UQAM

PROJET INDIVIDUEL

PAR

ERIC GAGNON - GAGE22037705

TRAVAIL PRÉSENTÉ

À

SÉBASTIEN MOSSER

MGL-7460
RÉALISATION ET MAINTENANCE DE LOGICIELS

12 AVRIL 2020

Introduction **4**

Cadre d'analyse de maintenance **4**

La dimension “Équipe de développement” 4

La dimension "Code source" 4

Dimension “Équipe de développement” **5**

Outils utilisés 5

Les développeurs principaux 5

Graphiques contributions (main dev per files) 5

L’Équipe “Core” 5

Ben Lesh : RxJs Project Lead, Senior Software Engineer 6

Contributions 6

OJ Kwon : Senior Software Engineer 6

Contributions 6

Nicholas Jamieson : front-end developer 6

Contributions 6

Paul Taylor : Software Engineer 6

Contributions 6

Autres membre de l’équipe “Core” 6

La relève 7

Les autres contributeurs 7

André Staltz 7

Contributions 7

Jason Aden 7

Contributions 7

La stabilité de l’équipe de développement 7

Nombre de contributeurs total versus actif 7

Nombre de mois de contributions 8

Paternité des composantes 8

Risques 9

Est-ce que Ben Lesh serait un risque pour le projet Rxjs ? 9

Bus Factor 9

Facteurs atténuants 9

Graphique - Paternité des composants: Individuelles et Équipe 10

Graphique - Pertes de connaissances 10

Dimensions “Code source” **11**

Outils utilisés 11

Les outils de construction utilisés 11

Reproductibilité de la compilation et des tests 11
Compilation et test avec l'intégration continue 11
Compilation et test en local 12
La qualité du code source 12
Métriques de Sonarqube 12
Maintenabilité 12
Duplication de code 13
Conclusion 13
Metriques de CodeScene 13
Technical Debt 13
Hotspots Refactoring / Defects 13
Composante Observable 13
Composante Subscription 15
Composante Testing 16
Hotspots Code Health 17
TestScheduler.ts 17
AjaxObservable.ts 18
WebSocketSubject.ts 18
Subscriber.ts 18
Conclusion **18**

Introduction

Ce travail est pour l'analyse du logiciel ReactiveX / RxJs afin d'estimer le risque d'accepter ce projet de tierce maintenance pour la compagnie.

ReactiveX est un API pour la programmation asynchrone en utilisant la combinaison des meilleurs concept du patron Observable, du patron Iterator et de la programmation fonctionnelle. ReactiveX est utilisé partout dans le développement logiciel, du backend au frontend, et il est multi-plateforme (Java, Scala, C#, C++, Clojure, JavaScript, Python, Groovy, JRuby, et autres).

Pour notre analyse c'est l'implémentation en Javascript (RxJs) qui sera étudié.

Cadre d'analyse de maintenance

Pour analyser la maintenance du logiciel RxJs je vais porter attention aux deux dimensions suivantes:

1. Dimension "Équipe de développement"
2. Dimension "Code source"

La dimension "Équipe de développement"

L'équipe de développement d'un logiciel est une partie importante de la pérennité d'un projet. Pour déterminer si l'équipe de RxJs est un enjeu majeur ou un atout pour le futur de ce projet, je vais répondre aux questions suivantes:

- Qui sont les développeurs principaux du projet ?
- L'équipe de développement est-elle stable ?
- Paternité des composantes

La dimension "Code source"

La qualité du code source d'un logiciel est très importante. Si la qualité du code est mauvaise elle peut mener à des pertes financières ou pertes de temps dû aux maintenances, modifications et ajustements. Pour valider la qualité du code je vais répondre aux questions suivantes:

- Quels outils de construction sont utilisés ?
- Dans quelle mesure la compilation du code est-elle reproductible ?
- Comment qualifiez vous la qualité du code source ?

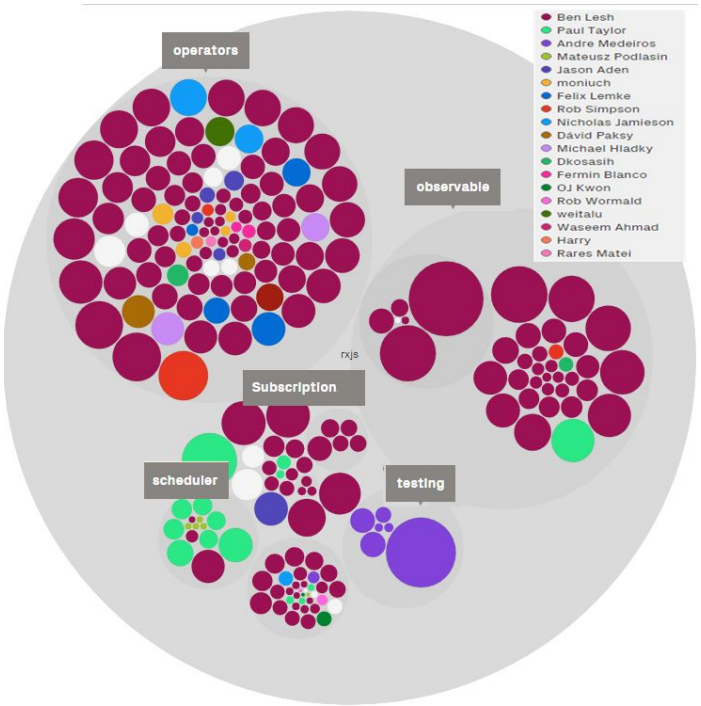
Dimension “Équipe de développement”

Outils utilisés

- La section “Team” du site web de RxJs (https://rxjs.dev).
- Pour les statistiques de contributions j’ai utilisé l’outils CodeScene (https://codescene.io/projects/7238/jobs/23304/results/) et la section “contributors” de GitHub .

Les développeurs principaux

Graphiques contributions (main dev per files)



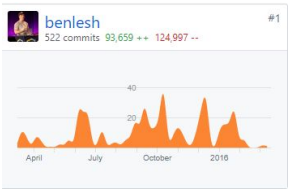
L’Équipe “Core”

Cette équipe est responsable du projet, de son évolution et des règles concernant les contributions et des règles conduites. Elle est aussi contributrice principale sur 4 (Observable, Operators, Subscription, Scheduler sauf Testing) des 5 modules principaux et possèdent toutes les connaissances requise pour la maintenance et l’évolution du projet.

Ben Lesh : RxJs Project Lead, Senior Software Engineer

Il est le principal contributeur et le “Project Lead” de RxJs. Par le passé il a contribué à des projets tel que Angular et React.

Contributions



Ben est contributeur depuis le tout début de RxJs le 12 mars 2015 et il est toujours très actif actuellement. Il est le contributeur le plus important de ce projet. Dans le graphique ci-dessus, vous pouvez constater l’ampleur de sa contribution au projet. Ses contributions touchent à la majorité des parties de l’architecture du projet. Il est l’auteur des composantes Observable à 59%, Operators à 43%, Subscription à 35%, Scheduler à 34% et Testing à 24%. Il est important de noter que la composante Observable est le coeur de l’application RxJs.

OJ Kwon : Senior Software Engineer

Contributions

OJ est le deuxième plus important contributeurs du projets. Il contribue au projet activement depuis le 26 août 2015. Quoique ses contributions ont été plus importantes au début du projet, il est toujours actif aujourd’hui. Il participe au développement des composantes Observable à 3%, Subscription à 6%, Scheduler à 2% et Testing à 3%.

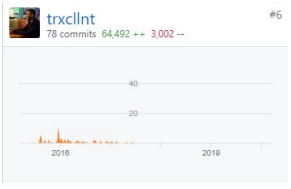
Nicholas Jamieson : front-end developer

Contributions

Il contribue au projet activement depuis le 12 août 2017 et il toujours actif sur le projet aujourd’hui. Il participe au développement des composantes Observable à 2%, Operators à 8%, Subscription à 2%, Scheduler à 3% et Testing à 3%.

Paul Taylor : Software Engineer

Contributions



Il a contribué au projet à partir du 28 juillet 2015 et il ne contribue plus sur le code source depuis le 14 juin 2017. Il a participé au développement des composantes Observable à 8%, Subscription à 27% et Scheduler à 35%. Nous comprenons pourquoi il est toujours membre de l’équipe “Core”. Il a contribué au projet de façon importante pendant deux ans.

Autres membre de l’équipe “Core”

Ces membres quoique importants, leur contributions sont moins importantes que les autres joueurs et je ne fournirai pas d’analyse de leur contribution au code source.

La relève

Il n'y a pas de documentation sur comment fonctionne cette équipe et quels sont les impacts sur le projet. De plus, aucun membre de cette équipe n'a encore contribué au projet selon les données provenant de Git.

Mais ce qui est intéressant c'est que cette équipe "Learning Team" est là pour s'assurer de la bonne continuité du projet dans le futur. Les connaissances des contributeurs principaux doivent être transmises pour que le projet continue son évolution.

Les autres contributeurs

En dehors de l'équipe "Core" il y a plusieurs contributeurs, mais je vous présente les deux plus importants.

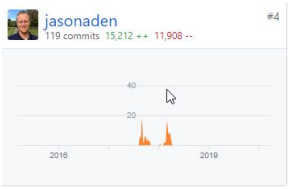
André Staltz

Contributions

André a contribué au projet activement du 30 octobre 2015 au 7 novembre 2016. Malgré sa courte période d'activité, il a participé au développement des composantes Observable à 2%, Subscription à 12% et Testing à 5%. Il a été très actifs pendant seulement 12 mois et ses contributions sont toujours très présentes dans le projet.

Jason Aden

Contributions



Jason a contribué au projet activement du 28 août 2017 au 5 avril 2018. Il a participé au développement des composantes Observable à 3%, Operators à 9%, Subscription à 1% et Scheduler à 1%. Sur le site de rxjs.dev, il est toujours dans la liste des contributeurs mais n'a pas fait de contribution au code source depuis 2018.

La stabilité de l'équipe de développement

Définissons la stabilité par le nombre de mois de contributions des principaux contributeurs.

Nombre de contributeurs total versus actif

Le nombre de contributeurs total depuis le début du projet est de 231 et il y en a actuellement 25 qui sont actifs.

Authors	231
Active Authors	25

Nombre de mois de contributions

Author	Added	Deleted	Net	Revisions	Months	Last Contribution
Ben Lesh	12755	11229	1526	421	52	2020-01-22
David Driscoll	570	405	165	27	40	2019-05-09
OJ Kwon	1196	774	422	100	34	2018-07-26
Alex Eagle	19	13	6	3	32	2018-10-10
Nicholas Jamieson	1029	832	197	56	28	2020-02-27
Jay Phelps	250	53	197	11	23	2018-08-27
SangKa.Z	3	24	-21	2	23	2019-05-11
Paul Taylor	1907	991	916	48	17	2017-06-14
Mateusz Podlasin	1389	230	1159	25	17	2018-07-27
Martin Sikora	269	228	41	7	15	2019-01-30
Martin Probst	6	4	2	3	15	2018-07-19
LongYinan	40	30	10	4	14	2018-01-23
Bowen Ni	5	5	0	3	14	2019-06-04
Andre Staltz	1361	277	1084	33	12	2016-11-07
Tetsuharu OHZEKI	337	231	106	31	10	2016-11-10
Rob Wormald	312	257	55	6	9	2016-07-12
Dzhavat Ushev	139	75	64	9	9	2019-12-24
Felix Becker	15	6	9	3	9	2018-08-27
Cotton Hou	4	4	0	2	9	2019-05-17
Michael Hladky	215	171	44	7	8	2019-12-01
Tim Deschryver	19	16	3	2	8	2019-05-10
Jason Aden	1272	1137	135	52	7	2018-04-05
Dkosasih	59	31	28	7	7	2019-04-23
Waseem Ahmad	84	4	80	5	6	2019-12-01
Jan-Niklas W	544	208	336	8	5	2018-11-27

Le nombre de mois de contribution moyen sur les 25 contributeurs les plus actif est de 17 mois (1,4 ans) et pour l'équipe "Core" cette moyenne est de 29 mois (2,4 ans). Le contributeur principal Ben Lesh est celui avec la plus grande stabilité (52 mois) car qu'il est actif de plus le tout début du projet.

La stabilité de l'équipe "Core" et des contributeurs en général est assez bonne si on compare à des compagnie tel que Uber (1.4 ans), DropBox et Tesla (2.1 ans), Square (2.3 ans), Facebook (2.5 ans). Les données proviennent de cette étude Silicon Valley techies get free food and dazzling offices, but they're not very loyal - here's how long the average employee stays at the biggest tech companies. De plus ces compagnies rémunèrent leurs employés grassement versus un projet open source.

Paternité des composantes

En regardant les graphiques ci-dessous, il peut être inquiétant de voir que Ben Lesh est le principal contributeur sur 3 des 5 composantes. De plus, les deux auteurs principaux des deux autres composants (Scheduler et Testing) Paul Taylor et Andre Medeiros ne sont plus actif dans le projet. Mais si on regarde

qui est le deuxième auteur en importance pour ces deux composantes, c'est Ben Lesh à 34% pour Scheduler et 24% pour testing.

Risques

Est-ce que Ben Lesh serait un risque pour le projet Rxjs ?

Je dirais que oui et non. S'il quitte le projet je suis certain que l'avancement de ce dernier sera perturbé mais la connaissance du projet est présente dans l'équipe.

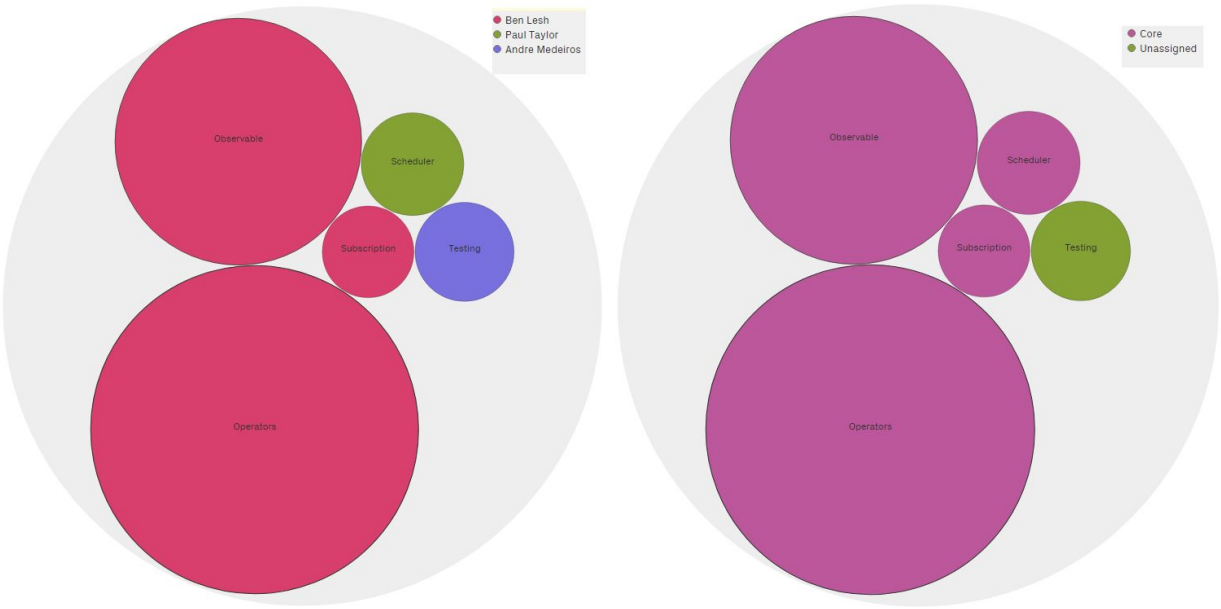
Bus Factor

Sommes nous en présence d'un bus factor de 1 ? Si on se fie aux chiffres de ses contributions, nous serions tenté de dire oui, mais d'un autre côté, il y a l'équipe "Core" et le "Learning Team" qui sont tout de même des contributeurs importants et qui sont présent pour la continuité du projet. Donc, je dirais que le Bus factor serait de 7 (nombre de personnes dans l'équipe "core") et ce c'est sans compter les contributeurs externes.

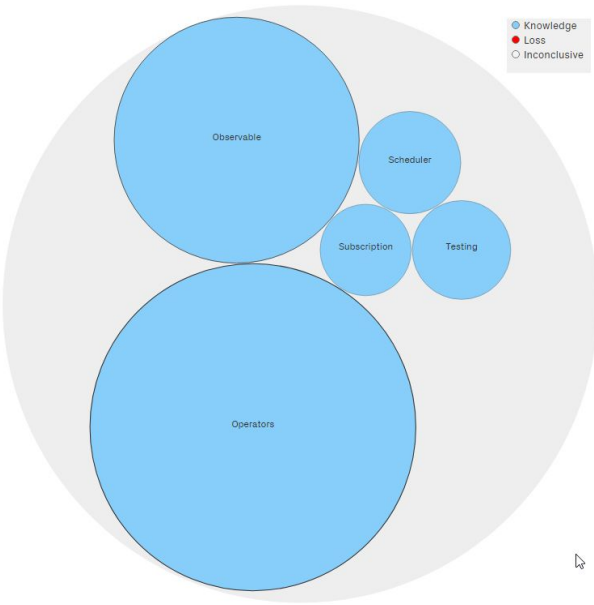
Facteurs atténuants

- L'équipe "Core": elle composée de plusieurs personnes d'expérience qui ont une très bonne connaissance du système.
- La "Learning Team": Équipe en apprentissage pour éventuellement contribuer au projet.
- Contributeurs externes: Plusieurs contributeurs actifs (25 actuellement).
- Knowledge Loss: Présentement, il n'y a aucune des composantes dont les connaissances ont été perdues par le départ d'un contributeurs. Le transfert de connaissances sur tout le système est possible actuellement. Voir graphique ci-dessous.

Graphique - Paternité des composants: Individuelles et Équipe



Graphique - Pertes de connaissances



Dimensions “Code source”

Outils utilisés

- Analyse de code statique avec Sonarqube (https://sonarcloud.io/dashboard?id=r_x_j_s).
- Analyse de l’historique de Git et Hotspots avec CodeScene (https://codescene.io/projects/7238/jobs/23304/results).

Les outils de construction utilisés

Le projet étant écrit en typescript, l’outil utilisé pour la compilation est le typescript compiler (tsc) qui est configurable avec un ou des fichiers “tsconfig.json”. Les scripts pour la compilation sont dans le fichier package.json et sont lancés à l’aide de NodeJs et NPM. Ils y a aussi un dossier “tools” qui contient plusieurs script javascript qui sont utilisés. Par exemple: ajouter le texte de la licence Apache 2.0 à tous les fichiers distribués ou générer le bundle UMD lors du packaging.

Pour les tests, le framework Mocha Js (qui roule sur Node.Js) est utilisé pour exécuter les tests unitaires et de Browser.

Autres tests de qualité:

- Validation des dépendances circulaire avec l’outils “Dependency cruiser”.
- Validation des règles de formatages de code avec “TsLint” pour améliorer la lisibilité, la maintenabilité et diminuer les erreurs de fonctionnalité.
- Validation des effets de bord lors de l’importation d’un module ES avec l’outil “Check side effects”.

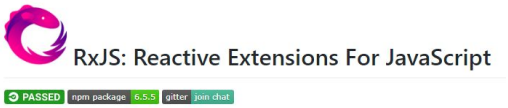
Il est important que tout soit scripté et que rien ne soit fait manuellement dans le processus de compilation et de test. Les outils utilisés sont la norme pour les projets de librairie web.

Reproductibilité de la compilation et des tests

Est-ce facile de générer un artefact de Rxjs ? Pour répondre à cette question je vais vous présenter comment est automatisé la compilation et tests avec le pipeline d’intégration continue (CI) et comment les reproduire à partir du code source sur un poste local.

Compilation et test avec l'intégration continue

L’équipe de RxJs utilisent CircleCi pour l’automatisation de la compilation et des tests. Dans la section “Readme” de leur GitHub il est possible de voir le résultat de leur pipeline et il est possible d’accéder à la liste de tous les builds en cliquant sur le tag de CircleCi.



Avec ce pipeline, nous sommes assuré que la compilation sera toujours la même parce que CircleCi utilise des images Docker pour avoir toujours le même environnement de compilation. De plus la configuration du pipeline évolue avec le code source et elle est conservée dans le dépôt Git. On retrouve la configuration du pipeline "config.yml" dans le dossier ".circleci". L'image utilisée pour la compilation est celle de Node:10.12. Les étapes du pipeline sont: build, lint, test et dtslint. Les commandes utilisées dans le pipeline sont contenues dans le fichier package.json (npm install, npm run build:package, npm run lint, npm run test, npm run test:side-effects) qui est à la racine du projet. Tous les scripts qui sont nécessaires pour la compilation et l'assurance qualité se retrouvent tous dans le même fichier (package.json).

Compilation et test en local

Pour exécuter la compilation et les tests en local la procédure est assez simple. RxJs est une librairie web et comme la plupart des librairies web il faut installer NodeJs sur notre poste. Pour le reste ce sont des commandes npm qu'il faut exécuter:

1. npm install (pour récupérer les dépendances). Cette étape est requise pour tous les projets utilisant du javascript.
2. npm run build:package (compilation du projet)
3. npm run test (tests de qualité)

Vous remarquerez que ce sont les mêmes commandes qui sont utilisées dans le pipeline d'intégration continue.

Il est donc assez simple de reproduire la compilation et les tests. Lors de commit sur le dépôt de code le processus de build et test se fait automatiquement, et sur un poste local il suffit que de quelques étapes pour arriver à les reproduire.

La qualité du code source

Pour répondre à la question "Comment qualifiez vous la qualité du code source ?", je vais utiliser l'analyse des outils SonarQube et de CodeScene pour voir s'il y a des symptômes qui affectent la qualité du code.

Métriques de Sonarqube

Maintenabilité

Dans la base de code de RxJs y a au total 77 "code smells". De ceux-ci 2 sont bloquant, 0 critique, 44 majeurs et les 31 restants sont mineurs. Cette dette technique représente environ

4H40 d’effort pour un développeur logiciel pour tous les corriger. Ceci nous donne un ratio de la dette technique de 0.1%. Selon SonarQube, ce ratio équivaut à une cote de maintenabilité de A.

Duplication de code

Le code dupliqué représente 1.5% de tout le code source. Ce qui est très peu.

Conclusion

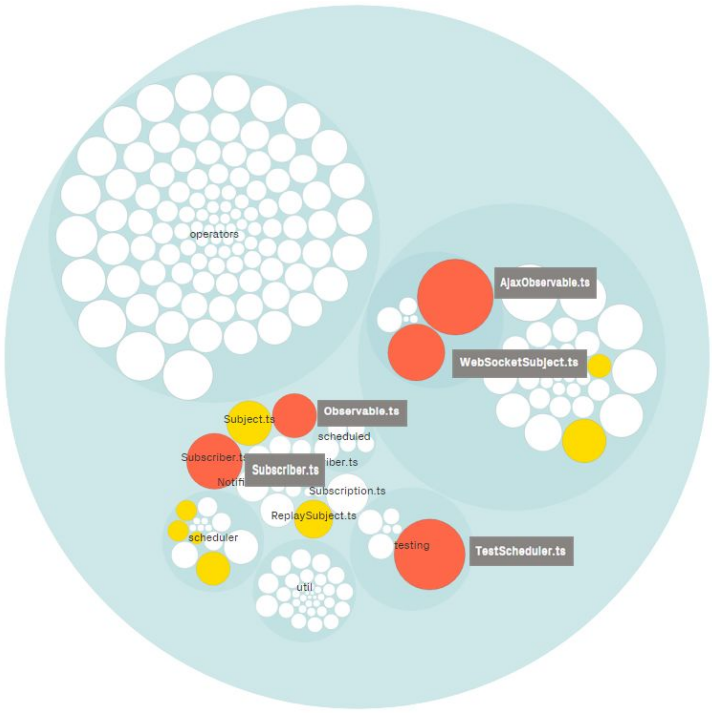
L’analyse statique du code de Sonarqube ne nous permets pas de trouver des éléments qui génèrent des problèmes de maintenance.

Metriques de CodeScene

Technical Debt

Pour l’analyse de la dette technique, j’ai ciblé les cinq fichiers sources les plus problématiques qui seront étudiés en détails. Dans le graphique ci-dessous, ils sont de couleur orange.

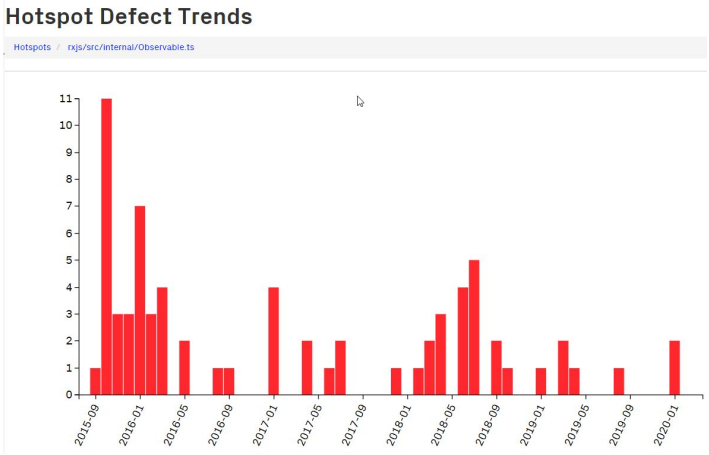
Hotspots Refactoring / Defects



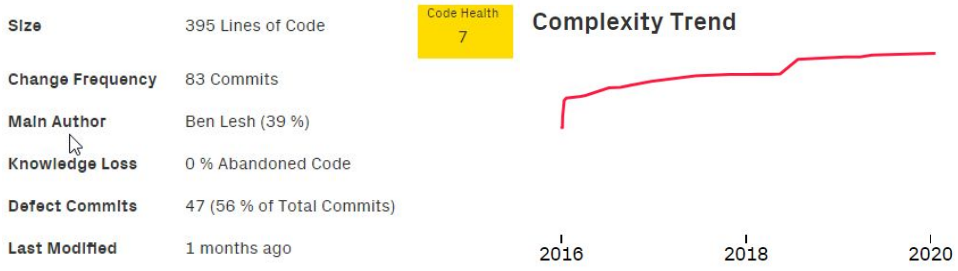
Composante Observable

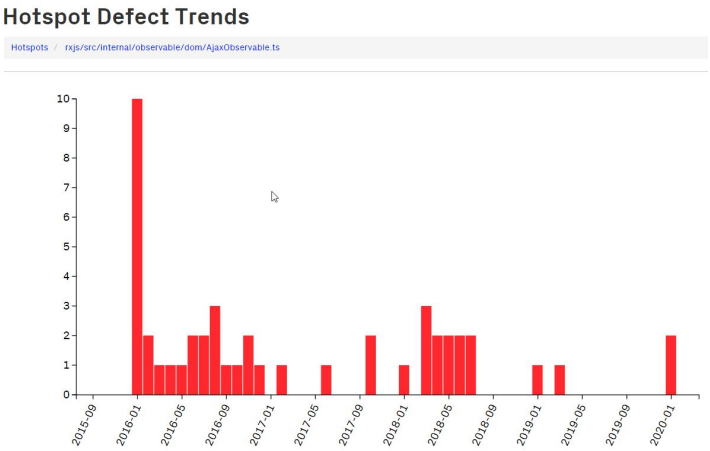
Le code du fichier observable.ts est celui qui a généré le plus de commit de correction de bugs. Ces bugs sont survenus majoritairement au début du projet. Nous pouvons voir une bonne corrélation du nombre de

bug avec la complexité du code. Il y a eu une grande diminution de la complexité en 2016 mais nous voyons que celle-ci a tendance à remonter. Il est une cible pour du refactoring.

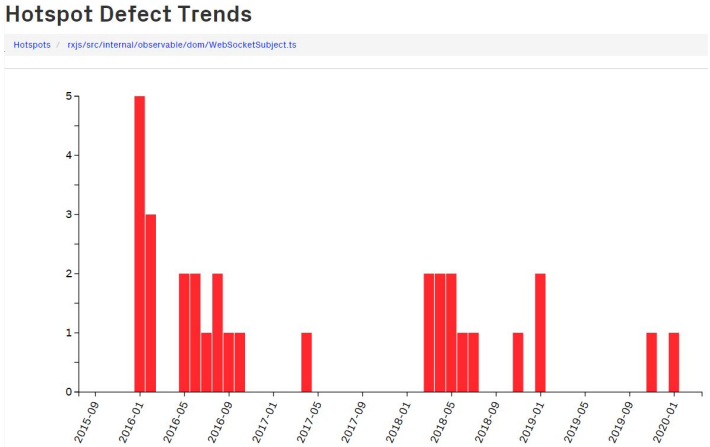


Le code du fichier AjaxObservable.ts lui aussi est une cible pour du refactoring. Sa complexité augmente progressivement depuis 2016 mais si nous regardons le nombre de bogues depuis 2019 est de 4. La majorité des problèmes sont antérieurs et réglés.





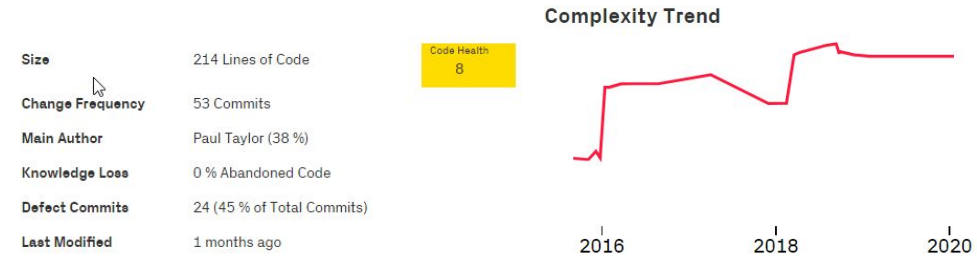
Le code source du fichier WebSocketSubject.ts lui aussi est une cible pour le refactoring. La complexité n'a pas augmenté énormément mais elle augmente lentement avec le temps. Le nombre de bogue depuis 2019 est de 4. La majorités des bogues sont antérieurs et ont été réglés.

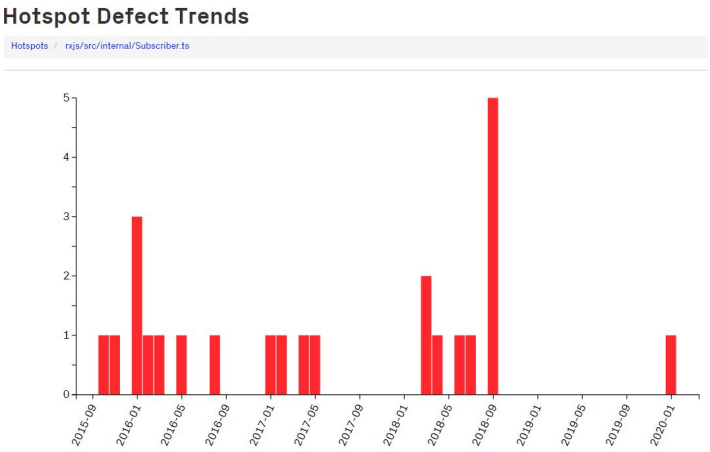


Composante Subscription

Dans cette composante, c'est le fichier Subscriber.ts qui est une bonne cible pour le refactoring. Sa complexité a augmenté de beaucoup depuis 2016. Par exemple, on peut voir qu'en 2018 il y a eu une augmentation significative de la complexité et du même coup le nombre de bogue lié à ce fichier. Depuis

2019 seulement 1 bug est lié à ce code ce qui est bon signe mais sa complexité croissante n'est pas encourageante.

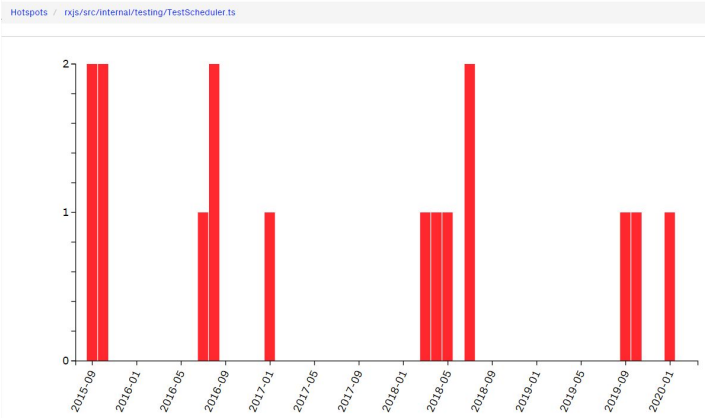




Composante Testing

Dans cette composante, c'est le fichier TestScheduler.ts qui est une bonne cible pour le refactoring. Sa complexité a augmenté de beaucoup en 2016 et 2018. Depuis 2019, il y a 3 bug est lié à ce code ce qui est bon signe si on compare aux "peaks" de 2015, 2016 et 2018.

Hotspot Defect Trends



Hotspots Code Health

Regardons pour ces mêmes fichiers sources, sauf pour Observable.ts qui a un code health de 10, les indications de “code smells”.

TestScheduler.ts rxjs/src/internal/testing/	6.4	6.4	6.4	Deeply Nested Logic Bumpy Road Ahead Brain Method Detected
AjaxObservable.ts rxjs/src/internal/observable/dom/	7.0	7.0	7.0	Bumpy Road Ahead Brain Method Detected Many Conditionals
WebSocketSubject.ts rxjs/src/internal/observable/dom/	7.7	7.7	7.7	Bumpy Road Ahead Brain Method Detected
Subscriber.ts rxjs/src/internal/	8.7	8.7	8.7	Bumpy Road Ahead Brain Method Detected

TestScheduler.ts

Deeply Nested Logic: The function TestScheduler.parseMarblesAsSubscriptions has a nested conditional depth of 6 (threshold: 4 levels deep). In addition, there are 1 other functions with deep conditional logic. Try to extract those nested conditions into named functions.

Bumpy Road Ahead: The code is complex to read due to its nesting with multiple logical blocks. The most complex function is TestScheduler.parseMarblesAsSubscriptions with 2 logical blocks. A bumpy road like TestScheduler.parseMarblesAsSubscriptions indicates a lack of encapsulation. Consider to extract smaller, cohesive functions from the bumpy functions.

Brain Method Detected: The function TestScheduler.parseMarbles has a McCabe complexity of 30 with 96 lines of code. The recommended complexity threshold is 9.

AjaxObservable.ts

Bumpy Road Ahead: The code is complex to read due to its nesting with multiple logical blocks. The most complex function is AjaxSubscriber.setupEvents with 4 logical blocks. A bumpy road like AjaxSubscriber.setupEvents indicates a lack of encapsulation. Consider to extract smaller, cohesive functions from the bumpy functions.

Brain Method Detected: The function AjaxSubscriber.setupEvents has a McCabe complexity of 19 with 93 lines of code. The recommended complexity threshold is 9.

Many Conditionals: The average function complexity is 4.55, and the total complexity in the file is 91 (McCabe)

WebSocketSubject.ts

Bumpy Road Ahead: The code is complex to read due to its nesting with multiple logical blocks. The most complex function is WebSocketSubject.constructor with 2 logical blocks. A bumpy road like WebSocketSubject.constructor indicates a lack of encapsulation. Consider to extract smaller, cohesive functions from the bumpy functions.

Brain Method Detected: The function WebSocketSubject._connectSocket has a McCabe complexity of 19 with 95 lines of code. The recommended complexity threshold is 9.

Subscriber.ts

Bumpy Road Ahead: The code is complex to read due to its nesting with multiple logical blocks. The most complex function is SafeSubscriber.error with 2 logical blocks. A bumpy road like SafeSubscriber.error indicates a lack of encapsulation. Consider to extract smaller, cohesive functions from the bumpy functions.

Brain Method Detected: The function SafeSubscriber.error has a McCabe complexity of 10 with 29 lines of code. The recommended complexity threshold is 9.

Il n'est pas surprenant que ces quatres fichiers sources font partie des cibles de refactoring. La complexité de ces fichiers est due à:

- l'imbrication de plusieurs blocs de logiques
- un manque d'encapsulation
- une complexité de McCabe élevée (de 10 à 30)
- fonctions avec un nombre de ligne de code élevé (de 29 à 96)

La cause à effet est assez visible, la complexité versus le nombre de bugs.

Conclusion

Suite à l'analyse des dimensions équipe de développement et code source de la librairie web RxJs, j'ai en ma possession assez de données pour donner ma recommandation sur l'état de ce logiciel.

Veuillez tenir compte que, même si je n'ai pas analysé en détails les dimensions architecture logicielle, tests, déploiement et livraison, j'ai tout de même des données intéressantes sur ces sujets me permettant d'appuyer ma recommandation.

Je recommande d'accepter le projet Rxjs en tierce maintenance pour les raisons décrites ci-dessous.

Revenons sur l'équipe de développement principale ("Core"). Elle est engagée dans le projet par le nombre de mois et d'années de contributions de ses membres, elle est stable (comparable à certaines

compagnies de Silicon Valley) et possède toutes les connaissances actuelles du projet. Je ne considère pas de risque lié à la perte d'un joueur clé (même si c'est Ben Lesh) compte tenu que c'est l'équipe principale qui est en charge du projet. Point positif à ajouter, est l'équipe en apprentissage ("Learning team"), qui nous assure d'une relève avec le transfert de connaissance et la communauté de contributeurs qui est aussi très impliqué, il y en a présentement 25 d'actifs.

Pour la partie code source, plusieurs points forts et certains moins, mais en général c'est un projet en bonne santé. Les outils utilisés pour la construction et test sont standard et reconnus dans le domaine. Il est facile pour un développeur web de comprendre comment lancer la compilation et les tests. L'automatisation est bien en place avec le logiciel CircleCi. La qualité du logiciel est prise au sérieux parce qu'il y a différentes stratégies de tests automatisées (test unitaires, browser, dépendances circulaires). Il ont même leur propre librairie pour tester les observable et les différents opérateurs. Il est certains que tout n'est pas parfait. Il y a des points chauds dans la base de code qui demanderont du refactoring mais ce n'est pas généralisé. Au total il y a 221 fichiers sources et environ 6 sont plus problématiques (complexité du code surtout).

Les autres points positifs de ce projet sont la documentation sur le Github et le site web officiel qui sont très bien fait, sans compter le nombre de ressources d'apprentissage sur le web. C'est un projet très en vue, il y a eu 19.8 millions de download dans la dernière semaine sur NPM. Il est utilisé dans des projets sérieux tel que Angular.