git

in a nutshell

Einführung in die Arbeit mit git

von Christian Grauer

Agenda

- **git**
  - Grundkonzepte
  - Versionierung
  - Branching
  - Merging
  - Deep-Dive: Everything is a Pointer
  - Tags
- **Remote / GitLab**
  - Remote Repositories
  - Synchronisation
  - What Else?
- **Tower**
  - Installation
  - Aufbau der Arbeitsoberfläche
  - Working Copy
  - History
  - Settings & Preferences
  - Branches, Tags und Remotes
  - Befehlsleiste
  - Navigationsleiste
- **Workshop**
  - Übungen, Fragen, freie Themen

Aufbau des Trainings

- **Monotasking**
  - Zuhören und Verstehen
  - Anwenden und Üben
- **Konzepte verstehen**
  - Verstehen statt auswendig lernen
  - Fokus auf git-Systematik
  - Tower durchschauen
- **Workshop**
  - Praxis und Übung
  - Individuelle Fragen

- **Terminplan:**

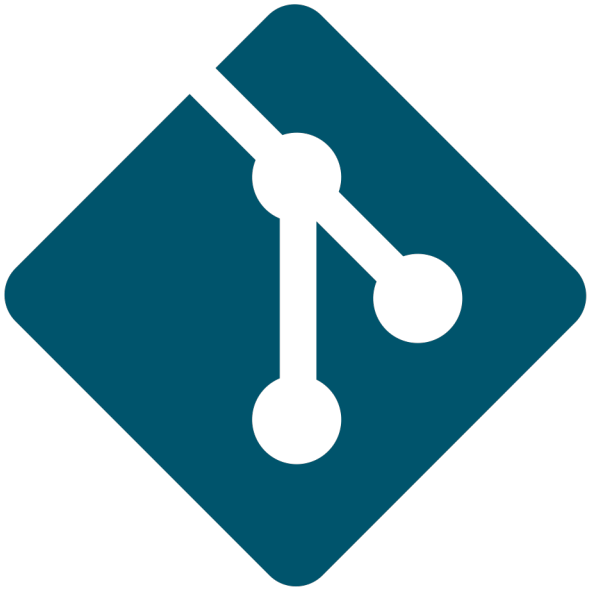
9.00 – 10.30	git
15 min. Pause	
10.45 – 12.00	Remotes
1 h Mittagessen	
13.00 – 14.30	Tower
15 min. Pause	
14.45 – 17.00	Workshop

Grundkonzepte

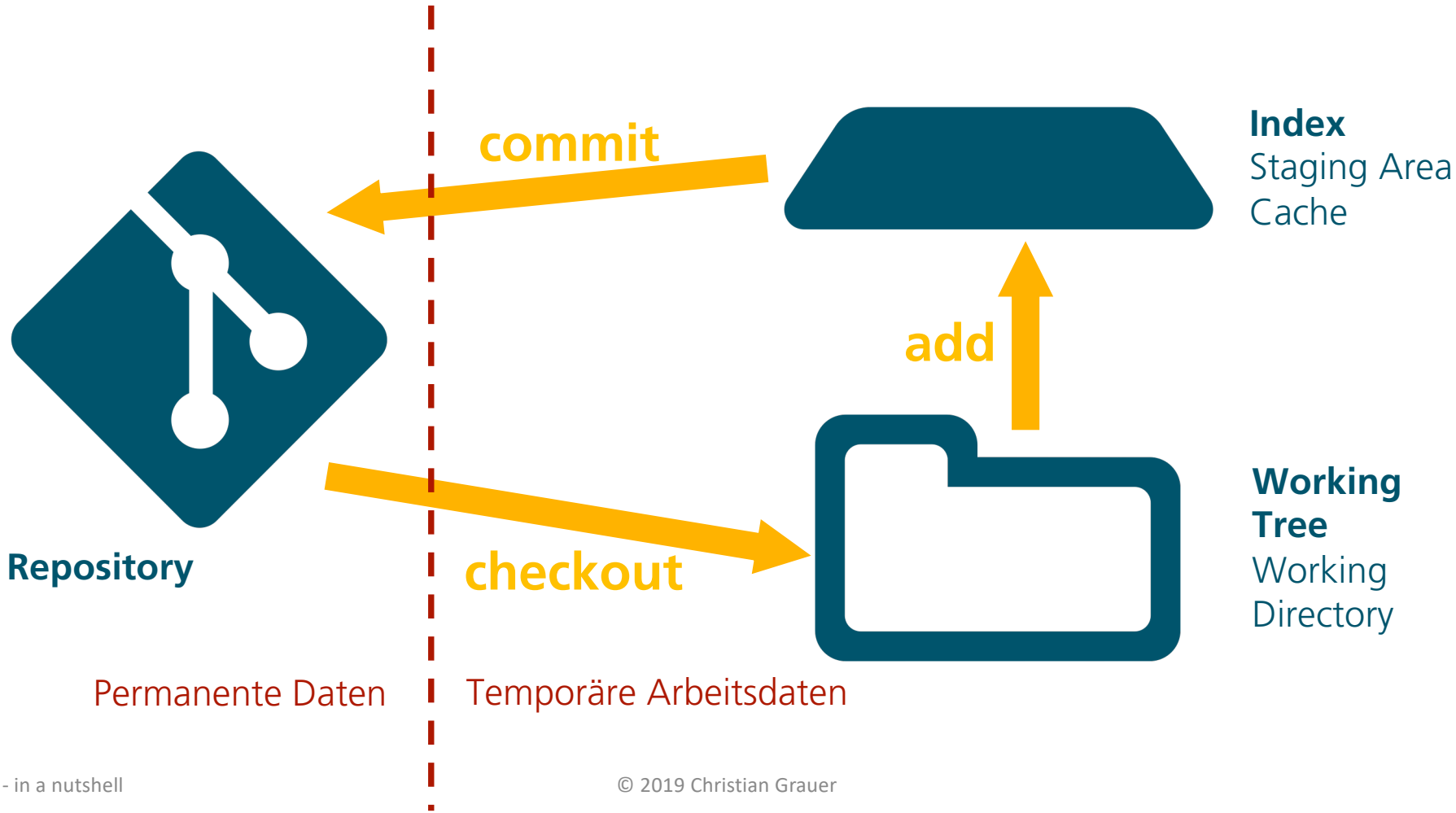
- Was ist git?
- Versionierung & Vergleich (diff)
- Branching und Merging
- Zusammenarbeit, Remote Repositories

Was ist git?

- **git**
  - Linus Thorwalds: Linux
- **Strukturierte Verwaltung von Quellcode**
  - Versionierung
  - Branching
  - Teamarbeit
  - Zentrale Datenhaltung
- **Interface**
  - Kommandozeilen-Tool (für *nix-Systeme)
  - git Bash: Kommandozeilen-Tool für Windows
  - GUI von Drittanbietern, z.B. TOWER



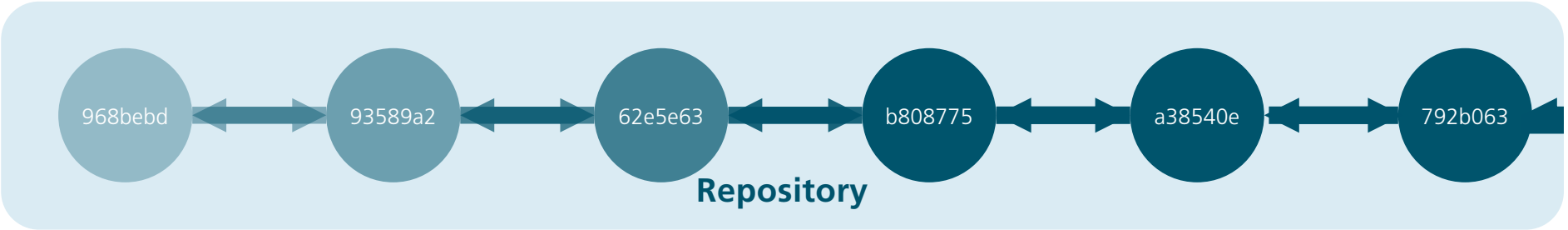
Das Grundprinzip



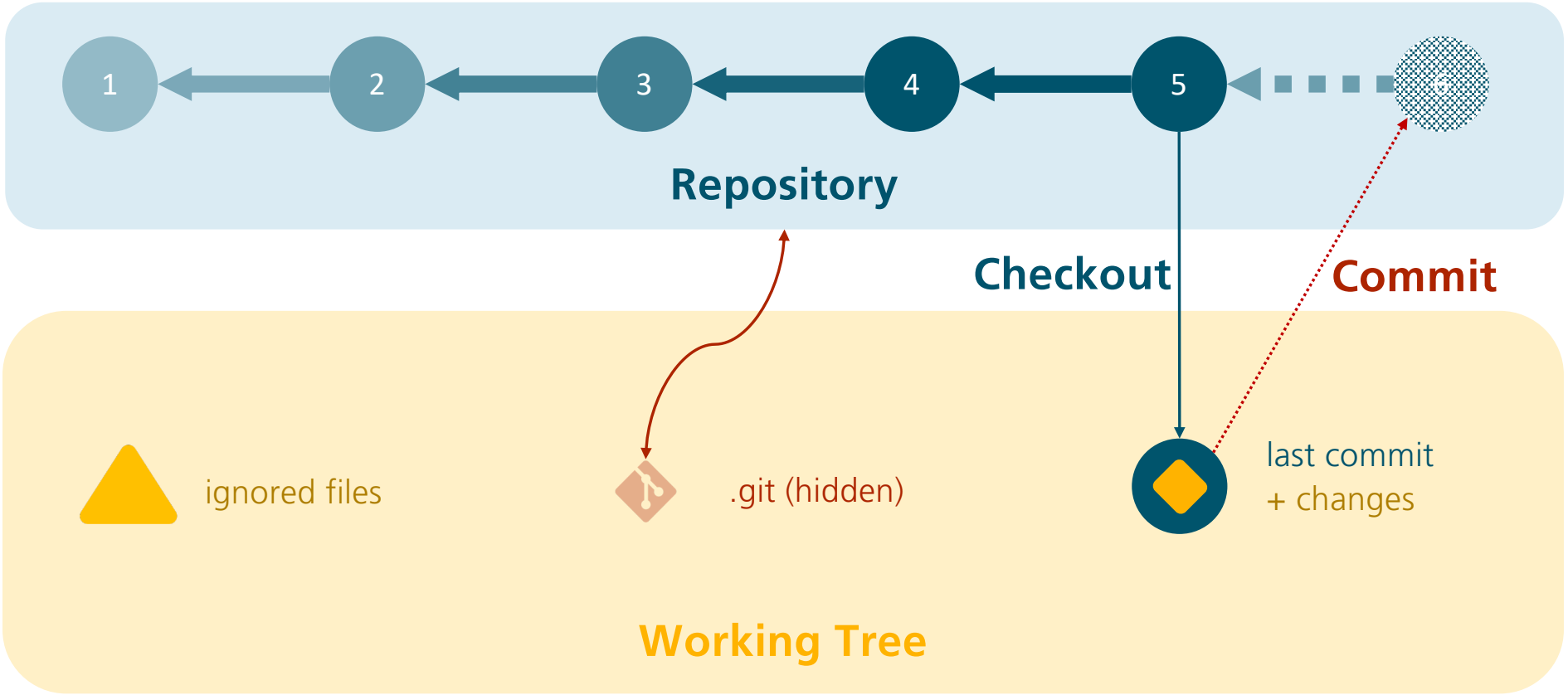
Commit

- **Commit (N) vs. commit (V)**
- **Gespeicherter Entwicklungs-Stand**
  - Commit-ID: SHA-1 Prüfsumme (605e9370b7f5134b679b2ae0...)
  - Autor, Datum, Uhrzeit
  - Kommentar
- **Differenz zum Commit davor**

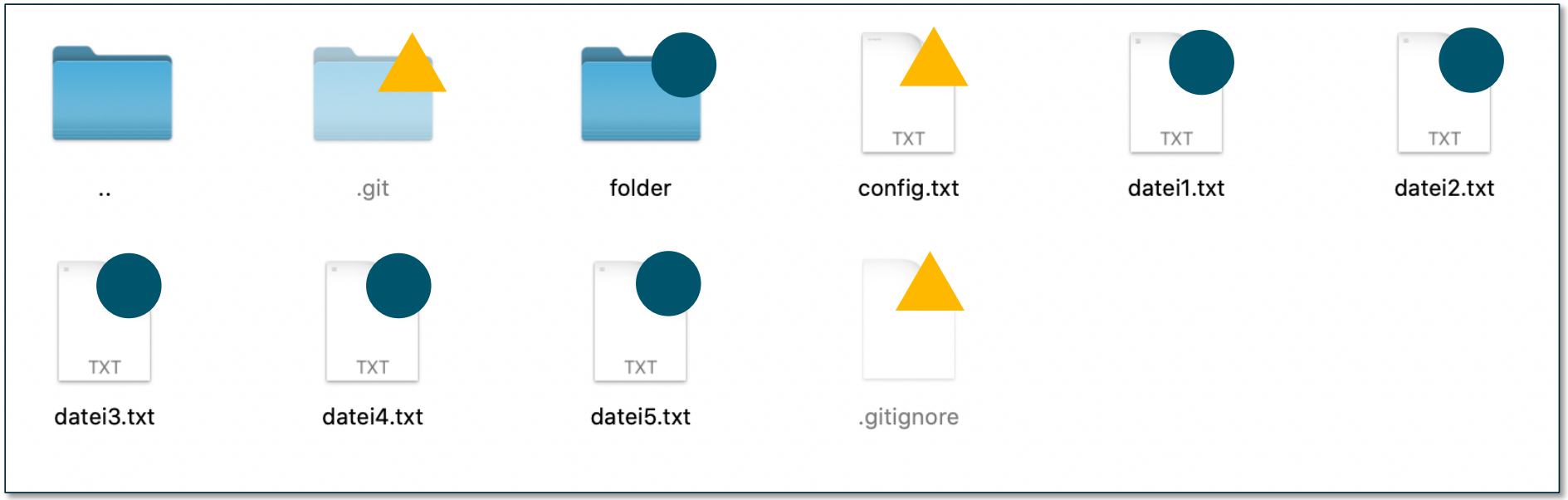
```
cgrauer@MacBook-Pro test % git log
commit 605e9370b7f5134b679b2ae0b2b09261619f1786 (HEAD -> master)
Author: Christian Grauer <mail@cgrauer.de>
Date:   Thu Aug 13 13:20:33 2020 +0200

    masters changes

commit 792b0638940e33ccc7fb42f1a898652b2fe0125f
Author: Christian Grauer <mail@cgrauer.de>
Date:   Thu Aug 13 13:19:42 2020 +0200

    Initial status

commit a38540e3b7ecb5ed466d22dc8e9887667ea0829e
Author: Christian Grauer <mail@cgrauer.de>
Date:   Thu Aug 13 13:14:06 2020 +0200

    diff after

commit b808775ead00d6631ff5e35037feb2babfd8a524
Author: Christian Grauer <mail@cgrauer.de>
Date:   Thu Aug 13 13:13:18 2020 +0200

    diff before

commit 62e5e633eeb37cb7db59b1720b73113c3fb8b19f
Author: Christian Grauer <mail@cgrauer.de>
Date:   Fri Aug 7 12:05:55 2020 +0200

    4.1
```



Repository & Working Tree



Working Tree & Ignored Files



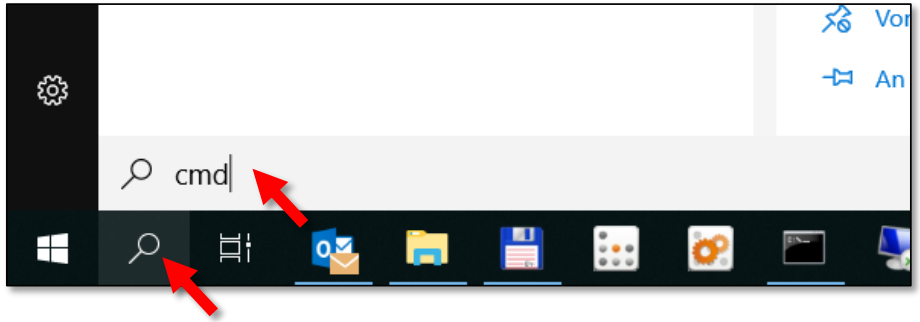

```
1 .*
2 config.txt
```

Kommandozeilen-Befehle

- **git Kommandozeilen-Tool**
  - `C:\Program Files (x86)\fournova\Tower\vendor\Git\bin\git.exe`
- **Erste Befehle**
  - Grundstruktur: `git <befehl> [parameter] ...`
  - Repository erstellen: `git init`
  - Status anzeigen: `git status`
  - Commit vorbereiten: `git add <file(s)>`
  - Commit mit Message: `git commit -m "bla"`
  - Commit auschecken: `git checkout xyz`
  - Log anzeigen: `git log` / `git lg`

Vorbereitungen

- **Terminal starten**
  - Windows Suchfeld
  - “cmd“
- **Suchpfad zu git**
  - `set PATH=%PATH%;"C:\Program Files (x86)\fournova\Tower\vendor\Git\bin"`
- **Arbeitsverzeichnis**
  - `cd Documents`
  - `md git`
  - `cd git`



Übung 1

- **Erstellen eines git Repositorys**
  - Erstell in einem beliebigen Pfad ein Verzeichnis für Dein Repository, gib ihm den Namen „helloworld“.
  - Wechsle in das Verzeichnis und mach es zu einem git-Repository mit dem Befehl: **git init**
  - Prüf das Ergebnis mit dem Befehl **git status**

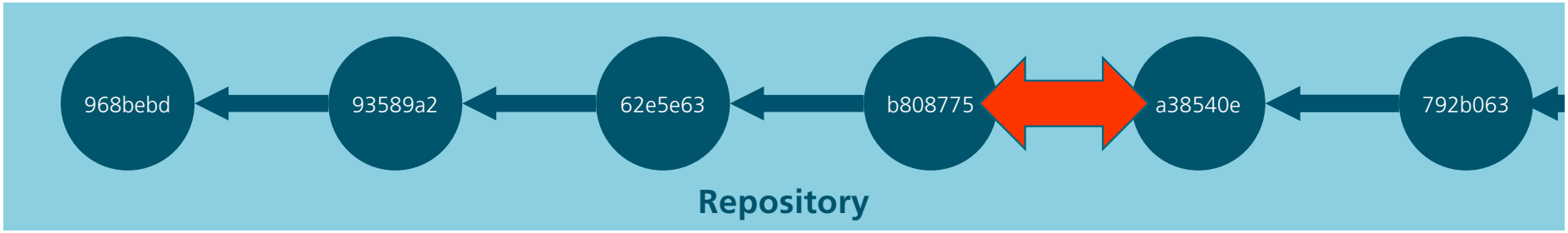
Übung 2

- **Die ersten beiden Commits**
  - Erstell eine Datei *datei1.txt* mit dem Inhalt „Meine erste Datei in git!“.
    - *Direkt aus dem Terminal mit **notepad datei1.txt***
    - *Bereite den Commit vor mit dem Befehl **git add .***
    - *Führ einen Commit durch mit dem Befehl **git commit -m "init"***
    - *Schau Dir zwischendurch immer wieder den Status (**git status**) und das Log an: **git log***
  - Erstell eine zweite Datei *datei2.txt* mit dem Inhalt „Meine zweite Datei in git!“.
    - *Wiederhol die Schritte „**git add .**“ und „**git commit…**“*
  - Schau Dir Dein Arbeitsverzeichnis an (**dir**): wie viele Dateien siehst Du?
  - Check den ersten Commit wieder aus
    - *mit dem Befehl **checkout xyz.** Ersetze xyz durch den SHA-1 Hash des ersten Commits*
  - Schaue Dir jetzt Dein Arbeitsverzeichnis an: was ist passiert?
    - *gibt den Befehl **checkout master** ein.*

Versionierung

- Commits und Vergleich (diff)
- HEAD
- Konkurrierende Commits
- Merge

Commits & Vergleich (diff)

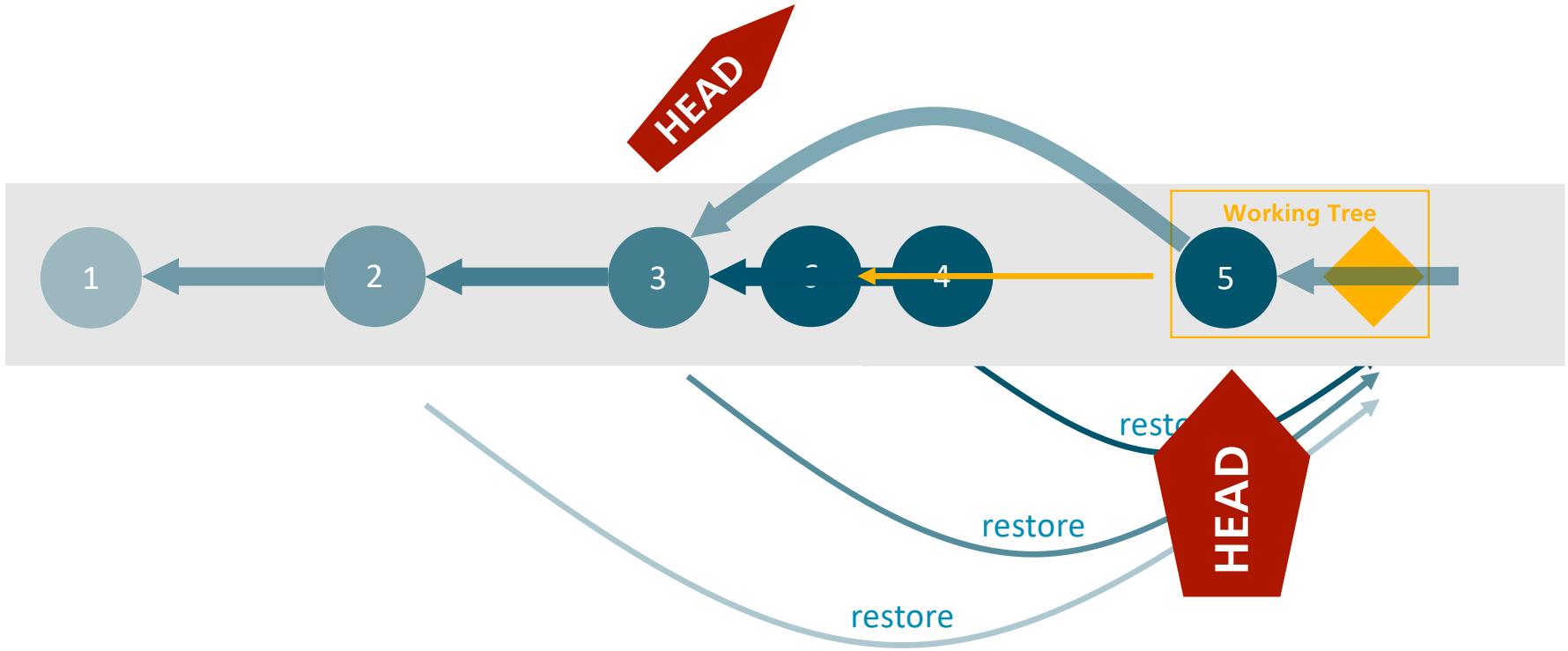



```
[cgrauer@MacBook-Pro test % git diff b808775 a38540e
diff --git a/code.txt b/code.txt
index 900203c..737b3df 100644
--- a/code.txt
+++ b/code.txt
@@ -1,5 +1,7 @@

   _create: function () {
+        // set parameter "widget" to this widget itself
+        this.options.widget = this;
         // Set parameter "element" to the DOM element to which the widget ist bind to
         this.options.element = this.element;
         // Set parameter "window" to the DOM element that will be scrolled. Normally t
@@ -7,10 +9,10 @@
             this.options.window = $('body, html');
             this.options.displayHeight = Math.max(document.documentElement.clientHeigh
         } else {
-            this.options.window = this.element;
+            this.options.window = version;
             this.options.displayHeight = this.options.window.height();
         }
-        this.options.contentHeight = this.options.window.height();
+
         // Set global parameters that are subject to changes or depend on such (can be
         this._recalculate();
```

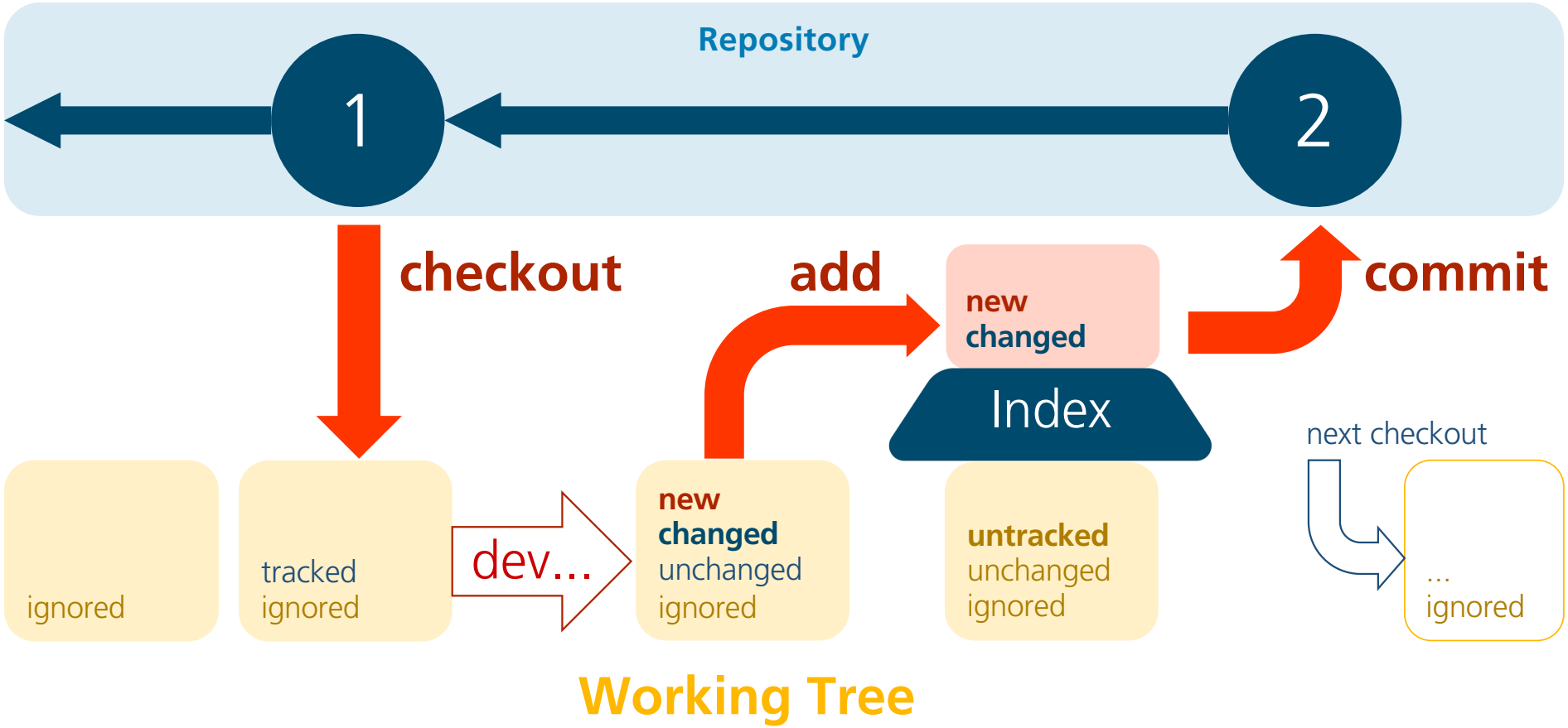
```
[cgrauer@MacBook-Pro test % git diff a38540e b808775
diff --git a/code.txt b/code.txt
index 737b3df..900203c 100644
--- a/code.txt
+++ b/code.txt
@@ -1,7 +1,5 @@

   _create: function () {
-        // set parameter "widget" to this widget itself
-        this.options.widget = this;
         // Set parameter "element" to the DOM element to which the widget ist bind to
         this.options.element = this.element;
         // Set parameter "window" to the DOM element that will be scrolled. Normally
@@ -9,10 +7,10 @@
             this.options.window = $('body, html');
             this.options.displayHeight = Math.max(document.documentElement.clientHeig
         } else {
-            this.options.window = version;
+            this.options.window = this.element;
             this.options.displayHeight = this.options.window.height();
         }
-
+        this.options.contentHeight = this.options.window.height();
         // Set global parameters that are subject to changes or depend on such (can b
         this._recalculate();
```

HEAD and ‚reset‘

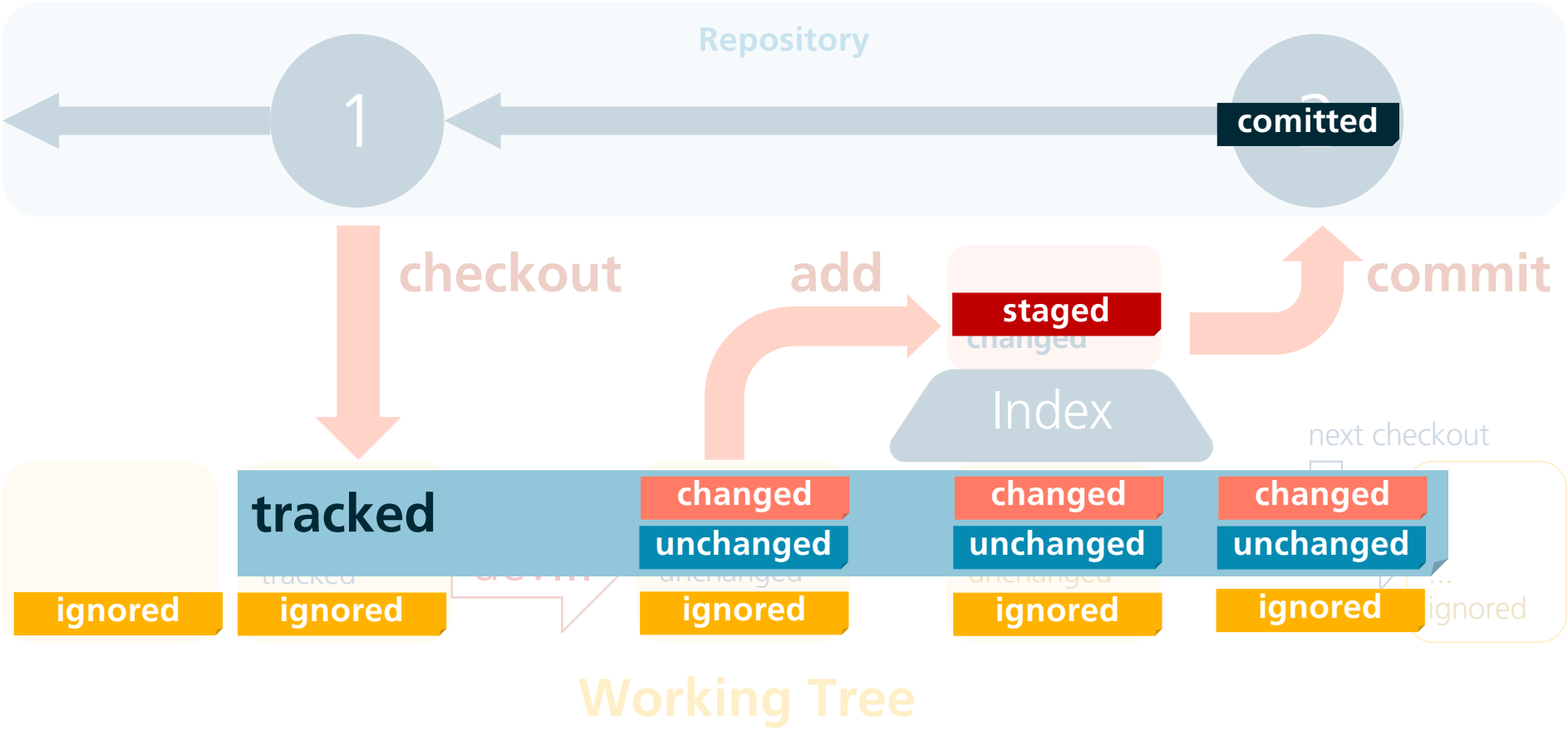
- **git reset 3**



Staging & Working Tree



Staging & Working Tree



Kommandozeilen-Befehle

- **Neue Befehle**
  - Commits vergleichen: `git diff <commit1> <commit2>`
  - Status anzeigen: `git status`
  - Staging: `git add <pattern>`
    - stage all changed: `git add .`
  - Unstaging `git reset <file>`

Übung 3

- **Commits vergleichen**
  - Erstelle die Datei code.js mit dem Inhalt im Kasten rechts
  - Committe Dein Working Tree.
  - Ändere die Datei code.js wie folgt:
    - *Ändere in Zeile 3 das „Hello World“ in „Hello TransnetBW“*
    - *Füge vor Zeile 3 eine neue Zeile mit folgendem Inhalt ein: /* Ausgabetext */*
  - Committe die Änderungen
  - Vergleiche die beiden Commits mit dem Befehl **git diff xyz abc**. Ersetze xyz und abc durch die Hashes der Commits
  - Vergleiche die beiden Commits „anders herum“

```
var World = {
    Hello: function() {
        var content = "Hello World";
        alert(content);
    },
}
World.Hello();
```

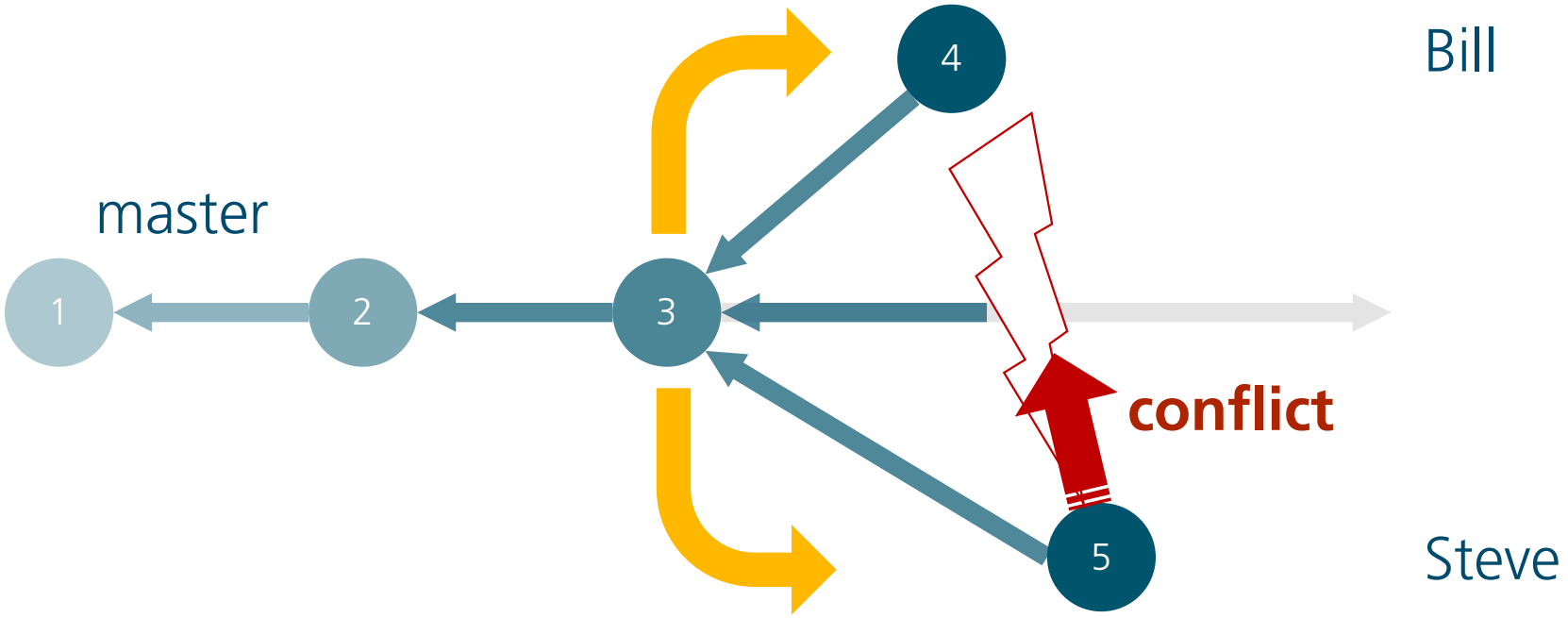
Übung 4

- **Aktuelle Änderungen vergleichen**
  - Bearbeite die Datei code.js wie folgt:
    - *Ändere in Zeile 4 das „Hello TransnetBW“ in „Hallo Stuttgart!“*
    - *Lösch die komplette Zeile 3 (/* Ausgabetext */)*
  - Vergleiche Deine Änderungen mit dem letzten Commit
    - *Tipp: Wenn Du bei diff im zweiten Parameter nichts angibst, wird automatisch mit dem Working Tree verglichen!*
  - Committe Deine Änderungen. Vergiss nicht, einen Kommentar anzugeben (-m „bla...“)

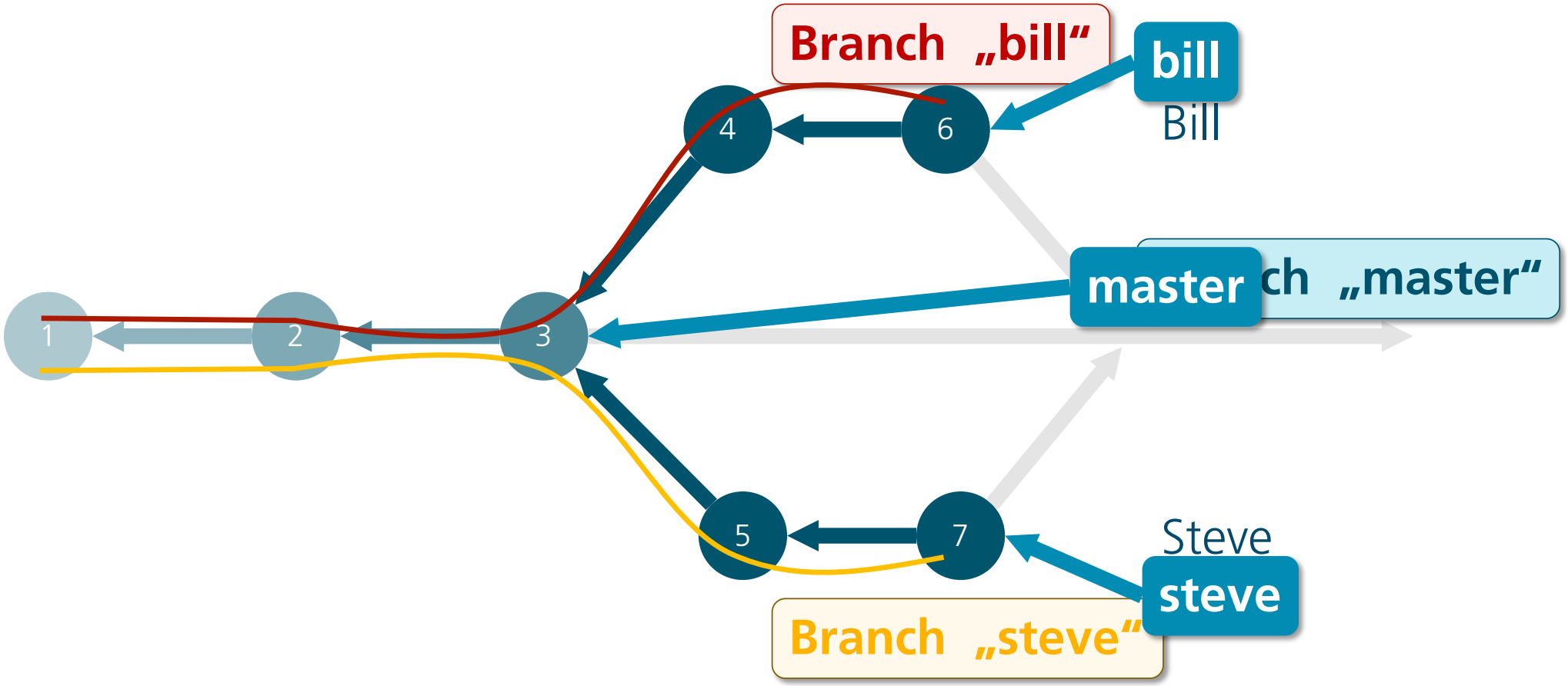
Branching

- Konkurrierende Commits
- Basic Branch
- Die 2 typischen Use Cases
- Branch-Struktur

Konkurrierende Commits



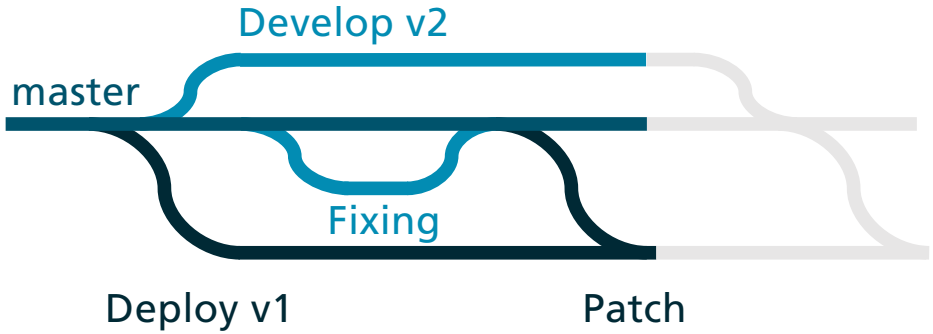
Basic Branch



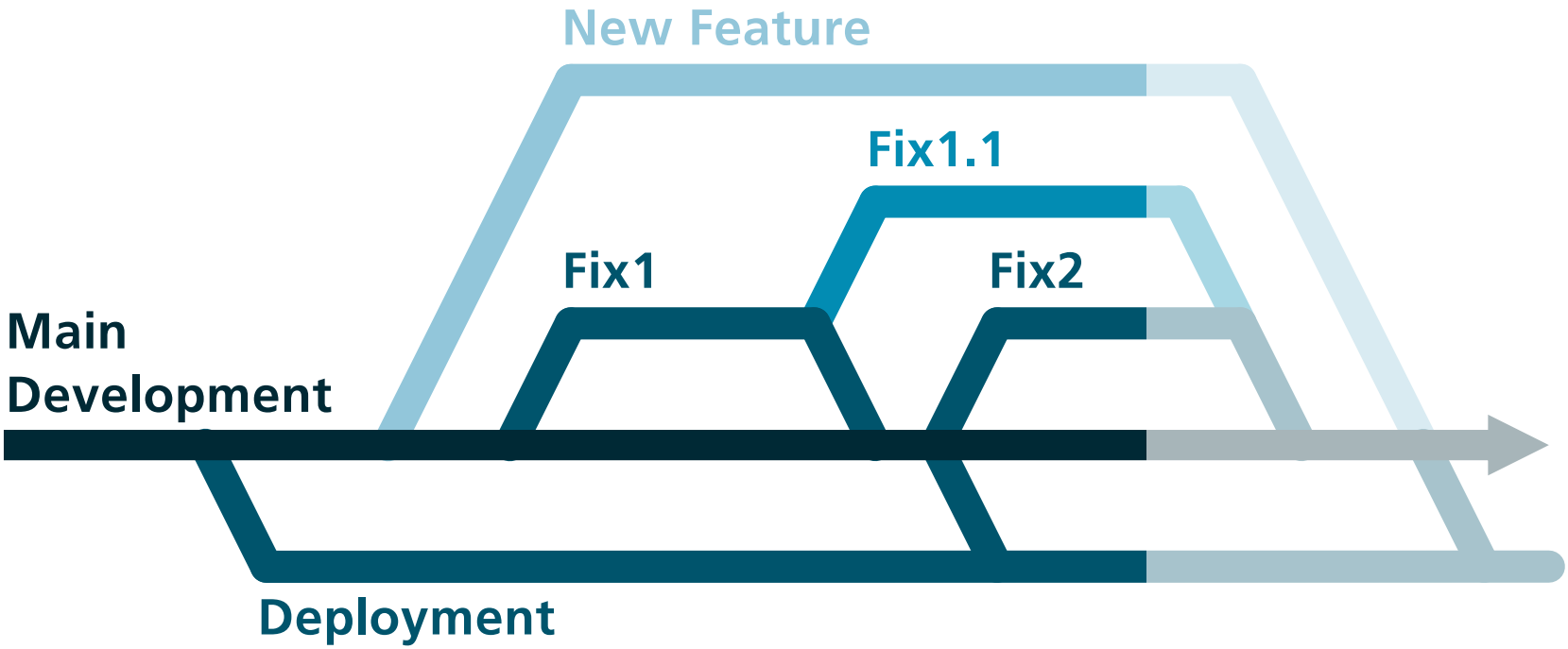
Die 2 typischen Use Cases

- **Multiuser**
  - Mehrere Benutzer arbeiten gleichzeitig an einem Projekt

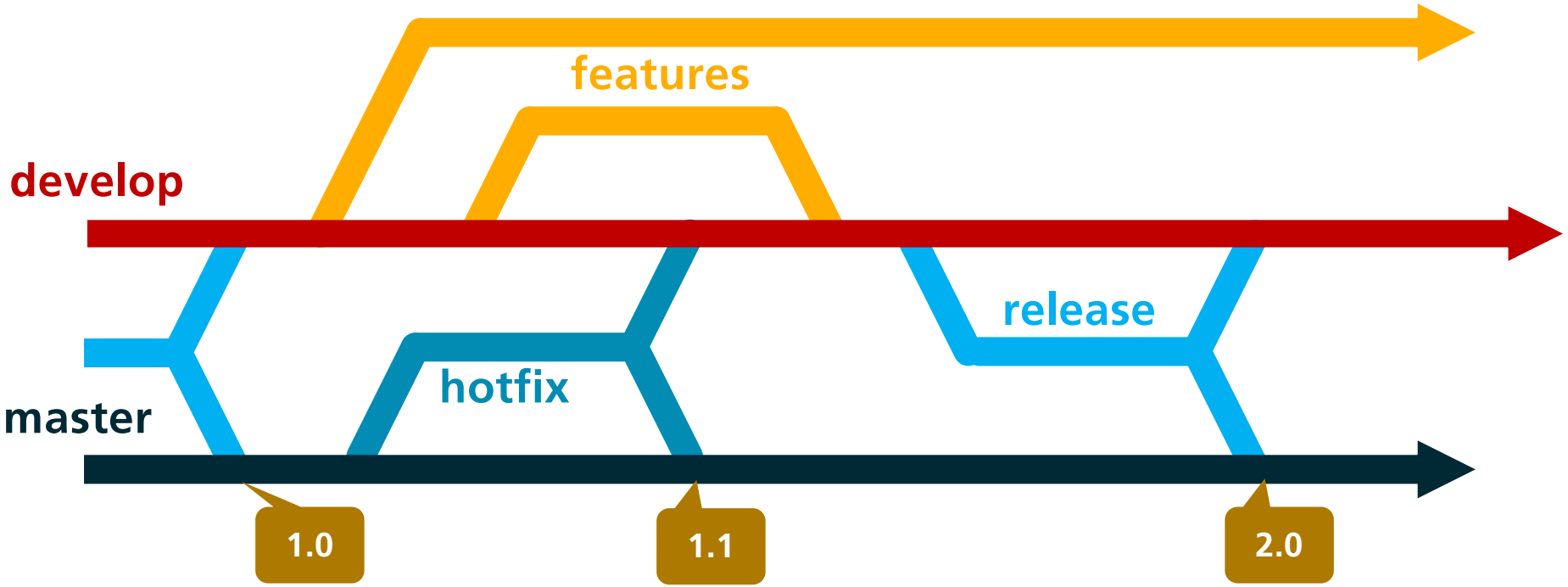
- **Paralelle Versionen**
  - Während der Entwicklung einer neuen Version muss ein Patch für die aktuelle Version erstellt werden



Branch-Struktur



Branch Struktur (nach Flow)



Kommandozeilen-Befehle

- **Neue Befehle**
  - Zeige alle branches: `git branch`
  - Branch auschecken: `git checkout <branch-name>`
  - Neuer Branch: `git checkout -b <branch-name>`

Übung 5

- **Branches anlegen**
  - Erstell einen Branch mit dem Namen „steve“, checke ihn aus
  - Erstell die Datei steve.txt mit beliebigem Inhalt
  - Ändere in der Date code.js den Ausgabetext in „Steve says Hello!“
  - Committe die Änderungen

  - Check den Branch „master“ aus
  - Erstell einen Branch mit dem Namen „bill“ und mach das Analoge zu Steve...
  - Schau Dir die Branchliste und das Log in allen drei Branches an. Wie würde die Branchstruktur aussehen?

Übung 6

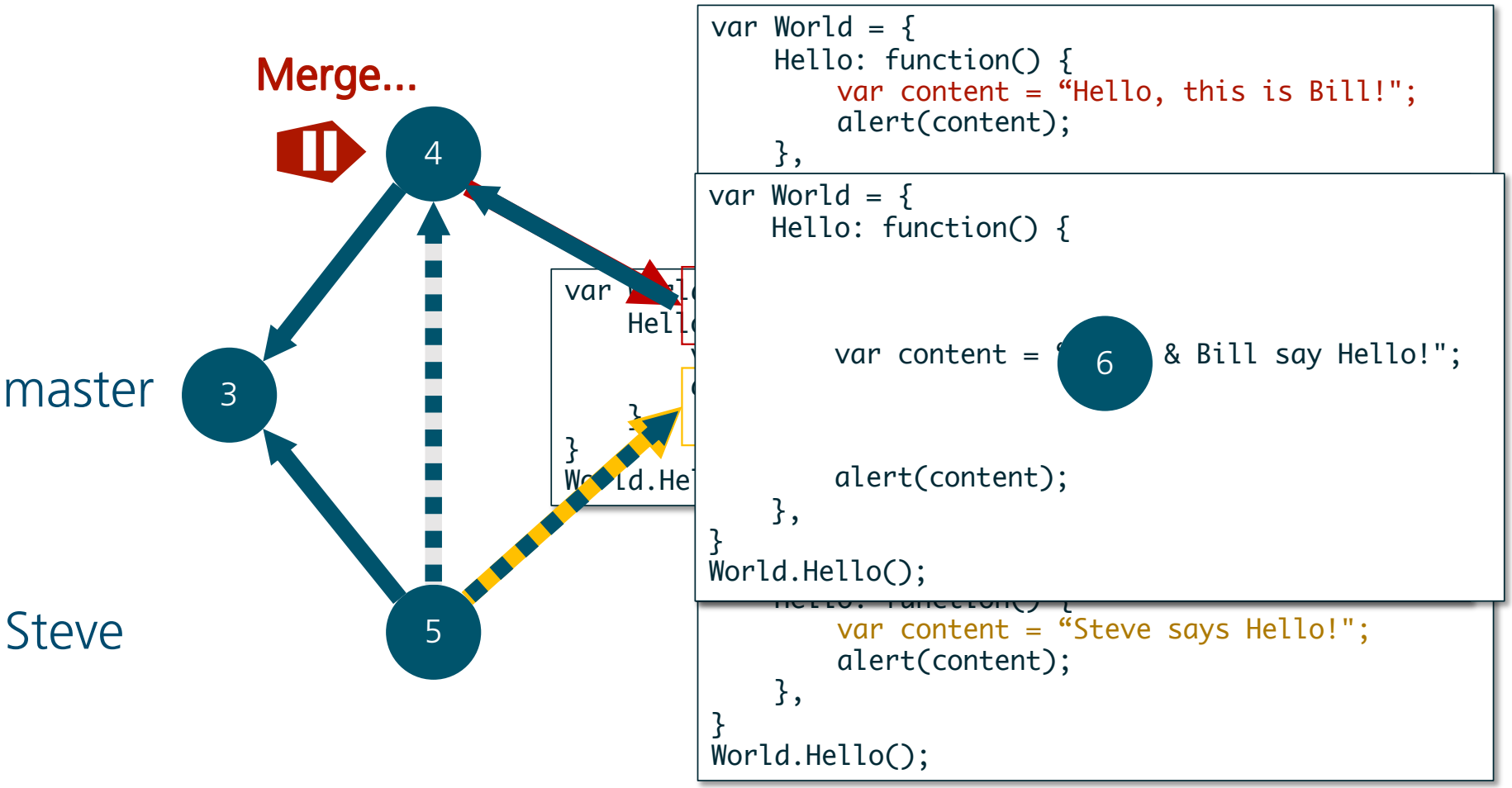
- **Branches vergleichen**
  - Vergleich die beiden Branches „steve“ und „bill“.
    - *Tipp: Branches werden genau gleich verglichen wie Commits*
  - Kopier Dir aus den beiden Branches steve und bill die jeweils neusten Commits heraus (git log)
  - Vergleiche nun diese beiden Commits
  - Vergleiche den Vergleich der Commits mit dem der Branches. Was fällt Dir auf?

Merging

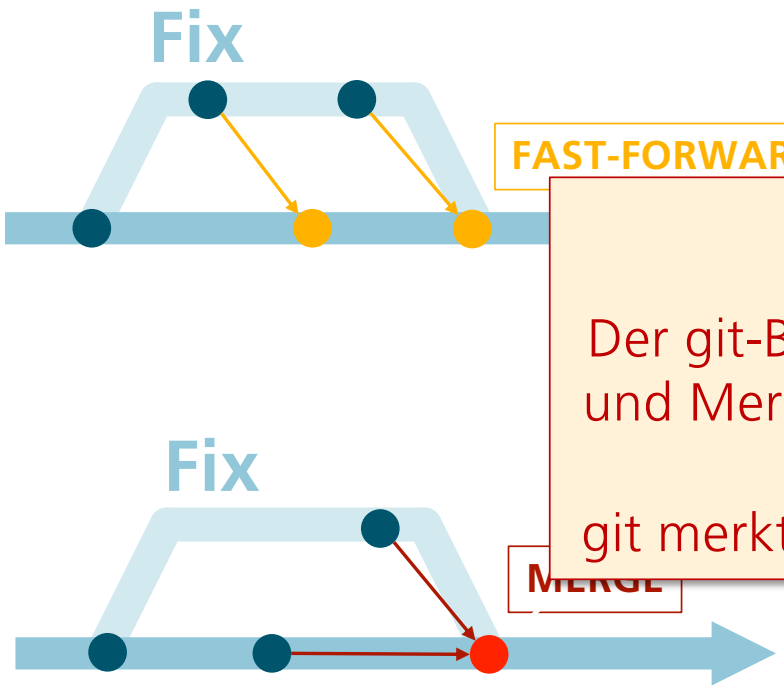
Mit dem „Merging“ werden Branches zusammengeführt, d.h. nicht aufeinander folgende Änderungen (Commits) konsolidiert.

- **Merge**
- **Merge & Fast Forward**
- **Rebase**

Merge



Merge: Fast-Forward



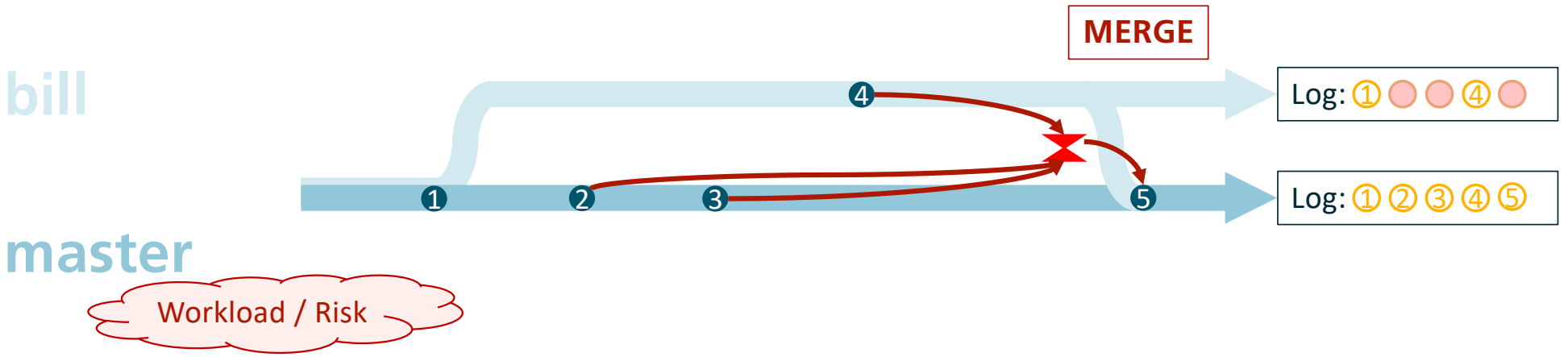
- **Fast Forward**
  - Da im Ziel-Branch keine Commits gemacht wurden, [illegible] Commits des Fix1 [illegible]gefügt werden

> **!**
>
> Der git-Befehl für Fast-Forward und Merge ist derselbe: **merge**
>
> git merkt selbst, was zu tun ist!

[illegible]Branches wurden [illegible]en gemacht und müssen nun "vermischt" (merge) werden.

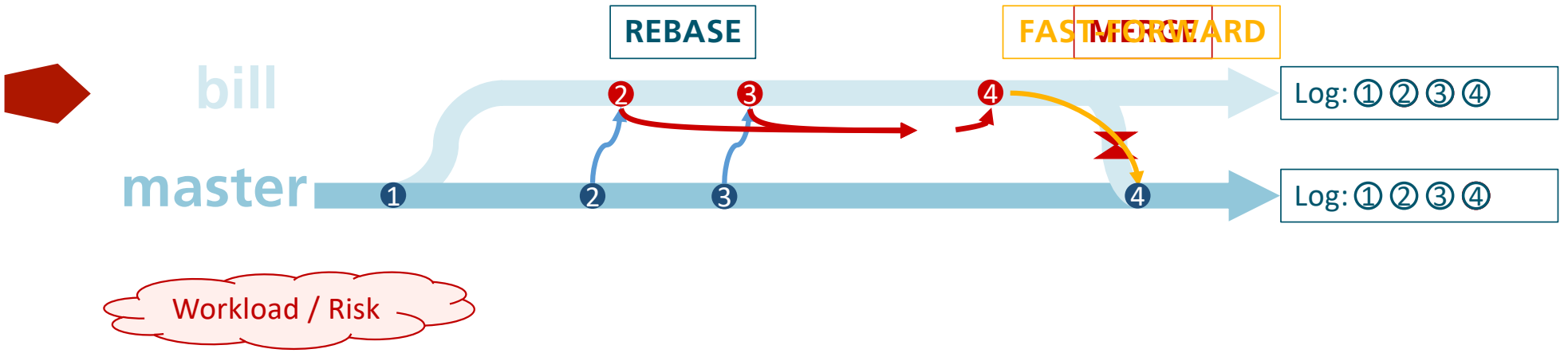
- *ggf. manueller Aufwand!*
- *mögliche Fehlerquelle!*

Merge: Conflict Management



Rebase

```
git checkout bill
git rebase master

git checkout master
git merge bill
```



Rebase Workflow

- Den Branch auschecken, der rebased werden soll („bill“)
  - `git checkout bill`
- Starte Rebase
  - `git rebase master`
- Bei Konflikten: manuell beheben od. mit Merge-Tool
- Bearbeitete Dateien dem Index hinzufügen
  - `git add .`
- Rebase fortsetzen (kein commit!!)
  - `git rebase --continue`

Kommandozeilen-Befehle

- **Neue Befehle**
  - Merge oder Fast Forward starten:
    ```
    git merge <branch (od. commit)>
    ```
  - Merge nach Konfliktbehebung abschließen:
    ```
    git add .
    git commit -m „blah...“
    ```
  - Rebase starten:
    ```
    git rebase <branch (od. commit)>
    ```
  - Rebase nach Konfliktbehebung fortsetzen:
    ```
    git add .
    git rebase --continue
    ```

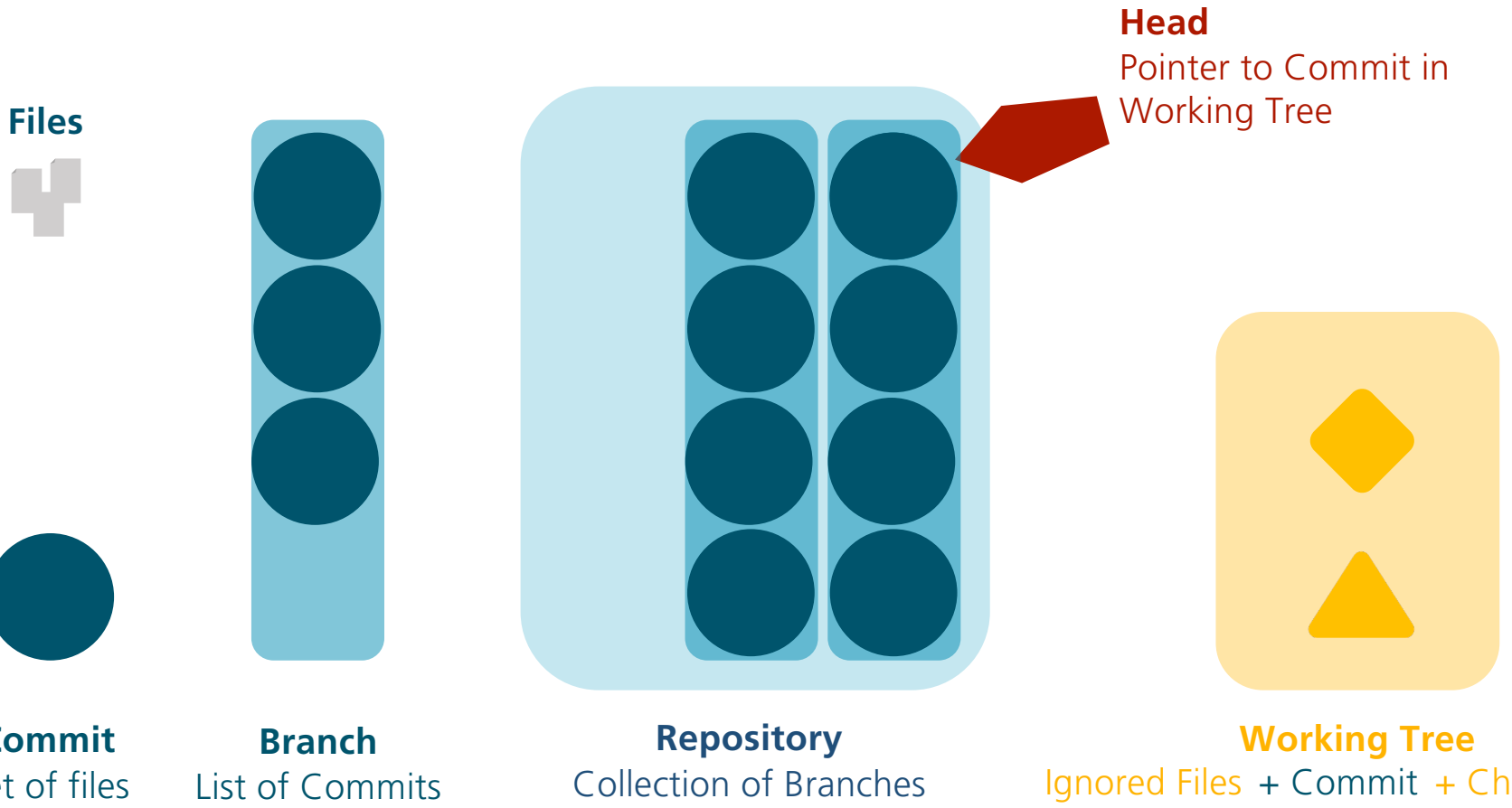
Übung 7

- **Steve und Bill mergen**
  - Checke den Branch „master“ aus und merge den Branch „steve“
    - *Welche Art von Merge wird git anwenden, „merge“ oder „fast forward“?*
  - Schau Dir das Log von „master“ und von „steve“ an. Was fällt auf?
  - Merge jetzt den Branch „bill“
    - *Schau Dir den Konflikt an, behebe ihn aber nicht! - Was machen wir stattdessen?*
  - Gib folgenden Befehl ein: git merge --abort
  - Checke „bill“ aus und mach einen Rebase
  - Behebe den Konflikt manuell mit „Steve and Bill say Hello!“ und setze den Rebase fort.
  - Checke „master“ aus und mach noch einen Merge

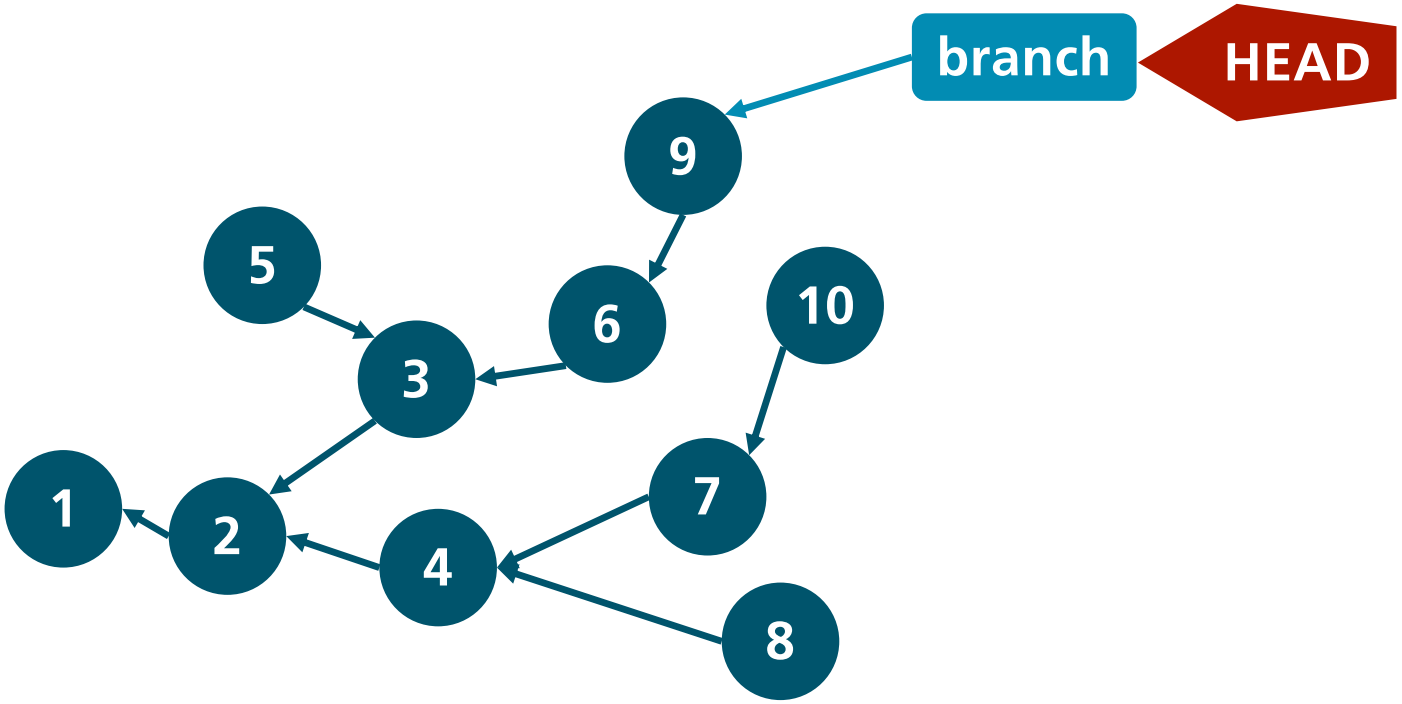
Everything is a Pointer!

- Everything is a Commit
- Everything is a Pointer
- Checkout vs. Reset
- Detached and Re-Attached HEAD
- Rollback

Everything is a Commit

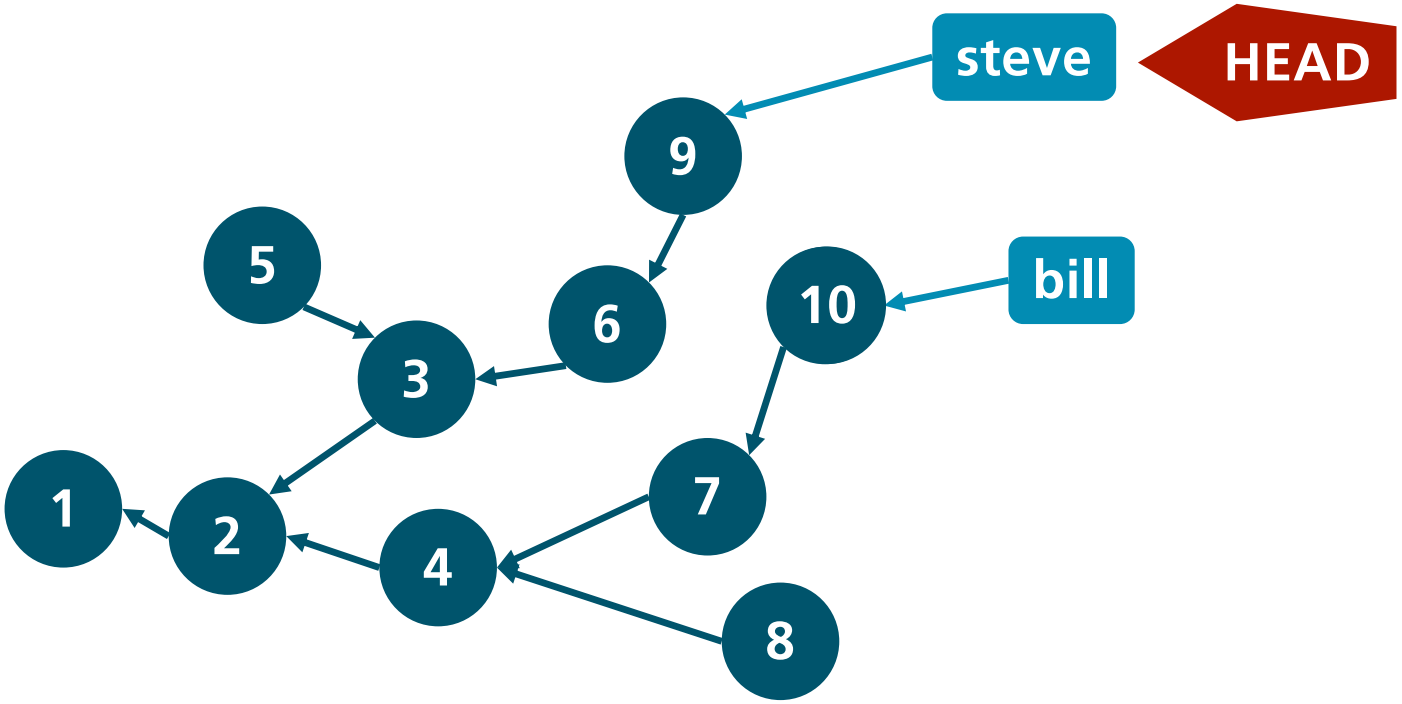


Everything is a Pointer



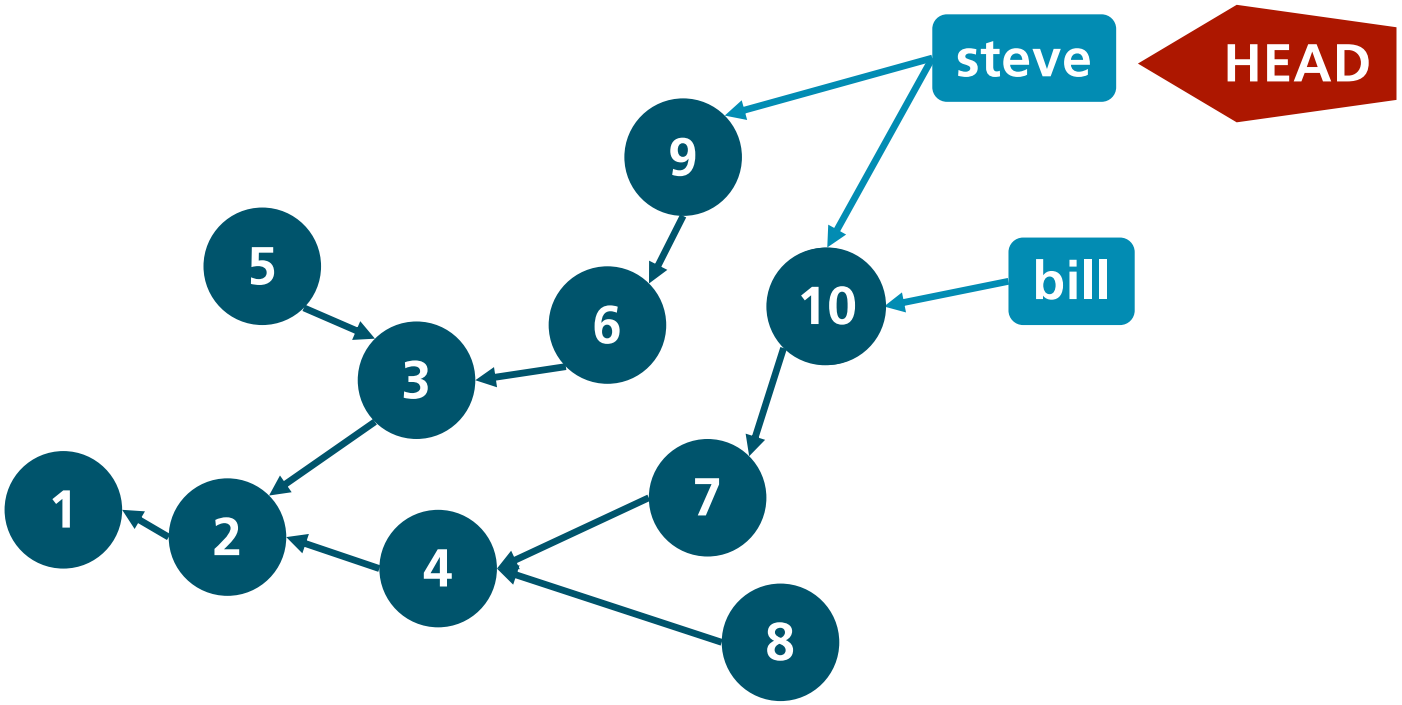
Checkout vs. Reset

git checkout bill

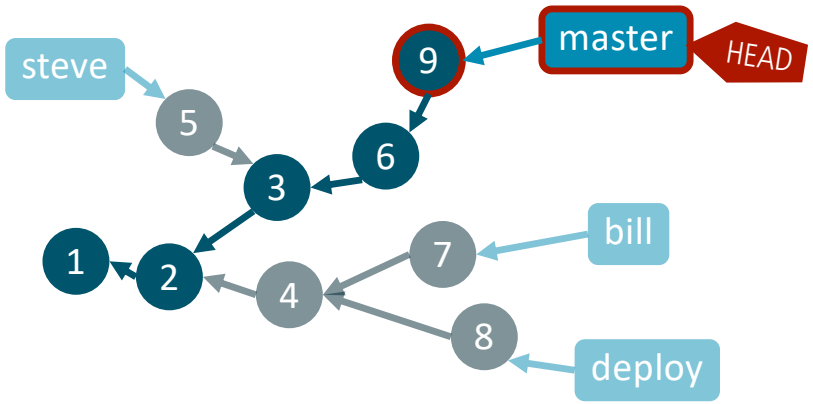


Checkout vs. **Reset**

```
git reset --hard 10
```

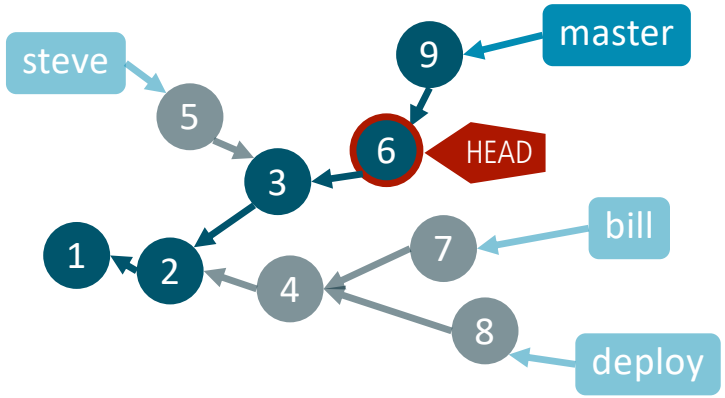


Detached HEAD



attached
HEAD!

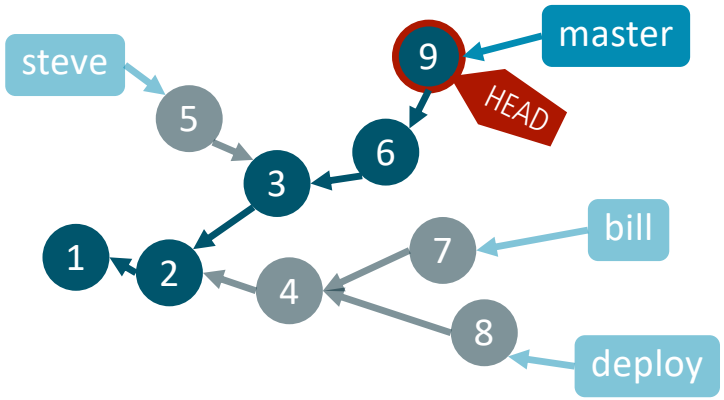
git checkout commit-6



detached
HEAD!

Re-Attach HEAD

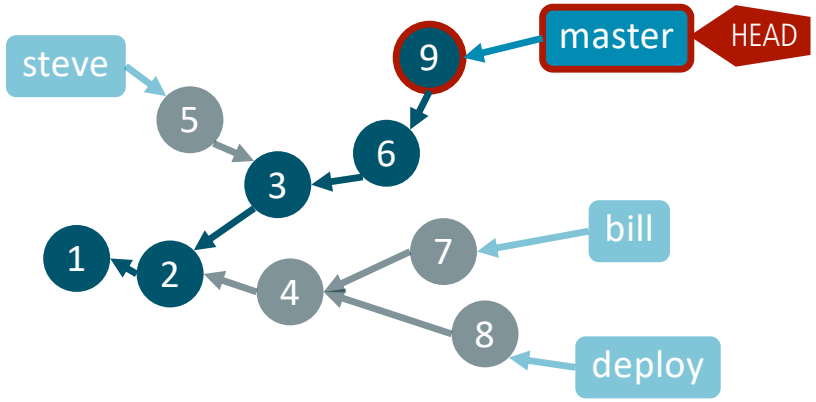
git reset master




```
* 7d8c316 - (HEAD, master) online (4 days ago)
```

detached
HEAD!

git checkout master

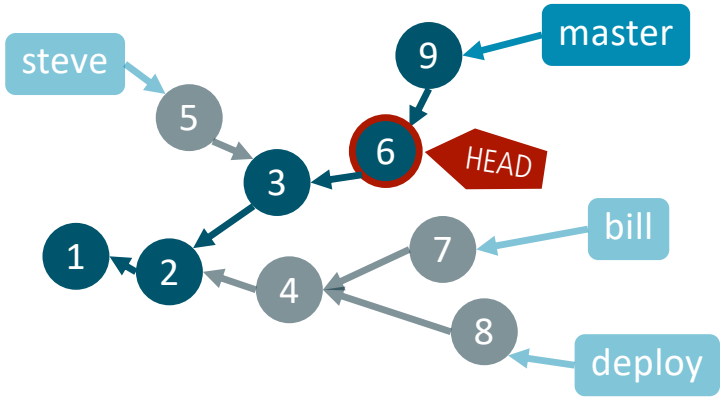



```
* 7d8c316 - (HEAD -> master) online (4 days ago)
```

attached
HEAD!

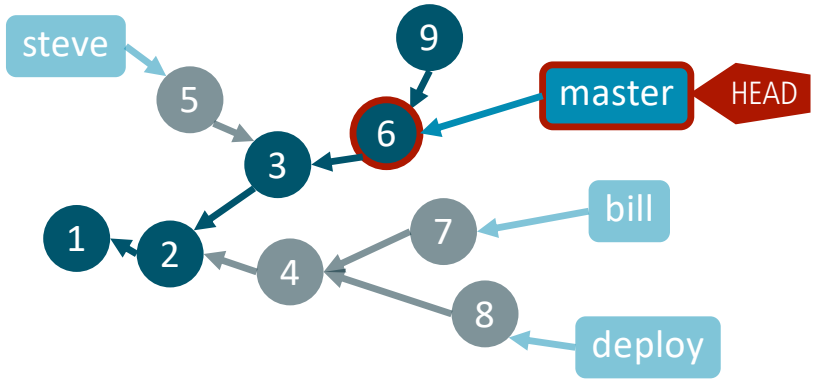
Rollback Branch

git checkout commit-6



detached
HEAD!

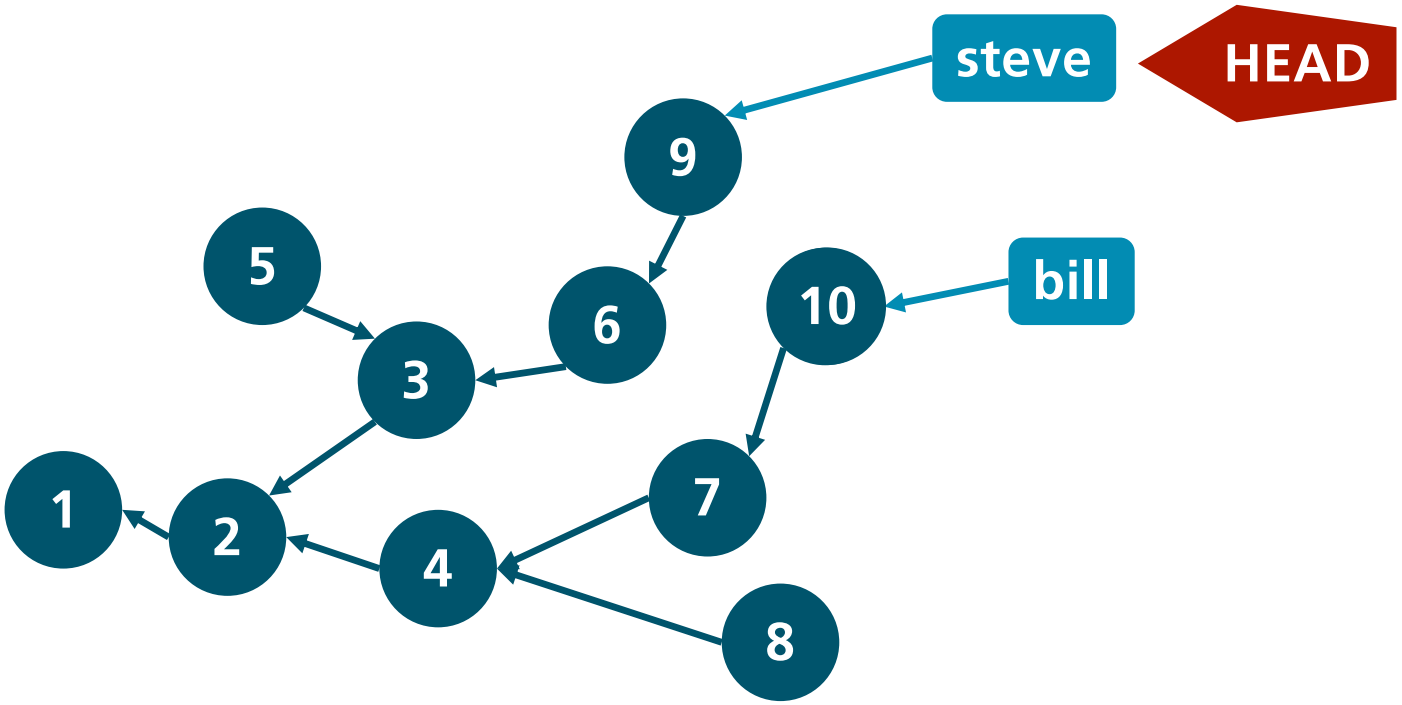
git reset --hard commit-6
git reset --hard HEAD~



attached
HEAD!

Checkout HEAD

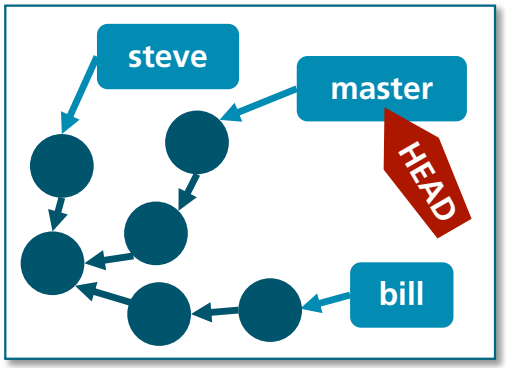
- git checkout HEAD file.txt



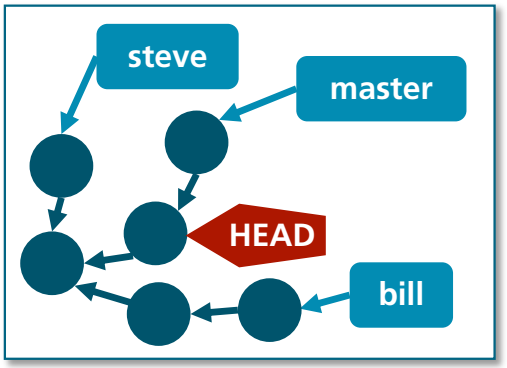
Übung 8

- **Detached HEAD**
  - Geh in den Branch „master“ und checke einen älteren Commit aus
    - *Lies die Meldung über das „detached“ HEAD durch*
    - *Welche der folgenden Grafiken gibt den jetzigen Zustand wieder?*

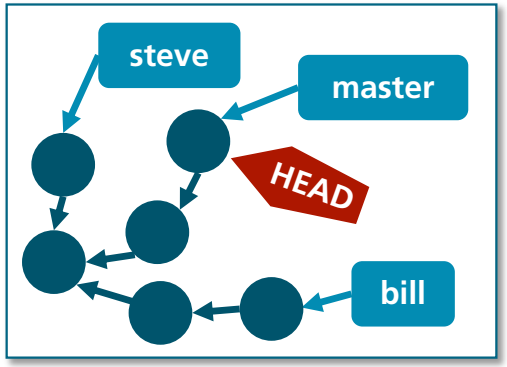
A



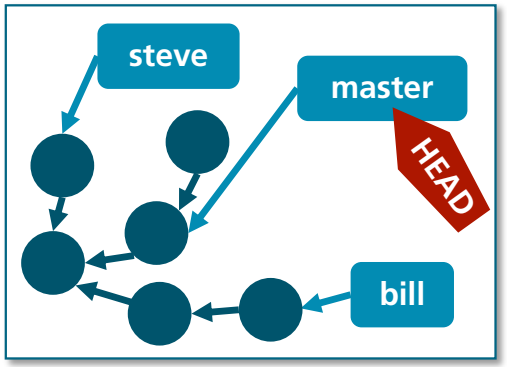
B



C



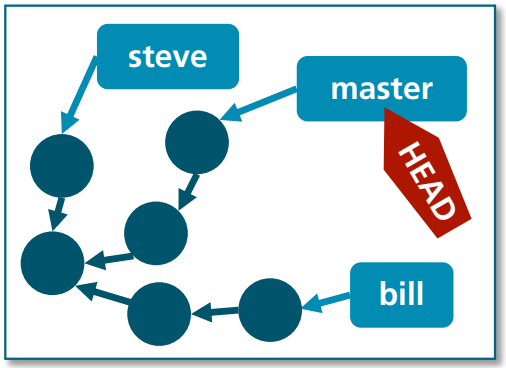
D



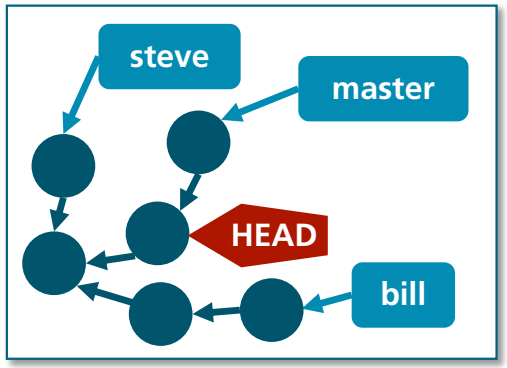
Übung 9

- **Detached HEAD (2)**
  - Checke nun den neuesten Commit wieder aus
    - *Welche Grafik gibt jetzt den Zustand wieder?*

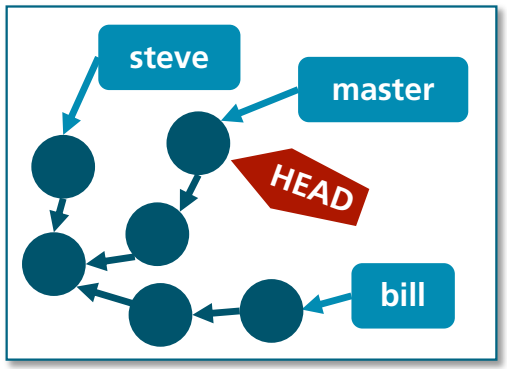
A



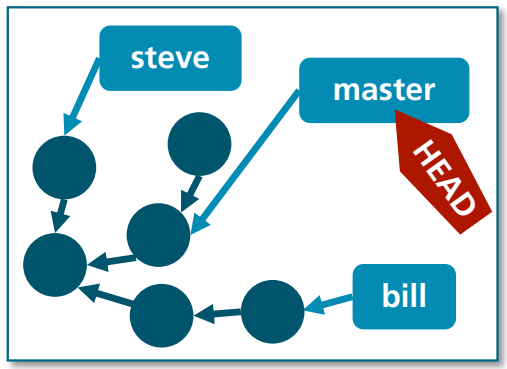
B



C



D

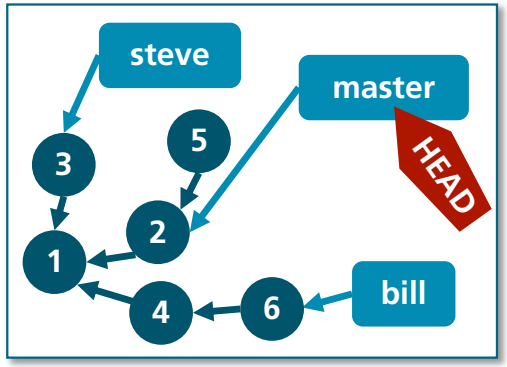


Übung 10

- **Attached HEAD**
  - Welche/n Befehl/e gibst Du ein, um das HEAD wieder zu „attachen“?
    a) *git attach HEAD*
    b) *git checkout master*
    c) *git reset master*
    d) *git reset <latest commit>*
    e) *git commit master*
  - Probier die Befehle einfach aus!
  - Prüf jeweils mit **git lg** und/oder **git status** ob Dein HEAD wieder „attached“ ist.

Übung 11

- **Branch auf ältere Version zurücksetzen**
  - Welchen Befehl musst Du eingeben, um den rechts abgebildeten Zustand zu erhalten?

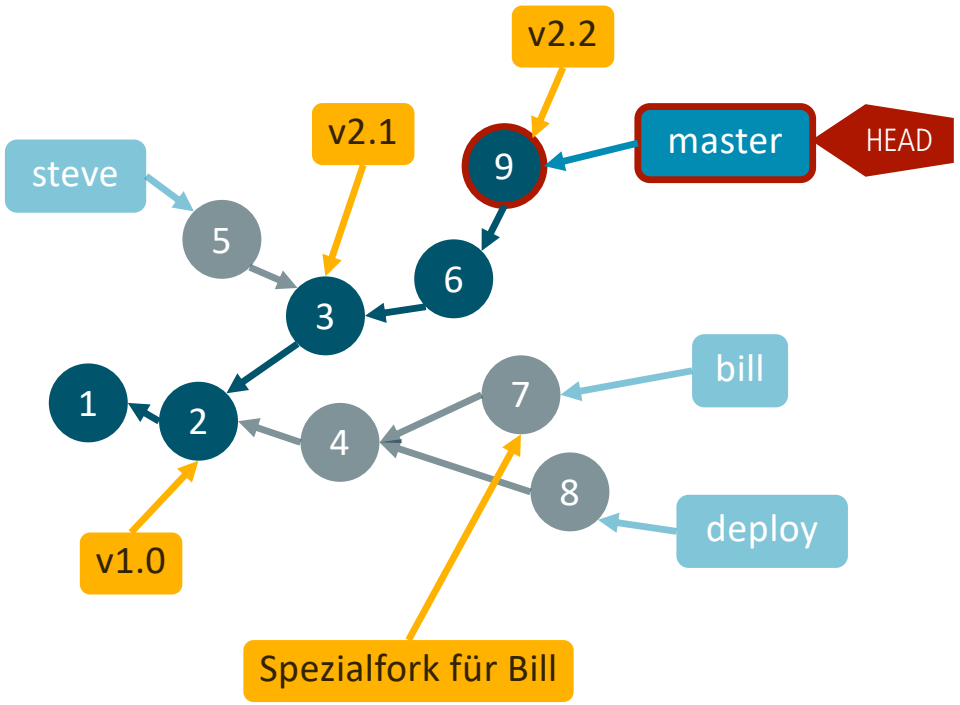


Tags

- Everything is a Pointer
- Annotated Tags

Everything is a Pointer (Tags)

- **Was sind Tags?**
  - Tags sind **Pointer** genau wie Branches
  - Unterschied:
    - *Sie zeigen immer auf den selben Commit, sind also sozusagen **schreibgeschützt***
    - *Sie haben **kein HEAD***
- **Wozu sind Tags gut?**
  - Um sich spezielle Commits zu **„merken“**
  - Insb. um den **Versionsverlauf** abzubilden



Annotated Tags

- **Lightweight Tag**
  - Nur ein simpler Name
  - wie ein Branch, der nicht geändert werden kann

- **Annotated Tag**
  - Objekt in der git Datenbank
  - Checksumme (SHA-1-Hash)
  - Autor
  - Datum
  - Beschreibung

Kommandozeilen-Befehle

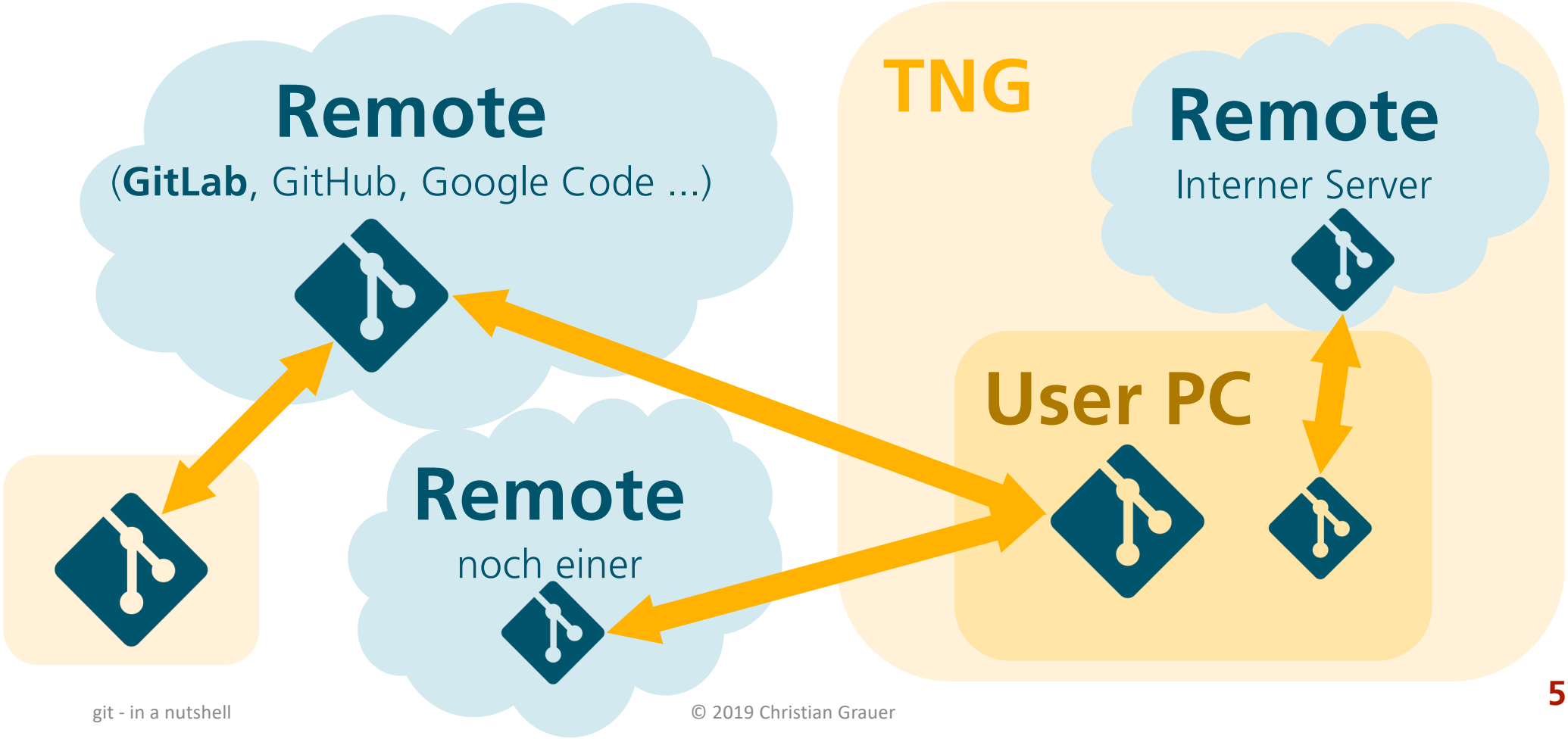
- **Tag erstellen**
  - Lightweight Tag: 
    ```
    git tag <name>
    ```
  - Annotated Tag:
    ```
    git tag -a <name> -m <msg>
    ```
  - Mit Editor:
    ```
    git tag -a <name> -m
    ```
  - Commit taggen:
    ```
    git tag -a <name> -m <msg> <commit>
    ```
- **Tags anzeigen**
  - Tag Liste:
    ```
    git tag
    ```
  - Einzelner Tag:
    ```
    git show <name>
    ```
- **Tags löschen**
  - Tag löschen:
    ```
    git tag -d <name>
    ```

Übung 12

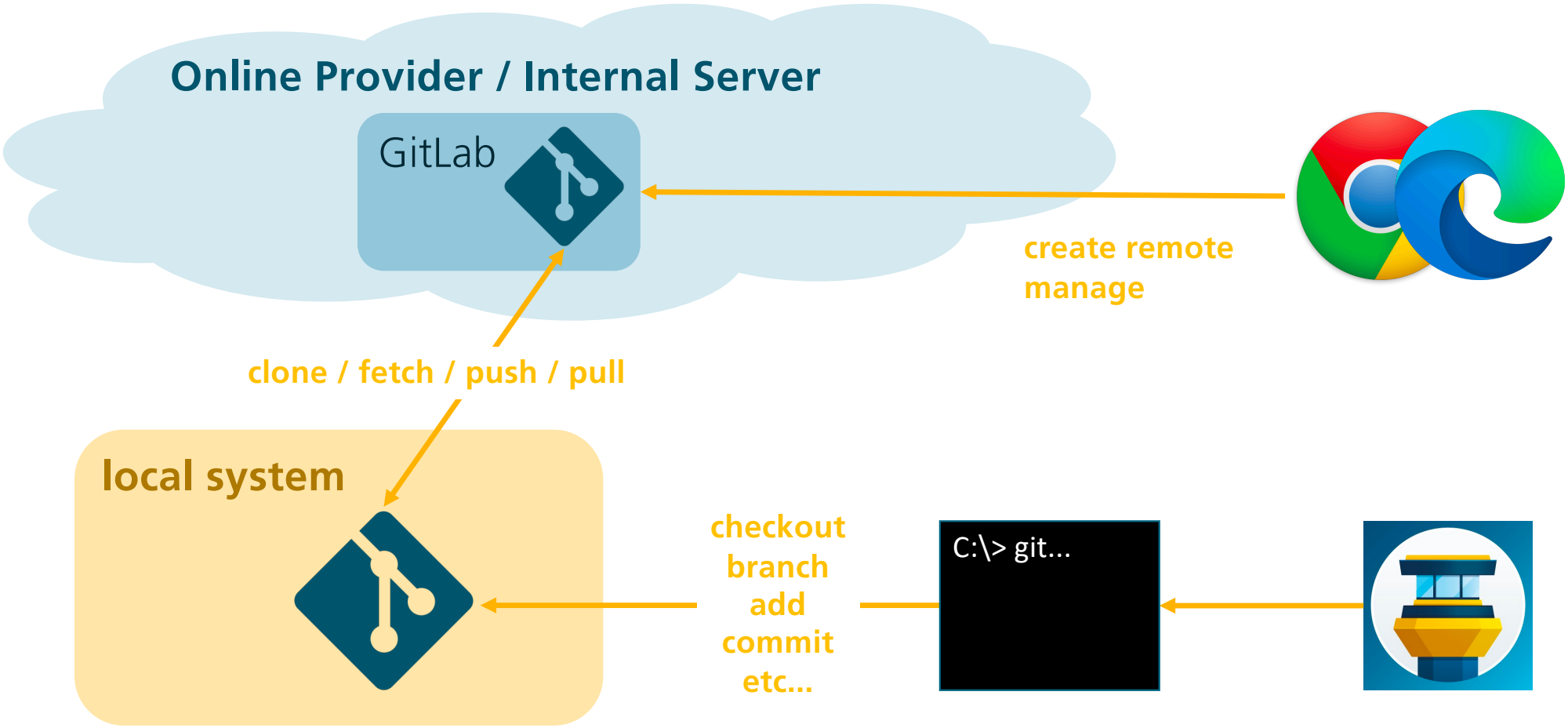
- **Tags setzen**
  - Check den „master“ aus und setze ein Tag mit dem Namen „v1.0“ und der Beschreibung „Unser erstes Release“
  - Setz einen Tag auf den vorletzen commit von “master“ mit dem Titel „beta“
  - Checke „steve“ aus und setze einen Tag „steve“
  - Checke „master“ aus und schau Dir das Log an. Was fällt Dir zu „beta“ und „steve“ auf?

Remote Repositories

Remote Repositories



Access Workflow



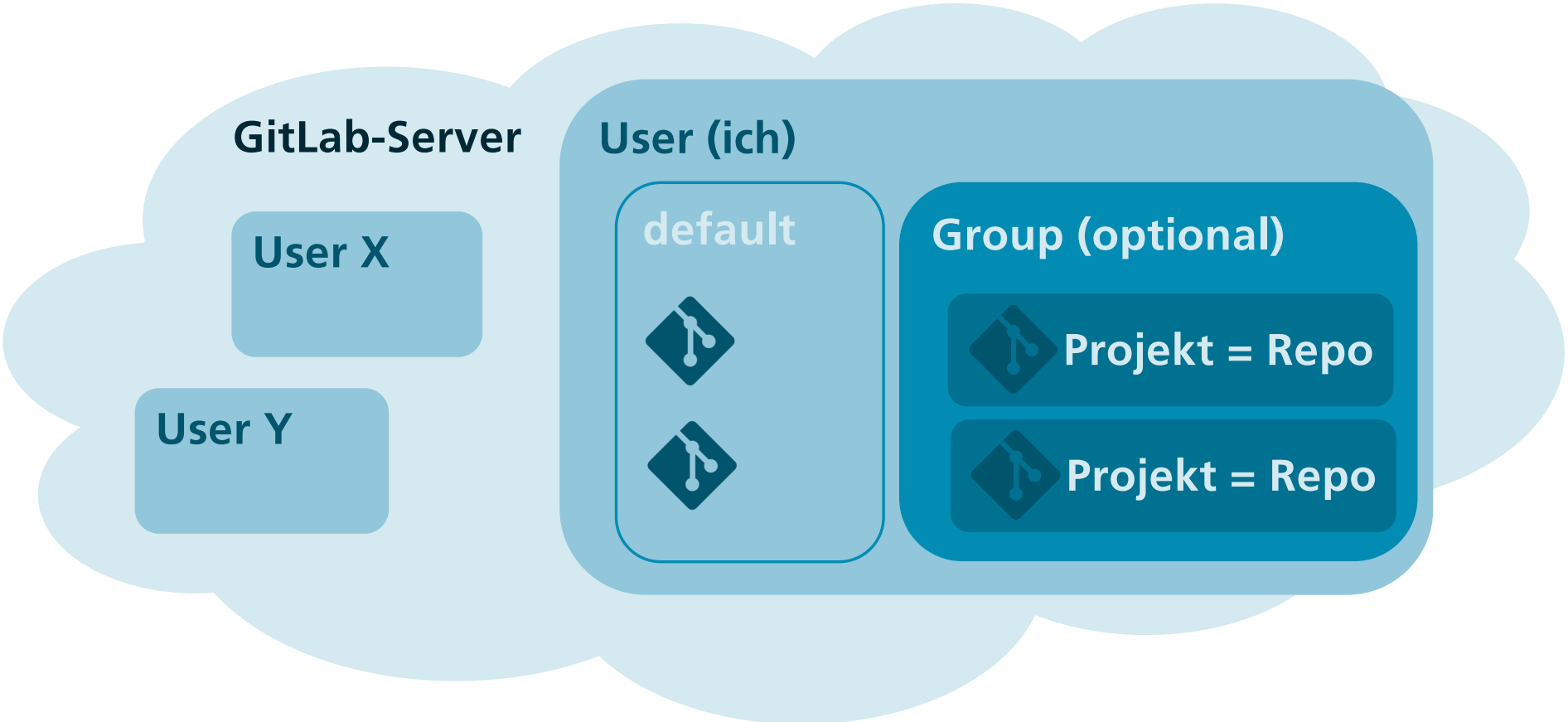
GitLab

- **Freie Software**
  für den Betrieb von Git-Servern
- **Online-Service**
  (Git-Server) der Fa. GitLab Inc

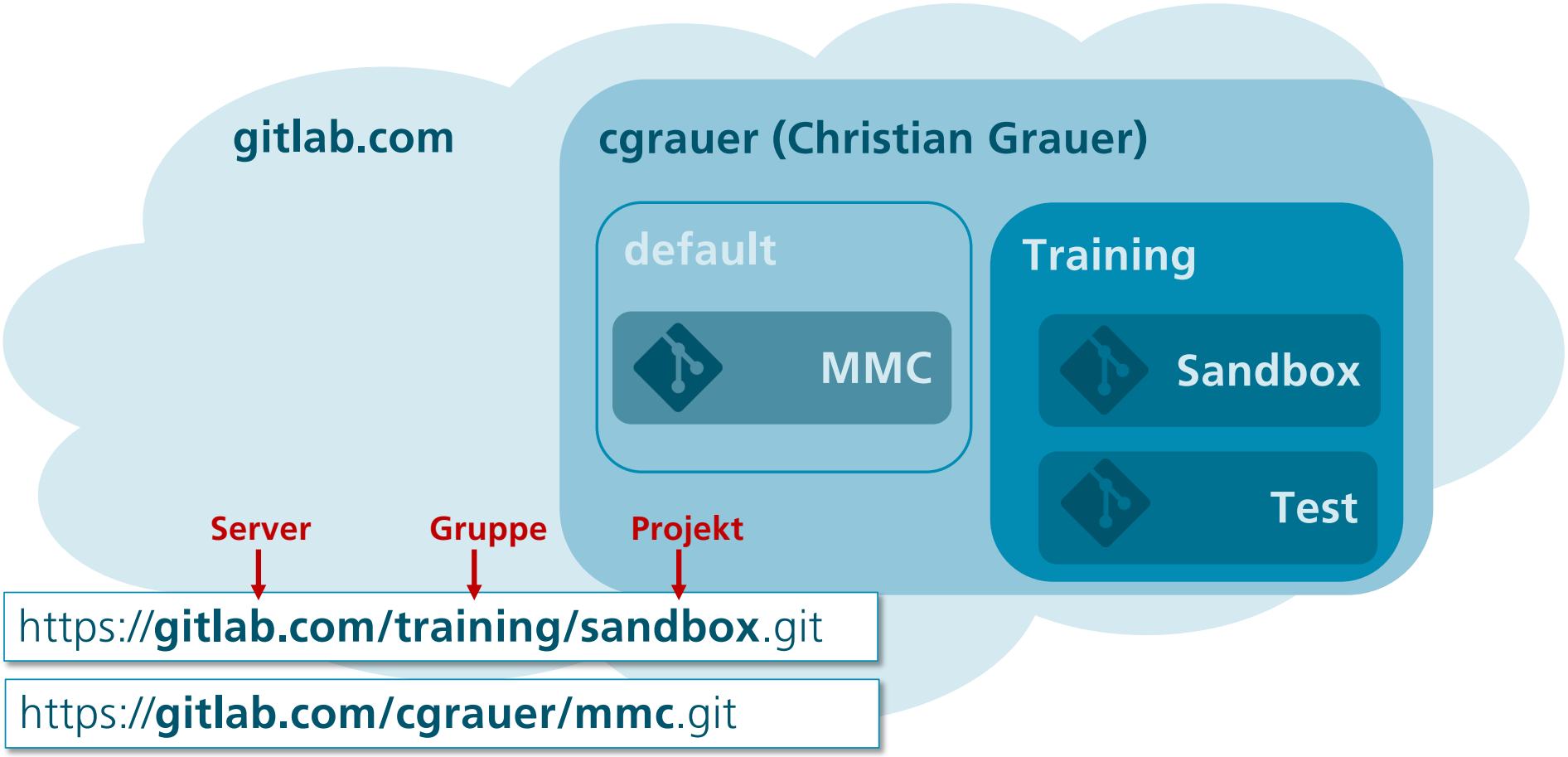
GitHub?

- **Und GitHub?**
  - Konkurrenzprodukt zu GitLab
    - Amazon AWS CodeCommit (Amazon)
    - BitBucket (Atlassian)
    - Codebase (Atlassian)
    - GitHub (Microsoft)
    - Google Code (Google)
    - Microsoft Azure DevOps (Microsoft)
    - SourceForge (Slashdot Media)
    - ...

GitLab Struktur

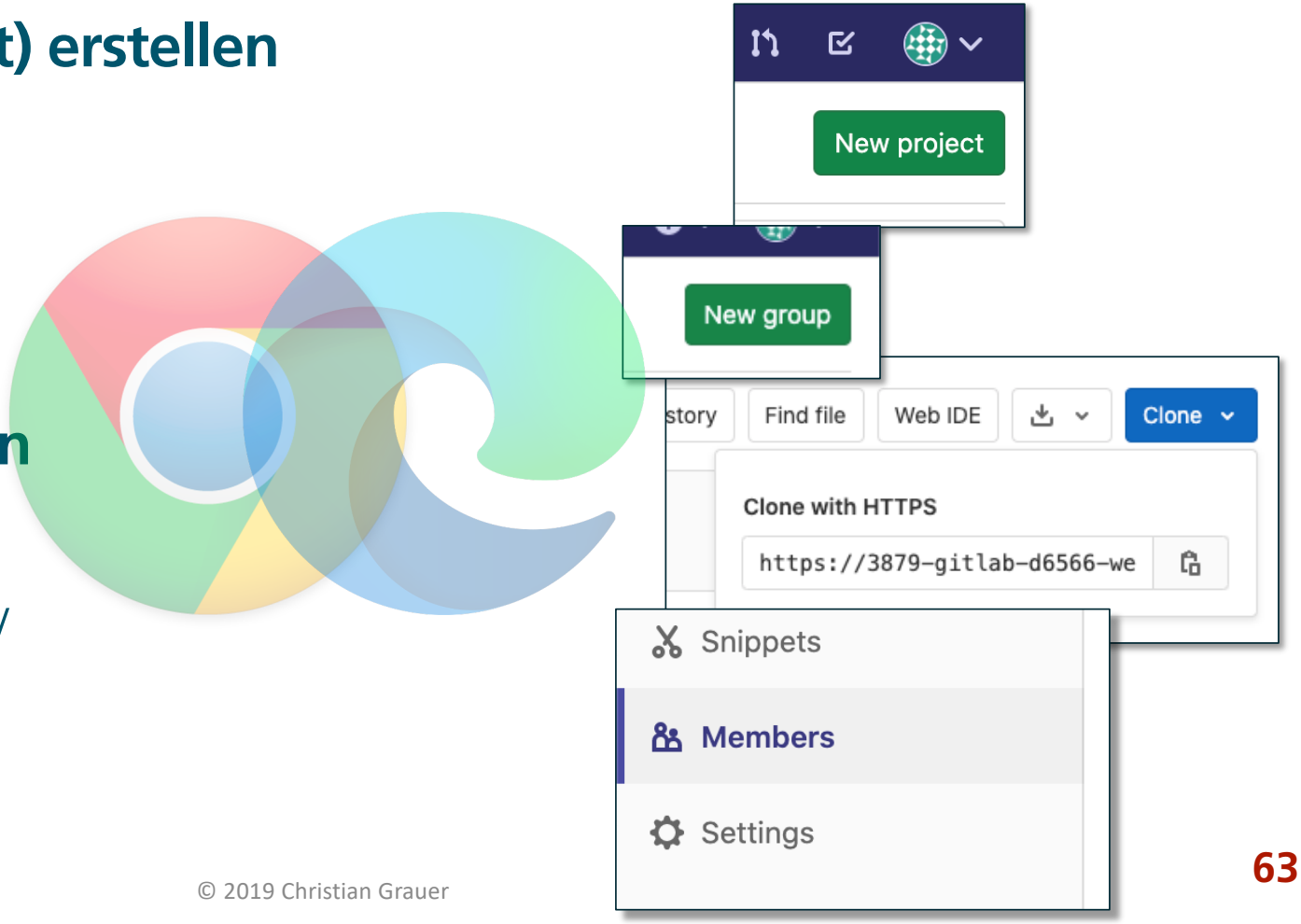


GitLab Struktur

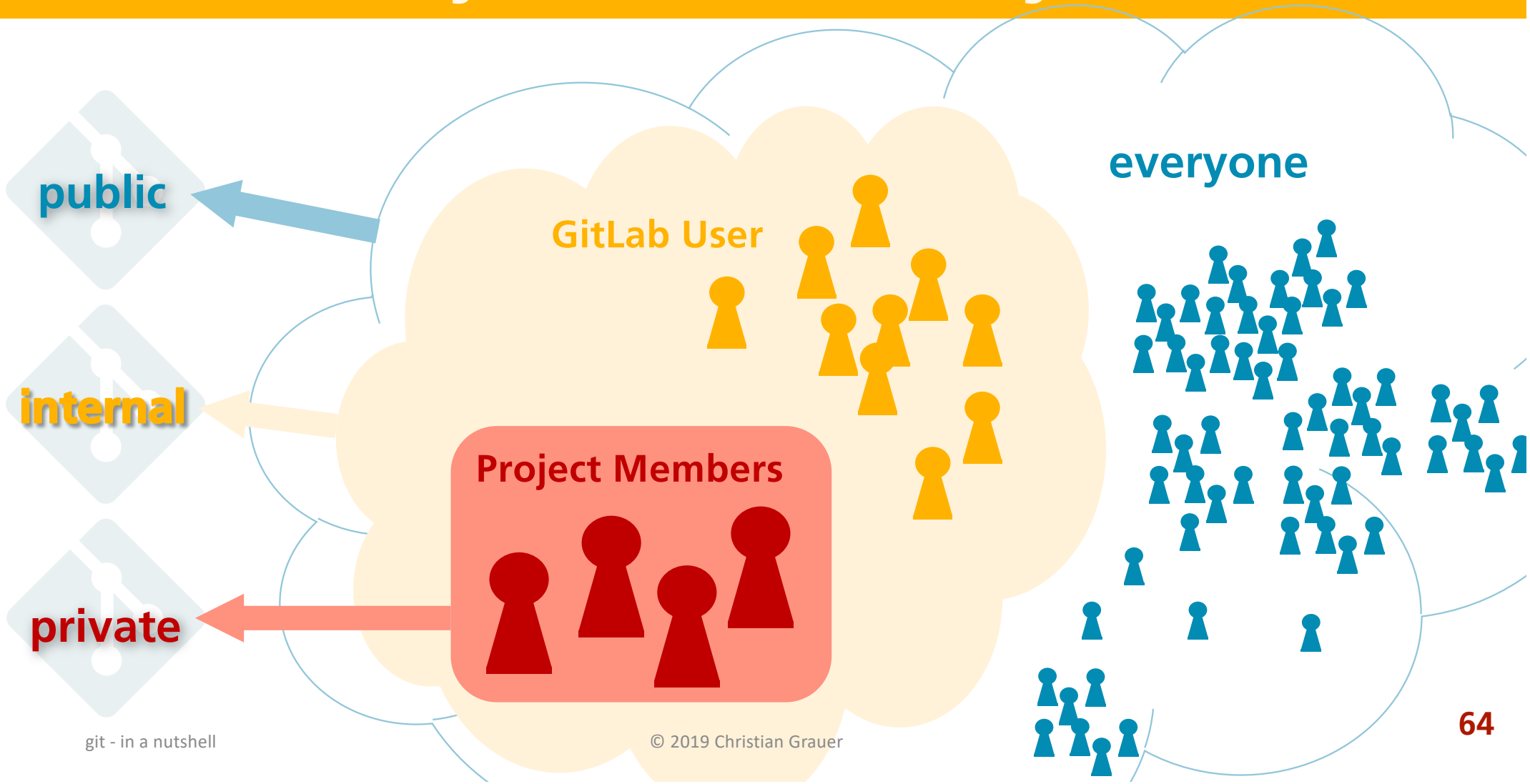


Wichtige Tasks im Web-Frontend

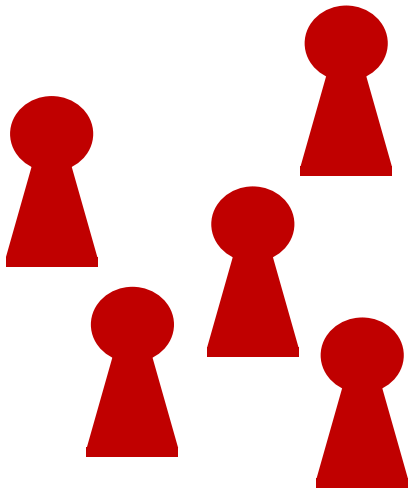
- **Repository (Projekt) erstellen**
  - Name
  - Slug
  - Description
- **Gruppen erstellen**
- **Clone-URL kopieren**
- **Zugriff einstellen**
  - Project-Accessability
  - Members
  - Roles



GitLab Project Accessability



GitLab Roles



Owner	Everything
Maintainer	Maintain Project, Manage Access
Developer	Add and Change Code
Reporter	Report Issues
Guest	View things
no Access	no Access

Kommandozeilen-Befehle

- **Neue Befehle**
  - Remotes anzeigen: `git remote`
  - Remote einbinden: `git remote add <name> <url>`
  - Remote klonen: `git clone <url>`

add remote ⇔ remote add

- **Beispiele**
  - `git remote add origin https://gitlab.com/test.git`
    - Verbindet das aktuelle Repo mit dem Remote-Repo „test“ unter dem Namen „origin“
  - `git clone https://gitlab.com/test.git`
    - Klont (1:1-Kopie) das Remote-Repo „test“ ins aktuelle Verzeichnis
    - Verbindet das Remote-Repo mit dem geklonten Repo unter dem Namen „origin“

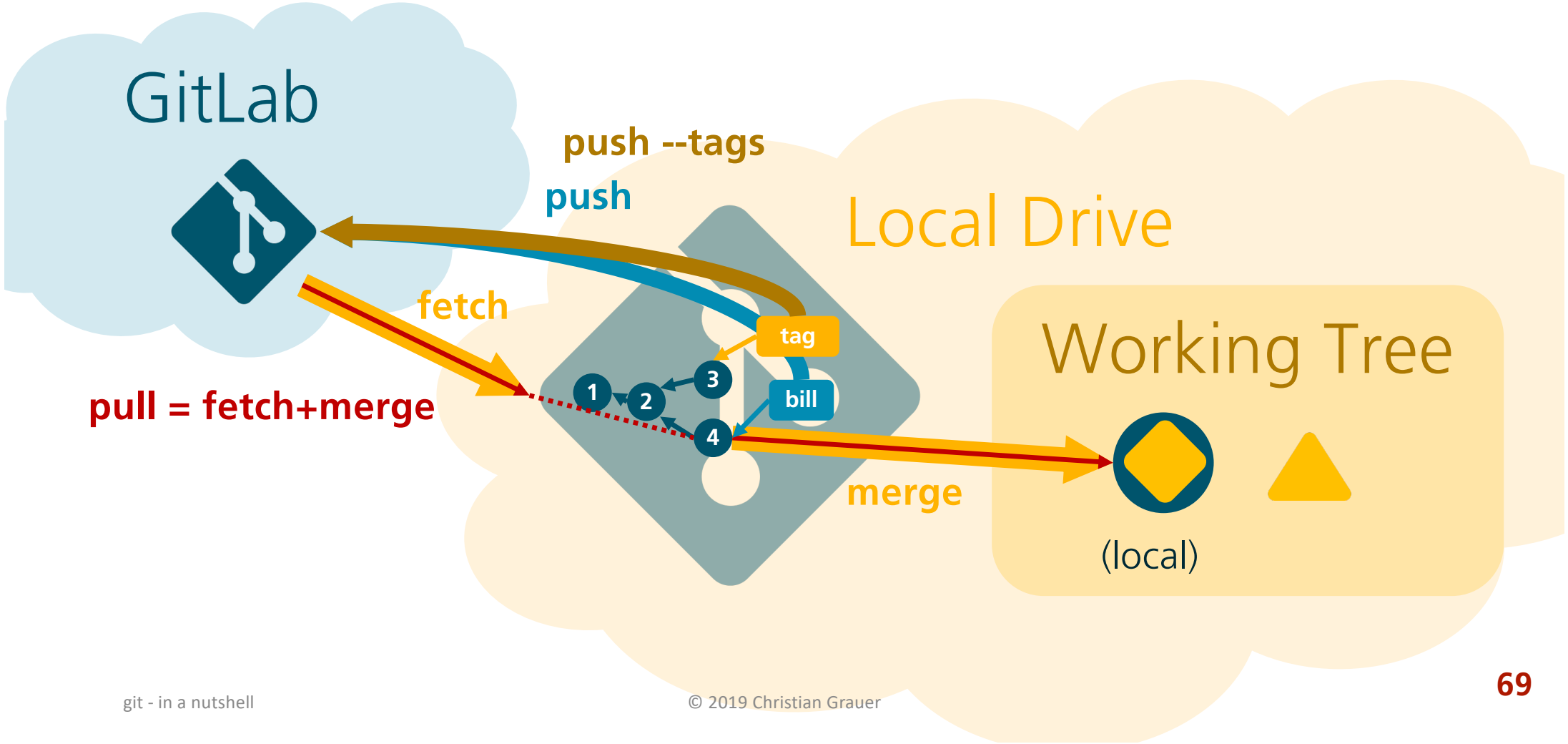
Übung 13

- **Remote Repository erstellen und verbinden**
  - Geh auf die GitLab-Platform und erstelle ein Neues Repository
    - *Gib ihm den Namen „Hello World“.*
    - *Achte darauf, dass der Project Slug „helloworld“ ist*
  - Füge das Remote als „origin“ deinem Repository hinzu
  - Liste die Remotes auf und prüfe, ob „origin“ angezeigt wird
- **Remote Repository klonen**
  - Geh auf die GitLab-Platform und suche das Repository tng_chgr/sandbox
  - Klone das Repository auf deinem lokalen Rechner

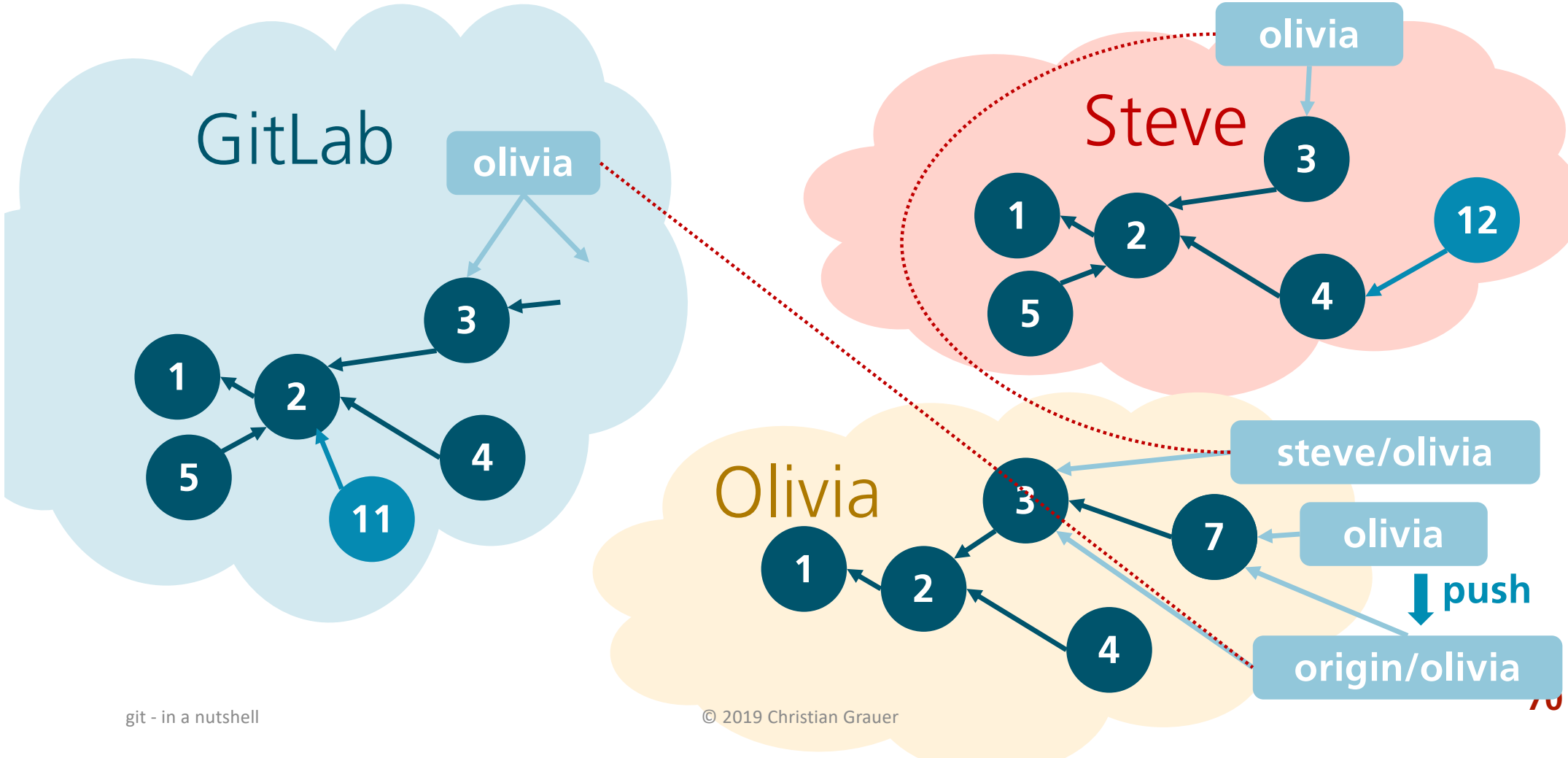
Synchronization

- fetch / pull / push
- Commit ist Commit!
- Push-Konflikte
- Everything is a Pointer (remote branches)
- Tracking

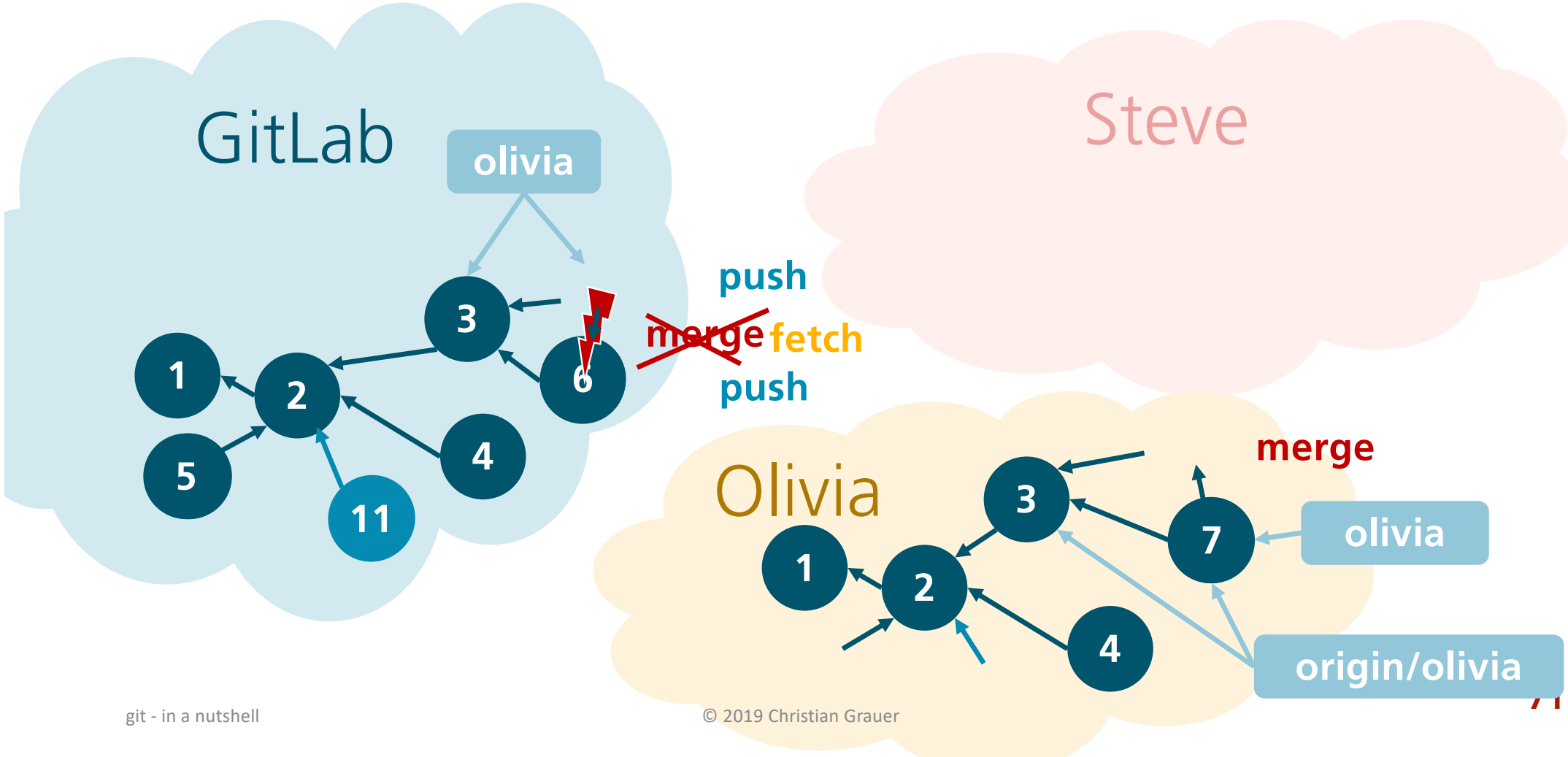
fetch / pull / push



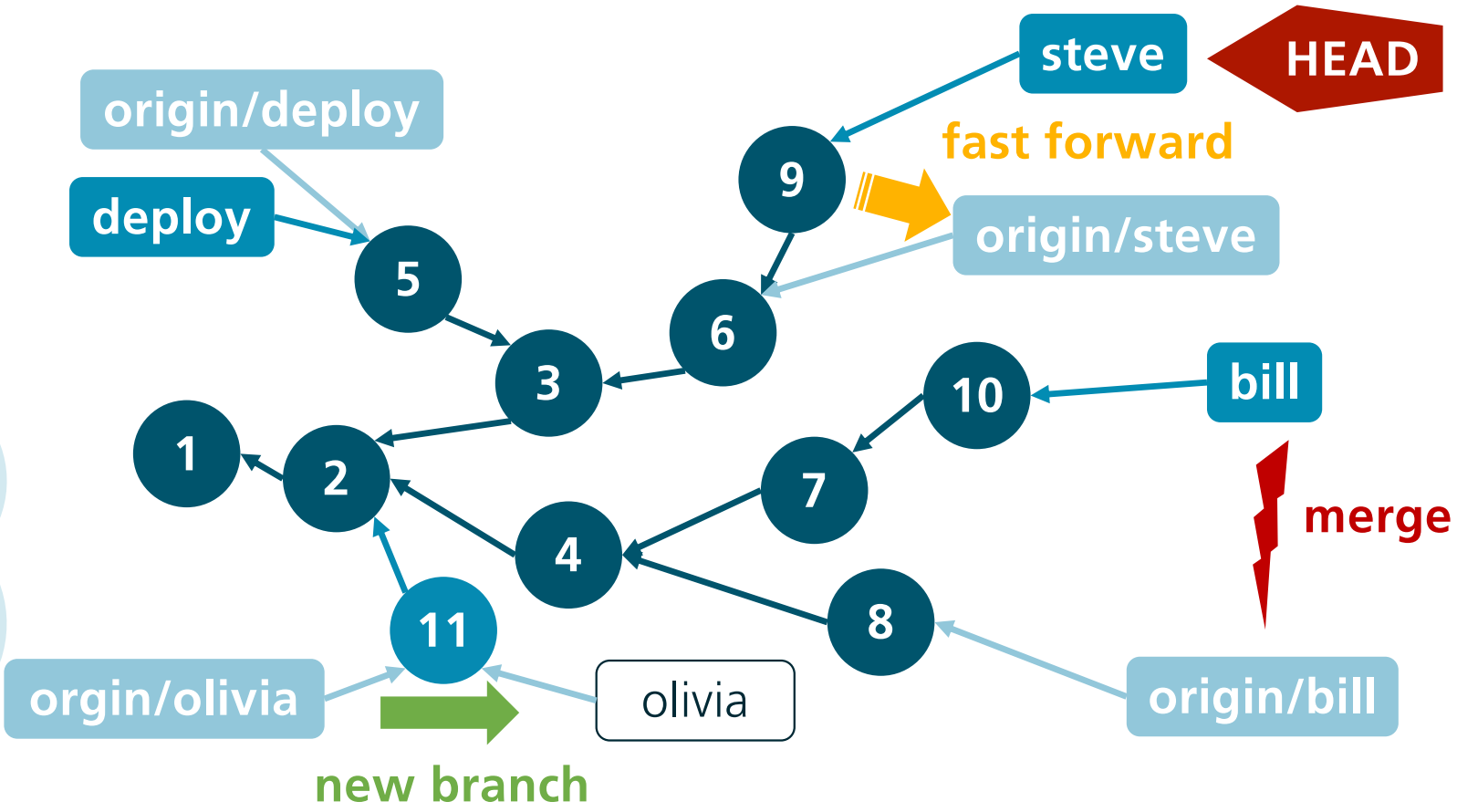
Commit ist Commit!



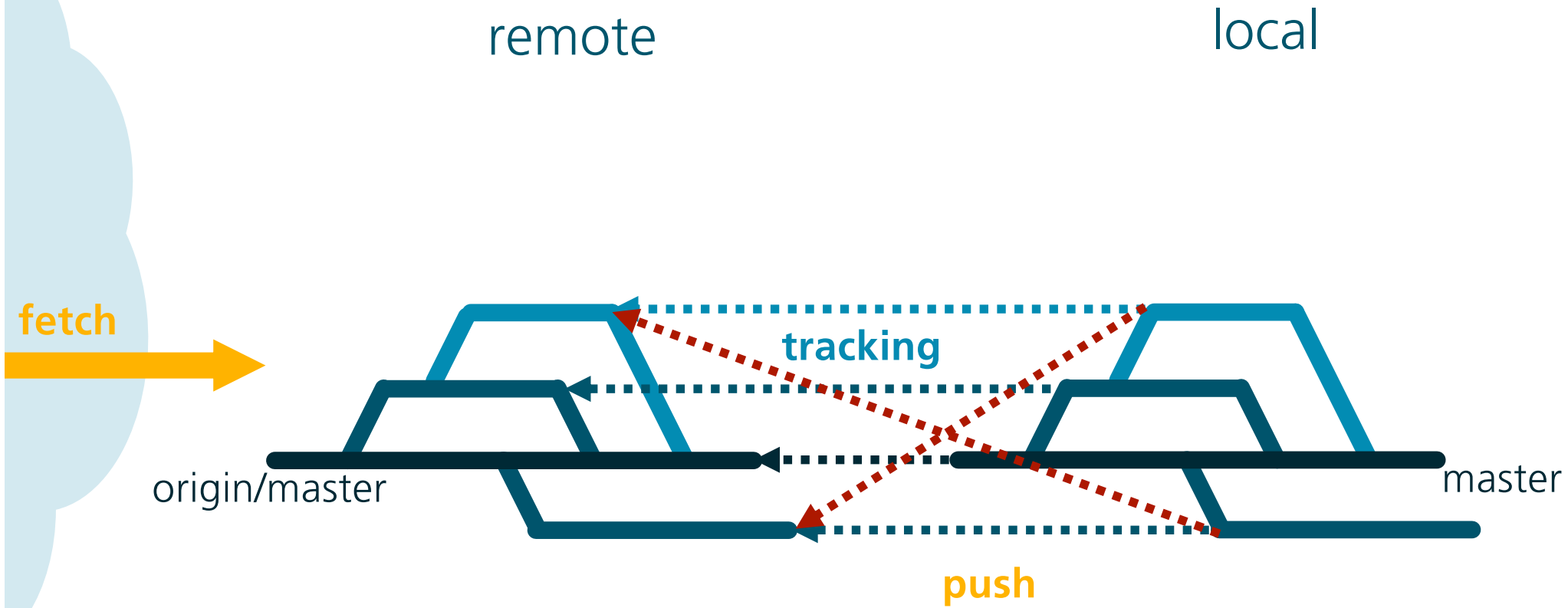
Push-Konflikte



Everything is a Pointer



Tracking



Neue Befehle

- **Fetch**
  - Remote herunterladen: `git fetch`
  - alle Remotes laden: `git fetch --all`
  - Ein Remote laden: `git fetch <remote>`
- **Pull (fetch und merge tracked branches)**
  - getracktes Remote: `git pull`
  - von beliebigem Remote: `git pull <remote>`
- **Push (upload branches)**
  - aktuellen Branch hochladen: `git push`
  - zu beliebigem Remote: `git push <remote>`

Neue Befehle

- **Tracking**
  - Tracking für aktuellen Branch setzen
    - remote branch existiert: `git branch -u <remote>/<branch>`
    - remote branch existiert nicht: `git push -u <remote> <branch>`
    - Beispiel: `git branch -u origin/steve`
  - Neuen lokalen Branch incl. Tracking erstellen:

```
git checkout -b <branch> <remote>/<branch>
git checkout --track <remote>/<branch>
git checkout <branch>
```

    - Beispiel:

```
git checkout steve
```

      - Erstellt einen lokalen Branch „steve“ und checkt ihn aus
      - Setzt das Tracking von „steve“ zu „origin/steve“, also dem remote-Branch
      - ACHTUNG: origin/steve muss existieren und es muss der einzige Remote-Branch mit dem Namen „steve“ sein! Sonst:

```
git checkout -b steve origin/steve
```

Übung 14

- **Sandbox**
  - Schau Dir die Branch-Struktur des geklonten Repositorys an
  - Erstelle einen neuen Branch mit dem Namen „develop“
  - Mach eine Änderung und erzeuge einen Commit
  - Synchronisiere den Branch mit dem Remote
- **Helloworld**
  - Verbinde alle lokalen Branches mit entsprechenden Remote Branches
  - Synchronisiere das Remote mit dem Lokalen Stand

What else?

- master / origin
- Remote-Namen (z.B. origin)
- Stash
- Konfiguration
- Repo Migration to GitLab

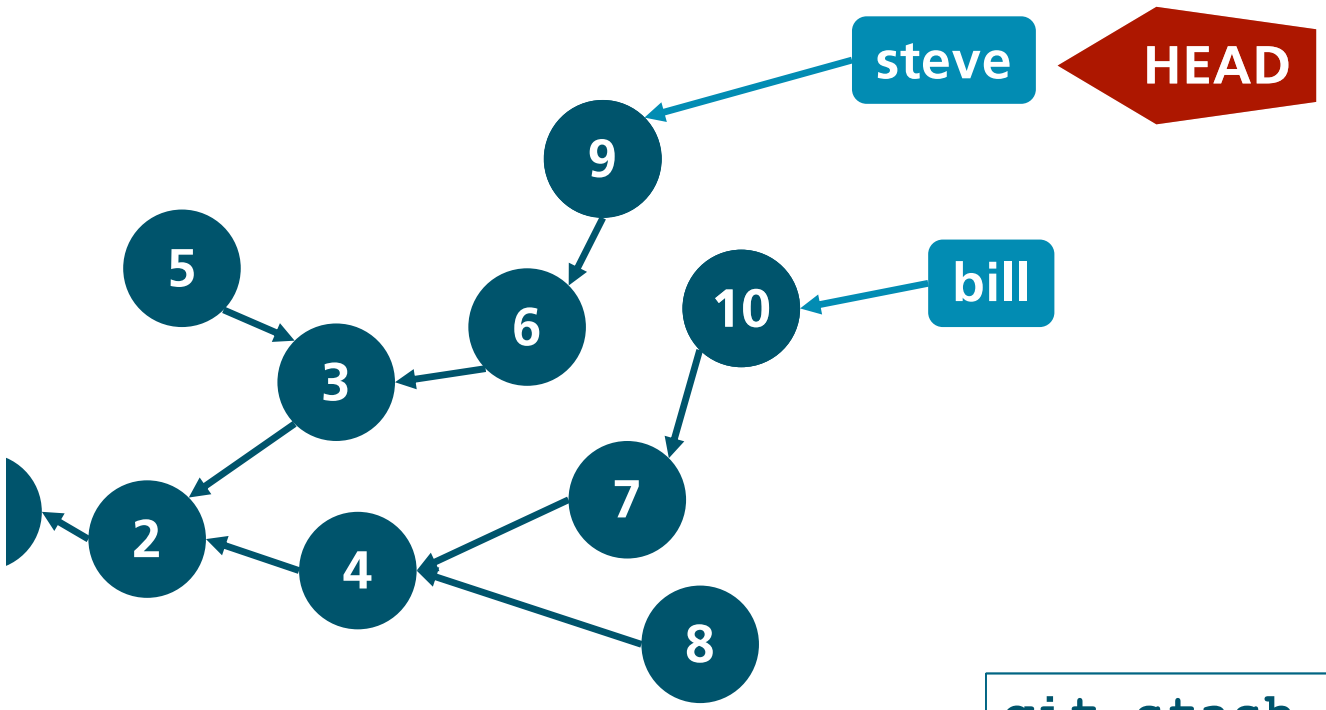
Master and Origin

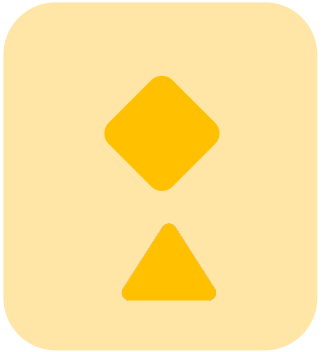
- **master**
  - kein Keyword!
  - beliebiger Branch-Name
  - Wird von git beim Erstellen automatisch als erster Branch erstellt
- **origin**
  - kein Keyword!
  - beliebiger lokaler Name für ein Remote (entferntes Repository, z.B. Gitlab)
  - Wird von git beim Klonen automatisch erstellt

Remote Namen (z.B. origin)

- **Remote Namen wie z.B. origin in „git fetch origin/master“ sind**
  - lokale Namen! D.h. sie gelten nur in „meinem“ Repository
  - Sie haben nichts mit dem Namen des GitLab-Repository zu tun!
  - Sie werden von mir (bzw. von git lokal) gesetzt
  - Andere User haben für den selben Server evtl. ganz andere Namen!
- **Aber:**
  - Beim Klonen verwendet git automatisch „master“ für den ersten Branch und „origin“ für den Remote, von dem geklont wird.

Stash





```
git stash pop
```

Konfiguration

```
git config -l
```

Lokale Konfiguration gilt für das aktuelle Repo

Datei: ../repo/.git/config

```
git config user.name "John Doe"
git config --local user.name "John Doe"
```

```
git config --local -l
```

Globale Konfiguration gilt für den aktuellen Systemuser

Datei: ~/.gitconfig

```
git config --global user.name "John Doe"
```

```
git config --global -l
```

System-Konfiguration gilt für den Rechner

Datei: <app>/etc/gitconfig

```
git config --system user.name "John Doe"
```

```
git config --system -l
```

Repo Migration to GitLab

Migration method	Options	Mögliche Quellen
Direkter Import		GitLab GitHub Bitbucket Cloud Bitbucket Server Google Code Fogbuz Gitea
Direkter Import meherere Repos auf einmal	Mit Manifest-Datei für die jew. Daten	
Klonen via URL	http:// https:// git://	Alle Platformen die `git clone` und das jew. Protokoll unterstützen

Allgemeine Voraussetzungen:
Username / Passwort für die Quelle

Tower

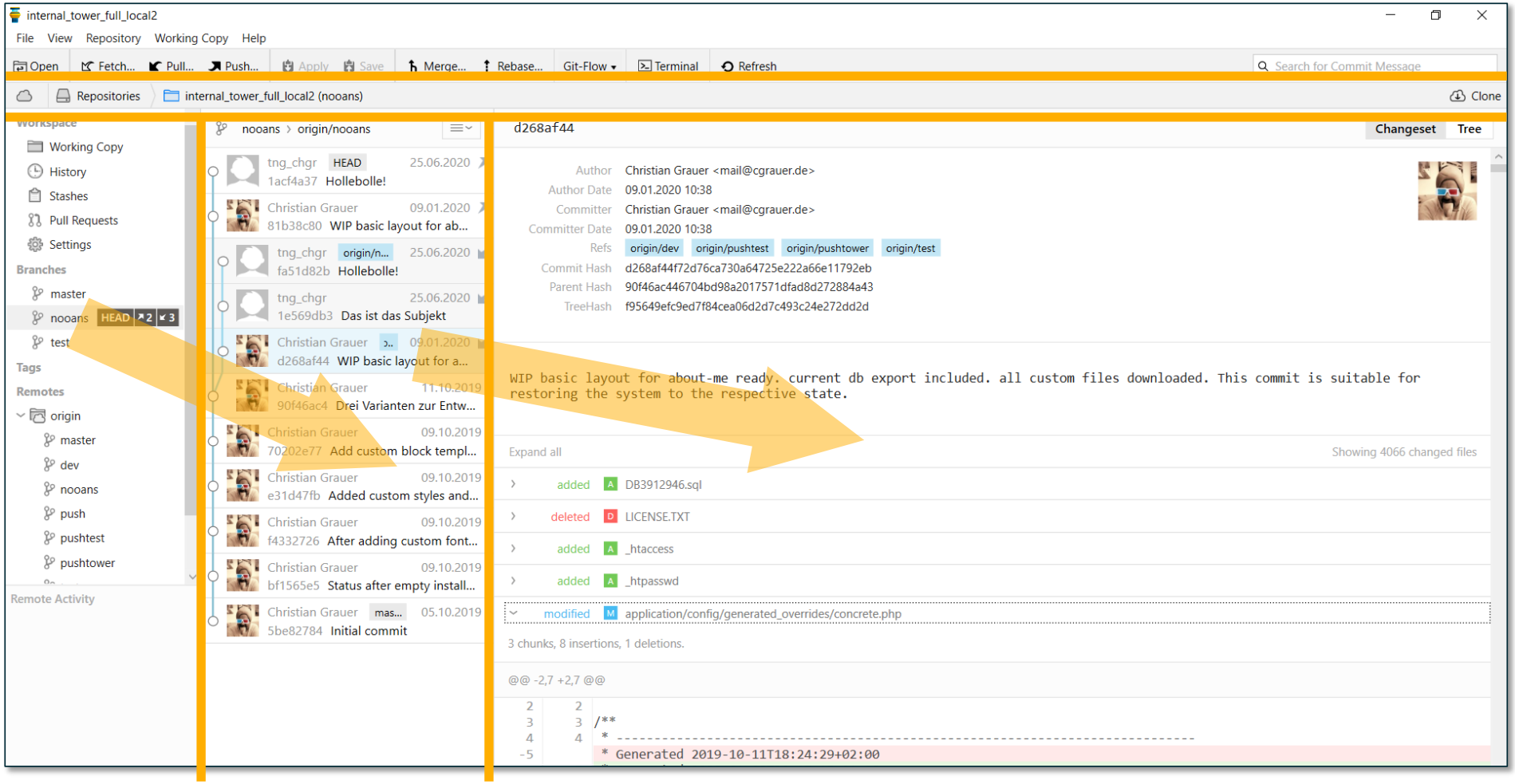
Installation

→ Vadym Elkind

GUI

- init, clone, config
- status, log, lg
- add, commit
- checkout, reset
- fetch, pull, push

Aufbau der Arbeitsoberfläche



Tower

→ Tower

Übung 15

- **Mach alle Übungen noch einmal von vorn, aber mit TOWER**
  - Benenne die Ordner „helloworld“ und „sandbox“ in „helloworld_cmd“ und „sandbox_cmd“ um
  - Lösch das Remote Repo „helloworld“
    - *Du findest den Befehl im Repo unter →Settings →General →Advanced →Delete Proejct*
  - Jetzt mach alle Übungen durch, aber nur mit Tower
- **Hinzufügen**
  - Füge die beiden Repos „helloworld_cmd“ und „sandbox_cmd“ in Tower hinzu

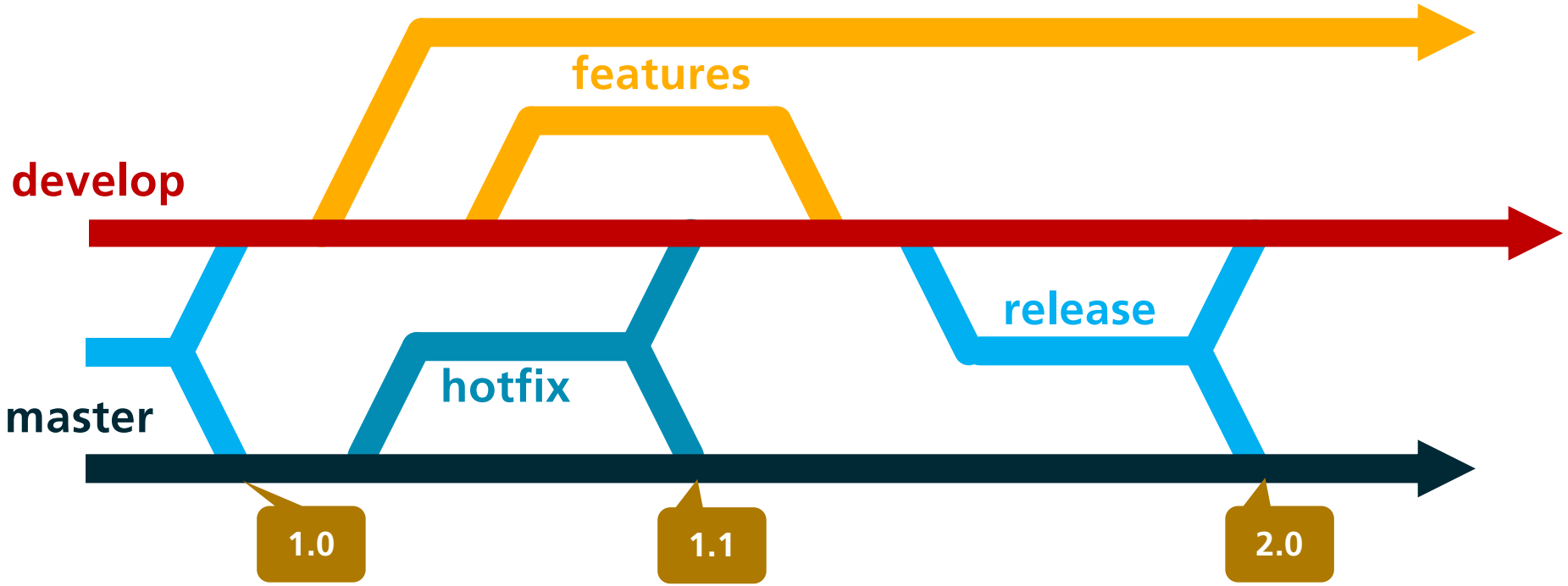
Übung 16

- **Konfiguration**
  - Öffne ein beliebiges Repo in Tower und geh in die Settings und ändere Deinen Username (Committer Identity) in Bugs Bunny
  - Wechlse zur Kommandozeile in das selbe Repo und schau Dir die Konfiguration an: `git config -l` und suche nach den Einträgen user.name=
  - Finde heraus, was Du in Tower machen musst, damit überall in git config Bugs Bunny als Name steht!
  - Schließe Tower und ändere den Namen zurück mit `git config --global user.name <Dein Name>`
  - Was ist in Tower passiert? Mach das gleiche auch für das Repository

Anhang

- Git Flow
- Reset

Git Flow



Reset

- **`reset --soft HEAD~`** ⇔ commit
  - Setzt den Branch, auf den HEAD zeigt, um einen Commit zurück
- **`reset HEAD~` (Abkürzung zu `reset --mixed HEAD~`)** ⇔ add
  - Setzt den Branch, auf den HEAD zeigt, um einen Commit zurück
  - Setzt den Index zurück auf den Stand vor dem letzten Commit
- **`reset --hard HEAD~`** ⇔ changes
  - Setzt den Branch, auf den HEAD zeigt, um einen Commit zurück
  - Setzt den Index zurück auf den Stand vor dem letzten Commit
  - Setzt den Working Tree zurück auf den Stand vor dem letzten Commit
- **`reset file.txt` (Abkürzung zu `reset --mixed HEAD file.txt`)**
  - Da nur eine Datei referenziert wird, bleibt der Branch unangetastet
  - Setzt die Datei file.txt im Index auf den letzten Stand zurück - Da sie dadurch keine Änderung mehr aufweist, heißt de facto, dass sie aus dem Index gelöscht wird.

Reset hard / mixed / soft