DOCUMENT À DUPLIQUER AVANT DE MODIFIER
(Mais ne pas hésiter à enrichir)

Accenture Digital

Audit de code

...

- CLIENT -

NUMERO DE REF OCTOPOD - Version N - mercredi 3 janvier 2020

Executive Summary

✔ Bon point

✔ Bon point

? Moyen

? Moyen

X Mauvais point

X Mauvais point

Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat

Evaluation de l'application:

01

Contexte et demande

02

Méthodologie d'audit

03

Résultats d'analyse

04

Synthèse et recommandations

05

Annexes

01

Contexte et Demande

Contexte et Demande V1

CONTEXTE

- xxx a mandaté la société xxx en 2016 pour réaliser l'application xxx
- Une première version de l'application a été livrée à xxx en octobre 2016, après 6 mois de développement
- L'application est alors passée en MCO, toujours chez xxx
- Cette première version livrée n'a jamais été utilisée par xxx , en raison de fonctionnalités non implémentées. De plus, avec un volume de données plus représentatif, des problèmes de performances sont apparus
- En juillet 2018, le XXX a été transféré à la société Scalian, qui refuse alors de s'engager au maintien de l'application à la suite d'un premier audit remontant la mauvaise qualité du code
- En novembre 2018, xxx mandate OCTO Technology pour un second avis

DEMANDE

- Évaluer la qualité de l'application et l'évolution de celle-ci entre la version d'octobre 2016 et celle de juillet 2018
- Évaluer l'avenir de l'application selon 3 scénarios :
  - Continuer le MCO à partir de la version d'octobre 2016
  - Continuer le MCO à partir de la version de juillet

xxx2.0

- Chez xxx, chaque recrutement, augmentation ou mutation doit être présenté devant les instances représentatives du personnel, dans ce qui est appelé une Commission Secondaire du personnel (xxx)
- Afin de faciliter ces xxxs, il est aujourd'hui indispensable pour xxx d'avoir une application pour en gérer les comptes rendus
- Face à l'absence de produit du marché répondant aux besoins fonctionnels de xxx, il est décidé de réaliser une application sur mesure : xxx2.0

Contexte et Demande V2

Contexte

Demande

- Description du contexte

- Demande

Équipe OCTO

Prénom Nom

RÔLE

Description du rôle

Lorem Ipsum. Ex quod recteque interrogatus eas ardenter curant eas ob plus Censorius quidam honeste sensu inquit delatum ad sit his superasset commendari est his figmentis regem autem figmentis per recteque his.

Prénom Nom

RÔLE

Description du rôle

Lorem Ipsum. Ex quod recteque interrogatus eas ardenter curant eas ob plus Censorius quidam honeste sensu inquit delatum ad sit his superasset commendari est his figmentis regem autem figmentis per recteque his.

02

Méthodologie d’audit

► THERE IS A BETTER WAY ◄

Démarche

1 Kickoff

- Réunion de lancement
- Première analyse documentaire
- Présentation fonctionnelle de l'application
- Entretien avec les parties prenantes : le sponsor de la mission, l'équipe de dév.

LIVRABLES

- Alignement avec *** sur le déroulement de la mission

2 Analyse technique et pratiques de dév.

- Analyse outillée du code et de l'architecture applicative
- Analyse approfondie du code par carottage

LIVRABLES

- Diagnostic technique

3 Préconisations et restitution

- Préconisations vis-à-vis des constats techniques au regard de l'état de l'art
- Maintenabilité de l'application
- Synthèse de l'audit
- Restitution finale

LIVRABLES

- Synthèse globale de l'audit et des préconisations

Détail des activités de l'audit

PRÉREQUIS

- Les différents éléments seront impérativement nécessaires à la mission :
  - Le code source, ou un accès au gestionnaire de sources
  - Toute la documentation possible
  - Disponibilité des personnes

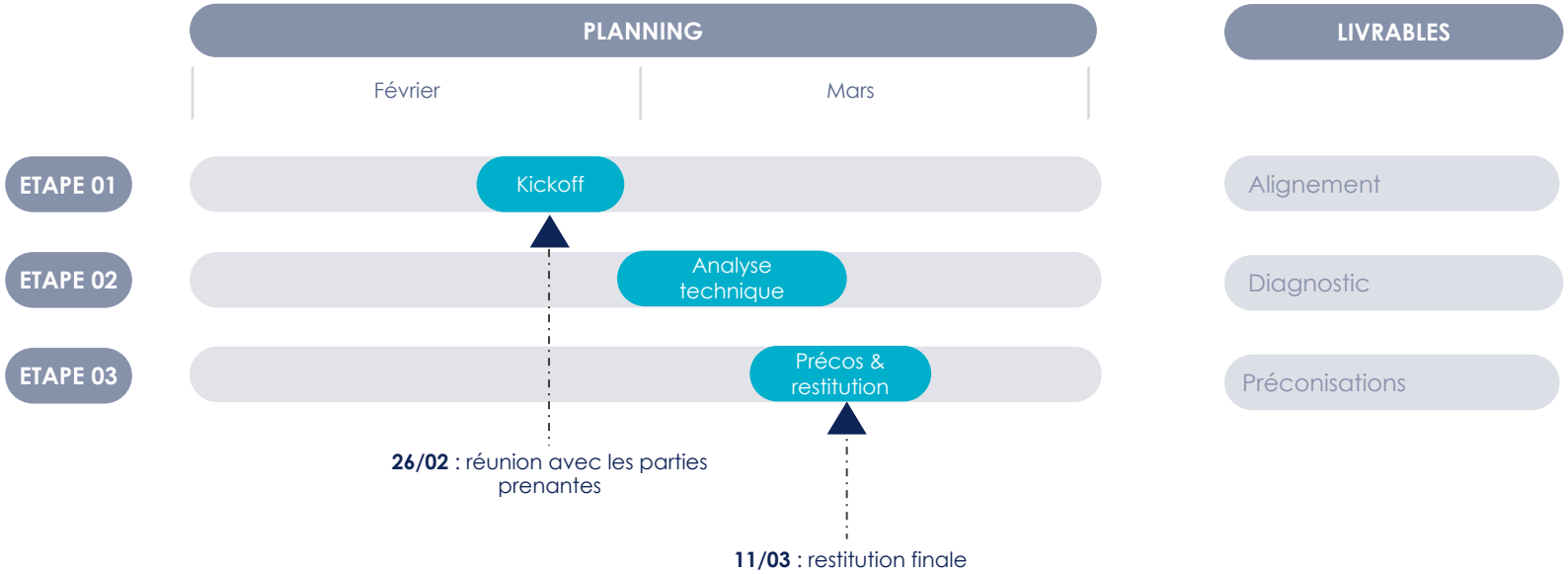
DÉMARCHE OCTO

- Analyse de l'architecture

Objectifs : comprendre la conception générale du produit et son évolutivité ainsi que la pérennité des choix techniques

- Analyse du code spécifique pour évaluer la scalabilité et la maintenance des applicatifs
  - Audit de code via une usine d'audit et par échantillonnage sur chaque application

Planning proposé



Architecture - échelle de notation

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art

Axes	Echelle de notation
Choix d'architecture	0 : l'architecture est non identifiable voir absente, 1 : l'architecture est difficilement identifiable et il y a un mélange de plusieurs concepts, 2 : l'architecture est découpée de manière cohérente, 3 : l'architecture a bien été pensée et implémente des pattern éprouvés, 4 : l'architecture est autoportante et utilise des concepts avancés.
Robustesse	0 : en cas de défaillance, l'application ne peut pas être relancée, 1 : en cas de défaillance, l'application offre partiellement l'accès aux services, 2 : en cas de défaillance, l'application offre des moyens de relance manuelle, 3 : en cas de défaillance, l'application offre des moyens de reprise automatique , 4 : en cas de défaillance, l'architecture de l'application permet de réagir aux pannes avec des mécanismes définis et automatisés.
Fiabilité	0 : l'application ne répond pas au besoin fonctionnel, 1 : l'application répond partiellement au besoin fonctionnel et ne garantit pas l'intégrité des données, 2 : l'application répond au besoin fonctionnel mais il y a quelques écarts dans l'attendu, 3 : l'application répond au besoin fonctionnel, 4 : l'application répond au besoin fonctionnel et garantit l'intégrité des données véhiculées en entrée, durant le traitement et à la sortie de l'application.
Evolutivité	0 : l'architecture rend les évolutions impossibles, 1 : l'architecture rend les évolutions difficiles, 2 : l'architecture rend les évolutions abordables, 3 : l'architecture rend les évolutions faciles, 4 : l'architecture de l'application permet de répondre de façon efficace et rapide aux besoins d' évolution du système.

Modélisation - échelle de notation

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art

Axes	Echelle de notation
Modularité	0 : il y a un couplage très élevé entre le métier et l'implémentation technique, 1 : il y a un couplage relativement élevé entre les couches, 2 : il y a peu d'adhésion entre les couches, 3 : l'adhésion entre les couches est faible, 4 : il y a un très bon découplage des composants et l'adhésion entre les couches est inexistante.
Responsabilité	0 : les composants réalisent des actions qui ne sont pas de leur responsabilité, 1 : les composants font trop de choses et qui ne sont parfois pas de leur responsabilité, 2 : les composants réalisent des traitements hors et dans leur périmètre de responsabilité, 3 : les composants réalisent des traitements uniquement dans leur périmètre de responsabilité, 4 : les rôles d'orchestration, de traitement sont bien séparés. L'infra et le métier sont bien séparés.
Performances	0 : l'application ne répond pas du tout aux critères d'acceptance du métier, 1 : l'application ne répond pas à 100% aux attentes du métier, 2 : l'application répond relativement bien aux attentes du métier, 3 : l'application répond bien aux attentes du métier, 4 : l'application répond au delà des attentes du métier.
Dépendances	0 : les frameworks ou librairies sont injustifiés et inutiles, et/ou sur des versions obsolètes, 1 : les frameworks ou librairies ne répondent pas au besoin fonctionnel, 2 : les frameworks ou librairies répondent au besoin fonctionnel, avec certaines mises à jour nécessaires, 3 : les frameworks ou librairies sont bien choisis et à jour. Certain(e)s pourraient être remplacé(e)s, supprimé(e)s, mises à jour, 4 : les frameworks ou librairies sont justifiés. Il n'y a pas de librairies superflues. Les versions sont à jour.

Lisibilité - échelle de notation

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art

Axes	Echelle de notation
Simplicité	0 : le code est illisible et incompréhensible, 1 : le code est difficilement lisible et compréhensible, 2 : le code est compréhensible mais la lisibilité peut être améliorée, 3 : le code est parfaitement lisible et compréhensible, 4 : le code est autoportant et il n'y a aucun doute sur les compoxxxments.
Conventions	0 : le code est complètement hétérogène et il n'y a pas de standards de développement, 1 : le code est hétérogène et il y a peu de standards de développement, 2 : le code est relativement homogène, 3 : le code est homogène et traduit des conventions et des standards bien marqués / respectés, 4 : le code est homogène et traduit un collective ownership évident avec des standards d'équipe.
Commentaires	0 : les commentaires noient le développeur dans le code et ne sont pas justifiés, 1 : les commentaires sont trop présents, 2 : les commentaires ne sont pas utilisés de manière abusive et sont justifiés, 3 : certaines portions de code pourraient être améliorées pour éviter les commentaires, 4 : les commentaires sont utilisés à bon escient. Le code est suffisamment autoportant pour éviter les commentaires superflus.
Code inutilisé	0 : plus de 50% de l'application contient du code mort, 1 : il y a beaucoup de code mort, 2 : il y a relativement peu de code mort, 3 : il y a très peu de code mort, 4 : il n'y a aucun code mort.

Maintenabilité - échelle de notation

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art

Axes	Echelle de notation
Débogage	0 : les phases de débogage sont extrêmement difficiles voir impossibles (logs inexploitables…), 1 : les phases de débogage ne se font pas sans douleur, 2 : les phases de débogage se font relativement avec peu de difficulté, 3 : l'application est pensée de manière à faciliter les phases de débogage, 4 : l'application remonte des traces pertinentes et permet de déboguer rapidement.
Dette technique	0 : il y a trop de dette technique accumulée et très complexe à reprendre sans tout casser, 1 : il y a beaucoup de dette technique et relativement complexe à reprendre, 2 : il y a de la dette technique acceptée et reprenable, 3 : il y a très peu de dette technique et celle-ci est facilement reprenable, 4 : il n'y a pas de dette technique.
Couplage faible	0 : il y a un couplage fort entre les composants, 1 : il y a un couplage élevé entre les composants, 2 : il y a un couplage faible entre les composants, 3 : il y a un couplage très faible entre les composants, 4 : il n'y a pas de couplage et il est très facile de basculer d'une librairie/framework à un autre.
Complexité	0 : le code est incompréhensible et la complexité n'est pas bien répartie, 1 : le code est abordable mais demande du temps pour l'assimiler, 2 : le code est compréhensible et peu complexe, 3 : le code est bien découpé et il y a très peu de dépendances inutiles. La complexité cyclomatique du code est relativement faible, 4 : le code est très bien découpé et les dépendances justifiées. La profondeur des appels n'est pas élevée et la charge cognitive requise n'est pas conséquente.

Testabilité - échelle de notation

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art

Axes	Echelle de notation
Testabilité du code	0 : le code n'est pas du tout testable (couplage fort des composants…), 1 : le code est testable difficilement (uniquement des tests d'intégration par exemple), 2 : le code est testable mais certaines portions de l'application ne le sont pas, 3 : le code est testable facilement même si la couverture de tests à 100% n'est pas atteignable, 4 : le code est testable sur tous les aspects (injection de dépendances, mocks, dépendances limitées).
Modélisation des tests	0 : il n'y a aucuns tests, 1 : il y a très peu de tests qui valident les compoxxxments de bout en bout, 2 : la pyramide des tests n'est pas complète, 3 : la proportion de tests est équilibrée : unitaires, intégration, end-to-end. D'autres niveaux peuvent être intégrés, 4 : la pyramide des tests est respectée.
Couverture	0 : la couverture de tests est égale à 0%, 1 : la couverture de tests est inférieure à 50%, 2 : la couverture de tests est correcte et supérieure à 50%, 3 : la couverture de tests est comprise entre 50% et 95%, 4 : la couverture de tests est comprise entre 95% et 100%.
Pertinence	0 : les tests posés sont inutiles, 1 : les tests posés ne couvrent pas les fonctionnalités critiques de l'application, 2 : les tests posés couvrent relativement bien le périmètre fonctionnel attendu de l'application, 3 : les tests posés sont pertinents mais certains pourraient être améliorés, supprimés ou intégrés dans le niveau inférieur/supérieur de la pyramide, 4 : les tests posés sont pertinents à 100%.
Automatisation	0 : il n'y a pas d'automatisation de la chaîne de tests, 1 : l'exécution des tests nécessite un déclenchement manuel, 2 : la pyramide de tests n'est pas complètement automatisée (uniquement les tests unitaires par ex.), 3 : la pyramide de tests est relativement complète et l'ordonnancement du lancement de ceux-ci peut être revu, 4 : la pyramide de tests est complètement automatisée : unitaires, intégration, end-to-end, sécurité, robustesse, performances.

03

Résultats d’analyse

Nos constats

Architecture

- L'architecture est basée sur des librairies spécifiquement développées pour des fonctionnalités standards (ex : BaseAdminController#zip)
- La partie front-end dynamique développée à la main est la plus endettée de l'application (duplication de librairies, pas de gestion de versions, css entremêlé, couplage fort JS/HTML/CSS)
- Il n'y a pas d'usine logicielle (CI/CD) permettant d'assurer la cohérence de l'application dans le temps

Modélisation

- On peut constater une foxxx adhésion entre les différentes couches (ex : BaseController qui contient aussi de la logique métier et de la logique de persistance).
- Les composants intègrent un nombre très important de fonctions (ex : *simulateur.php* intègre 142 fonctions en 2 400 LOC. Une des fonctions compte 600 LOC).
- Il n'y a pas actuellement de signe évident de problèmes de performances (attention cependant sur la persistance : xml et sqlite peuvent représenter un risque en cas d'augmentation des flux)

Lisibilité

- L'abondance de commentaires dans le code, pas toujours justifiée, permet d'aider à la compréhension globale (27,9% de commentaires sur la partie PHP)
- La foxxx adhésion entre les couches et la taille des classes/fonctions rendent la lisibilité compliquée (ex : SimulatorsAdminController qui fait du routing, de la persistance, du métier, …)
- On retrouve des concepts métiers dans le code, ce qui est une bonne chose (ex : SimulatorsAdminController, DatasourcesAdminController, ViewsAdminController, …)

Nos constats

Maintenabilité

- L'application a été développée par une seule personne qui est partie, en emportant avec elle sa connaissance de l'application
- Cette contrainte est renforcée par le fait que peu de librairies standards aient été utilisées (le code à maintenir est plus vaste)
- Par ailleurs, le regroupement de la logique applicative dans des classes de plusieurs milliers de lignes rend le travail d'identification, de corrections des anomalies et d'analyse d'impact plus complexe

Testabilité

- Il n'existe pas aujourd'hui de stratégie de tests bien identifiée.
- Les tests sont réalisés manuellement : il n'y a pas d'outils d'automatisation (du type PHPUnit, Codeception, ...)
- Le mélange des responsabilités au sein de mêmes classes (traitement technique de la requête, traitement fonctionnel de la requête, persistance) rend le code difficilement testable

Architecture - KPIs

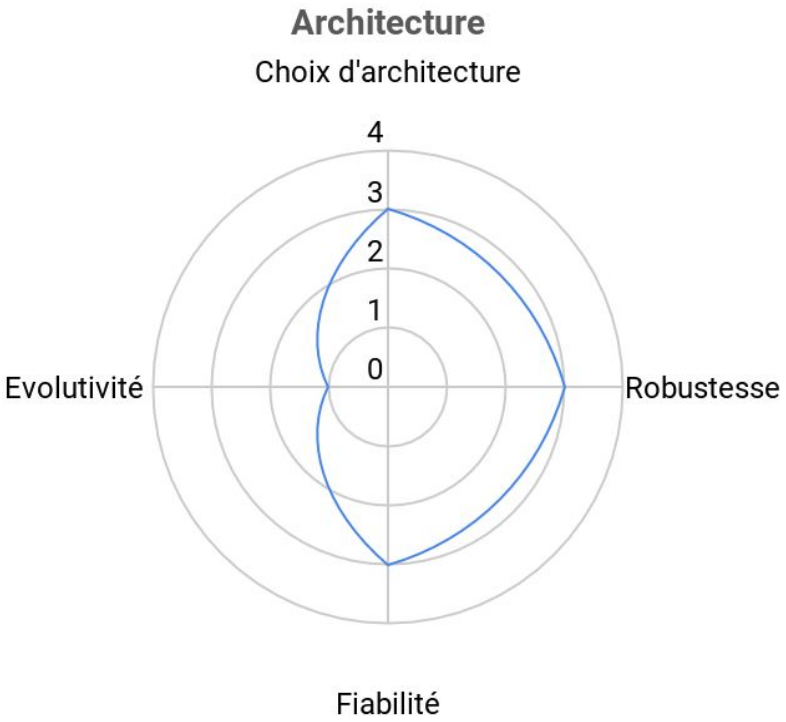
KPIs	Statut	Observations
Contraintes Standards imposés ? code existant ?		• Base de code écrite par un seul développeur, parti à la retraite
Contexte Nb utilisateurs, trafic observé/prévu ? Système tierce/externe ?		• Utilisateurs prestataire externes (5<>10) • Nombre de calculs/générations par semaine ?
Technologies Frameworks, langages utilisés ? stockage ?		• PHP v7.2, SYMFONY v4.2 (dernières versions majeures) • Stockage dans des fichiers XML et SQLite • Front: peu de librairies, tout est fait en JS + JQuery
Usine logicielle Outils utilisés pour le build et le déploiement		• Les synchronisations se font par échange de mails • Pas de CI/CD. CI/CD en projet sur le modèle de SP. Le partage des responsabilités n'est pas établi • Déploiement à la main

Interactions
Comment les données sont transmises/échangées entre les composants (HTTP, REST, BDD, ...) ?
		• Peu d'interactions identifiées :
		  ○ Encapsulation HTML
		  ○ API Web non décrites
		  ○ Stockage dans des fichiers XML
		  ○ L'authentification est gérée dans G6K

Prod Backups réguliers ? Métriques de prod suivies ? ELK ?		• Backup à la main du code en Prod (reverse jamais testé) • Pas de backup des données • Pas de monitoring de prod

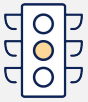
Architecture - Évaluation Finale

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art



Axes	Note
Choix d'architecture	1
Robustesse	1
Fiabilité	2
Evolutivité	1

Modélisation - KPIs

KPIs	Statut	Observations
Composant Rôle de chaque composant du système ? Clair et explicite ?		• Un composant faisant le stockage, le traitement et le rendu • Des fichiers statiques pour l'aspect dynamique et la mise en page se trouvent dans le dossier /calcul
Layering Layered architecture, hexagonal, CQRS, ...?		• On retrouve une séparation Model/Vue/Controller • Mais les responsabilités de chaque couche ne sont pas claires
Choix des frameworks/librairies Pour résoudre un problème générique ou réinvente la roue ?		• Symfony est utilisé dans son plus simple appareil • Le front-end aurait mérité l'utilisation d'un framework et/ou librairies afin de gérer l'aspect dynamique • Des librairies auraient pu être utilisées plutôt que de refaire à la main (zip, sql, moteur de règle, ...)
Conception Modèle objet adapté aux use cases ? Utilisation de getter/setter ? De null ?		• Les modèles semblent mappés sur la représentation des tables/XML • Certaines classes dédiées aux modèles font trop de choses et ne sont pas limitées à leur seul périmètre de responsabilité
Délégation flux d'exécution respecté ? (orchestration au début/traitement spécifique à la fin)		• Peu de délégation, le code se trouve souvent regroupé dans les même classes (ex: SimulatorAdminController)

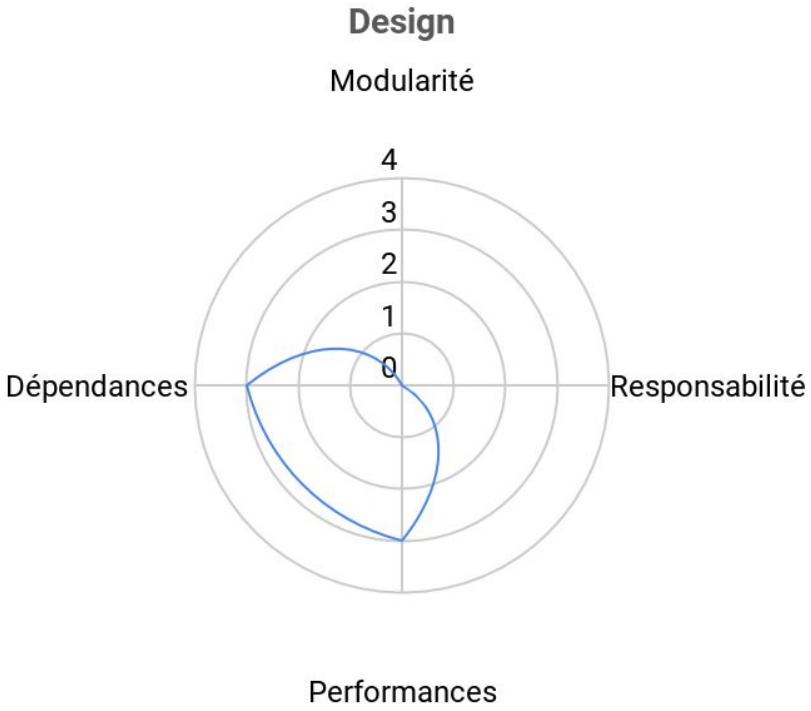
Modélisation - Observations Supplémentaires

- **Exemple de *SimulatorsAdminController* :**
  - Gère un état interne (via les attributs de classe $simu, $dataset, …) alors qu'un contrôleur devrait être stateless.
  - Gère les verbes HTTP à la main plutôt que d'utiliser le framework pour s'occuper du routing (indexAction l.171).
  - S'occupe de la création, mise à jour, suppression des fichiers XML de stockage des simulateurs (doDelete l.662, doRename l.691, … devrait être géré dans une couche de persistance, via des Repositories par exemple).
  - Assure la transformation de données métier en données envoyable par la requête (serialization).
  - Créait des requêtes SQL à partir de données en entrée (composeSimpleSQLRequest l.1398, très technique et plutôt de la responsabilité de la couche de persistance, via un service/librairie spécifique).
  - Assure le déploiement de simulateur sur serveur (doDeploySimulator l.1633).
  - Contient de la logique de création de BusinessRules (makeBusinessRule l.1156, rôle du Domain).
  - Contient de la paramétrisation (ex: getRegionalSettings l.3911).

- **Exemple de *BaseAdminController l.103***
  - Création d'un zip "à la main", alors que l'utilisation d'une librairie pour ce type de fonction permettrait de réduire le nombre de LoC complexes à gérer.

Modélisation - Évaluation Finale

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art



Axes	Note
Modularité	0
Responsabilité	0
Performances	3
Dépendances	1

Lisibilité - KPIs

KPIs	Statut	Observations
Taille Petit module, petite classe, petite fonction ?		• Les classes liées à la simulation sont beaucoup trop grosses (SimulatorAdminController, Simulator) • Certains contrôleurs et modèles compoxxxnt trop de logique (Simulator, BusinessRule, DataBase, BaseController, DataSourcesAdminController) • Des fonctions à plus de 200 lignes (BaseController, DDLParser, Simulator, …)
Nommage Non abrégé ? explicite (package, classe, …) ?		• Le nommage des packages n'est pas très explicite • Les Controllers sont nommés en fonctions des pages utilisateur (plutôt explicite) • Certaines classes n'expriment pas leurs intentions (Simulator)
Commentaires pertinents et utiles ? à jour ?		• Beaucoups de commentaires de fonctions, qui n'appoxxxnt pas toujours de la valeur, d'autant plus que les dernières versions de PHP inclus les types de retour et de paramètres
Standards de développement Standards d'équipe/entreprise respectés ? Code homogène ?		• Code "homogène" parce qu'écrit par un seul développeur, mais pas de standards de développement
Conventions de code Des langages/frameworks sont suivis ?		• 1,3k "code smells" SonarQube (dont 616 de "magic strings") • 500 "code smells" sur le JS
Code mort Code commenté ? Code inutile (méthodes, variables, …) ? Dépendances inutiles ?		• Très peu de code mort et/ou commenté

Lisibilité - Observations Supplémentaires

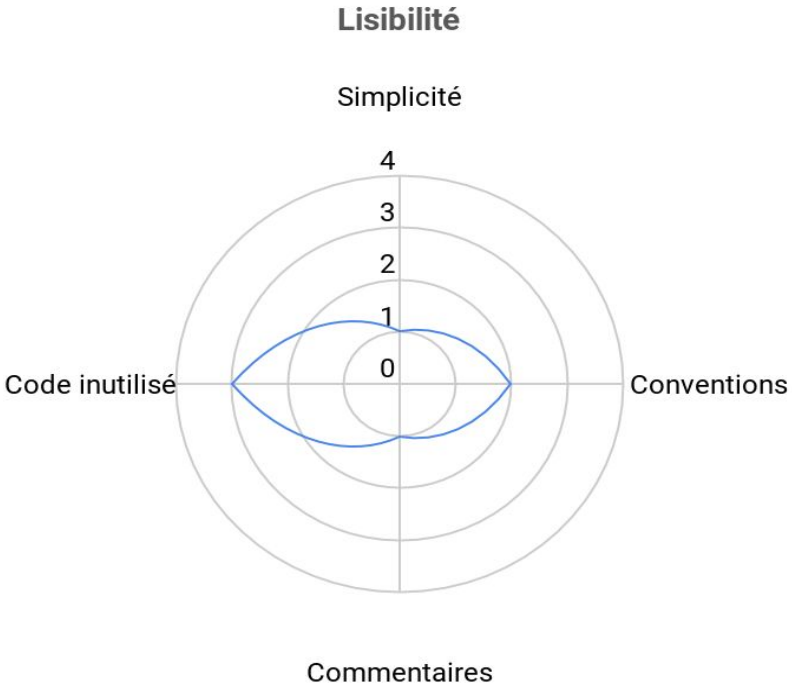
- **Certains fichiers, certaines classes contiennent beaucoup de lignes de code :**
  - BaseController => 1400 LOCs
  - DatasourceAdminController => 1544 LOCs
  - SimulatorsAdminController => 3981 LOCs
  - ControllersTrait => 973 LOCs
  - Simulator => 3692 LOCs
  - g6k.admin.simulators.sources.js => 2400 LOCs

D'après les standards de qualité logicielle, un fichier ne devrait pas contenir plus d'~300 LoC

- Des fonctions peuvent être très longues
  - Simulator#save => environ 600 LOCs
  - BaseController#runStep => environ 300 LOCs

Lisibilité - Évaluation Finale

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art

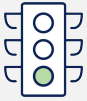


Axes	Note
Simplicité	1
Conventions	2
Commentaires	1
Code inutilisé	3

Maintenabilité - KPIs

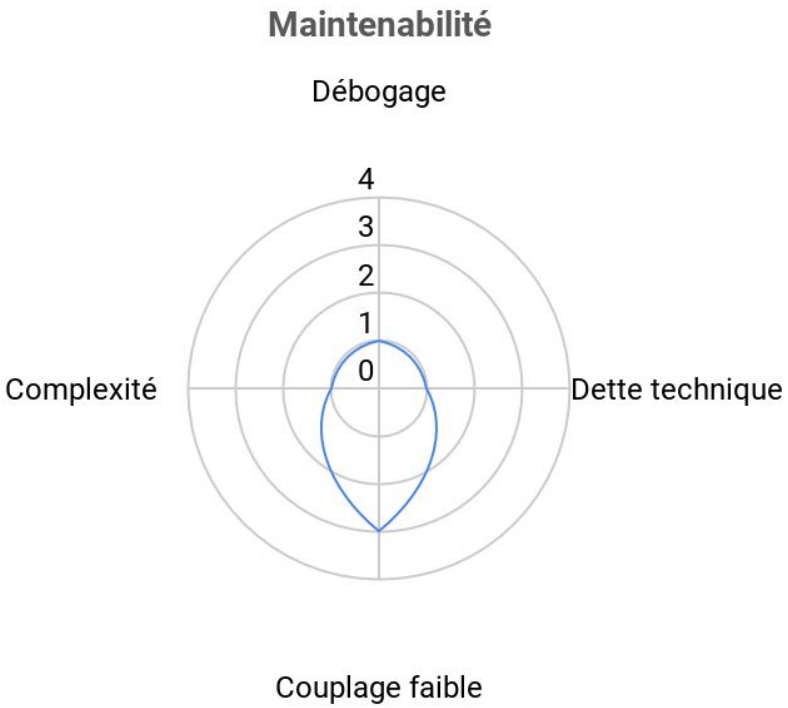
KPIs	Statut	Observations
Dépendances internes entre les modules, entre les packages ? que des classes publiques ?		• Peu de dépendances internes, le code étant regroupé dans de grosses classes
Logging Qualité des logs ? monitoring des points critiques ? exceptions nommées ?		• Logs Symfony présents dans l'application (info, warn, error, fatal) • Sur différentes parties de l'application • Très peu de logs système
Duplication de code		• 23,5% de duplication de code sur 365 fichiers (JS et PHP) • 7,9% de duplication sans les librairies (126 fichiers) • Beaucoup de duplication de fichiers et librairies JS, notamment entre "particuliers" et "professionnels-entreprises"

Potentiel d'anomalies
combien de bugs retournés par les outils comme SonarQube ou IntelliJ ?
		• PHP:
		  ○ Intellij : 13 potentiels bugs, 352 warnings
		  ○ SonarQUBE : 2 bugs
		• JS :
		  ○ Intellij : 900 warnings
		  ○ SonarQUBE : 31 bugs
		• CSS : SonarQUBE: 512 bugs

Code legacy Dépendances à jour ? Code déprécié ? Alexxxs lors de la phase de compilation/build ?		• 'Code smell' remontés par SonarQube : 301K CSS, 1,3K PHP, 471 JS
Utilisation des frameworks/librairies Intrusifs ? Enchevêtrés ?		• Peu de frameworks/librairies utilisées, donc peu d'adhérence technique.

Maintenabilité - Évaluation Finale

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art



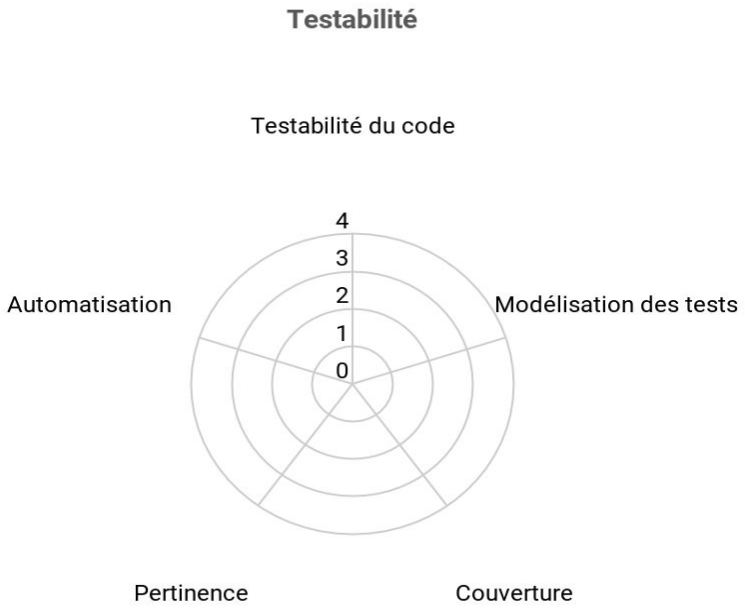
Axes	Note
Débogage	1
Dette technique	1
Couplage faible	3
Complexité	1

Testabilité - KPIs

KPIs	Statut	Observations
Nombre de tests		• Aucun tests automatisés
Couverture		• 0%
Testabilité du code		• Couplage fort entre les différentes couches (Controller, Model, Accès au stockage) • Code regroupé dans des “grosses” classes (Simulator, SimulatorsAdminController, BaseController, …)
Stratégie de tests		• Tests manuels
Pertinence		• N/A
Lisibilité des tests		• N/A

Testabilité - Évaluation Finale

Légende **0** Urgence critique, **1** Insuffisant par rapport à l'état de l'art, **2** Acceptable, **3** Point fort, **4** Etat de l'art



Axes	Note
Testabilité du code	0
Modélisation des tests	0
Couverture	0
Pertinence	0
Automatisation	0

03

Synthèse et Recommandations

► THERE IS A BETTER WAY ◄

Synthèse

- L'application xxx est endettée, ce qui rend sa maintenance et son évolution difficile. S'orienter vers une remise en état technique minimale semble être un choix pertinent afin de :
  - Mettre en place une base saine en permettant la mise en place de tests dédiés afin de sécuriser le périmètre fonctionnel actuel et futur
  - Ré-aligner les standards de développement en vue d'assurer l'homogénéité du code et le collective ownership
  - Mieux séparer la logique applicative afin d'éviter de concentrer trop de complexité dans les composants

- L'absence de tests automatisés ainsi que les différents défauts remontés sur la lisibilité du code et sa maintenabilité démontre un probable manque d'expertise sur les pratiques de qualité de développement

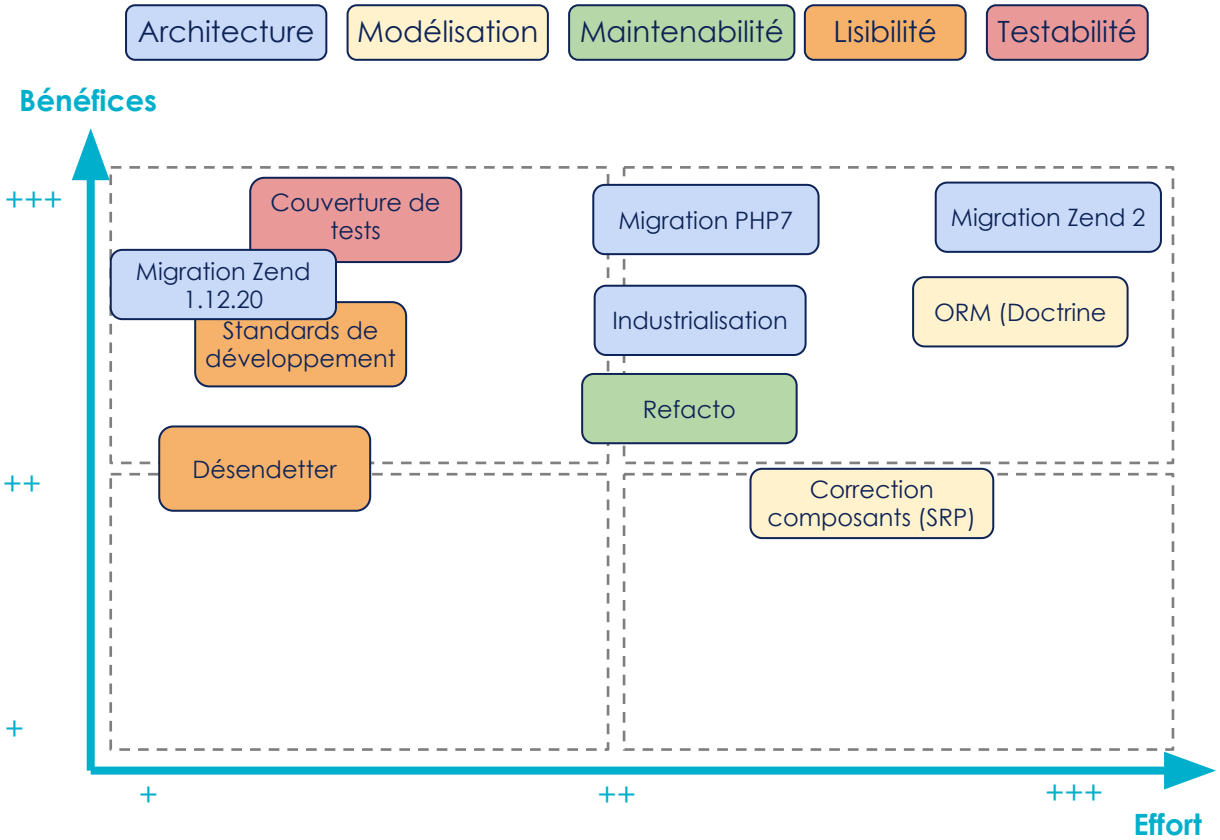
- xxx n'atteint pas le niveau d'industrialisation que nous pouvons habituellement voir sur le marché

- Le passage vers autre équipe nécessite un accompagnement et l'implication des personnes historiques au projet et poxxxuses de la connaissance du produit

Notre proposition d'actions

Point d'amélioration	Proposition	Complexité	Priorité
Architecture	Mettre en place un processus d'industrialisation du build au déploiement (gestionnaire de package, de dépendance, migration SQL…)	++	**3**
	Migrer vers Zend 1.12.20	+	**1**
	Migrer vers Zend 2 (la version 1 est obsolète et non maintenue)	+++	**5**
	Migrer vers PHP7 (PHP5 n'est plus maintenue)	++	**4**
Modélisation	Migrer les interactions BDD vers un ORM (Doctrine)	+++	**4**
	Mieux séparer la logique applicative : des composants poxxxnt trop de responsabilités (Single Responsibility Principle)	++	**4**
Maintenabilité	(Re)factoriser des compoxxxments car des fonctions ont une complexité cyclomatique très élevée (= nombre de chemins possibles dans une portion de code)	++	**5**
Lisibilité	Désendetter le code mort et commenté	+	**2**
	Mettre en place des standards de développement (nommage, code style, d'équipe…)	+	**2**
Testabilité	Améliorer la couverture de tests pour sécuriser l'existant : unitaires, intégration et end-to-end (le code actuel est testable, et des librairies sont disponibles dans le framework)	+	**1**

Matrice Bénéfices/Efforts du plan d'action



Chantiers à mener

Grandes catégories

Désendettement de l'application

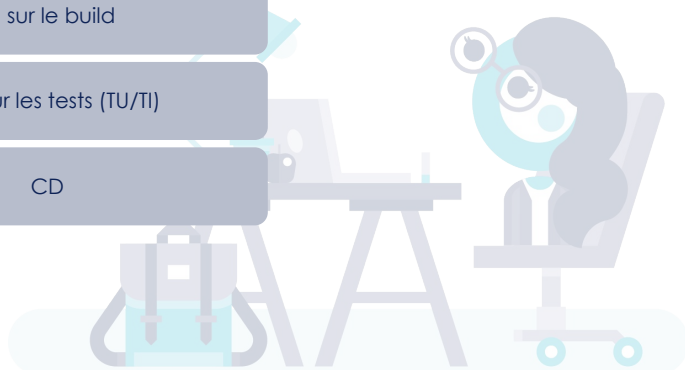
- Tests de caractérisation
- Redécoupage de l'application
- Tests unitaires
- Refonte du front
- Affranchissement du simulateur (POC)
- Stratégie de désendettement
- Standardisation des développements

Enrichissement fonctionnel

- Injection PDF
- Mapping front/PDF dynamique
- Contrôle sur les champs
- Automatisation du mapping
- Webservices pour les sources de données
- Harmonisation des templates Twig
- Encapsulation SP
- Autofill sur les champs

CI/CD*

- Forker G6K
- Versionner le code (Git)
- CI sur le build
- CI pour les tests (TU/TI)
- CD



*** CI/CD :** Continuous Integration/Continuous Deployment

Chantiers à mener

Priorité : 1 : Très prioritaire, 2 : Moyennement prioritaire, 3 : Non prioritaire

Complexité :  Peu complexe,  Complexe,  Très complexe

Désendettement

		Priorité	Complexité
Tests de caractérisation	Poser des tests de caractérisation en s'engageant sur une stratégie de tests par API afin de sécuriser l'existant fonctionnel	2	
Redécoupage de l'application	Mieux séparer la logique applicative, (ex : réduction de la taille des Contrôleurs afin qu'ils aient moins de logique métier, en déplaçant la logique métier dans une couche "Domain" avec plus de services et moins de code dans les Modèles, et la logique de persistance dans une couche Infra/Repositories)	2	
Tests unitaires	Poser des Tests Unitaires sur le nouveau code et le code déplacé afin de s'assurer du bon fonctionnement et d'éviter les régressions dans les développements futurs	2	
Refonte du front	Redévelopper entièrement le front (très endetté) en tenant compte des standards de développement actuels, dans une technologie connue	3	
Affranchissement du simulateur (POC)	Sécuriser le découpage de l'application en réalisant un POC d'extraction du générateur de simulateurs	1	
Stratégie de désendettement	Définir une stratégie pour identifier les axes prioritaires à traiter (modules à découper, codes à simplifier etc…)	1	
Standardisation des développements	Définir des règles relatives à la conception des applications, partagées par l'équipe de développement	2	

Chantiers à mener

Priorité : 1 : Très prioritaire, 2 : Moyennement prioritaire, 3 : Non prioritaire

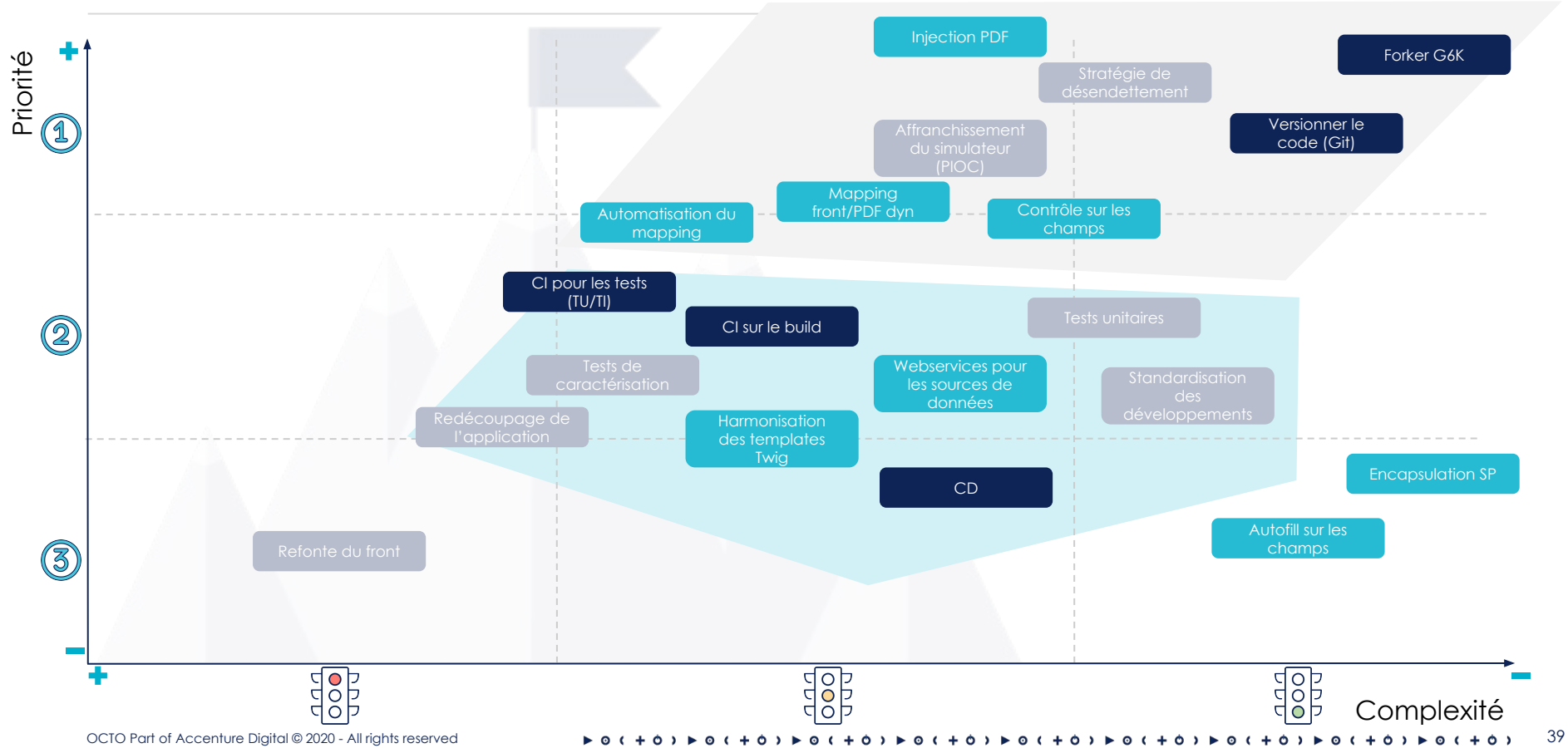
Complexité :  Peu complexe,  Complexe,  Très complexe

Intégration/Déploiement continus

		Priorité	Complexité
Forker G6K	Décider de créer une branche de G6K spécifique à la xxx	1	
Versionner le code (Git)	Mettre en place un outil de gestion de version (type Git) pour permettre le développement collaboratif et initialiser une chaîne d'intégration continue.	1	
CI sur le build	Mettre en place une plateforme d'intégration continue pour s'assurer au fil de l'eau que le code poussé compile correctement	2	
CI pour les tests (TU/TI)	Étendre la plateforme d'intégration continue aux tests unitaires et aux tests d'intégration	2	
CD	Mettre en place une plateforme de déploiement continu pour décharger l' équipe de cet aspect en l'automatisant	3	

Priorisation des chantiers

CI/CD | Fonctionnel | Désendettement



Revue de code

Objectifs :

Détection des défauts, discussion des standards, improve collective awareness

Faits

- Fait 1
- Fait 2
- Fait 3

Risques

- Risque 1
- Risque 2
- Risque 3

Approche

- Explicit the current standards that the team use, and make sure that they are shared and understood.
- The code review should be done by every member of the team, not necessary the technical leader.

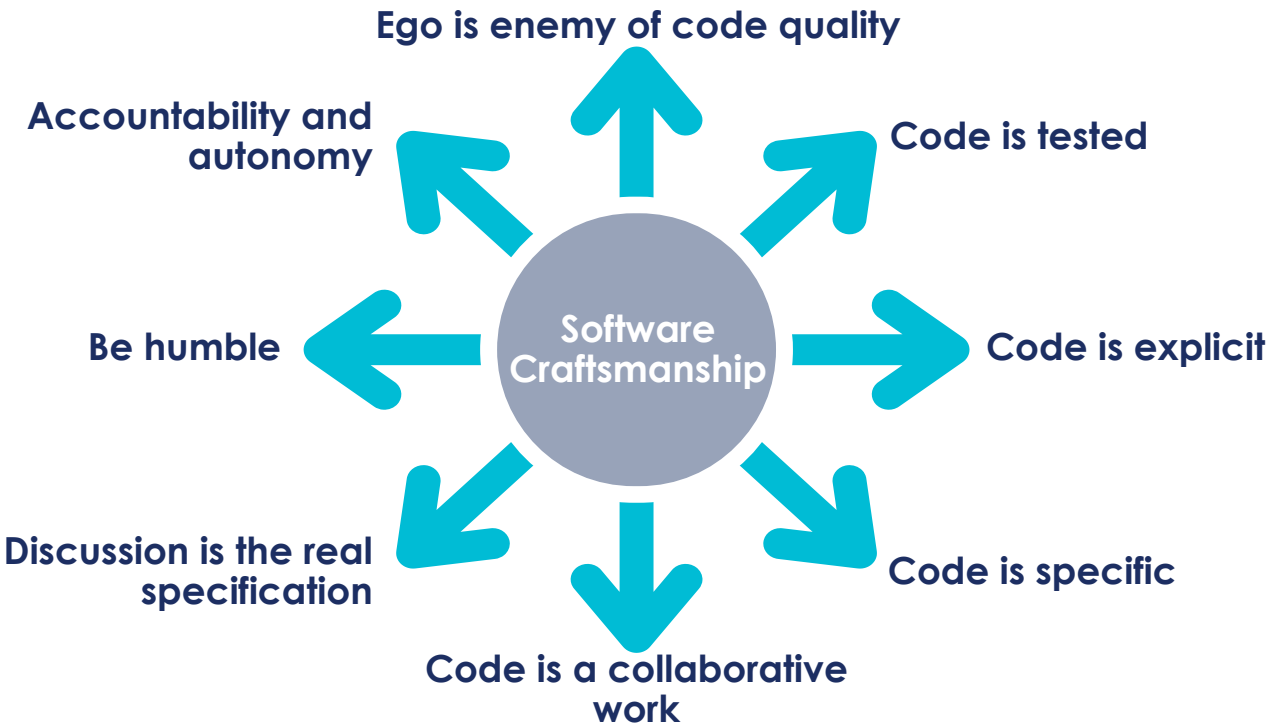
Outcomes:

- According to a Jones, Capers study conducted over 12000 projects, code review detect bugs in 65% of cases when the review is collective, in 50% when doing a pair review.
- Homogeneous codebase: easier for newcomers to get on board, easier for maintainers to enforce

04

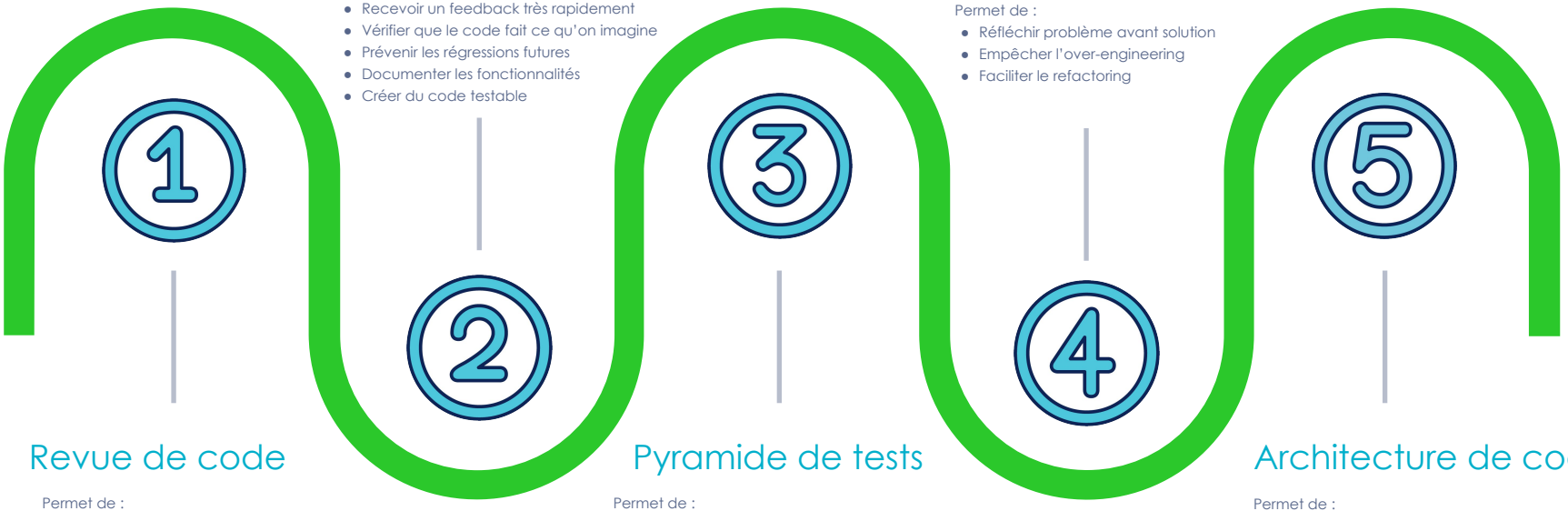
Annexes

Software Craftsmanship



Étapes de mise en place d'une culture de la qualité

Échelle de maturité



Revue de code

Permet de :

- Relever des bugs
- Améliorer la qualité du code
- Faire évoluer les standards de code
- Renforcer la propriété collective du code
- Faciliter l'apprentissage
- Améliorer la qualité des échanges

Tests unitaires

Permet de :

- Recevoir un feedback très rapidement
- Vérifier que le code fait ce qu'on imagine
- Prévenir les régressions futures
- Documenter les fonctionnalités
- Créer du code testable

Pyramide de tests

Permet de :

- Recevoir un feedback exhaustif sur le fonctionnement de l'application
- Vérifier que les composants communiquent correctement entre eux
- Optimiser le temps passé à exécuter, écrire et maintenir les tests

Test Driven Development

Permet de :

- Réfléchir problème avant solution
- Empêcher l'over-engineering
- Faciliter le refactoring

Architecture de code

Permet de :

- S'inspirer de patterns d'architecture pour répondre au contraintes du projet : Clean Code, Clean Architecture, Architecture Hexagonale, etc.
- Structurer l'organisation du code pour répondre à l'augmentation de la code base

Qualité de Code : quelques métriques

Nombre de lignes de code par classe

À partir de 200 LOCs dans une classe, il faut commencer à se poser des questions sur le nombre de responsabilités de la classe, et si on ne devrait pas séparer la logique.

Nombre de méthodes par classe

À partir de 15/20 méthodes dans une classe, il faut commencer à se poser des questions sur le nombre de responsabilités de la classe, et si on ne devrait pas séparer la logique.

Nombre de lignes de code par méthode

À partir de 20 lignes de code par méthode, il faut commencer à se poser des questions sur le nombre de responsabilités de la méthode, et si on ne devrait pas séparer la logique.

Complexité Cognitive

La Complexité Cognitive (métrique calculée par des outils comme Sonar), représente la difficulté de lecture et de compréhension du code. Elle est notamment calculée à partir du nombre de if/else d'une méthode. Au delà de 8, il est nécessaire d'optimiser le code.

Duplication

La duplication de code doit rester peu élevée, ce qui ne veut pas dire qu'il ne peut pas y en avoir, notamment pour éviter des abstractions non nécessaires (et de perdre en compréhensibilité)

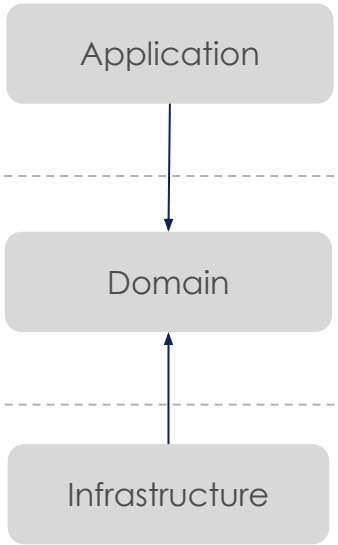
Tests

Le nombre de tests sur l'applicatif donne une indication sur la maintenabilité du code. Plus il y a de tests, plus on peut travailler sereinement sur la base de code sans avoir peur des régressions

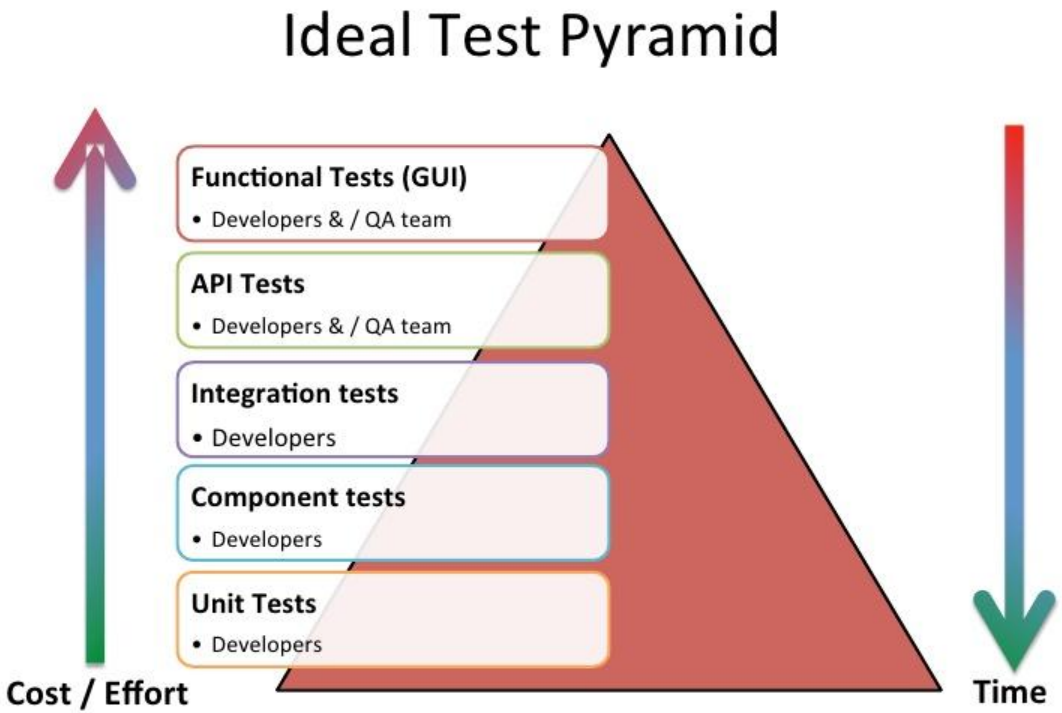
Qualité de Code : quelques principes de Design Applicatif

Parmi les différents pattern de design applicatif, que ce soit de l'onion architecture, de la clean architecture ou encore de l'hexagonale architecture, on retrouve un principe de base de séparation des couches ayant chacune leur responsabilité

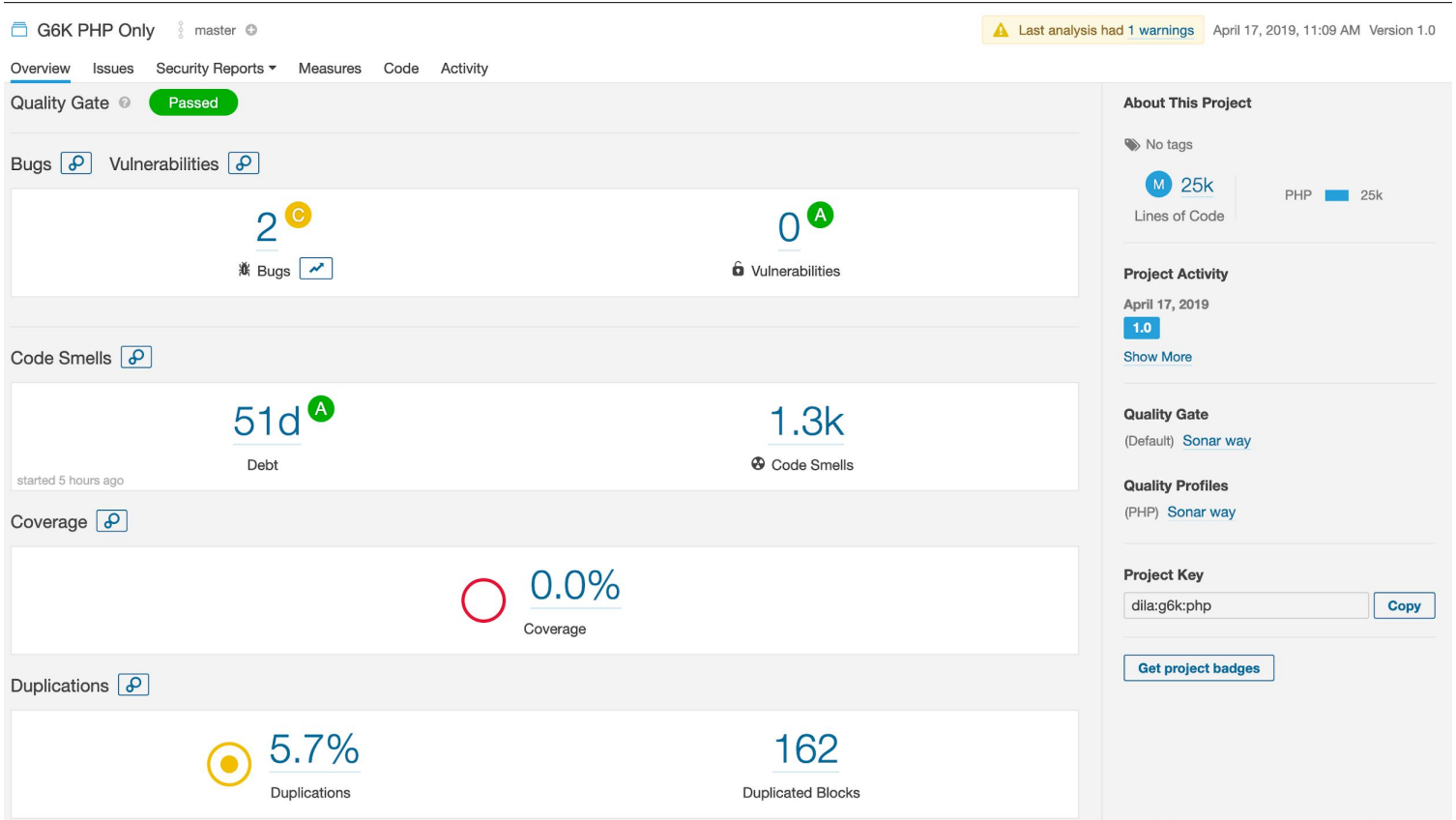
- **L'Application**
  - Relatif à la présentation des données (couche "UI")
  - Reçoit des demandes émanant de l'extérieur de l'application
  - Appel le Domaine qui va exécuter la logique métier
  - Retourne le résultat de l'exécution de la logique métier à l'appelant
  - ex : Controller qui reçoit une requête HTTP, la décode, appel le service du domaine adéquat, retourne le résultat (en JSON par exemple)
- **Le Domaine**
  - Relatif au métier
  - Implémente les règles de gestion
  - Contient le modèle de donnée conceptuel (Entitées)
  - Ne contient pas d'I/O
- **L'infrastructure**
  - Relatif aux données et aux appels à l'extérieur de l'application
  - Contient la logique de stockage et de requêtage des données
  - ex : persistance et récupération des utilisateurs dans une base de donnée PostgreSQL



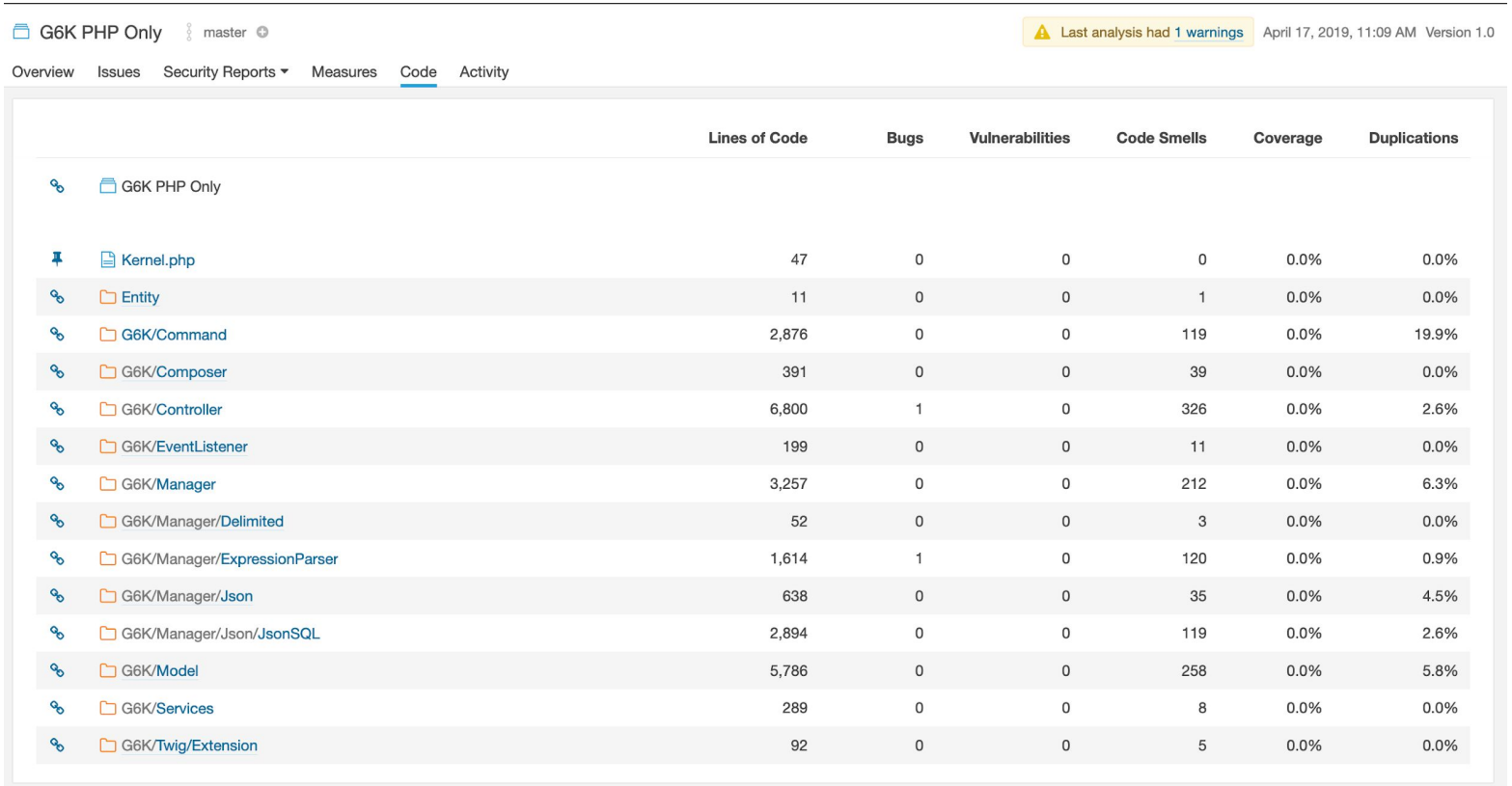
Pyramide des tests



Analyse Globale PHP - SonarQube - Overview

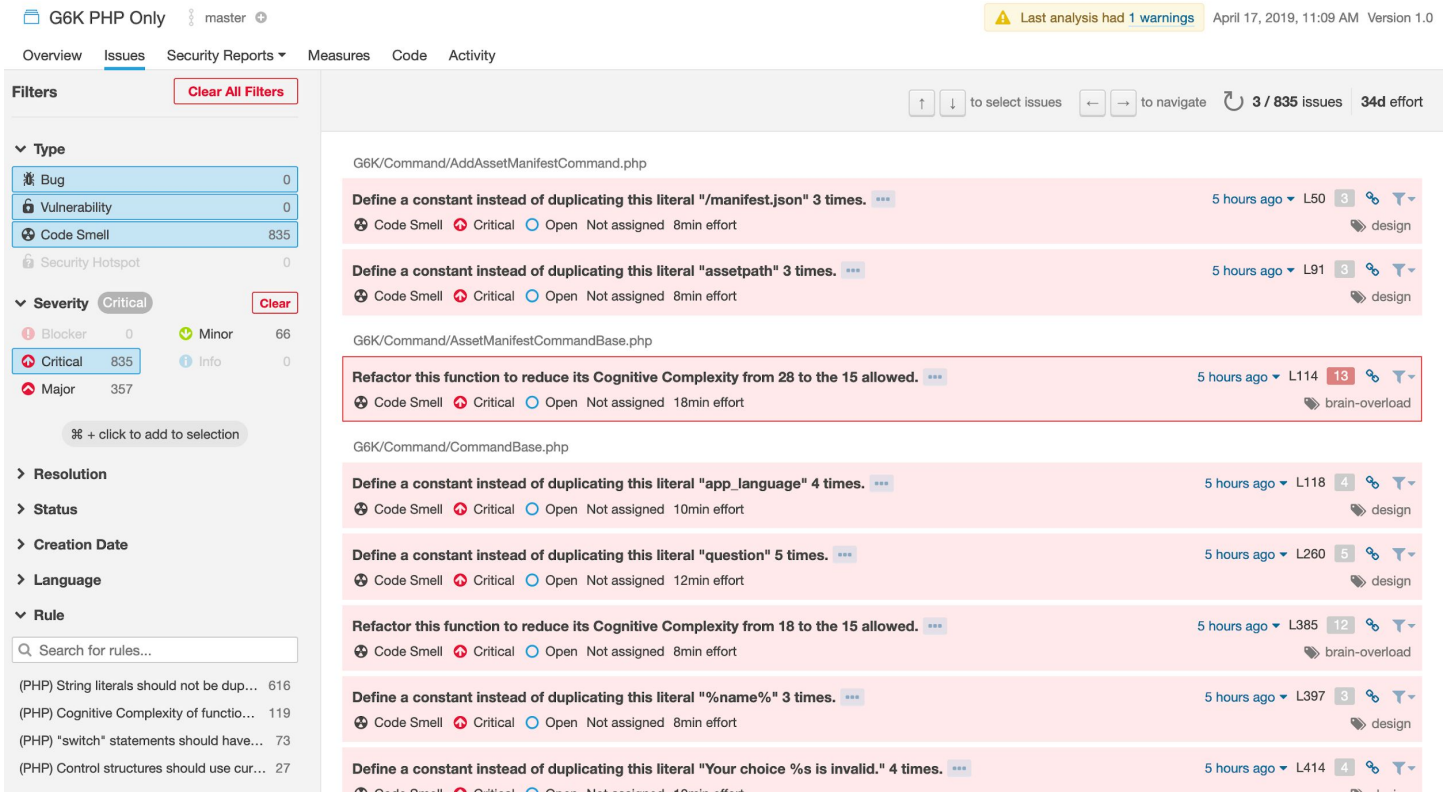


Analyse Globale PHP - SonarQube - Packages Details

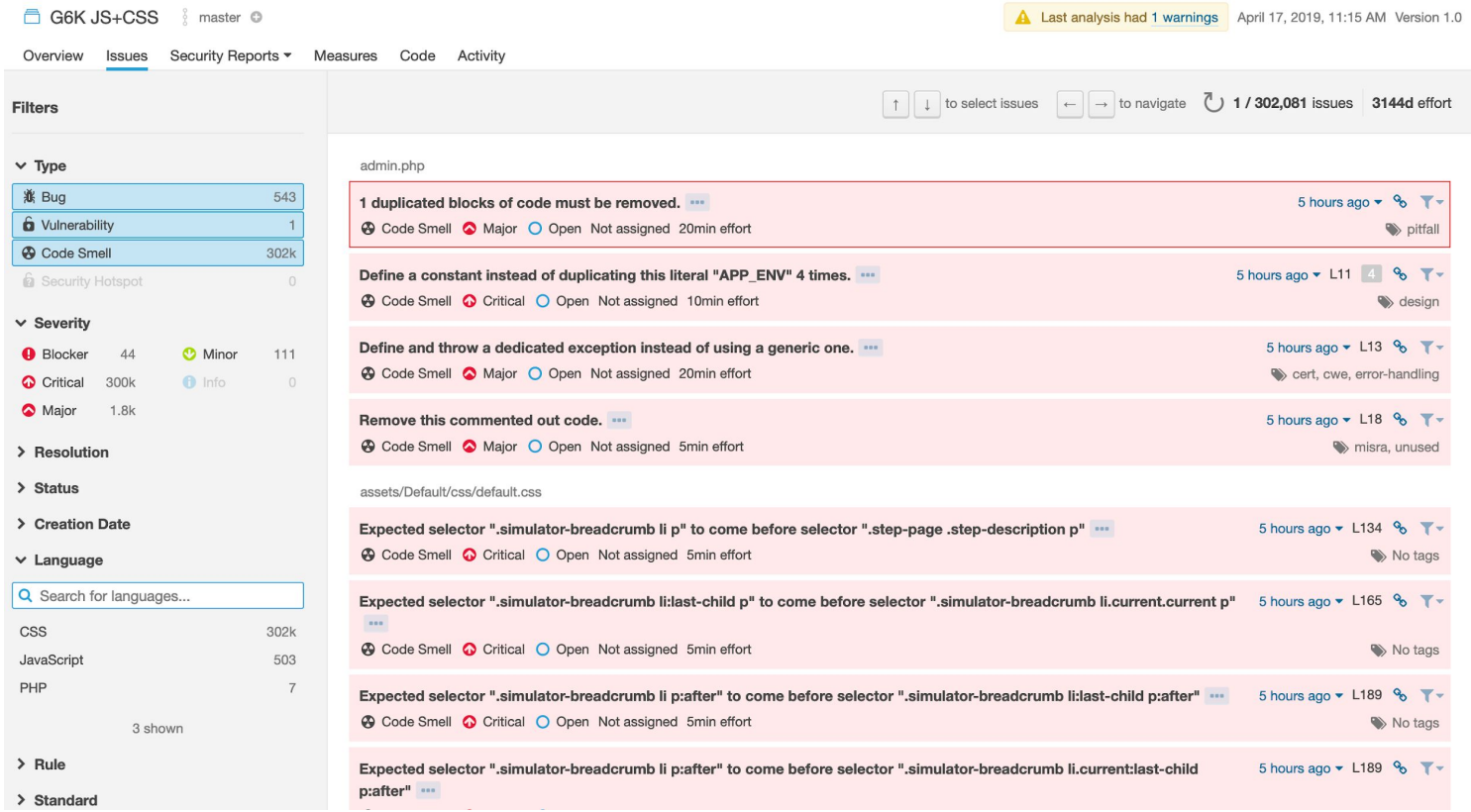


	Lines of Code	Bugs	Vulnerabilities	Code Smells	Coverage	Duplications
G6K PHP Only						
Kernel.php	47	0	0	0	0.0%	0.0%
Entity	11	0	0	1	0.0%	0.0%
G6K/Command	2,876	0	0	119	0.0%	19.9%
G6K/Composer	391	0	0	39	0.0%	0.0%
G6K/Controller	6,800	1	0	326	0.0%	2.6%
G6K/EventListener	199	0	0	11	0.0%	0.0%
G6K/Manager	3,257	0	0	212	0.0%	6.3%
G6K/Manager/Delimited	52	0	0	3	0.0%	0.0%
G6K/Manager/ExpressionParser	1,614	1	0	120	0.0%	0.9%
G6K/Manager/Json	638	0	0	35	0.0%	4.5%
G6K/Manager/Json/JsonSQL	2,894	0	0	119	0.0%	2.6%
G6K/Model	5,786	0	0	258	0.0%	5.8%
G6K/Services	289	0	0	8	0.0%	0.0%
G6K/Twig/Extension	92	0	0	5	0.0%	0.0%

Analyse Globale PHP - SonarQube - Issues



Analyse Globale JS/CSS - SonarQube - Issues



Du fait de la duplication des librairies au sein des dossiers du front, l'outil n'est pas en mesure de restituer des indicateurs pertinents (filtrage complexe)


There is a Better Way