Introduction à la programmation en `Java`

Cours 11

Stéphane Airiau

Université Paris-Dauphine

dernier cours

Bilan

Des choses que l'on ne verra pas cette année :

- reflections
- gérer des expressions régulières
- programmation concurrente (thread)
- compiler depuis un programme, exécuter d'autres programme (pas forcément codés en `Java`), une API pour utiliser des langages de scripts.
- beaucoup de détails et de choses plus avancées

javaFX

- moteur graphique appelé `Prism`
- un système de fenêtre appelé `Glass`
- un moteur de media
- un moteur web

On peut donc créer des formulaires, des graphiques, etc...

Interface utilisateur : contrôle

Interface utilisateur : organisation

On peut utiliser différentes classes pour organiser son interface de façon dynamique

layout

- `BorderPane` organise par région (haut, bas, gauche, centre...)
- `HBox` organise le contenu sur une ligne
- `VBox` organise le contenu sur une colonne
- `StackPane` organise le contenu comme un tas
- `GridPane` organise le contenu dans une grille
- `TilePane` organise le contenu selon des cellules de même taille

javaFX :

- transformer chaque élément : rotation, zoom, translation
- on peut utiliser les couleurs, utiliser des dégradés, ajouter des effets (ombre, flou, reflections...)
- on peut modifier l'aspect du curseur
- faire des animations

bref, on peut faire beaucoup !

javaFX : Application

- pour faire une interface graphique, on crée un objet qui va hériter d'une classe `Application`
- ainsi, pour réaliser l'interface, il suffit d'écrire la méthode `public void start(Stage primaryStage)`

La méthode `main` appellera simplement la méthod `launch` de la classe `Application`.

```
public static void main(String[] args){
   Application.launch(MaClass.class, args);
}
```

javaFX : Stage

La méthode `launch` de la classe `Application` va appeler la méthode `start` qui prend en paramètre une **scène**.

La scène (stage en anglais) représente l'interface que l'on bâtit.

```
1 @Override
2 public void start(Stage primaryStage) {
3     primaryStage.setTitle("JavaFX Welcome");
4
5     primaryStage.show();
6 }
```

La fenêtre créée aura pour titre "JavaFX Welcome" et sera visible.
Elle sera à présent complètement vide.

javaFX : Scene

Pas de chance en français, Stage et Scene se traduisent par le même mot : la scène. On va donc essayer de dire la pièce pour Scene.
Tout ce qui apparaîtra dans notre pièce doit être inséré dans un objet `Scene` qui lui même se trouvera dans un objet `Stage`.

```
1 GridPane grid = new GridPane();
2 grid.setAlignment(Pos.CENTER);
3 grid.setHgap(10);
4 grid.setVgap(10);
5 grid.setPadding(new Insets(25, 25, 25, 25));
6
7 Scene scene = new Scene(grid, 300, 275);
8 primaryStage.setScene(scene);
```

- Ici, on prépare un layout sous la forme d'une grille.
- on crée une pièce avec ce layout
- on insert la pièce sur scène.

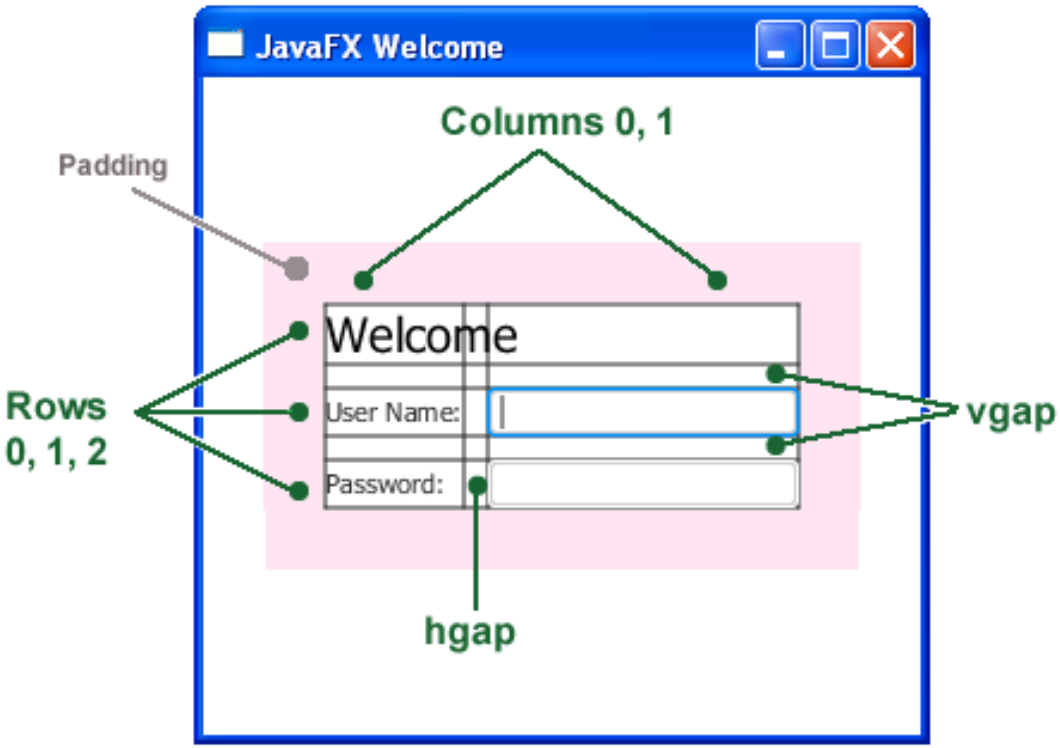
Il ne nous reste plus qu'à insérer les différents composants (noeuds graphiques) de notre interface.

Exemple

On pourrait ajouter des formes (ex un objet de la classe `Circle`), etc... Ici, on va ajouter du texte, un champ pour entrer du texte.

```
1  Text scenetitle = new Text("Welcome");
2  scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
3  grid.add(scenetitle, 0, 0, 2, 1);
4
5  Label userName = new Label("User Name:");
6  grid.add(userName, 0, 1);
7
8  TextField userTextField = new TextField();
9  grid.add(userTextField, 1, 1);
10
11 Label pw = new Label("Password:");
12 grid.add(pw, 0, 2);
13
14 PasswordField pwBox = new PasswordField();
15 grid.add(pwBox, 1, 2);
```

Exemple


JavaFX Welcome
Columns 0, 1
Padding
Welcome
User Name:
Password:
Rows
0, 1, 2
vgap
hgap

Exemple : ajout d'un bouton et d'un texte

```
1  Button btn = new Button("Sign in");
2  HBox hbBtn = new HBox(10);
3  hbBtn.setAlignment(Pos.BOTTOM_RIGHT);
4  hbBtn.getChildren().add(btn);
5  grid.add(hbBtn, 1, 4);
6
7  final Text actiontarget = new Text();
8  grid.add(actiontarget, 1, 6);
```

Exemple : ajout d'un bouton et d'un texte

Gestion d'évènement

A chaque objet graphique, on peut associer un objet de type `EventHandler` qui exécutera une fonction à chaque fois qu'un certain type d'évènement se produit.
par exemple :

- entrée ou sortie de la souris dans une zone
- clic ou relâchement du clic
- mouvement de la souris
- ....

Dans l'exemple ci-dessous, on fera quelque chose à chaque fois que l'on clique sur ce qui est représenté par `object`.

```
object.setOnMouseClicked(new EventHandler<MouseEvent>(){
    public void handle(MouseEvent me){
        // code à exécuter après un clic
    }
});
```

Qu'est ce le code argument de la méthode `setOnMouseClicked` ?

Exemple : Gestoin d'un évènement

Pour un bouton, si on enfonce le bouton (par un clic ou par le clavier), on "actionne" le bouton, donc la méthode s'appelle `setOnAction`.

```
1 btn.setOnAction(new EventHandler<ActionEvent>() {
2
      @Override
      public void handle(ActionEvent e) {
          actiontarget.setFill(Color.FIREBRICK);
          actiontarget.setText("Sign in button pressed");
      }
  });
```

Quand l'utilisateur va presser le bouton, le text "sign in button pressed" apparaîtra en couleur rouge brique.

Exemple : ajout d'un bouton et d'un texte

Exemple

on a donc des actions

- `setOnMouseEntered`
- `setOnMouseExited`
- `setOnMousePressed`
- `setOnMouseReleased`
- `onKeyPressed`
- `onKeyReleased`

grosse librairie

- on peut utiliser CSS
- faire des animations
- ajouter du contenu multimedia
- ...

Git

Gestion des versions

- Collaboration avec d'autres développeur
- Pourquoi une modification, qui l'a effectué ? Ou est-ce que ça ne marche plus ?
- Quel a été l'ordre des modifications ?
- Peut-on revenir en arrière ?

⇨ logiciel de gestion des *versions*

- `git` outil développé par Linus Torvald, auteur du kernel Linux

Différents types d'outils

- modèle centralisé : le code est stocké sur un serveur central
  `SVN`, `CVS`
  problème du point unique de panne ⇨ aucun client ne peut enregistrer des changements.
- modèle distribué : pas de serveur central, toutes les machines ont accès à tout le code
  `git`

⇨ le code est accessible par plusieurs sources, pas de problèmes si le serveur plante...

⇨ pas besoin de se connecter sur un serveur central pour travailler. Il faudra internet pour collaborer.

⇨ chaque extraction est une sauvegarde complète des données !

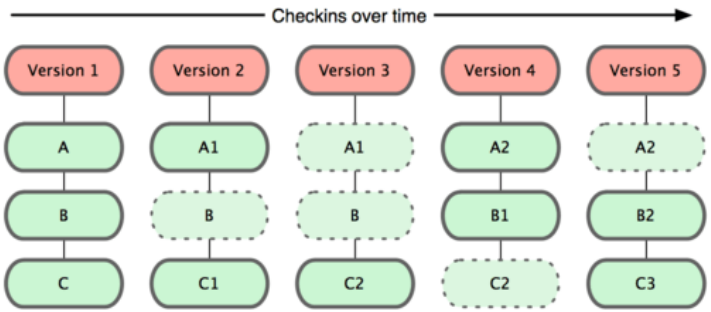
`git` en un slide

- enregistrer les modifications de temps en temps ⇨faire un `commit`
- un `commit` ⇨version du code à un instant
- on obient un historique du projet

⇨ on peut se déplacer dans l'historique du projet

`Git` va enregistrer un nouvel état :

- il n'enregistre pas les fichiers qui n'ont pas changé (ref vers le fichier le plus récent)
- il enregistre les fichiers qui ont changé (pas juste les modifications)
- un peu un mini système de fichiers



⇨ `Git` crée une série d'instantanées, un pour chaque `commit`.

Pertes

Une fois les modifications entrées dans la base, il est très difficile de perdre de l'information.
(avec un dépôt distant)

Beaucoup d'opérations en local

- grande vitesse pour parcourir l'historique
- connaître la différence entre l'état courant et un état de l'historique donné ⇨calculé en local

Intégrité

- Git gère l'intégrité
- tout est vérifié avant d'être stocké
- utilisation d'une somme de contrôle

⇨ si on modifie le contenu d'un fichier, Git va s'en apercevoir !

- identifiant unique : le SHA-1 - une *empreinte* de 40 caractères hexadécimaux (qui contient de l'information)

Démarrer un dépôt `Git`

- initialiser un dépôt `Git`

```
$ git init
```

⇨

création d'un répertoire `.git` contiendra les fichiers générés par `Git`

- pour suivre les versions des fichiers existants

```
$ git add
```

- Obtenir une copie d'un dépôt existant

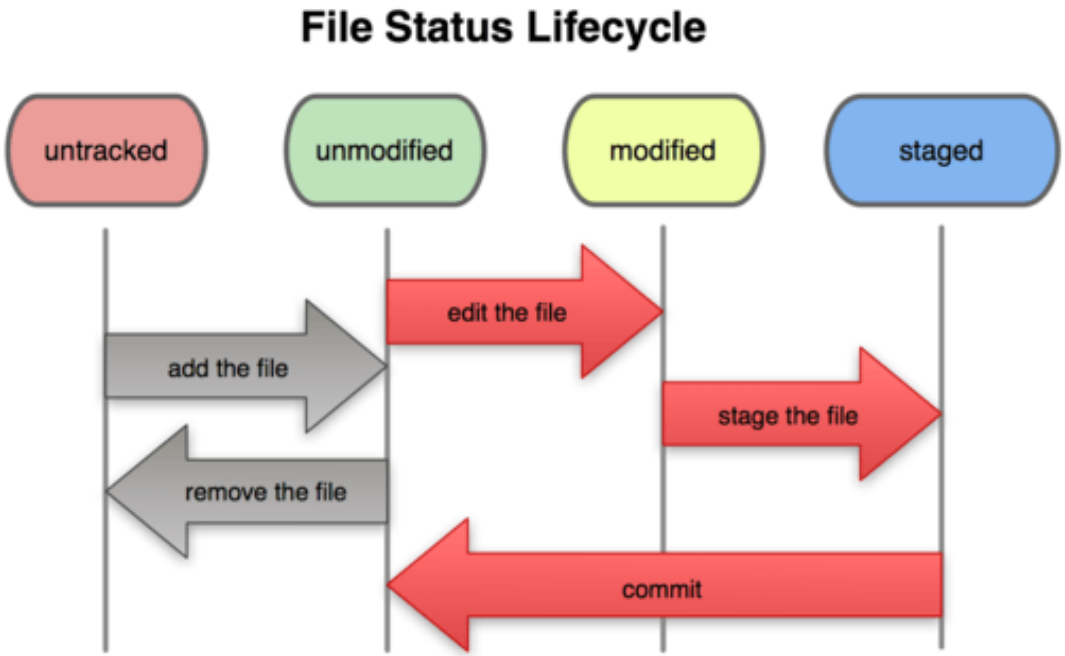
```
$ git clone
```

⇨ reception de toutes les donnés dont le serveur dispose (en particuliers, toutes les versions)

Etat des fichiers

Les fichiers peuvent être sous les états suivants :

- sous suivi de version : fichiers inchangés, modifiés ou indexés
  - modifiés : différents de ceux enregistrés
  - indexés : on dit qu'il faudra les placer dans le prochain dépôt
  - lorsqu'on *commit*, on enregistre les fichiers indexés
- non suivi : on n'est pas obligé de suivre tous les fichiers




```
$ git status
```

déterminer l'état des fichiers

Suivre des fichiers

Pour commencer à suivre un nouveau fichier

```
$ git add
```

Attention, il faut relancer `git add` pour prendre en compte l'état actuel de la copie de travail

Valider les modifications : `commit`

```
$ git commit
```

- vous pouvez/devez ajouter un message d'explication sur la modification que vous validez

⇨ on vous indique sur quelle branche la validation a été effectuée (cf plus tard)

⇨ somme de contrôle `SHA-1`

⇨ combien de fichiers ont été modifié

⇨ quelqes stats sur les lignes changées

L'option `-a` à la commande `git commit` place tout fichier déjà en suivi dans la zone d'index

⇨ évite d'avoir à taper les commandes git add

Historique des validations

```
$ git log
```

donne l'historique des commits réalisés (empreinte sha, auteur, date, message)
On peut

- demander les changements (avec un diff)
- avoir quelques stats (option –stat) (nombre de lignes ajoutées, retirées, liste des fichiers modifiés, ...)
- pas mal d'options sont disponibles...

Annuler des actions!

- erreur classique : on a oublié d'ajouter des fichiers, ou bien on a fait une erreur dans le message ⇨ on aura un seul commit ammendé.

```
$ git commit – amend
```

- désindexer un fichier
- réininitialiser un fichier modifié (`Git` vous dit comment faire dans l'output du commit)

Etiquettage

A vous de choisir ou de nommer les commits importants.

```
$ git tag -a v1.8 -m 'super nouvelle version
1.8'
```

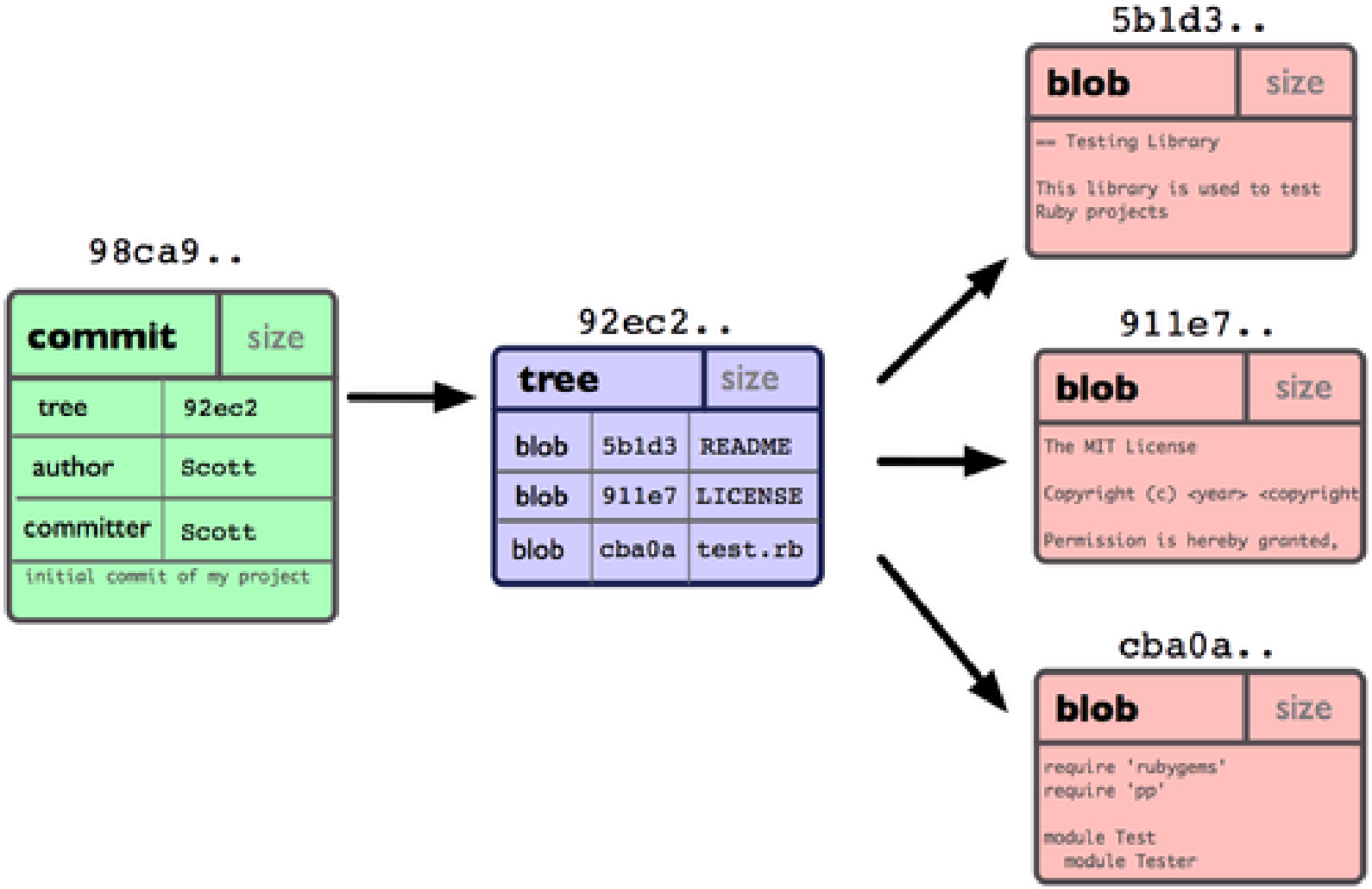
```
$ git show v1.4
```

Les branches

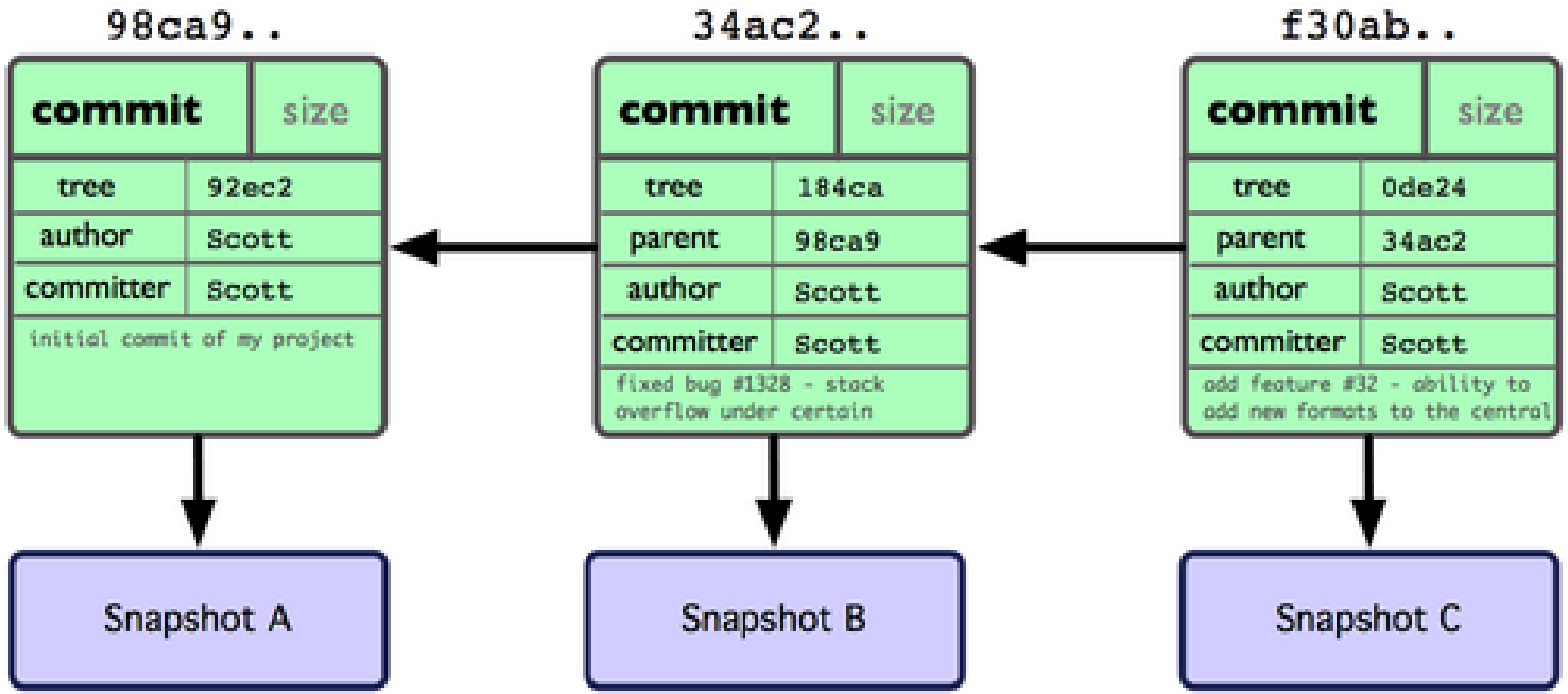
Parfois, on veut commencer une fonctionalité sans complètement savoir si ce sera utile :
On peut commencer à coder cette fonctionalité, tout en laissant les autres continuer sur la ligne principale.

- *ouvrir une branche* : Diverger de la ligne principale sans se préocupper de cette ligne principale.
- une des particularités de `Git` ➪ assez légère
- créer une branche puis fuisionner peut être une bonne façon de développer

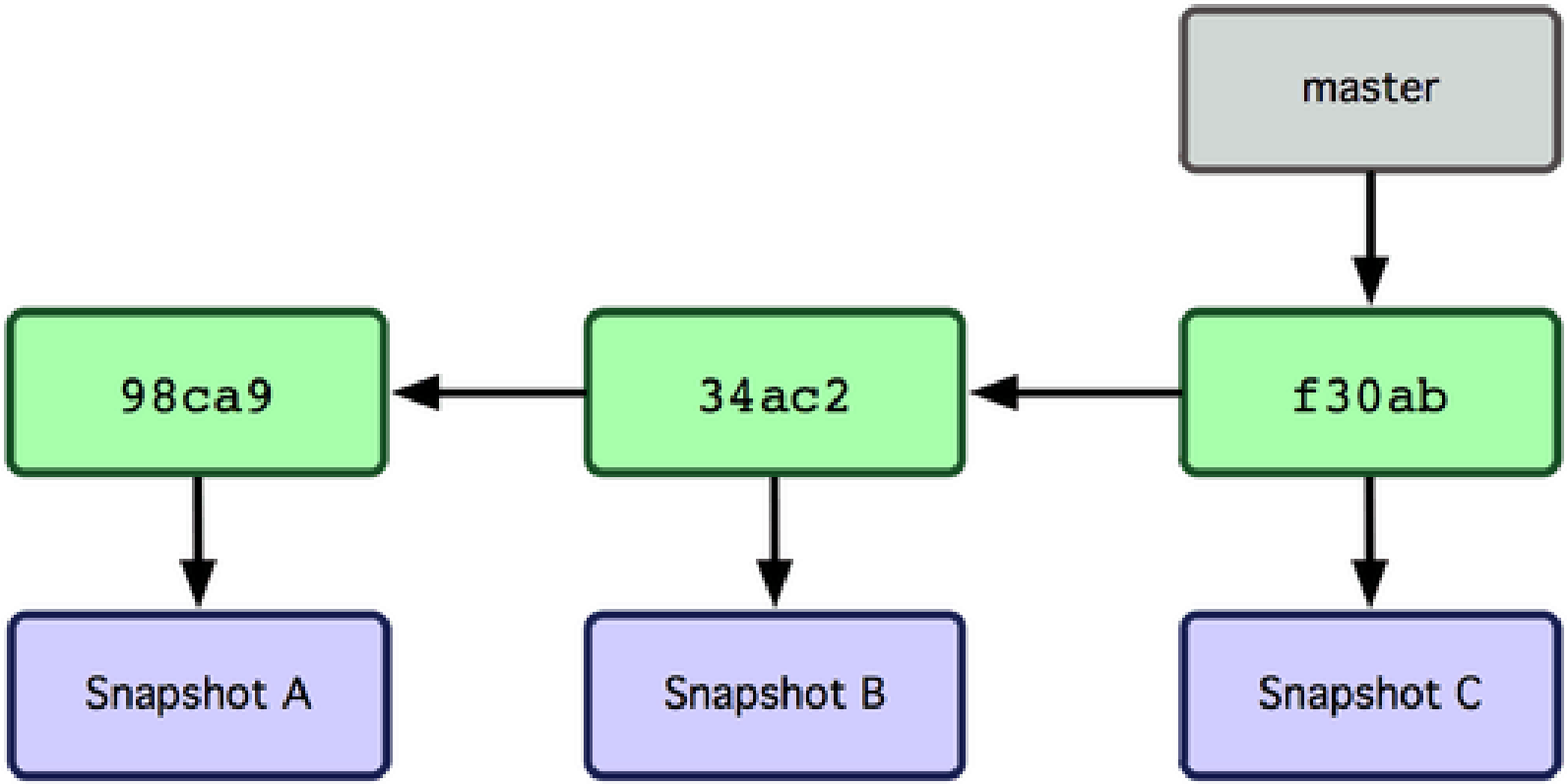
Le commit et son arbre



commit et les parents

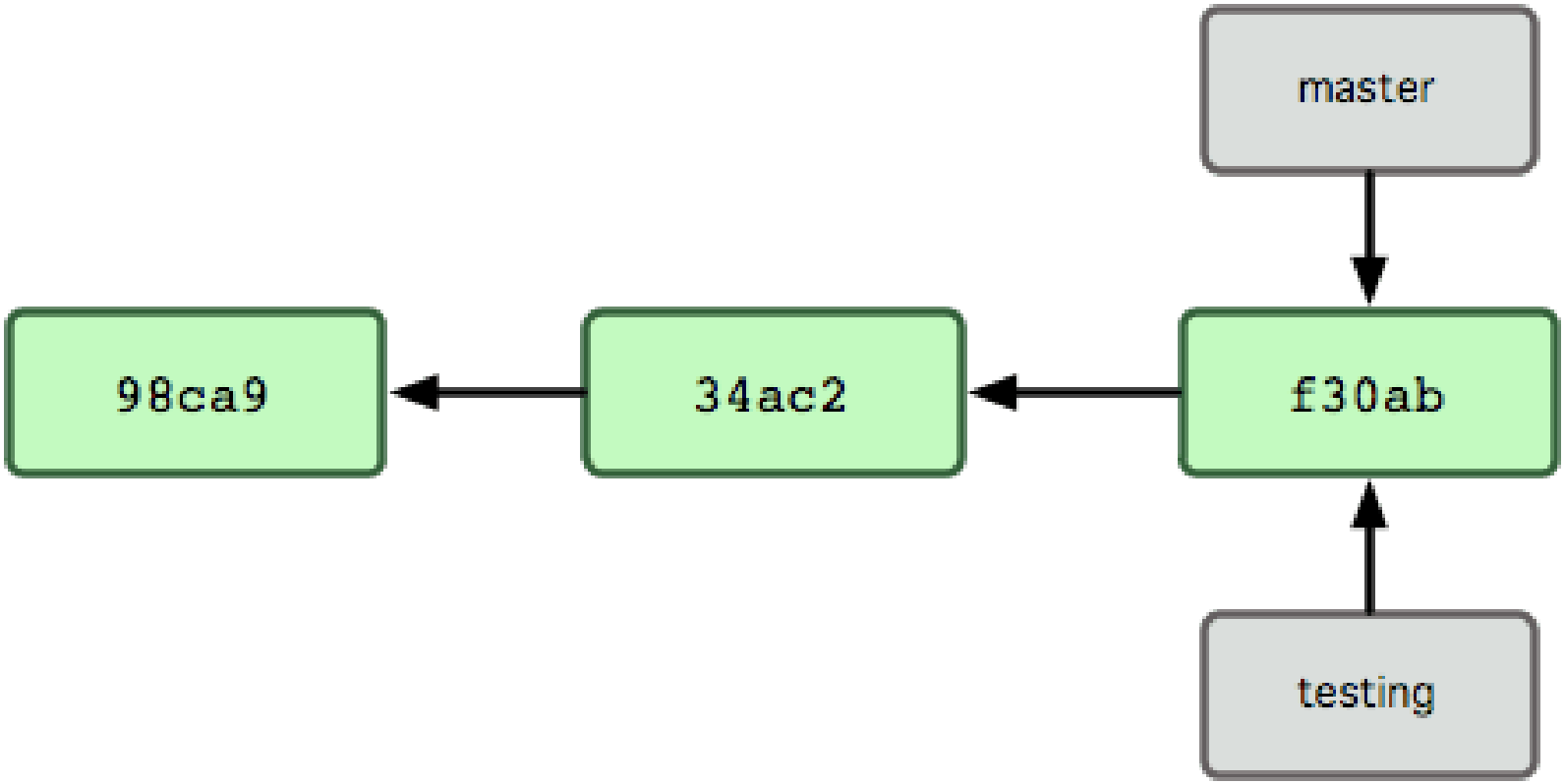


Le master



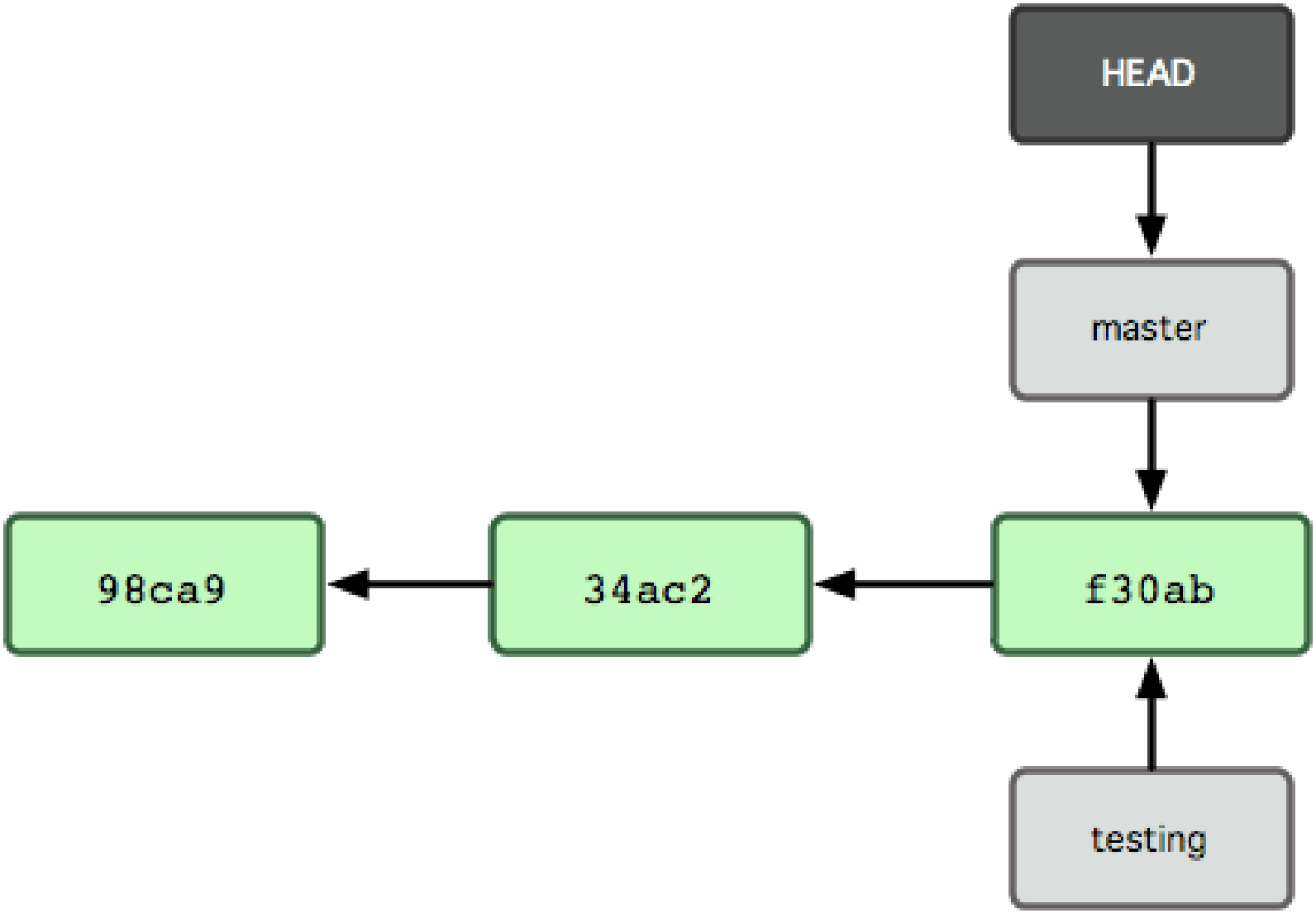
Une branche (qui pointe sur le même commit)

```
$ git branch testing
```



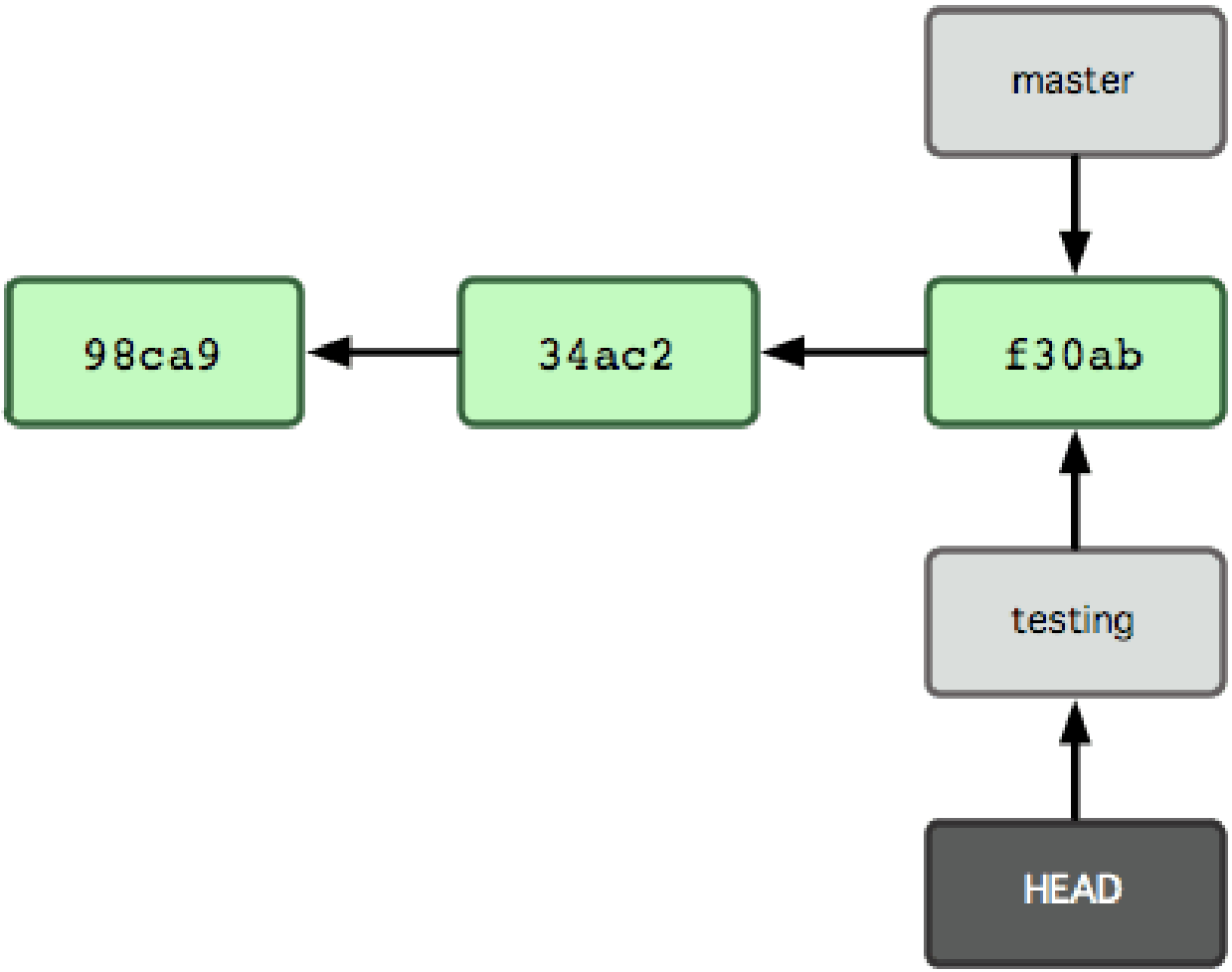
On pointe sur la nouvelle branche

Pointeur spécial appelé HEAD pour désigner la branche sur laquelle vous travaillez.

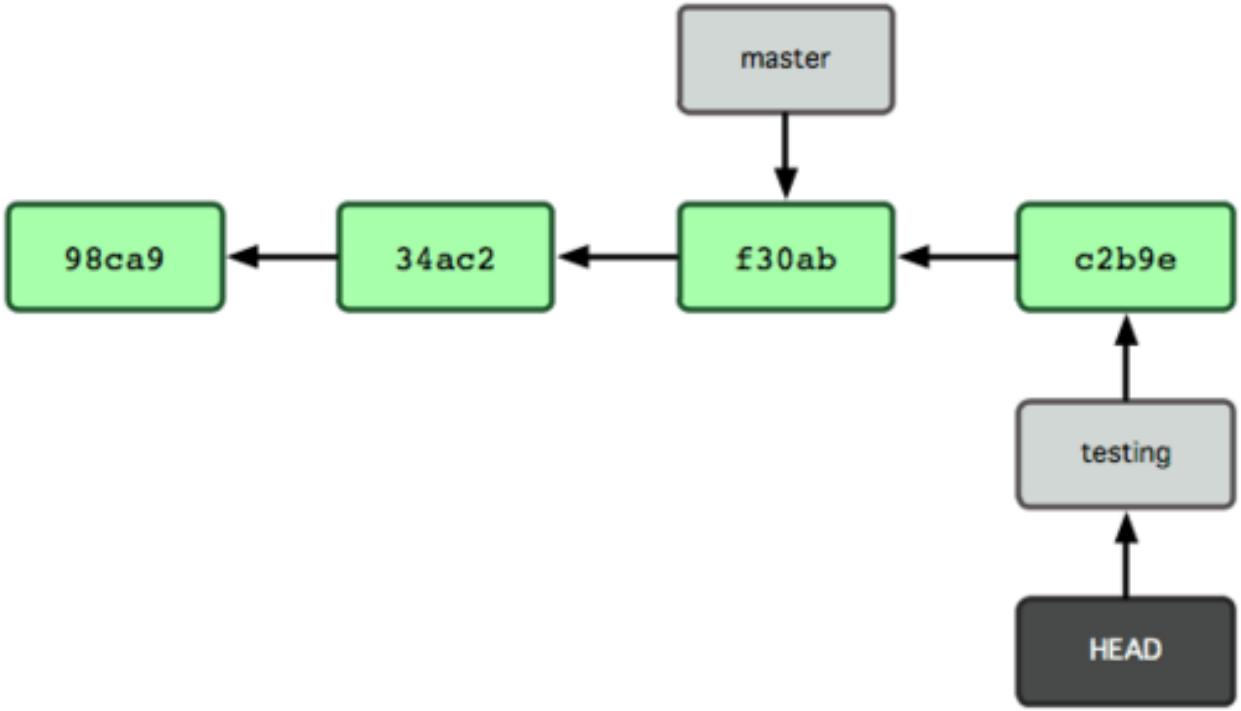


On peut se déplacer sur une autre branche

```
$ git checkout testing
```

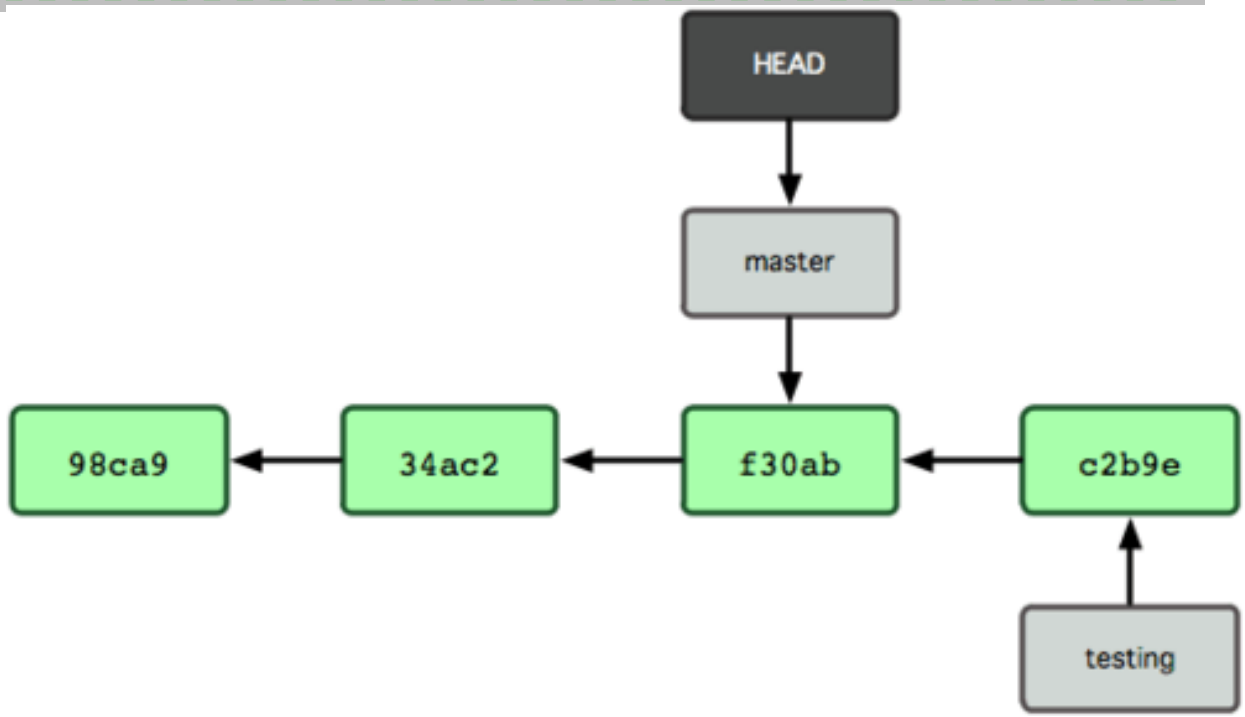


On peut continuer à travailler sur la branche



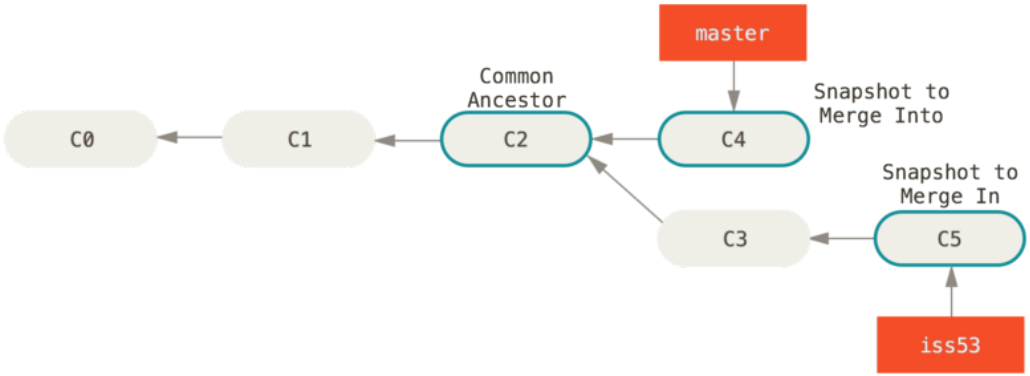
On peut retourner au master

```
$ git checkout master
```

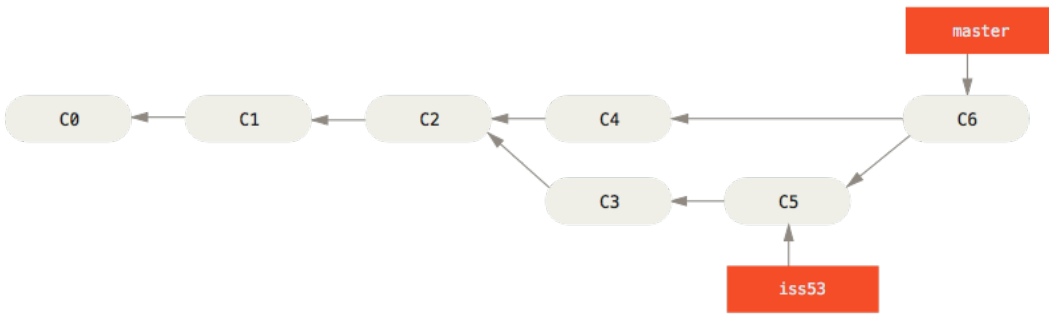


par exemple, quelque chose ne marchait pas et il fallait revenir rapidement à une version qui marche !

Fusion de branches!




```
$ git checkout master
Switched to branch 'master'
$ git merge prob53
Merge made by the 'recursive' strategy.
README | 1 +
1 file changed, 1 insertion(+)
```



Gestion des conflits!

Tout ce qui comporte des conflits et n'a pas été résolu est listé comme `unmerged`
(des régions du code sont différentes et `Git` ne peut pas savoir quelle est la bonne)
`Git` vous indique les zones de conflit
C'est à vous de les régler ! (i.e. éditer les fichiers et régler les conflits !)

Protocoles

- en local : dépôt sur un autre répertoire dans le système de fichier
  ⇨dangereux si la machine plante
- protocole `SSH`
  - accès authentifié
  - facile à mettre en oeuvre
  - mais pas d'accès anonyme
- protocole `git`
  - le plus rapide
  - pas d'authentification (mais on peut coupler avec `SSH`)
  - plus difficile à mettre en place
- avec `HTTP HTTPS`
  - facile à mettre en place, donner un accès public en lecture
  - pas très efficace pour le client

GitHub : hébergeurs de dépôts Git