l'ensemble du code source du système.

Une analyse à l'aide l'outil Cloc nous permet de visualiser les informations suivantes :

Language	files	blank	comment	code
Java	171	3271	3877	11461
Objective C	70	2169	1987	7268
XML	68	289	75	2786
C/C++ Header	72	1035	2141	1363
SQL	21	228	96	895
MUMPS	1	41	0	237
Bourne Shell	1	54	39	210
Maven	2	28	13	126
Bourne Again Shell	1	20	21	123
JSON	2	0	0	85
DOS Batch	1	24	2	64
HTML	1	0	0	20
SUM:	411	7159	8251	24638

Dans le cadre de cet audit, les lignes code labélisée par Java, Objective C et C/C++ Header sont analysées. Les autres sont donc ignorées et aucun commentaire ne sera donné sur le présence ou contenu.

1.1.2 Périmètre de l'analyse

Une analyse d'un tel système nécessite d'identifier le code à analyser mais également la manière dont la découpe est réalisée. En effet, le système dispose de plusieurs composants présentant des caractéristiques et rôles différents. Il est donc pertinent de les analyser à part.

Ce système est constitué de 2 niveaux distincts :

- Un backend représenté par une application Java exposant un ensemble d'interfaces vers les systèmes de VNF
- Un frontend consommant les interfaces proposées par le backend et exposant les fonctionnalités à l'utilisateur final.

Le frontend existe sous forme de deux déclinaisons mobiles différentes : une déclinaison pour les systèmes de type Android et une déclinaison pour les systèmes de type iOS.

Le code analysé dans cet audit concerne donc les 3 composants suivants :

- Le code de l'application de type web service que nous nommerons « **backend Java** » dans la suite du document
- Le code de l'application frontend pour les systèmes Android que nous nommerons « **frontend Android** » dans la suite du document
- Le code de l'application frontend pour les systèmes iOS que nous nommerons « **backend iOS** » dans la suite du document

2. Analyse du système

Cette section présente et discute les résultats de l'audit. Elle est organisée en fonction des 3 composants identifiés (c.f. section 1.1.2).

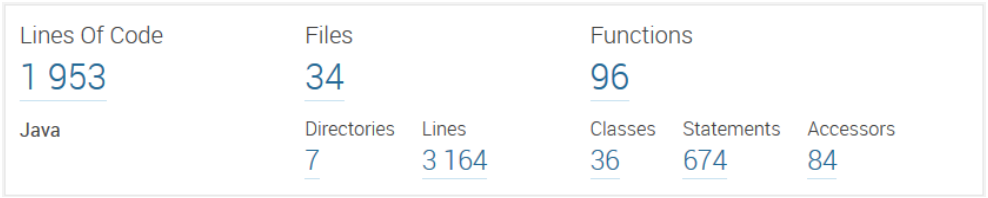
2.1 Analyse du backend Java

Le « backend Java » correspond au code du composant webservices exposant ces interfaces aux 2 autres composants. Il est développé en Java.

Le modèle de qualité utilisé pour l'évaluation est le modèle défini par l'outil « SonaQube » pour le langage Java.

2.1.1 Taille

La taille de l'application met en évidence les métriques déterminant la taille selon plusieurs points de vues (nombre de fichiers, de classes, etc.). Les valeurs de ces métriques sont présentées dans le cadre suivant.



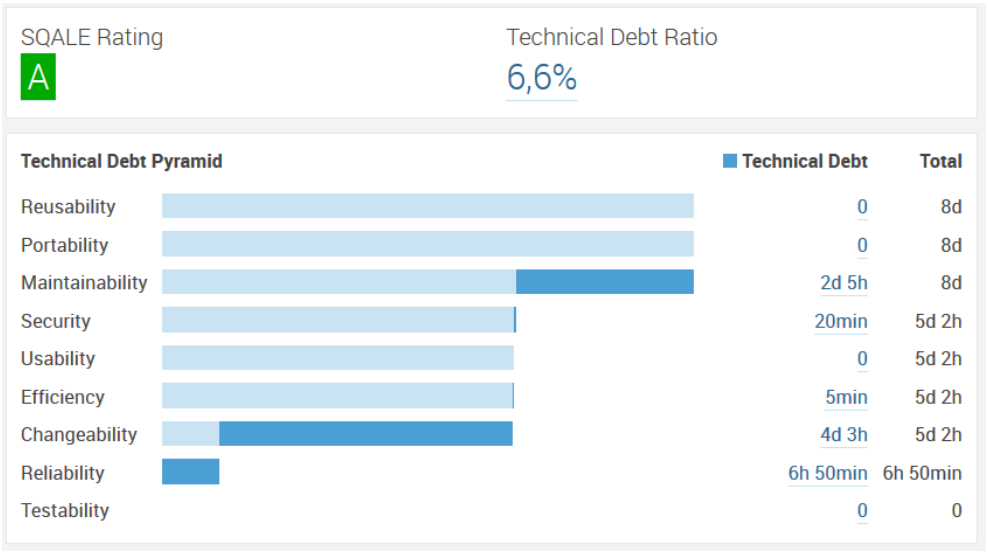
L'application comporte 1953 lignes de code reprises dans 34 fichiers. Il s'agit d'une application de très petite taille.

2.1.2 Dette technique

La dette technique est un indicateur indispensable à la bonne gestion du développement d'un logiciel. La dette technique correspond à la somme des efforts de correction de toutes non-conformités identifiées dans le code. Plus cette dette est grande plus l'effort de correction est important (il peut même dépasser le temps de développement d'une application).

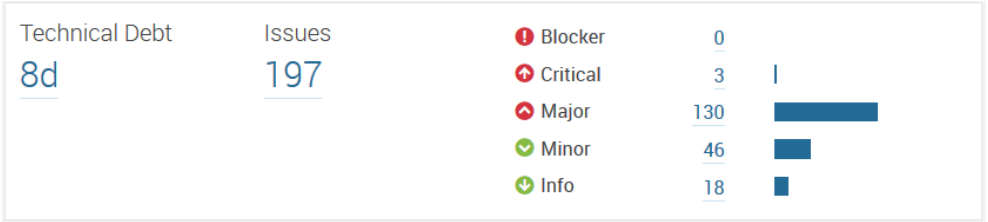
Le fait de ne pas rembourser sa dette technique induit le paiement d'intérêts. Ces intérêts correspondent au surcoût engendrer par la dette. Le manque d'une bonne documentation augmente les temps de développement de nouvelles fonctionnalités ou la maintenance des anciennes.

Pour identifier si une dette technique risque d'induire une grande quantité d'intérêts. On utilise un score sur base du modèle d'évaluation SQALE (http://www.sqale.org/). Cette note dépend du taux de dette technique par ligne de code. Le cadre suivante présente la note SQALE identifiée pour le backend Java ainsi que la décomposition de la dette selon les différentes caractéristiques de la qualité logicielle.



Le composant présente une note SQALE de A. Il s'agit de la meilleure note SQALE disponible. La maintenance et l'évolution de l'application ont peu de risque de susciter un surcoût. Notons que la plupart de la dette technique se trouve au niveau de l'évolutivité (« changeability ») de la plateforme.

La dette technique identifiée provient de non-conformités présentent dans le code. Le cadre suivant expose et trie les non-conformités identifiées.



La dette technique (8 jours) provient de 197 non-conformités identifiées. La plupart sont jugées majeures et proviennent du non respect des conventions de nommage standards pour le Java et de l'incertitude sur la fermeture des objets de type « ResultSet » dans la classe « DatabaseManager ». De plus, 3 sont considérées comme critiques. Elles proviennent de l'absence d'implémentation de la méthode « hashCode » alors que la méthode « equals » est implémentée (ou vice-versa). Les 3 non-conformités identifiées à ce sujet sont :

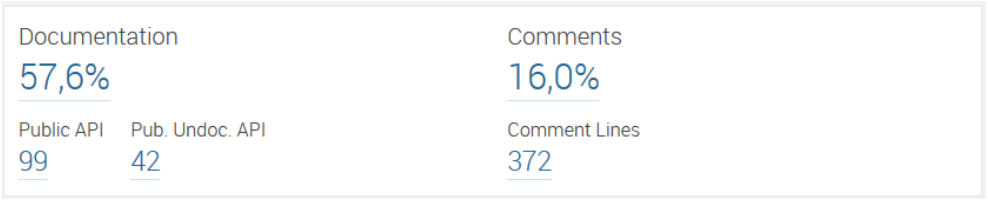
- La classe VNFPeriode
- La classe VNFCategorie
- La classe VNFCalendrier

2.1.3 Maintenabilité

La maintenabilité est un aspect important de l'application car elle va influer son évolution à court et moyen termes.

La documentation est un facteur de coût important dans une application. Cependant, dans les développements de type agile ou itératif, il est commun que la seule source de documentation soit le code. Dans ce cas, il est important de documenter le code afin de faciliter sa compréhension.

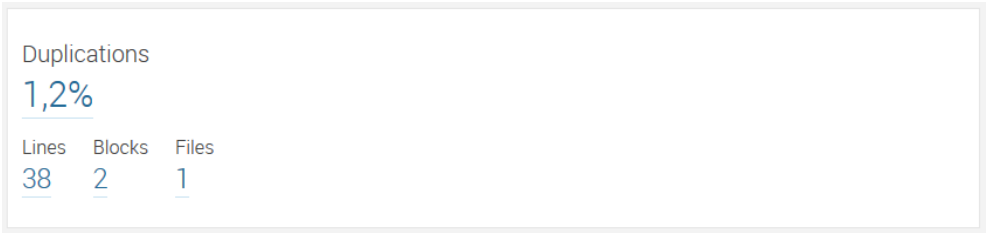
Le cadre suivant présente les métriques relatives à la documentation interne du code.



Globalement, l'application dispose d'un taux adéquat de documentation. Notons malgré tout que le taux d'API documentées est trop faible. Il convient que ce taux dépasse 95%.

La duplication est un autre problème lors de la maintenance. Elle indique que des comportements sont dupliqués à plusieurs endroits du code. Cela implique de devoir maintenir plusieurs fois le même comportement. De plus, cela induit un surcoût pour identifier et maintenir les liens entre ces duplications, mais également un risque de désynchronisation entre les doublons.

Les indicateurs de duplication sont présentés dans le cadre suivant.



Le taux de duplication est dans les bornes adéquates (inférieur à 2%). La duplication concerne 2 blocs correspondant à 38 lignes de code dans 1 fichier.

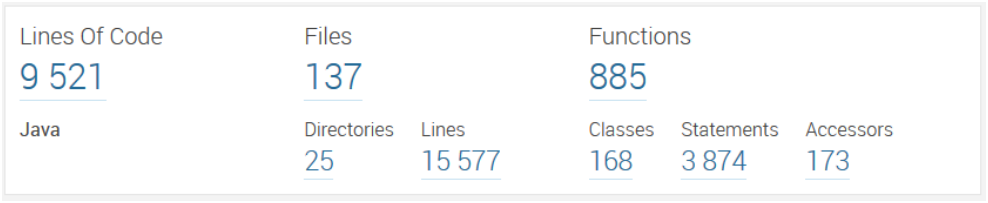
2.2 *Analyse du frontend Android*

Le « frontend Android » correspond au code du composant déployé dans les environnements Android. Il est développé en Java sous le Framework Android.

Le modèle de qualité utilisé pour l'évaluation est le même que le modèle utilisé pour le langage Java sur lequel des règles spécifiques aux bonnes pratiques du Framework Android ont été ajoutées. Ces règles proviennent de l'outil Android Lint (http://developer.android.com/tools/help/lint.html).

2.2.1 Taille

La taille de l'application met en évidence les métriques déterminant la taille selon plusieurs points de vues (nombre de fichiers, de classes, etc.). Les valeurs de ces métriques sont présentées dans le cadre suivant.



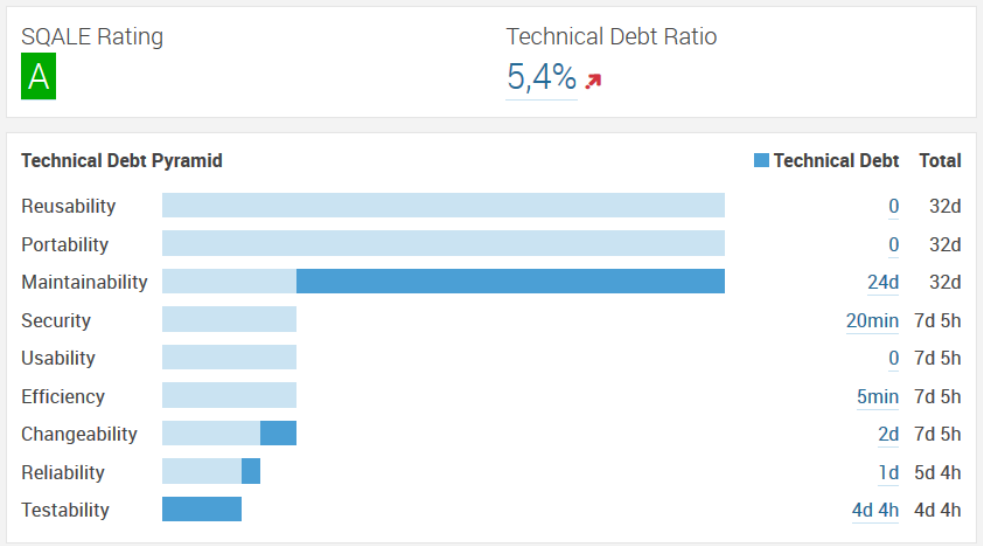
L'application comporte 9521 lignes de code reprises dans 137 fichiers. Il s'agit d'une application de petite taille.

2.2.2 Dette technique

La dette technique est un indicateur indispensable à la bonne gestion du développement d'un logiciel. La dette technique correspond à la somme des efforts de correction de toutes non-conformités identifiées dans le code. Plus cette dette est grande plus l'effort de correction est important (il peut même dépasser le temps de développement d'une application).

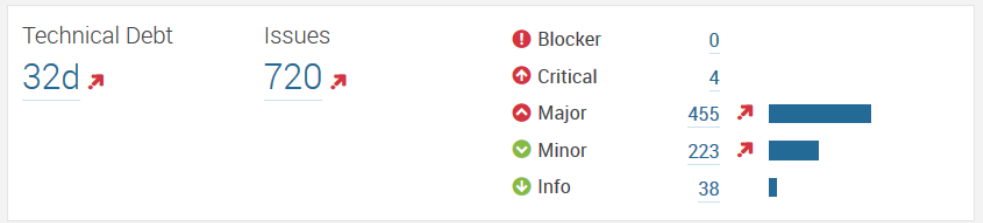
Le fait de ne pas rembourser sa dette technique induit le paiement d'intérêts. Ces intérêts correspondent au surcoût engendrer par la dette. Le manque d'une bonne documentation augmente les temps de développement de nouvelles fonctionnalités ou la maintenance des anciennes.

Pour identifier si une dette technique risque d'induire une grande quantité d'intérêts. On utilise un score sur base du modèle d'évaluation SQALE (http://www.sqale.org/). Cette note dépend du taux de dette technique par ligne de code. Le cadre suivante présente la note SQALE identifiée pour le Frontend Android ainsi que la décomposition de la dette selon les différentes caractéristiques de la qualité logicielle.



Le composant présente une note SQALE de A. Il s'agit de la meilleure note SQALE disponible. La maintenance et l'évolution de l'application ont peu de risque de susciter un surcoût. Notons que la plupart de la dette technique se trouve au niveau de la maintenabilité de l'application.

La dette technique identifiée provient de non-conformités présentent dans le code. Le cadre suivant expose et trie les non-conformités identifiées.



La dette technique (32 jours) provient de 720 non-conformités identifiées. La plupart sont jugées majeures mais ne proviennent que l'absence de documentation ou des problèmes de style de codage. Cependant, 4 sont considérées comme critiques :

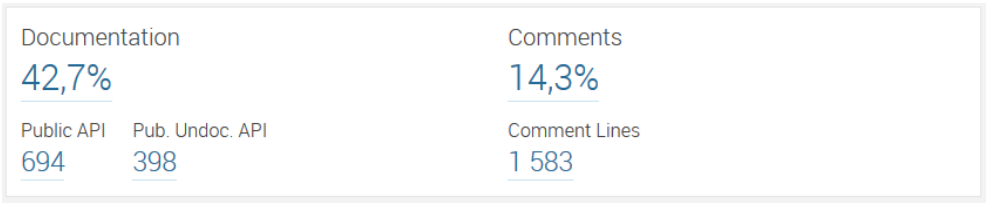
- La fonction updateArrow du fichier DisclosureHeaderPanel.java contient un bloc if vide
- La fonction updateArrow du fichier DisclosureHeaderPanel.java contient un bloc else vide
- La fonction equals du fichier FilterGroupItem.java contient une signature proche de la méthode standard equals
- La fonction setDataImg du fichier Event.java stocke directement un tableau au lieu d'en faire une copie

2.2.3 Maintenabilité

La maintenabilité est un aspect important de l'application car elle va influer son évolution à court et moyen termes.

La documentation est un facteur de coût important dans une application. Cependant, dans les développements de type agile ou itératif, il est commun que la seule source de documentation soit le code. Dans ce cas, il est important de documenter le code afin de faciliter sa compréhension.

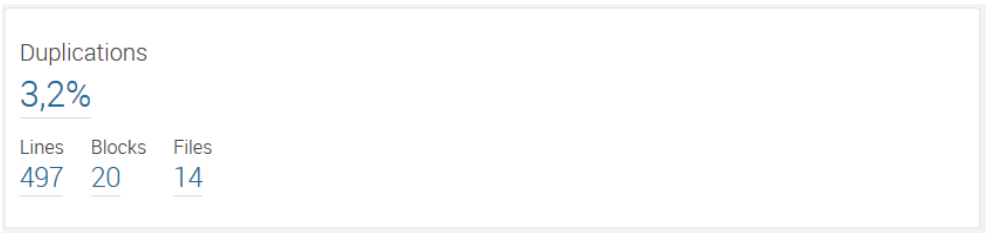
Le cadre suivant présente les métriques relatives à la documentation interne du code.



Globalement, l'application dispose d'un taux adéquat de documentation. Notons malgré tout que le taux d'API documentées est trop faible. Il convient que ce taux dépasse 95%.

La duplication est un autre problème lors de la maintenance. Elle indique que des comportements sont dupliqués à plusieurs endroits du code. Cela implique de devoir maintenir plusieurs fois le même comportement. De plus, cela induit un surcoût pour identifier et maintenir les liens entre ces duplications, mais également un risque de désynchronisation entre les doublons.

Les indicateurs de duplication sont présentés dans le cadre suivant.



Le taux de duplication est un peu élevé (inférieur à 5%). La duplication concerne 20 blocs correspondant à 497 lignes de code dans 14 fichiers. Il convient qui soit en dessous de 2%. La plupart des blocs (9) se trouvent dans le package « app/src/main/java/com/cgi/pogo/adapters ».

2.3 Analyse du frontend iOS

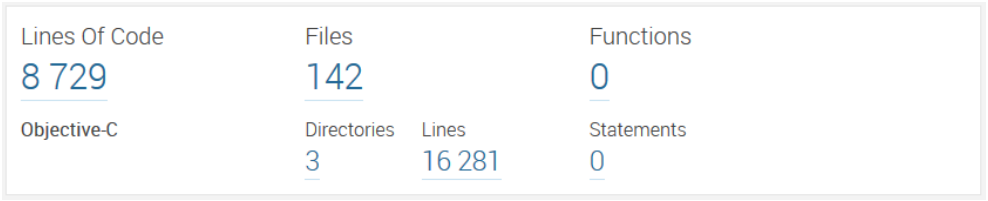
Le « frontend iOS » correspond au code du composant déployé dans les environnements iOS. Il est développé en Objective C.

Le modèle de qualité utilisé pour l'évaluation est basé sur les métriques fournies par le plugin expérimental Objective-C de SonarQube (https://github.com/octo-technology/sonar-objective-c) et sur l'outil OCLint (http://oclint.org/) lequel des règles spécifiques aux bonnes pratiques Objective-C identifiée par la communauté iOS.

Note importante : L'ensemble des métriques n'est pas disponible pour l'Objective C car le plugin est encore en développement. Les résultats ne sont donc donnés a

2.3.1 Taille

La taille de l'application met en évidence les métriques déterminant la taille selon plusieurs points de vues (nombre de fichiers, de classes, etc.). Les valeurs de ces métriques sont présentées dans le cadre suivant.



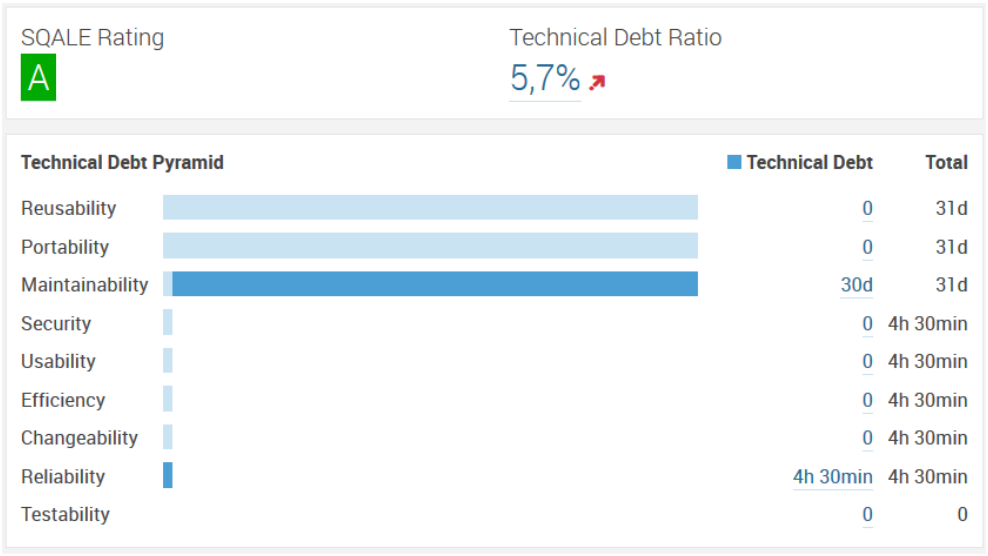
L'application comporte 8729 lignes de code reprises dans 142 fichiers. Il s'agit d'une application de petite taille. Le chiffres nulles correspondent à des métriques non-implémentées.

2.3.2 Dette technique

La dette technique est un indicateur indispensable à la bonne gestion du développement d'un logiciel. La dette technique correspond à la somme des efforts de correction de toutes non-conformités identifiées dans le code. Plus cette dette est grande plus l'effort de correction est important (il peut même dépasser le temps de développement d'une application).

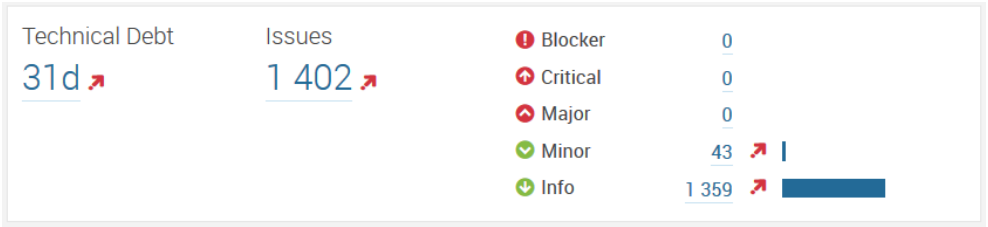
Le fait de ne pas rembourser sa dette technique induit le paiement d'intérêts. Ces intérêts correspondent au surcoût engendrer par la dette. Le manque d'une bonne documentation augmente les temps de développement de nouvelles fonctionnalités ou la maintenance des anciennes.

Pour identifier si une dette technique risque d'induire une grande quantité d'intérêts. On utilise un score sur base du modèle d'évaluation SQALE (http://www.sqale.org/). Cette note dépend du taux de dette technique par ligne de code. Le cadre suivante présente la note SQALE identifiée pour le frontend iOS ainsi que la décomposition de la dette selon les différentes caractéristiques de la qualité logicielle.



Le composant présente une note SQALE de A. Il s'agit de la meilleure note SQALE disponible. La maintenance et l'évolution de l'application ont peu de risque de susciter un surcoût. Notons que, au même titre que pour le frontend Android, la plupart de la dette technique se trouve au niveau de la maintenabilité de l'application.

La dette technique identifiée provient de non-conformités présentent dans le code. Le cadre suivant expose et trie les non-conformités identifiées.



La dette technique (31 jours) provient de 1402 non-conformités identifiées. La plupart sont jugées mineures voire juste informatives. Les non-conformités les plus graves concernent 5 bloc if vide lors de l'appel à la méthode « initWithStyle » des composants suivants :

- VNFMapCategoryCell.m
- VNFEventCustomCell.m
- VNFCategoryCell.m
- VNFToolbarViewController.m

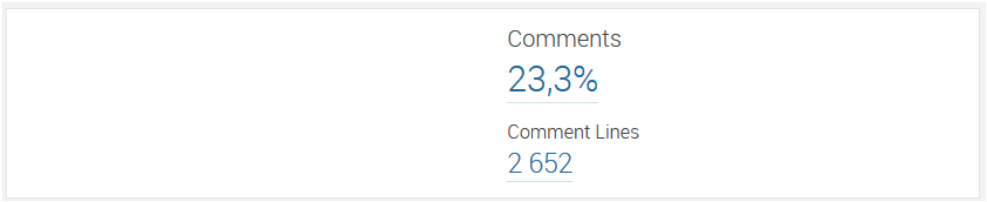
De plus, la méthode « persistentStoreCoordinator » du fichier VNFCoreDataService.m contient un bloc if vide. Ce bloc est en plus identifié par le commentaire « Error for store creation should be handled in here » qui renforce le besoin d'y ajouter la gestion de l'exception adaptée.

2.3.3 Maintenabilité

La maintenabilité est un aspect important de l'application car elle va influer son évolution à court et moyen termes.

La documentation est un facteur de coût important dans une application. Cependant, dans les développements de type agile ou itératif, il est commun que la seule source de documentation soit le code. Dans ce cas, il est important de documenter le code afin de faciliter sa compréhension.

Le cadre suivant présente les métriques relatives à la documentation interne du code.



Globalement, l'application dispose d'un taux adéquat de commentaires.

La duplication est un autre problème lors de la maintenance. Elle indique que des comportements sont dupliqués à plusieurs endroits du code. Cela implique de devoir maintenir plusieurs fois le même comportement. De plus, cela induit un surcoût pour identifier et maintenir les liens entre ces duplications, mais également un risque de désynchronisation entre les doublons.

Les indicateurs de duplication n'est pas présent dans notre outil de base. Les informations sur la duplication ont été évaluées grâce à l'outil Simian (http://www.harukizaemon.com/simian/). Nous avons calculé que la duplication touchait 38 blocs de code dans 6 fichiers pour un taux de 3,3%. Ce qui est un peu trop élevé. Il faudrait diminuer ce taux en dessous de 2%.

3. Conclusion

Le système analysé est de petite taille et constitué de 3 applications. Le pattern utilisé consistant en 2 applications frontend, l'une pour Android et l'autre pour iOS, et un webservice est un pattern standard qui ne présente peu de risque compte tenu de la taille du système et des différentes applications.

Le système présente une dette technique faible. En effet, le total de l'effort de correction des non-conformités ne dépasse pas 5 hommes.mois alors que l'effort de développement d'un tel système est estimé à l'aide de la méthodologie COCOMO à 56 hommes.mois. De plus, cette dette technique ne présente que peu de risque.

Notons que la documentation interne du code est adéquate. Le code est donc adapté à une méthodologie de développement agile.

Un point d'attention serait la vérification que les règles métiers sont bel et bien implémentées dans le backend. En effet, le backend est très petit en comparaison des applications fontend. Il est important que ces règles soit positionnées dans le backend pour éviter de devoir maintenir plusieurs fois les mêmes règles métiers dans des composants et des technologies différentes.

Les soucis identifiés touchait principalement la maintenabilité du système, notamment la présente d'une duplication un peu trop élevé et sur un taux trop faible d'API documentées.

En conclusion, le système correspond au niveau de qualité attendu pour un système de cette taille. Nous n'avons pas identifié, à ce stade, de gros problèmes impactant le développement de l'application à court ou moyen termes.