

# Formation Git

## XI - Debugger son code

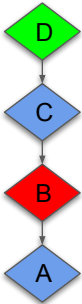


**Arnaud MERCIER**  
arnaud.mercier@hexotech.fr



Nous allons voir un ensemble d'outils et méthodes autour de Git afin d'améliorer la qualité du code ou retrouver plus rapidement les commits qui entraînent une régression

# Voir les évolutions du code



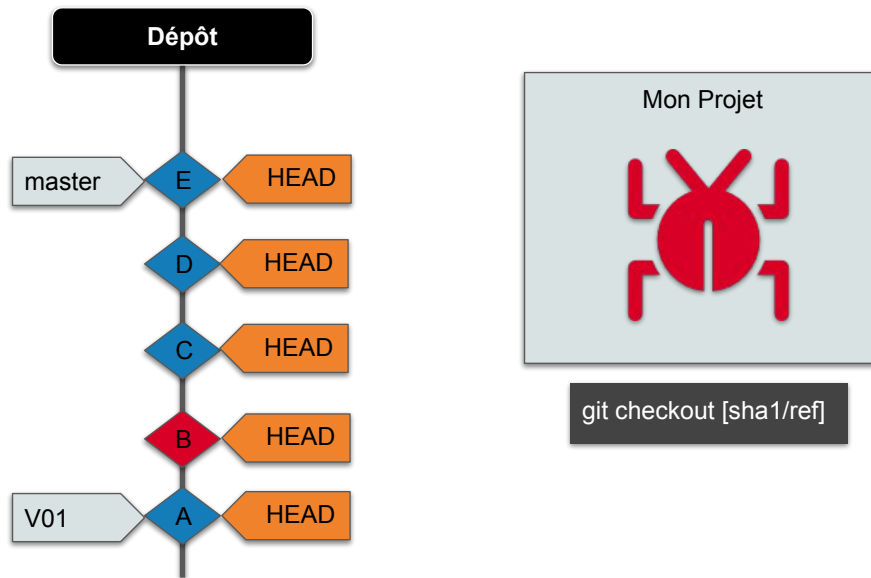
```
$ git diff 92df0e310f1bce1023e8aecdf7e78d7a919dbba9 af6361b380074f135c6b5e2e0baed1fe96885a0b
diff --git a/.gitignore b/.gitignore
index cdf8a87..ae7e9eb 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1,2 +1,4 @@
+tmp/todo.txt
+tmp/*
+*.txt
diff --git a/hello.html b/hello.html
index e733468..ed7a15d 100644
--- a/hello.html
+++ b/hello.html
@@ -26,7 +26,7 @@
<!-- recompense -->
<div>
  <h1> Mes récompenses </h1>
  <p>   </p>
+  <p>  </p>
</div>
<!-- mon gist -->
<div>
  <p> <script src="https://gist.github.com/amgitformation/6ffb2f05ea6ad2a58ef129a15541c978.js">
</div>
<!-- commandes git -->
<div>
  <h1> Mes commandes git </h1>
  <p> git pull </p>
</div>
</body>
</html>
```

diff entre deux commits

```
git diff [sha1 / ref] // Comparaison HEAD et la référence
git diff [sha1 / ref] [sha1 / ref] // Comparaison entre 2 références
```

La commande “git diff” peut également être utilisée pour comparer des versions d’un fichier dans l’historique. Cela permet de voir ce qui évolue entre 2 commits (1 qui est ok et l’autre ko)

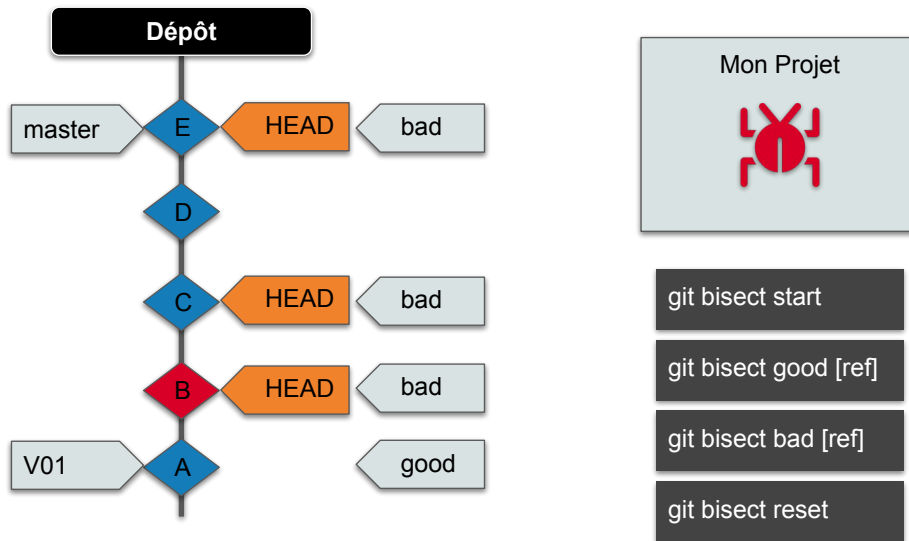
## A la recherche du bug perdu



lorsque l'on recherche un bug dans son code, il est parfois pratique de revenir à des versions antérieures pour trouver à partir de quand le bug est apparu et ainsi isoler les modifications incriminées.

il est parfaitement possible de le faire manuellement via la commande "git checkout". on passe alors de commit en commit et à chaque déplacement, on test notre programme. c'est simple mais pas forcément très efficace.

# Dichotomie: git bisect



Pour accélérer la recherche du bug, on peut penser à la méthode dichotomique. le principe est le suivant:

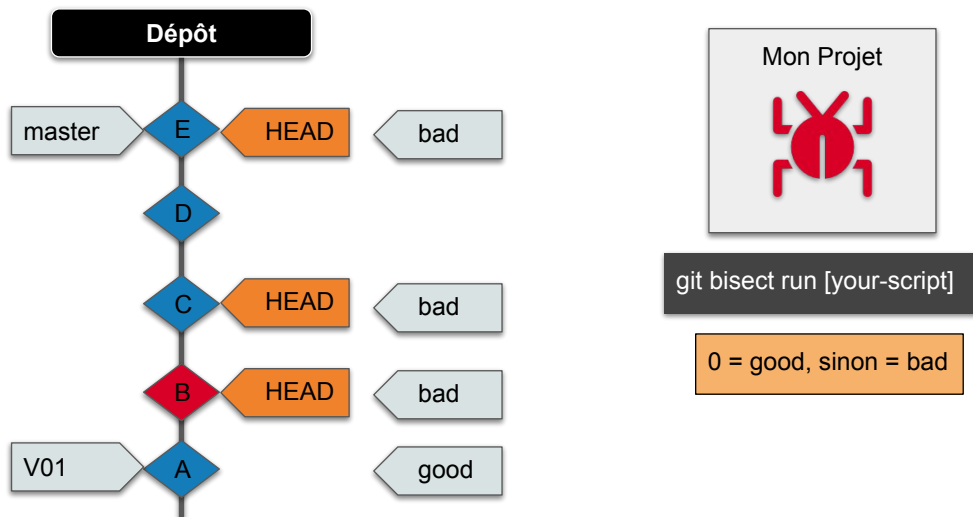
- 1- on prend le commit ou le bug a été découvert et un commit plus ancien pour lequel on est sure qu'il n'existe pas encore (par exemple une livraison client)
- 2- on prend le commit entre ces deux commits sélectionnés. Si le bug est présent alors il est apparus avant le commit en question, sinon il est apparu après. puis on recommence sur ce nouveau sous ensemble et ainsi de suite jusqu'à arriver a un sous ensemble de un commit. A ce moment là nous avons trouvé le coupable!

Biensure on peut appliquer cet algo à la main mais ce n'est clairement pas simple et efficace. heureusement git fournis un outil pour ça aussi: git bisept

la commande fonctionne comme suite:

- 1- lancer la commande "git bisept start" pour démarer la recherche
- 2- utiliser la commande "git bisept bad sha1" pour indiquer le commit avec le bug (sans sha1, git prendra le commit courant)
- 3- utiliser la commande "git bisept good sha1" pour indiquer le commit sans le bug (sans sha1, git prendra le commit courant)
- 4- ensuite l'algo tourne et git vous demande a chaque itération si le bug est présent ou non. Pour dire bug présent il faut indique "git bisept bad" et biensure "git bisept good" pour dire que le bug n'est pas présent.
- 5- quand il reste un seul commit, git vous donne alors les infos sur celui-ci

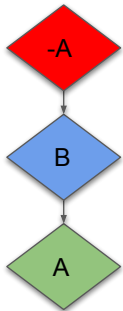
## Dichotomie automatique: git bisect run



L'utilisation de "git bisect" nous aide dans la recherche manuel, mais il y a encore mieux!

utiliser "git bisect" avec en plus en paramètre "run <script>". Cela permet d'exécuter un script à chaque itération pour automatiser la recherche. si le script retourne 0, cela correspond à "git bisect good" sinon cela correspond à "git bisect bad"

# Créer l'inverse d'un commit



```
<body>
- <!-- TITRE -->
- <div>
+ <div id="top-banner">
+   Bienvenue dans cette formation GIT !
</div>
</body>
```

commit A

```
<body>
+ <!-- TITRE -->
+ <div>
- <div id="top-banner">
-   Bienvenue dans cette formation GIT !
</div>
</body>
```

commit -A

git revert [sha1/ref]

Une fois le commit qui pose problème est trouvé, il est possible de le revert pour faire un commit inverse de ce dernier

Attention: cela ne supprime pas le commit de l'historique mais permet de créer un autre commit qui est l'inverse de celui passé en paramètre

# Qui a fait quoi? **Git blame**

```
$ git blame -L 3,7 index.html
```

b1aa30ad	(A.mercier	2018-06-26 15:34:29 +0200	3)	<div>
b1aa30ad	(A.mercier	2018-06-26 15:34:29 +0200	4)	<a href="index.html">
4adb40f7	(John Snow	2019-02-11 10:56:34 +0100	5)	Home
b1aa30ad	(A.mercier	2018-06-26 15:34:29 +0200	6)	</a>
b1aa30ad	(A.mercier	2018-06-26 15:34:29 +0200	7)	</div>

sha1	author	timestamp	line number	line content
------	--------	-----------	-------------	--------------

```
git blame -L [line_start],[line_stop] [file_name]
```

Vous vous êtes sûrement déjà demandé en lisant une code source “mais qui à écrit ça et quand?” et bien git fournit un outil qui répond à vos rêves les plus fous: git blame

la commande affiche alors l'ensemble des lignes du fichier passé en paramètre. Pour chaque ligne, git affiche les informations suivantes:

- sha1 du commit qui a pour la dernière fois modifié cette ligne
- auteur et mail de la modification
- date et heure de la dernière modification
- numéro de la ligne dans le fichier
- le contenu de la ligne en question

# Relecture de code

## Les avantages de la relecture de code:

- Trouver les bugs plus facilement
- Former les nouveaux
- Apprendre de ses erreurs sans rien casser
- Norme de codage
- Faire plus confiance et déléguer
- Maintenir une bonne lisibilité et qualité

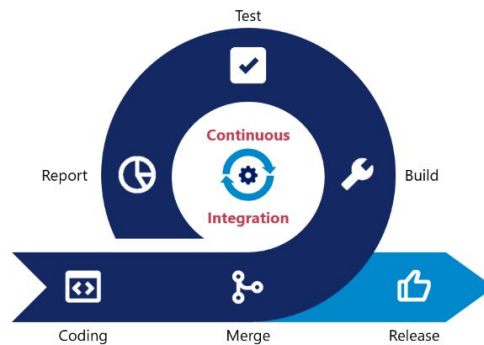


Gerrit est une surcouche à git initié par Google. son objectif, rajouter la relecture de code dans les workflows Git.

l'avantage de gerrit c'est qu'il ne nécessite pas, côté client, d'installer un outil complémentaire ou d'utiliser des commandes particulières. tout peut être fait avec les commandes de base de git.



# CI: Vérifier automatiquement le code



 Bitbucket



Enfin, il est préférable d'avoir une CI qui lance les tests automatiquement sur les commits push

En génie logiciel, CI/CD (parfois écrit CICD) est la combinaison des pratiques d'intégration continue et de livraison continue ou de déploiement continu<sup>1</sup>.

Le CI/CD comble le fossé entre les activités et les équipes de développement et d'exploitation en imposant l'automatisation de la création, des tests et du déploiement des applications. Les pratiques DevOps modernes impliquent le développement continu, le test continu, l'intégration continue, le déploiement continu et la surveillance continue des applications logicielles tout au long de leur cycle de vie. La pratique CI/CD, ou pipeline CI/CD, constitue l'épine dorsale des opérations DevOps modernes.

# TP 11.1

## Scénario

Une régression est apparue dans le code mais vous ne savez pas quelle est l'origine de celui-ci. Vous savez simplement que cette régression est apparue entre la version SITE\_V02 et la dernière version disponible sur le dépôt.

## Objectifs

- 1- Retrouver le commit qui a introduit cette régression. (pour l'exercice, nous allons dire que le bug c'est l'ajout de la médaille d'argent)
- 2- Une fois le bug trouvé, vérifier que cette ligne est toujours présente dans le dernier commit et trouver le nom de la dernière personne à l'avoir modifié.

## SOLUTION:

- il faut utiliser git bisect

## Bilan

```
$ git blame -L [line_start],[line_stop] [file_name]
```

```
b1aa30ad (A.mercier 2018-06-26 15:34:29 +0200 3)  
b1aa30ad (A.mercier 2018-06-26 15:34:29 +0200 4)  
4adb40f7 (John Snow 2019-02-11 10:56:34 +0100 5)  
b1aa30ad (A.mercier 2018-06-26 15:34:29 +0200 6)  
b1aa30ad (A.mercier 2018-06-26 15:34:29 +0200 7)
```

```
<div>  
  <a href="index.html">  
    Home  
  </a>  
</div>
```

sha1	author	timestamp	line number	line content
------	--------	-----------	-------------	--------------

git bisect start

git bisect good [ref]

git bisect bad [ref]

git bisect reset

Nous avons vu ensemble les outils autour de Git afin d'améliorer la qualité du code ou retrouver plus rapidement les commits qui entraînent une régression