Live Documentation : vous allez aimer la documentation
ou Documention is Dead, Celebrate Living Documentation!

Speaker : Cyrille Martraire (Arolla)
Format : Conférence
Date : 22 avril 2016
Slides : http://fr.slideshare.net/cyriux/living-documentation-bdxio-2015-dddx-london-2015

Est-ce que la documentation peut-être aussi fun que du code ?
Constat : soit y'en a pas assez, soit elle est pas à jour, soit on n'a pas envie de la faire.

Le développeur aime le code car çà s'exécute. La documentation est statique.
La documentation n'est pas Agile.

Pour faire mieux, il faut adopter une nouvelle façon de penser.
Quel est le but de la documentation ?
C'est une question de transmission du savoir, d'un développeur à ses pairs ou à des gens non techniques, tout de suite ou plus tard.

Le savoir : long-run, large audience, critical.
L'information utile pour peu de temps ou peu de monde : pas besoin de documentation. Pas de perte de temps.

La conversation est le mode de transfert de connaissance le meilleur : high-bandwich, interactive, just in time

1ière règle : conversations over documentation

Plutôt que converser, on pourrait faire du pair programming ou bien du mob-programming.
Travailler ensemble, c'est de la documentation.

2ième règle : la documentation passe également par le code

Les commentaires compensent les problèmes de code. Etes vous fiers de votre commentaire ?

Le nommage des classes, interfaces racontent le code.
« Est-ce qu'en lisant le code, on comprend comment le métier fonctionne ? »

Quel rapport avec DDD ?
Le savoir métier peut être appris lors d'« Event Storming ». En 2h on peut découvrir un métier. Très efficace.
Autre technique pour apprendre le métier : aller sur le terrain (camions, entrepôts …)
Tous les nouveaux arrivants peuvent apprendre le métier au contact des opérationnels.

Utilisation d'un panneau où on affiche des photos à l'instar du Investigation Wall.
Lorsque le métier est stable, on peut faire des evergreen document. Pas besoin de le maintenir. Toujours vrai. Ne pas mettre des infos qui changent vites dans des infos qui ne changent pas vite. Le marketing bouge, les noms, les personnes.

Comment faire pour les comportements métiers ? La solution s'appelle BDD.
Utilisation de conversation qu'on matérialise sous forme de scénarios (exemples concrets) utilisables par des outils.
Le code implémente les scénarios : il y'a donc redondance. Comment savoir que l'un ou l'autre diverge ?
Avec des outils comme Cucumber et SpecFlow. Ce ne sont pas que des outils de tests. Ils font de la réconciliation entre les 2. On a un moyen de vérifier la documentation. C'est ce qu'on appelle de la Living Documentation. Peut faire office de spécs. La doc est versionnée dans le même référentiel de code.
Cette pratique peut être généraliser à d'autres domaines que les comportements métiers.

Dans la documentation, on peut mettre des tags.
L'outil Pickles fait un site avec moteur de recherche + filtrage avec les tags.
Arnaud Loyer a écrit un plugin pour générer le PDF. Les départements compliance adorent çà. On peut leur envoyer à chaque build.

Use case : mécanisme de détection de collision
La plupart des diagrammes ont une durée de vie relativement courte. On ne s'en sert que pour coder. Sauf dans certains cas où on a besoin. On peut être tenter par de gros outils payant.
Il existe des librairies comme Ditaa, diagrammr, des générateurs de diagramme.
A chaque fois qu'on utilise du plain-text, il y'a du gain : on peut faire des diff, l'IDE sait faire des renommages lors des refactoring, le versionner ….

La questions clés pour faire un choix : à quelle fréquence est-ce que çà change ?

Retour au DDD. Reproche : très abstrait et exemples difficiles à trouver.
Comment représenter des Bounded Contexts (modules) dans mon code ?
Le découpage est implicite. Tout le monde ne comprend pas pourquoi le code est découpé ainsi. Quelle est la logique de découpage ?
On peut utiliser des annotations Java. Par exemple, l'annotation @BoundedContext
Le package-info.java est un excellent emplacement pour décrire les packages.

On peut mettre des liens vers des articles qui expliquent le concept (ex : blog de Martin Fowler)
On peut également annoter les classes métiers avec des annotations dédiées.
Cela permet de générer le glossaire métier.
Moulinette avec doclet.
On peut également générer un fichier Excel. Doc métier extrait du code Java.

Comment gérer la doc de design ?
Exemple de l'architecture hexagonale.
La littérature explique déjà le concept. Une simple URL suffit. Tout ce qui existe déjà, on le référence.
S'appuyer sur les conventions de nommage (ex : .domain à l'intérieur et .infra à l'extérieur)
Si besoin de diagrammes, on génère le diagramme à partir du code (DOT).
Démo dans laquelle la classe est renommée => le diagramme est régénéré et reste à jour.
L'important est de filtrer l'infos pour ne pas avoir de gros diagrammes.

Ca ressemble à du MDA (et à de mauvais souvenirs)

S'il est difficile d'écrire de la documentation, c'est que le design est faible (code mauvais)

Les compétences pour faire de la doc sont les mêmes que pour faire du design.

Livre Living Documentation à prix libre.