**420-C63**

**Projet – Sprint et organisation**

# Standards de programmation

Afin de produire du code de qualité l’équipe doit établir des standards de programmation. Cela permet entre autres de produire du code de qualité avec une forme à peu près équivalente entre les programmeurs. Les différents programmeurs de l’équipe ont souvent des préférences au niveau de la nomenclature et des design patterns qui sont utilisés. Le but n’est pas de définir une « meilleure » façon de programmer, mais bien que de façon générale chaque programmeur soit assez satisfait afin qu’il respecte les conventions qui ont été établies.

## Formatage automatique

Peu importe le projet, il est essentiel d’utiliser le formatage automatique même si le formatage diffère parfois de nos préférences en tant que programmeur. Dans une équipe de travail, il est essentiel d’avoir le moins de conflits possible dans notre logiciel de version afin de travailler plus vite et d’éviter de polluer les commit avec des changements non essentiels comme l’espace blanc. Lorsque chaque programmeur de l’équipe utilise la même configuration de formatage automatique, le problème est éliminé.

Des logiciels plus avancés comme StyleCop ou Resharper permettent de définir des formatages avancés et sont recommandé, mais à petite échelle celui de Visual Studio peut être utilisé. Ne pas oublier de supprimer les lignes vide superflues et le code non utilisé.

## Nomenclature et organisation du code

Il est recommandé de suivre les standards suggérés par les créateurs du langage de programmation utilisés. Par exemple Pour C#, suivre les standards suggérés par Miscrosoft. Cependant, il est possible de faire certaines modifications tant que l’équipe est d’accord. Par exemple, il est possible d’ajouter un underscore devant le nom des variables membres (ex : \_maVariableMembre) afin de ne pas les confondre avec les variables locales. Afin de sauver du temps, il est suggéré d’utiliser le fichier *Nomenclature.cs* comme standards.

## Design Patterns

Les design patterns sont utiles pour régler des problèmes complexes de façon prouvée. Cependant, ils ajoutent de la complexité au code et ne sont pas toujours la meilleure solution. Le but principal en gestion de projet est de réduire le plus possible ce qu’on appelle le Bus Factor. Le Bus Factor est une situation hypothétique si un des programmeurs se faisait frapper par un bus, est-ce que l’équipe serait capable de continuer le projet. Comme la plupart des programmeurs restent 2-3 ans à un même emplois, la situation est souvent beaucoup plus qu’une simple hypothèse.

Afin de réduire le Bus Factor, chaque programmeur doit comprendre facilement chaque partie du code et pouvoir la modifier. Habituellement, c’est pourquoi on force les programmeurs à parfois travailler sur des parties de code qu’ils n’ont pas mis en place. D’un autre côté, c’est aussi pourquoi certains design patterns complexes ne sont souvent pas la meilleure solution pour certaines équipes. Comme la plupart des programmeurs auront à toucher à tout, il faut qu’ils soient à l’aise à toucher à tout. Il faut donc adapter les designs patterns utilisés avec la force de l’équipe.

C’est pourquoi il y a souvent des discussions d’équipe lorsqu’un nouveau design pattern doit être utilisé. Il faut évaluer la complexité ajoutée et le temps nécessaire à l’équipe pour apprendre à utiliser le nouveau design pattern. Souvent cette décision est prise par le Lead Programmer de l’équipe qui connait assez bien les forces et les faiblesses de l’équipe.

Cependant ne pas utiliser de designs patterns peut aussi être un problème, car avoir du code trop simple peut le rendre difficile à modifier. Il est très facile de créer des classes monstres de 1000 lignes qui font trop de travail. Il faut quand même diviser le code et utiliser des design patterns simples lorsque possible. Par exemple, diviser le GameManager en un nouveau manager ou utiliser des State Machines au lieu de plusieurs variables disinctes. Le but est aussi de réutiliser le plus possible les mêmes design patterns afin d’unifier le code et de le simplifier.

Bref, il n’y a jamais de meilleure solution, seulement la meilleure solution pour une situation et une équipe donnée. Le code et les programmeurs sont toujours en constante amélioration.

# Git – Organisation des branches

Afin de pouvoir travailler à plusieurs et de gérer un projet qui doit être livré en tout temps, il est suggéré d’utiliser la structure suivante pour organiser Git.

## Branche *main (ou master)*

* Contient la version en cours du programme qui n’est pas encore approuvée pour être la prochaine version officielle du programme.
* Sert d’endroit commun afin de travailler et préparer la prochaine version officielle du programme.
* On ne programme pas dans cette branche directement, on y merge des branches personnelles.

## Branche personnelle

* Pour chaque fonctionnalité à implémenter, un programmeur créé une branche personnelle à partir de la branche *main* (ex : branche *mainMenu*, *enemy1*, *enemy2*, *player*, etc.). Si un programmeur travaille sur plusieurs fonctionnalités, il créera une branche différente par fonctionnalité.
* Flot normal d’utilisation :
  + Mettre une tâche dans JIRA en cours
  + Créer une branche personnelle à partir de *main*
  + Programmer la fonctionnalité
  + Commit et pusher régulièrement le travail sur Git
  + Tester que tout fonctionne
  + Merger la branche *main* dans sa branche personnelle et régler les conflits
  + Tester que tout fonctionne
  + Merger et pusher la branche personnelle dans *main*
  + Supprimer la branche personnelle
  + Fermer la tâche dans JIRA
* Il peut arriver que 2 programmeurs travaillent sur une fonctionnalité reliée qui n’est pas encore prête à être ajoutée à *main*, par exemple si un programmeur travaille sur le joueur et un autre sur un ennemi en même temps. Dans ce cas :
  + Chaque programmeur créé sa propre branche (ex : branche *enemy1* et branche *player*)
  + Chaque programmeur peut périodiquement merger les mises à jour d’une autre branche dans sa branche au besoin (ex : merger la branche *player* dans *enemy1*)
* Note : régler un bogue est considéré une fonctionnalité, donc un programmeur doit créer une branche par bogue. Dans ce cas, il est possible d’utiliser un nom de branche générique comme *mael* ou *maelBugFixes* et de la réutiliser pour chaque bogue.

## Branche *release*

* Contient une version du jeu qui est relativement stable et testée afin de pouvoir être compilée et déployée pour des clients ou des utilisateurs.
* Rarement mise à jour, la plupart du travail est effectué dans la branche *main*.
* À la fin de chaque sprint :
  + Tester qu’il n’y a pas de bogue majeur et que tout fonctionne relativement bien.
  + Merger la branche *main* dans la branche *release*
* Exceptionnellement, des « hotfix » peuvent être déployés directement sur une branche *release* dans le cas ou des bogues critiques (ex : crash) qui n’ont pas été détectés lors des tests empêchent la version officielle de fonctionner.
  + Créer une branche *hotfix* à partir de *release*
  + Fixer le bogue
  + Merger la branche *hotfix* dans *release*
  + Supprimer la brancher *hotfix*
* *Note : parfois une branche supplémentaire dev ou develop est créée et prend le rôle de la branche main et la branche main contient plutôt une version stable du jeu servant à créer les branches release. Pour le cours, il est suggéré d’utiliser simplement la branche main comme branche principale et de ne pas créér de branche develop afin de ne pas ajouter trop de complexité. En entreprise, il est suggéré de toujours créer une branche develop.*

# Git – Standards

Voici quelques standards à respecter pour Git qui sont ceux utilisés dans la plupart des entreprises peu importe le style de projet ou le langage de programmation utlisé.

* Nom des branches
  + Toujours en anglais.
  + Pas d’abréviation.
  + Respecter la casse camelCase
  + Avoir un lien direct avec la fonctionnalité qu’elle contient
* Messages
  + Toujours en anglais.
  + Respecter les règles de grammaire et de ponctuation.
  + Messages agissent comme documentation à long terme sur le contenu de chaque commit et peuvent aussi servir pour faire les messages de changement contenus dans chaque version du programme. Le but n’est pas d’être utile en un jour, une semaine ou un mois, mais 3-6 mois, un an, etc. Des bons messages sont indispensables sur un gros projet.
  + Avoir le format Verbe + description détaillée :
    - Verbe  
      *Choisir un des verbes suivants*
      * Added
      * Removed
      * Fixed
      * Moved
      * Changed
    - Description détaillée
      * Souvent similaire au nom de la tâche dans JIRA
      * Essayer le plus possible de diviser en plusieurs blocs distincts.
    - Exemples
      * Mauvais
        + ahdsha
        + Enemy
        + add player,enemy
      * Bien
        + Small fixes.
        + Added Goomba.  
          Removed old enemy.
        + Added Player.  
          Added Goomba.
      * Excellent
        + Fixed typo in Main Menu Quit button.  
          Fixed typo in HUD player name.
        + Added new Goomba enemy.  
          Removed enemy placeholder assets and GameObject.
        + Added PlayerController.  
          Added Player GameObject.  
          Added new Goomba enemy.  
          etc.
  + Lorsque possible diviser les changements en commit différent afin de pouvoir tracer rapidement les changements ou de faire un revert de certains changements.
    - Faire 2 commits
      * Fixed typo in Main Menu Quit button.
      * Fixed typo in HUD player name.
    - Faire 2 commits
      * Added new Goomba enemy.
      * Removed enemy placeholder assets and GameObject.
  + Dans le cas où il s’agit d’un fix pour bogue qui a un numéro assigné dans JIRA, il est recommandé de mettre comme message le numéro de tâche et la description. Ex :
    - [PRO-152] Player can walk through walls in level 2.
    - [PRO-195] Goomba does not take damage from explosions.
    - [PRO-234] Main menu quit button does not work.

# Git – Cheat Sheet

Voici les commandes les plus utiles pour 95% des cas, soit dans l’ordre créer une branche, faire des modifications et merger la branche dans la branche principale une fois les modifications terminées.

## Créer une branche

git fetch

git checkout -b *maBranche* origin/main

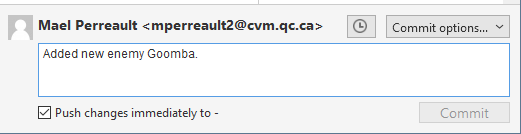
## Envoyer des modifications

git add .

git commit -m "Added ..."

git push origin *maBranche*

Note : il est fortement conseillé d’utiliser SourceTree afin de faire des commit/push, car il est beaucoup plus facile de valider les changements ou d’en envoyer seulement une partie. Cocher *Push changes immediately to...* afin de faire un push automatique sur la branche remote sur le serveur de GitHub.



## Merger sur *main*

git fetch

git checkout main

git merge *maBranche*

## Supprimer branche locale

git branch -d *maBranche*

## Supprimer branche remote

git push origin --delete *maBranche*