Berney Alec, Forestier Quentin, Herzig Melvyn

GEN – 11.03.2021

Portfolio

GÉnie LOgiciel



Table des matières

[Introduction 2](#_Toc66353379)

[Outils 2](#_Toc66353380)

[Questions préalables 2](#_Toc66353381)

[Pourquoi signer les commits avec GPG 2](#_Toc66353382)

[Commit ealry, commit often 2](#_Toc66353383)

[Que doit contenir un message de commit ? 2](#_Toc66353384)

[Conventions de codage 3](#_Toc66353385)

[Langues 3](#_Toc66353386)

[Accolades 3](#_Toc66353387)

[Entête de fichier 3](#_Toc66353388)

[Commentaires 3](#_Toc66353389)

[Classes 3](#_Toc66353390)

[Méthodes 3](#_Toc66353391)

[Nommage 3](#_Toc66353392)

[Workflow 4](#_Toc66353393)

[Processus piloté ou agile 4](#_Toc66353394)

[Collaboration 4](#_Toc66353395)

[Branches 4](#_Toc66353396)

[Issues 4](#_Toc66353397)

[KanPan 5](#_Toc66353398)

[Commit 5](#_Toc66353399)

[Description du projet 6](#_Toc66353400)

[Sprint 1 6](#_Toc66353401)

# Introduction

Ce document a été réalisé dans le cadre du cours de génie logiciel de la HEIG-VD. Durant ce cours nous avons dirigé un petit projet. Le but de ces pages est de marquer les décisions que nous avons dû prendre au cours de ce travail ainsi que les réponses à différentes questions.

# Outils

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Langage : | Java | IDE : | IntelliJ | Tests : | Maven/JUnit |
| Additionnel : | PicoCLI | **Versionning :** | Git/GitHub | **Déploiement :** | Maven Shade |

# Questions préalables

## Pourquoi signer les commits avec GPG

Grâce à la signature GPG, la source des commits peut être authentifiée. De cette façon, dans certains projets critiques, il est possible de s’assurer de la provenance des modifications du code.

## Commit ealry, commit often

Cette philosophie de commit est bénéfique sous plusieurs aspects. Elle permet d’identifier facilement dans quel commit une fonctionnalité a été implémentée. De cette manière, il est plus facile de revenir en arrière puisque chaque commit implique une quantité limitée de code. De plus, lors de travaux collaboratifs, il est plus facile d’intégrer plusieurs petits commits ensemble que de devoir débugger une grosse intégration multifonctionnelle.

## Que doit contenir un message de commit ?

Ce que l’on souhaite voir dans un commit est subjectif. Toutefois, nous pouvons trouver quelques lignes directrices.

Le titre doit résumer en quelques caractères qu’est ce qui a été modifié, ajouté ou supprimé.

Le corps du commit sert à expliquer en quelques mot ce qui a été impacté et éventuellement pourquoi mais pas comment. La donnée comment est obtenue en lisant les différences.

Il faut être précis mais concis de manière à obtenir rapidement et précisément ce à quoi chaque commit se rapporte.

# Conventions de codage

## Langues

* Code -> En anglais
* Commentaires -> En Français

## Accolades

Toujours après un retour à la ligne

if(...)

{

/\* Du code \*/

}

## Entête de fichier

/\*

-----------------------------------------------------------------------------------

Cours : Génie logiciel (GEN)

Fichier : <NomDuFichier>.h

Auteur(s) : Forestier Quentin & Melvyn Herzig

Date : JJ.MM.AAAA

-----------------------------------------------------------------------------------

\*/

## Commentaires

Les classes et les méthodes sont commentées au format Javadoc.

### Classes

/\*\*

\* Description de la classe

\* @author auteur 1

\* @date jj-mm-aaaa

\*/

### Méthodes

/\*\*

\* Description de la méthode

\* @param param1 Description du premier paramètre.

\* @param param2 Description du second paramètre.

\* @return Que retourne la méthode..

\*/

### Nommage

Tout les noms seront en camel case sauf pour les constantes où les espaces seront des ‘\_’ et en majuscules.  
**Classe** : Première lettre en majuscule, MaClasse.java  
**Fonction** : Première lettre en minuscule, maFonction()  
**Ma constante** : final int UNE\_CONSTANTE.

# Workflow

## Processus piloté ou agile

Nous avons décidé de choisir un processus agile.

Nous n’avons pas choisi le processus piloté pour une raison principalement. Nous ne sommes pas encore des experts en gestion de projet. De ce fait, il est obligatoire à un moment qu’une tâche à laquelle nous n’avions pas pensé fasse surface ou que nos tâches soient mal agencées. En d’autres termes, nous sentons que nous n’avons pas assez d’expérience pour choisir ce type de pilotage qui laisse peu de marge de manœuvre. Notre manque d’expérience pose également problème au niveau de l’estimation du temps à passer sur ces chacune des tâches.

Au contraire, nous préférons choisir un processus agile. Il nous permettra d’avancer et d’améliorer notre projet itération après itération. Nous trois, avons pour habitude de programmer de manière à obtenir rapidement un MVP (Minimum Viable Product) que nous améliorons par la suite avec les différentes spécifications du projet. Ainsi en optant pour cette méthode de travail nous seront plus aptes à faire face aux imprévus et aux changements en cas de besoin.

## Collaboration

L’outil principal pour la collaboration sera GitHub. <https://github.com/gen-classroom/projet-berney_forestier_herzig>

## Branches

Chaque itération aura sa branche dédiée appelé iteration-x où x est le numéro de l’itération. Puis, dans les branches d’itérations nous aurons des sous branches qui serviront à l’implémentation des différentes fonctionnalités au sein de l’itération. Puis à la fin d’une itération, la branche sera merge sur main. Le nom des branches est en anglais.

Le nom des branches doit être écrit en minuscules et les espaces doivent être remplacés par des « - ». Les branches qui ajoutent une feature se nomment « ft-<featurename> ». Les branches qui résolvent des bugs « bg-<bugsolving> ».



Figure 1Exemple branches

## Issues

Chaque fonctionnalité/tâche fera l’objet d’une « Issue ». Lorsqu’un collaborateur travaille sur une issue, il se désigne comme « assignee ». Lorsque la tâche est terminée, il effectue une « pull request » de sa branche sur la branche de l’itération, en spécifiant l’issue concernée. De cette manière, la personne qui validera le merge fermera également l’« issue ». Les issues sont en français

Les personnes assignées à l’issue posteront l’état de leur travail dans l’issue en cas de besoin.

## KanPan

Nous avons intégré un projet KanPan de GitHub. Chaque issue fera l’objet d’une étiquette automatiquement créée. Nous verrons l’avancement et la répartition des tâche d’une meilleure manière qu’uniquement par l’onglet « Issue ».

Nous y avons inséré 4 colonnes :

* **Todo :** Regroupe les issues qui n’ont pas été commencée.
* **In progress :** Regroupe les issues qui sont en cours de traîtement
* **Waiting to be validated** Regroupe les “Pull request” en attente de validation.
* **Done** Regroupe les « issues » et les « Pull request » teminée

Il y aura un kanban par sprint.

## Commit

Les commits seront signés sinon ils ne seront pas acceptés par GitHub. Ils sont écrits en français.

**Entête** : Très court, max 50 caractères Résumé des modifications apportées.

**Message :** Max 4-5 lignes, explique plus en détail ce qui a été changé, éventuellement pourquoi, mais pas comment. Le comment est obtenu en lisant les modifications par rapport à l’état précédent.

# Description du projet

Le but de ce projet et d’implémenter un générateur de sites statiques. Le site sera généré à partir à partir de deux fichiers.

Un fichier JSON, YAML ou TOML. Ce fichier permettra de créer les métadonnées de la page dans le <head>.

Un fichier Markdown. Ce fichier générera le contenu de la page dans le <body>.

L’utilisateur aura 4 commandes à disposition :

* **statiqueSite init** : Initialisera le répertoire donnée en argument avec les fichiers pour créer le site.
* **statiqueSite clean** : Nettoie le répertoire de build.
* **statiqueSite serve**: ?
* **statiqueSite build** : Génère le site en fonction des fichiers, le tout dans un dossier build.

# Sprint 1