Légende :

- *En jaune clair italic :* l'histoire
- `En gris clair :` du code à copier/coller; des extraits de console
- : les corrections (gris foncé sur gris foncé)
- Certaines étapes ne sont pas indispensables ; elles sont marquées (**bonus**) ou (**conseil**)

1. Exercice 1 : `git stash`

https://git-scm.com/docs/git-stash/2.16.0

Lundi matin 7h30 *(Oui oui, en Lorraine on commence tôt !)*
Autour d'un café / croissant, le business analyst arrive et demande une nouvelle fonctionnalité de clic partout sur tous les champs cliquables d'un écran. Cette fonctionnalité doit-être implémentée d'ici 10 min, GO!!

- *Nouvelle API* `Ecran#clicAleatoire()`
- *Nouvel utilitaire :* `UtilitaireDeClicPartout`
- *Et création de la classe de TU associé*

1.1. Récupérez le dépôt suivant via le terminal (`git bash` sous windows) : https://github.com/efluid/jeTestDoncJeSuis-hol (Ou via la clé USB)

1.2. Vérifiez que ça compile avec maven : `mvn clean install`

1.3. Ouvrir **Eclipse** / **Intellij (ou l'IDE de votre choix)** :

- Importez le projet
- Lancez le test `UtilitaireDeClicAleatoireTest` et vérifier qu'il passe

1.4. Ajoutez une nouvelle fonctionnalité : "**clic partout**"

- Proposition d'ajout dans la classe `js.jetestdoncjesuis#Ecran`

```
public void clicPartout() {
    for (Bouton bouton : boutons) {
      bouton.clic();
    }
}
```

- Proposition de classe à ajouter : `js.jetestdoncjesuis#UtilitaireDeClicPartout`

```
package js.jetestdoncjesuis;

import java.util.Collection;

public class UtilitaireDeClicPartout {

  public void clicsPartout(Collection<Ecran> ecrans) {
    System.out.println("\n\nDémarrage des clics partout :");
    for (Ecran ecran : ecrans) {
      ecran.clicPartout();
    }
    System.out.println("Fin des clics partout\n\n");
  }
}
```

- Proposition de classe de test associé : `js.jetestdoncjesuis#UtilitaireDeClicPartoutTest`

```
package js.jetestdoncjesuis;

import static org.junit.Assert.fail;

import java.util.Collection;

import org.junit.*;

public class UtilitaireDeClicPartoutTest {

  private Collection<Ecran> ecrans = null;

  @Before
  public void init() {
    ecrans = ContexteTest.creerEcrans();
  }

  @Test
  public void quand_j_ouvre_un_ecran_je_clique_partout_et_rien_ne_bug() {
    UtilitaireDeClicPartout utilitaireDeClic = new UtilitaireDeClicPartout();
```

```
    try {
      utilitaireDeClic.clicsPartout(ecrans);
    } catch (Exception ex) {
      fail("Bug en approche !!");
    }
  }
}
```

***5 min plus tard** : l'expert test fonctionnel demande une correction en urgence sur une branche de maintenance (`bugfix`) sur la fonctionnalité de test aléatoire qui ne lui semble pas vraiment aléatoire.*
Une fois le bug fixé, il faut retourner sur la fonctionnalité pour la terminer car le chef de projet vient d'arriver dans le bureau avec son café et met la pression.

1.5. Stashez toutes les modifications avec un message "`mon premier stash`" (`dos2unix <nomDuFichier>` en cas de problème de CRLF)

[redacted]

1.6. Regardez l'état du dépôt et l'état de la liste des stash :

[redacted]

- L'ensemble du contenu est-il bien stashé ?
- Le message du stash est-il propre ?

⇒ Ne passez pas à la suite tant que vous n'avez pas un **stash unique** avec l'ensemble du contenu et un message "`mon premier stash`"
(`dos2unix <nomDuFichier>` en cas de problème de CRLF)

[redacted]

1.7. Consultez le contenu du stash (En visualisant les lignes de code Java modifiées)

[redacted]

1.8. Remettre le contenu du stash dans le dépôt **tout en conservant le stash dans la liste**

[redacted]

1.9. Créer **2 nouveaux stashs** (avec une partie des modifications dans chaque stash) :

- Un premier avec la modification dans `Ecran` nommé "`modification Ecran`"
- Et un deuxième avec `UtilitaireDeClicPartout` et `UtilitaireDeClicPartoutTest` nommé "`ajout utilitaire de clic partout`"

1.10. Vérifiez l'état de la liste des stash ; il en contient normalement 3

```
stash@{0}: ajout utilitaire de clic partout
stash@{1}: modification Ecran
stash@{2}: mon premier stash
```

1.11. Dépiler le contenu du stash "`modification Ecran`" dans le répertoire de travail

1.12. Supprimez le stash nommé "`mon premier stash`"

1.13. Oooops ; je ne voulais pas tout perdre. Je souhaite récupérer le contenu du stash "`mon premier stash`" et le mettre dans un stash nommé "`oops`"

⇒ le numéro de commit est indiqué en résultat de la commande précédente (§1.13)

(**Option :** Une commande "mystique" est dispo dans l'aide pour retrouver le numéro de commit)

1.14. Créez une branche "`feat-clic-partout`" à partir du stash nommé "`oops`" et commiter le code avec le message "`ajout de l'utilitaire clic partout`"

1.15. Nettoyez la liste des stash

2. Exercice 2 : `git worktree`

https://git-scm.com/docs/git-worktree/2.16.0

***Mardi matin** :*
On a de nouveau une correction à réaliser en parallèle de la réalisation d'une nouvelle fonctionnalité.
Mais cette fois-ci, on veut être plus productif !!

2.1. Positionnez-vous sur "`master`" proprement : `git reset --hard; git clean -fd; git checkout master`

2.2. Créez un worktree nommé "`jeTestDoncJeSuis-hol-bugfix`" (Ici, un nouveau "répertoire" va être créé ; positionnez-le à côté de l'actuel : "`../jeTestDoncJeSuis-hol-bugfix`")

[redacted]

2.3. Consultez la liste des worktree ; voici le résultat que vous devez avoir :
`../jeTestDoncJeSuis-hol` ⇒ main `worktree`
`../jeTestDoncJeSuis-hol-bugfix`

[redacted]

2.4. Comparez l'espace occupé par ces 2 répertoires (commande "`du -hs <path>`")

[redacted]

2.5. Dans votre IDE : ouvrir le worktree "nouveau worktree" (`jeTestDoncJeSuis-hol-bugfix`) via un import de projet

2.6. Réalisez la correction (décommentez le code dans `Ecran#clickAleatoire()`) dans le worktree `jeTestDoncJeSuis-hol-bugfix`

2.7. Ouvrir 2 terminaux (`git bash` sous windows) (un worktree différent sur chaque fenêtre) : vous pouvez lancer les compilations / tests sur chaque worktree en même temps

2.8. Commitez la correction sur le worktree "`jeTestDoncJeSuis-hol-bugfix`"

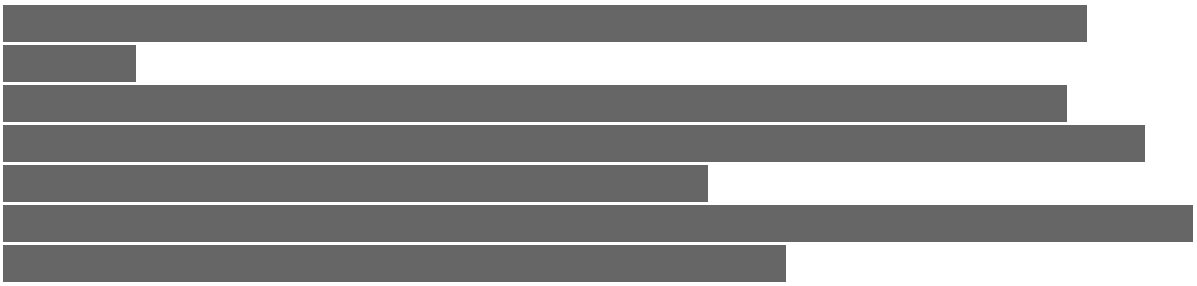
[redacted]

2.9. Depuis le worktree master (`jeTestDoncJeSuis-hol)` :

- Est-ce que le commit est visible ? (`git show` par exemple)
- Est-ce qu'on peut faire un cherry-pick dessus ?

2.10. (**Bonus**) Lancez une commande sur les différents worktree en même temps à partir du même terminal (`git bash` sous windows) ?

2.11. Le travail est terminé, supprimez le worktree "`jeTestDoncJeSuis-hol-bugfix`"



2.12. Consultez la liste des worktree pour vérifier que le worktree est bien supprimé

2.13. Positionnez vous sur le worktree principal (`../jeTestDoncJeSuis-hol`) ; est-ce que vous arrivez à retrouver la branche "`jeTestDoncJeSuis-hol-bugfix`" associé au worktree supprimé à l'étape 2.11 ?

3. Exercice 3 : `git rebase interactif`

https://git-scm.com/docs/git-rebase/2.16.0

***Mardi après-midi** : le Business Analyst souhaite, pour une raison inconnue, traduire les messages de log en anglais (oui oui, les messages de log …).*

Un rebase tout court pour commencer :

3.1. Positionnez-vous sur “master” proprement : `git reset --hard; git clean -fd; git checkout master`

3.2. Récupérez localement la branch “`rebase`” qui doit suivre la branche distante “`rebase`”

3.3. Créez une branche “`exercice_3a`” à partir de la branche “`master`” et placez-vous dessus.

3.4. Traduisez le contenu des messages de log de `UtilitaireDeClicAleatoire#clicsAleatoires`.

3.5. Commitez.

Le Business Analyst souhaite que cette feature englobe les modifications faites par une équipe off-shore sur la branche ‘rebase’.

3.6. Effectuez un rebase avec la branche “`rebase`”

3.7. Que se passe-t-il ?

3.8. Il va falloir résoudre le conflit avec classe, c’est à dire dans un outil fait pour ça car dans l’IDE ça donne quelque chose comme ça :

```
package js.jetestdoncjesuis;

import java.util.Collection;
```

```
public class UtilitaireDeClicAleatoire {

  public void clicsAleatoires(Collection<Ecran> ecrans) {
<<<<<<< HEAD
      System.out.println("\n\nDémarrage des clicks aléatoires :");
=======
      System.out.println("\n\nStarting random clicks :");
>>>>>>> [feat] traduction en anglais des messages de log.
      for (Ecran ecran : ecrans) {
      ecran.clicAleatoire();
      }
      System.out.println("Random clicks done\n\n");
  }
}
```

- Ajoutez l'outil de votre choix dans votre `.gitconfig` (PS : ça marche aussi pour `diff`/`difftool`) :

```
[merge]
    tool = bc3
[mergetool "bc3"]
    path = D:/Programs/beyondcompare/bcomp.exe
```

- Résolvez le conflit (avec classe) [redacted]

Pour info, quelques outils : Beyond Compare, kdiff3, meld, TortoiseMerge

- (**Bonus**) Des fichiers ont été créés, lesquels ? Que contiennent-ils ? [redacted]
    - (**Conseil**) Ok c'est cool mais comment on fait pour que git ne les génère pas parce qu'en fait je les supprime tout le temps : `git config --global mergetool.keepBackup false`

- Lors d'un rebase, si jamais vous vous êtes mis les pieds dans le tapis, il y a une commande qui permet de revenir dans l'état pré-rebase. Quelle est-elle ? [redacted]

3.9. (**Bonus**) Vous êtes fatigué et vous supprimez votre branche "`exercice_3a`". Zut, il faut tout recommencer, même la résolution du conflit … sauf si git se souvient de votre résolution et l'applique tout seul : nous allons donc activer le ***Re**use **Re**corded **Re**solution*. [redacted]

- Si vous avez le temps, faites le test :)
- Un peu plus de documentation : https://delicious-insights.com/fr/articles/git-rerere/

3.10. **Et maintenant le rebase intéractif**

Vous devez développer 2 features :

- Calcul du temps des tests effectués
- Génération des données au format texte des temps des tests (pour être consommer par un grafana ou équivalent)

Ici, vous avez 2 solutions :

- Soit vous lancez le script magique `exercice_31.sh` qui va vous créer une branche locale "`exercice_rebase_interactif`" et qui va cherry-picker les bons commits à partir de la branche "`rebase_interactif`"
    - Ils sont aux nombres de 8 : 4 commits par feature
        - 2 pour les implémentations
        - 2 pour les tests

- Soit vous êtes aventurier et vous codez dans votre IDE préféré les 2 features (3.11 Mode aventure)

3.11. (**Bonus**) Mode aventure

- Un rapide guide d'implémentation est disponible ci-dessous
- Attaquez le développement de la première feature (en faisant 2 commits si possible)
- Développez les TUs de la première feature (en faisant 2 commits si possible)
- Attaquez le développement de la seconde feature (en faisant 2 commits si possible)
- Développez les TUs de la seconde feature (en faisant 2 commits si possible)

- Guide d'implémentation :
    - un package *`calculTemps`* dans `jetestedoncjesuis`, une interface *`ICalculDeTemps`* et une implémentation qui porte 2 long (un temps de début et un temps de fin), une interface *`ICalculateurDeTempsDexecutionService`* avec 3 signatures (*`lancerCalculTempsExecution(ICalculDeTemps)`*, *`arreterCalculTempsExecution(ICalculDeTemps)`* et *`calculerTempsExecution(ICalculDeTemps))`* et une implémentation simple.
    - Un TU du service de calcul
    - Un package *export* dans jetestedoncjesuis, une interface *`IServiceDExportTempsExecution`* avec une seule signature qui prend en paramètre un *`ICalculDeTemps`* et qui utilise le service de calcul pour exporter ça dans un format quelconque (JSON, texte plat, sortie standard, etc.)
    - Un TU du service d'export

Nous sommes désormais le soir 18h00 et y'a un tournoi de babyfoot dans votre bar préféré du coup vous partez sans merger votre travail sur master, il fera jour demain.

*Nous sommes maintenant **mercredi matin**, **9h00** (bah oui on a bu des bières aussi en plus de jouer au baby) et on rallume le PC. On s'apprête à pousser et tout d'un coup le CTO débarque dans l'open space :*
"Et les krombolz, dans 15 minutes je check la master pour voir à qui je donne les primes sur objectifs alors s'il vous reste un truc à pousser vous avez 15 minutes".

Et là, vous vous rappelez les règles de qualité de dév. qu'il y a sur le Wiki interne.
Dont deux :

- *"Découper au maximum ses commits"*
- *"Effectuer les features en TDD".*

Et là, c'est le drame : sur nos 2 dernières features, on a fait le dev et ensuite on a fait les tests. Alors certes la première règle est respectée (peut-être même un peu trop) mais pas la deuxième. Et le CTO, c'est un robot : bye bye la prime. Bon on a 15 minutes comment je fais pour avoir ma prime → on envoie le rebase interactif (git rebase -i) pour remettre les commits au propre et remettre ceux de tests avant ceux des implémentations, et on gagne notre prime.

3.12. (**Conseil**) Si vous n'aimez pas vi, vous pouvez configurer un éditeur de texte pour réaliser les modifications lorsque git le demande (rebase interactif, message de commit, etc.) ; dans votre gitconfig

```
[core]
editor = 'D:/Programs/Sublime Text 2.0.2 x64/sublime_text.exe'
--wait
```

3.13. Commencez par regrouper vos commits par feature via le rebase intéractif :
Faites en sorte de n'avoir plus que 4 commits (1 d'implémentation, 1 de tests,1 d'implémentation, 1 de tests)


3.14. Vous arrivez sur un écran comme ci-dessous

```
pick 04f192ac premier commit pour calcul (interfaces).
pick a55627ee implémentation d'un calculateur via interpolation quantique.
pick 65aada58 test unitaire pour
CalculateurDeTempsDexecutionViaInterpolationQuantiqueService.
pick aa95588b fix TU
CalculateurDeTempsDexecutionViaInterpolationQuantiqueService.
pick 665a5307 dépendance Jackson + interface service d'export.
pick d7d6cf05 service d'export à fichier plat.
pick e8b33f68 service d'export sous forme de fichier.
pick 9b50792b TU de ServiceDExportTempsExecutionPlainTextTest.

# Rebase 3d30c6db..9b50792b onto 3d30c6db (8 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
```

```
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

3.15. Ici, le but est de regrouper les commits : quelle commande utiliser ? Sachant que la commande mise devant l'id de commit s'applique 'sur' le commit du dessus.

Rappel : Faites en sorte de n'avoir plus que 4 commits (1 d'implémentation, 1 de tests,1 d'implémentation, 1 de tests)
Soyez cool avec votre communauté : renommez proprement les nouveaux commits fusionnés :)

Résultat à obtenir à ce stade :

```
tambori@PC2215 MINGW64
/d/java/workspaces/developpement_dev/jeTestDoncJeSuis-hol (rebase_interactif
u+4-8)
$ git log -5
commit 85dfb369f2678940f8fbe4564fea254e84e12509 (HEAD -> rebase_interactif)
Author: Jerome Tamborini <j-tamborini@efluid.fr>
Date:   Mon Apr 2 11:08:35 2018 +0200

      TU de ServiceDExportTempsExecutionPlainTextTest.

commit f526315a111b4e128f6a136bc302303937fd1372
Author: Jerome Tamborini <j-tamborini@efluid.fr>
Date:   Wed Mar 28 11:25:18 2018 +0200

      [feat] service d'export sous forme de fichier plat.

commit 2dce2556ea2951db5351de2083967a276d42c08d
Author: Jerome Tamborini <j-tamborini@efluid.fr>
Date:   Tue Mar 27 13:16:55 2018 +0200

      [test] TU du service de calcul de temps.

commit f4f7a446d96c717dc97adb8da57de2b6ef03b8b4
Author: Jerome Tamborini <j-tamborini@efluid.fr>
```

```
Date:   Mon Mar 26 17:10:06 2018 +0200

    [feat] Implémentation des beans et des services pour le calcul de
temps.

commit 3d30c6db3b9f450bde23a589ca2feea9f938e567
Author: DMYTRYK Alexis <a-dmytryk@efluid.fr>
Date:   Sun Mar 25 22:05:28 2018 +0200
```

3.16. Amendez le dernier commit avec un message propre

3.17. Inversez le sens des commits lors d'un second rebase interactif

(Parce que le but c'est quand même de réussir à avoir la prime :))

Résultat à obtenir à ce stade :

```
tambori@PC2215 MINGW64
/d/java/workspaces/developpement_dev/jeTestDoncJeSuis-hol (rebase_interactif
u+4-8)
$ git log -5
commit 2c0911989f5b3f2a1bae69b64043bca8736c376a (HEAD -> rebase_interactif)
Author: Jerome Tamborini <j-tamborini@efluid.fr>
Date:   Wed Mar 28 11:25:18 2018 +0200

    [feat] service d'export sous forme de fichier plat.

commit bf4fe2ad386e933e4a123bbaeaccce0acec299e5
Author: Jerome Tamborini <j-tamborini@efluid.fr>
Date:   Mon Apr 2 11:08:35 2018 +0200

    [test] TDD->TU de ServiceDExportTempsExecutionPlainTextTest.

commit 587be14cf8f37451b4b18986bfb1add9894af0cd
Author: Jerome Tamborini <j-tamborini@efluid.fr>
Date:   Mon Mar 26 17:10:06 2018 +0200

    [feat] Implémentation des beans et des services pour le calcul de
temps.

commit 8af60264e2078c193aa997e8e6ef5a17a8d350ec
Author: Jerome Tamborini <j-tamborini@efluid.fr>
Date:   Tue Mar 27 13:16:55 2018 +0200

    [test] TDD->TU du service de calcul de temps.

commit 3d30c6db3b9f450bde23a589ca2feea9f938e567
```

Et voilà la prime est pour nous :).

4. Exercice 4 : `git reflog`

https://git-scm.com/docs/git-reflog

***Jeudi Matin** :*
La veille au soir, on est sorti au bar (normal)
On attaque un nouveau développement.
Puis coup de téléphone !! On switch sur autre-chose (sur un commit précis par exemple) et on ne se rappelle plus de l'emplacement du nouveau développement qu'on était en train de réaliser.

4.1. Positionnez-vous sur "master" proprement : `git reset --hard; git clean -fd; git checkout master`

4.2. Faites une modification de code
- Par exemple : "`touch jeCommenceParEcrireUneDoc.adoc`"

4.3. Commitez cette modification de code

4.4. Supprimez votre commit

4.5. Vous consultez l'historique et vous vous positionnez sur votre précédent commit (en mode "A l'arrache") :
- `git log --oneline -n 10` ⇒ `sélectionner un précédent commit`
- `git checkout -f <mon commit>`

4.6. Consultez le reflog
- Filtrez par branche (ici par la branche `master`)
- Affichez la date et/ou la date relative sur chaque ligne de log
- Identifiez votre commit

4.7. Revenez sur `master` et récupérez le code via un `cherry-pick` sans le commiter directement (car il n'est pas terminé)

4.8. Commitez le code

```
git commit -a -m "XXX"
```

4.9. Oups, une petite coquille : faites une modification (ex : `echo "toto" >> jeCommenceParEcrireUneDoc.adoc`) et amendez le commit

```
git commit --amend
```

4.10. Consultez le reflog

5. Exercice 5 : `git bisect`

https://git-scm.com/docs/git-bisect

Vendredi matin :
Le chef de projet débarque en furie car un bug vient d'être détecté en production. Personne ne l'avait vu avant. Ah bon ?
Vous devez trouver au plus vite le commit à l'origine de ce bug en utilisant la commande git bisect.

5.1. Positionnez-vous sur "`master`" proprement : `git reset --hard; git clean -fd; git checkout master` ou si vous voulez supprimer le commit de la fin de l'exercice précédent : `git reset --hard HEAD^1; git clean -fd; git checkout master`

5.2. Se placer sur la branche de production : "`production`"

5.3. Lancez les tests avec maven => quel est le constat ?

5.4. Quel est le commit qui pose problème ?
Lancez un bisect automatique entre le commit courant et le commit `6d140c7c0176c683269dc9d897dbdb1a6b4b7554` car on sait que celui-la fonctionne.

Notez le temps global qu'a pris le mode automatique. Sortez du mode bisect

5.5. (**Bonus**) Comment, de manière plus visuelle, voir le commit qui a fait planté les tests ?

5.6. Effectuez la même manipulation mais en mode manuel (mode basic dans la documentation de `git bisect`)

(**Bonus**) **On voudrait améliorer le processus, car c'est assez long, pour cela on propose d'utiliser Gradle.**

5.7. Convertir le projet au format Gradle

```
gradle init

=> vérification du fichier build.gradle généré
=> ajout de :

test {
  filter {
      //specific test method
      includeTestsMatching "org.junit.tests.AllTests"
   }
}

Et si nécessaire en cas de problème de récupération des dépendances

repositories {
      jcenter()
}
```

5.8. Lancez un build avec Gradle

5.9. Relancez un bisect automatique. Les performances sont-elles meilleures ?

5.10. Tentez d'améliorer les performances avec les possibilités offertes par Gradle (build cache, parallélisme, etc…)

6. Exercice 6 : `hook` / `alias`

https://git-scm.com/book/en/v2/Git-Basics-Git-Aliases
https://git-scm.com/book/uz/v2/Customizing-Git-Git-Hooks

Samedi matin (oui dans une startup c'est 6 J / 7)
Débat interminable au café sur le Space vs Tabs car dès qu'on fait un commit on arrête pas de devoir changer le formatage du code. Y'a pourtant bien un fichier de formatage éclipse à la racine du projet mais personne ne l'utilise. Pour solutionner le problème la décision est prise d'automatiser le formatage.

6.1. Créez un hook client permettant de remplacer automatiquement tous les tabs des fichiers modifiés par des spaces lors du commit.

[redacted]

[redacted]

[redacted]

[redacted]

- Dans quelle moment du process git le mettre ?

[redacted]

- Créer le fichier de hook au bon endroit en vous basant sur les exemples qui sont fournis dans le référentiel git avec l'extension `.sample`

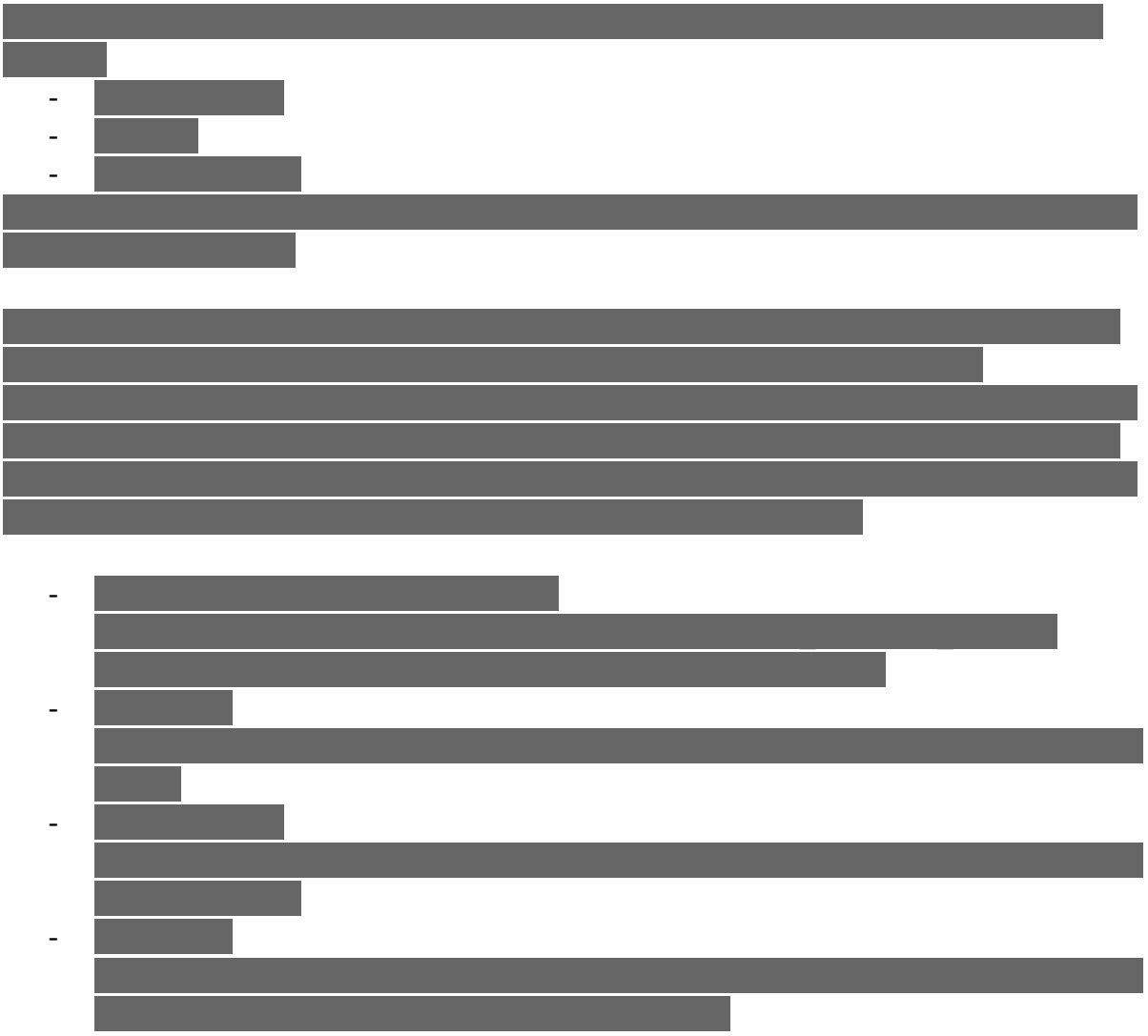
[redacted]

6.2. Testez sur une branche locale avec un fichier source contenant des tabs !

[redacted]

6.3. Comment faire coté serveur pour s'assurer que les fichiers ne contiennent aucun tabs et rejeter les commits si ce n'est pas le cas ?

[redacted]



Ambiance détendue l'après-midi : *c'est le week end après tout ! Réflexion autour de l'efficience de développement , et notamment "comment améliorer les outils du développeur ?" En effet aujourd'hui les développeurs font souvent la même chose mais n'ont pas automatisé tout cela.*

Aliases

6.4. Quels sont les différentes façons/emplacements d'ajouter un alias git

6.5. Créez un alias git '`git oneline`' qui log proprement en une ligne les derniers commit

6.6. Créez un alias git '`git st`' pour écrire plus vite le status

6.7. Créez un alias git '`git logme`' qui log tous ses commits

6.8. Créez un alias git '`git listfiles`' prend en entrée un id de commit et qui liste seulement les fichiers du commit

Bonus'

- En modifiant une seule classe (modifiez plusieurs lignes non consécutives), comment faire pour ne choisir que certaines modifications pour faire un premier commit ?

(A l'aide de la commande magique (`git help`) et de la commande usuellement utilisée pour ajouter les fichiers modifiés dans l'index vous devriez pouvoir vous en sortir ;)!)

- Modifiez l'invite de commande git pour qu'elle affiche le statut courant en permanence

- Ajoutez la complétion des commandes maven dans un terminal (`git bash` sous windows)

[redacted]

[redacted]

[redacted]

- Vous recherchez des métriques sur votre dépôt ? https://github.com/github/git-sizer
- Testez une interface graphique sympa :
    - Git up : *http://gitup.co/*
    - Atlassian sourcetree : https://fr.atlassian.com/software/sourcetree
- Pour s'entraîner à Git : https://learngitbranching.js.org/
- Ajouter des emoji dans les messages de commit (en utilisant par exemple *https://gitmoji.carloscuesta.me*)
- Tapez une commande qui compiler le code, et si tout est ok, qui va créer une pull request github

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

7. Annexe

7.1. Configuration gradle pour un utilisateur d'artifactory

Fichier `build.gradle :`

```
maven {
```

```
  url "${artifactory_contextUrl}/${repo_key_resolve}"
  credentials {
    username = "${artifactory_user}"
    password = "${artifactory_password}"
  }
}
```

Fichier `gradle.properties` dans le projet ou `.gradle` dans le dossier de l'utilisateur

```
artifactory_user=user
artifactory_password=password ou encrypted password
artifactory_contextUrl=http://server:port/artifactory
repo_key_resolve=libs-release
```

7.2. Trucs et astuces

- Première installation de Git ?

Lorsque l'on installe Git depuis le début, il faut penser à faire les configs du mail et name.
Sinon au premier stash on a un message d'erreur :
git stash push -m "mon premier stash"

* Please tell me who you are.
Run
git config --global user.email "you@example.com"
git config --global user.name "Your Name"