Hochschule Kempten
University of Applied Sciences

GIT - WORKSHOP

WiSe 2024/25

Jan Nothacker & Felix Gold

1. Termin - Grundlagen

- Was ist Git & warum Git?
- Verwendung des CLI
- Grundlegender Workflow
- Kooperatives Arbeiten
- Gitignore
- LFS
- Reset & Clean
- Stashing
- Git Flow
- Tags & Versionierung
- Forks & Pull Requests

2. Termin - Fortgeschrittenes

- (Interactive) Rebase
- Maintenance
- Branch Protection
- Grep
- Blame
- Cherry picking
- Worktrees
- Submodules
- Bisect
- Reflog
- CI/CD-Pipeline

Wer sind wir?

Jan Nothacker

- Studiengang GE
- 7. Semester
- Discord: @jan0h4ck
- linklist.jan-nothacker.de

Felix Gold

- Studiengang GE
- 5. Semester
- Discord: @matoowastaken

Was ist Git?

Git ist ein sog. **V**ersion **C**ontrol **S**ystem (VCS) bzw.
Source **C**ode **M**anager (SCM)

Das bietet uns folgende Features:

- Snapshots
- Backups
- Timelines
- Kollaboration

Warum Git?

- Weltweit **der** Standard unter den VCS
- Auch offline verwendbar
- Flexible Wahl des Workflows
- Open-Source und somit unabhängig

Verwendung des Git CLI

- CLI ist kurz für **Command Line Interface**. aka die Konsole
- Git wurde für Linux entwickelt, weshalb die Konsolen-Kommandos einem bestimmten Schema folgen:

```
command subcommand --options <params>
```

z.B.:

```
git push --force
git clone feature/fireball
git merge --no-ff feature/fireball
```

Per Konvention sind Optionen mit `--` ausgeschrieben, während Optionen mit `-` Abkürzungen aus einzelnen Buchstaben sind

So ist z.B. `git commit -a` äquivalent zu `git commit --all`

Ebenso gibt es Shortcuts, die wichtigsten darunter:

`.` ist das aktuelle Directory (Ordner)
`..` ist das parent Directory
`~` ist das home Directory
`*` ist eine Wildcard, d.h. „alles“

z.B. bedeutet `~/Downloads/*` „alle Dateien im Downloads Directory“

Navigation im CLI

Ein paar wichtige Commands, damit ihr euch im CLI zurecht findet:

- `cd <path>` **welchselt** das aktuelle **Directory** (`<path>` ist relativ!)
- `mkdir <name>` erstellt ein **neues Directory**
- `touch <file>` erstellt eine **neue Datei**
- `mv <path1> <path2>` **bewegt** eine **Datei/Directory**
  - Dadurch können diese auch umbenannt werden!
- `rm <file>` **löscht** eine **Datei/Directory** (direkt, kein Papierkorb!)
- `man <command>` öffnet die **Anleitung** eines Commands

Aufgabe

1. Öffnet das Git CLI (auf Windows „Git bash“)
2. Navigiert an eine Stelle, an der ihr euer Projekt aufsetzen wollt (z.B. Dokumente)
3. Erstellt ein neues Directory
4. Erstellt eine neue Datei (z.B. README.md)
5. Verschiebt die Datei in das neue Directory
6. Löscht die Datei

Nützliche Commands:

- `cd <path>`
- `mkdir <directoryname>`
- `touch <filename>`
- `mv <origin> <destination>`
- `rm <file/direcotry>`

Interner Aufbau

- Ein **Directory**/Ordner, welches von Git verwaltet wird, wird als **Repo**, kurz für **Repository**, bezeichnet
- Innerhalb eines solchen Repos gibt es 4 "Stufen":

Der **Worktree** sollte euch schon bekannt sein, das ist das selbe, wie wenn ihr Git nicht verwendet (IDE, Engine, Assets, etc.)

Von dort aus arbeiten wir uns jetzt Stückweise vorwärts

Initialisierung

Um aus einem bestehendem Directory ein Git-Directory zu machen, verwendet man:

```
git init --initial-branch=<name>
```

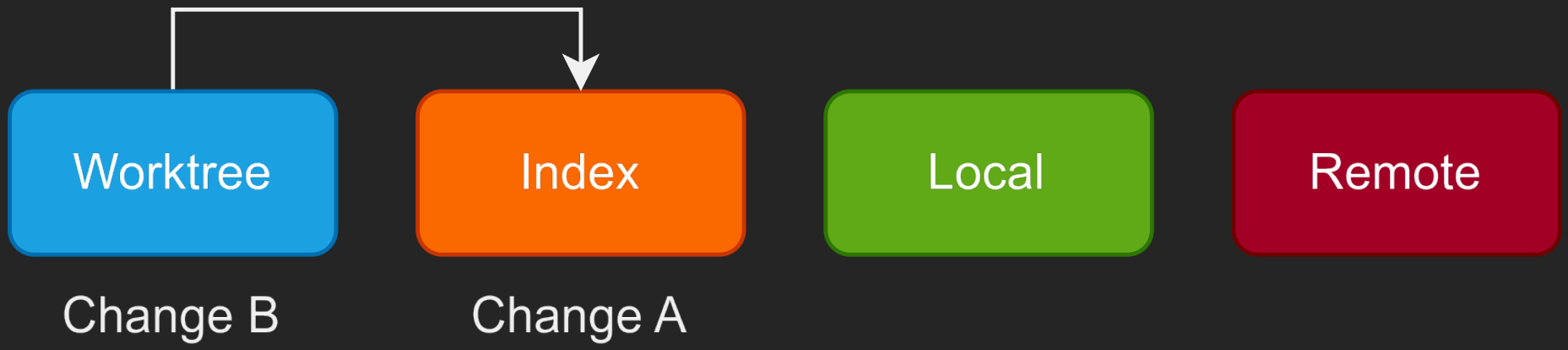
Wir können auch ein komplett neues directory anlegen, indem wir den Parameter `<directory>` anhängen

Dadurch wird ein `.git` Directory erstellt, welches von Git intern verwendet wird. Löscht man dieses Directory, wird Git hier nicht mehr verwendet

Den Status unseres aktuellen Git-Directories können wir mit `git status` abfragen

Staging

Wir haben nun ein neues Feature implementiert, und wollen davon einen Snapshot machen. Dazu müssen wir auswählen, was genau darin enthalten sein soll. Diesen Prozess nennt man **staging**



Die ausgewählten Änderungen werden dabei vom Worktree in den sog. **Index** verschoben.

Um eine Datei dem Index hinzuzufügen, verwendet man

```
git add <file>
```

Um eine Datei aus dem Worktree & Index zu Löschen, verwendet man

```
git rm <file>
```

Vorsicht: `git rm` ist nicht das selbe wie `rm`!

Löscht man z.B. eine bestehende Datei im Worktree mit `rm`, muss diese manuell aus dem Index entfernt werden.

`git rm` macht nur beides auf einmal

Mit der Option `--cached` können wir eine Datei nur aus dem Index entfernen, ohne den Worktree zu beeinflussen

Committing

Wir haben nun alle gewünschten Änderungen in den Index verschoben. Nun wollen wir einen Snapshot vom aktuellen Zustand erstellen. In Git wird dieser als ein **Commit** bezeichnet

Der Commit & die damit verbundenen Daten werden dabei lokal im .git Directory gespeichert

Ein Commit zeigt immer auf einen Vorgänger wie bei einer verketteten Liste, wodurch eine Art „Zeitstrahl“ entsteht

Intern ist ein Commit ein Container, bestehend aus den folgenden Komponenten:

- Commit message (z.B. „added Fireball to Enemy“)
- Author Information (Name, Email, etc.)
- Metadaten (z.B. Datum)
- Eindeutige Signatur (sha)
- Tree-Pointer (Interner Datei-Zeiger)
- Previous-Commit-Pointer (Vorgänger-Commit)

Die Metadaten, Signatur & Pointer werden von Git automatisch erstellt

Die Author Information muss nur zu Beginn einmal aufgesetzt werden

Meistens interessiert uns nur die Commit message / sha!

Um einen Commit zu erstellen, verwendet man

```
git commit
```

Dadurch wird ein Texteditor geöffnet, in welchem man die Commit Message eingeben kann.

```
git commit -m "<message>"
```

Ist hier ein nützlicher shortcut, um die Message direkt einzugeben

Häufig sieht man auch

```
git commit -am "<message>"
```

Hierbei werden automatisch alle Änderungen aus dem Worktree mit einbezogen

Remote Repos

Aktuell leben alle Commits noch in unserem .git Directory. Um ein Backup davon zu haben & mit anderen zusammen arbeiten zu können, möchten wir einen Server festlegen. Dieser wird in Git als ein **remote** bezeichnet

Ein Repo kann beliebig viele Remotes besitzen. Der primäre remote wird i.d.R. als **origin** bezeichnet

Mit dem folgenden Command können wir unserem Repo ein Remote hinzufügen:

```
git remote add <name> <url>
```

Um ein bereits bestehendes Repo vom remote herunterzuladen:

```
git clone <url>
```

Pushing / Pulling

Wir möchten nun unsere Commits auf den Remote hochladen.

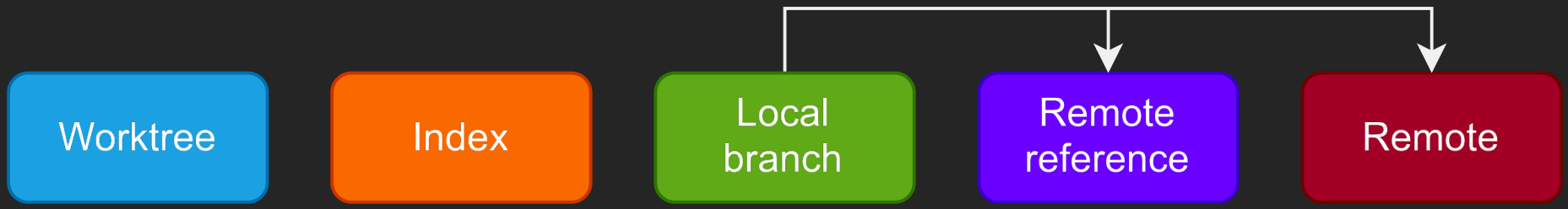
Dafür verwenden wir `git push`, wodurch alle neuen lokalen Commits auf den Remote hochgeladen werden.

`git pull` lädt alle für uns neuen Commits vom remote herunter

Pushing / Pulling

Wir möchten nun unsere Commits auf den Remote hochladen.

Dafür verwenden wir `git push`, wodurch alle neuen lokalen Commits auf den Remote hochgeladen werden.

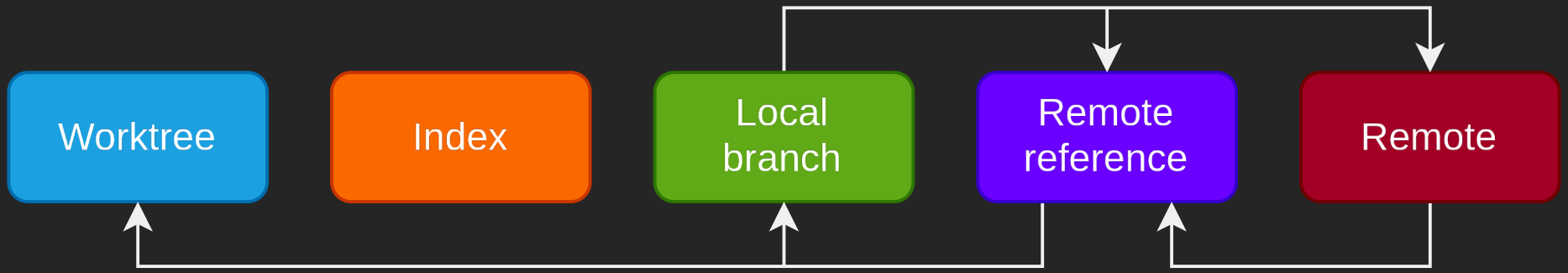


`git pull` lädt alle für uns neuen Commits vom remote herunter

Pushing / Pulling

Wir möchten nun unsere Commits auf den Remote hochladen.

Dafür verwenden wir `git push`, wodurch alle neuen lokalen Commits auf den Remote hochgeladen werden.



`git pull` lädt alle für uns neuen Commits vom remote herunter

Force pushing

Ist unsere lokale version outdatet, so hindert uns Git als Vorsichtsmaßnahme am pushen.

Um die remote-Version dennoch mit unserer Local-Version zu überschreiben, können wir die Option `--force` anhängen

Allgemein sollte man jedoch lieber `--force-with-lease` verwenden!

Aber Vorsicht: Bei Force-Pushes können Daten (quasi) unwiederbringlich verloren gehen!

Aufgabe

1. Erstellt ein neues Git Repo namens „workshop-example“
2. Erstellt „README.md“ und schreibt etwas hinein
3. Committet & Pusht die Datei
4. Erstellt ein neues Repo auf GitHub
5. Fügt das GitHub Repo als Remote hinzu
6. Pusht eure Änderungen

Nützliche Commands:

- `git init`
- `git remote add <name> <url>`
- `touch <filename>`
- `git add <file1> <file2> ...`
- `git commit -m "<message>"`
- `git push`

Branches

In unserem aktuellen Aufbau besteht noch ein Problem: Was passiert, wenn mehrere Leute gleichzeitig am Projekt arbeiten?

Um uns nicht gegenseitig in die Quere zu kommen, können wir gleichzeitig auf verschiedenen parallelen „Zeitstrählen“ arbeiten. In Git werden diese als **Branches** bezeichnet

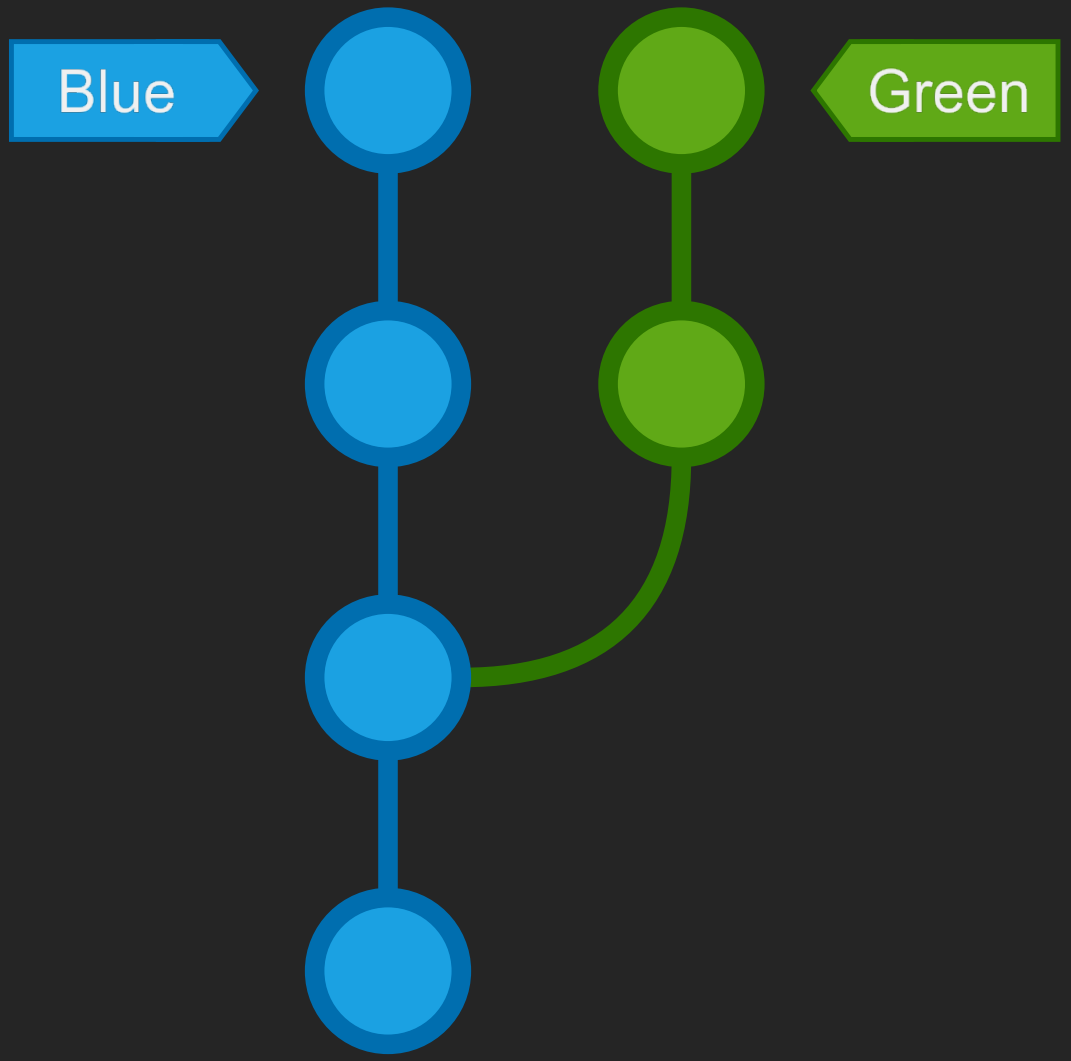
`git branch <name>` erstellt einen neuen Branch, der sich vom aktuellen Commit abspaltet

Mit `git branch -a` können alle Branches angezeigt werden

Mit `git branch -d` wird ein Branch gelöscht

Der sog. **HEAD** bezeichnet immer die Position, an der man sich aktuell befindet, und ist unabhängig von Branches!

In diesem Beispiel gibt es 2 Branches, Blue & Green. In der Vergangenheit hat sich Green von Blue abgespaltet



Logging

Mit `git log` können wir die Commit History von unserem aktuellen Commit einsehen

Dies ist standardmäßig eine Text-Ansicht

Mit der Option `--graph` können wir im CLI einen simplen Graphen anzeigen lassen

Checkout

Um zu einem anderen Branch oder einem alten Commit zu kommen, verwendet man einen **Checkout**

```
git checkout <branchname>
```

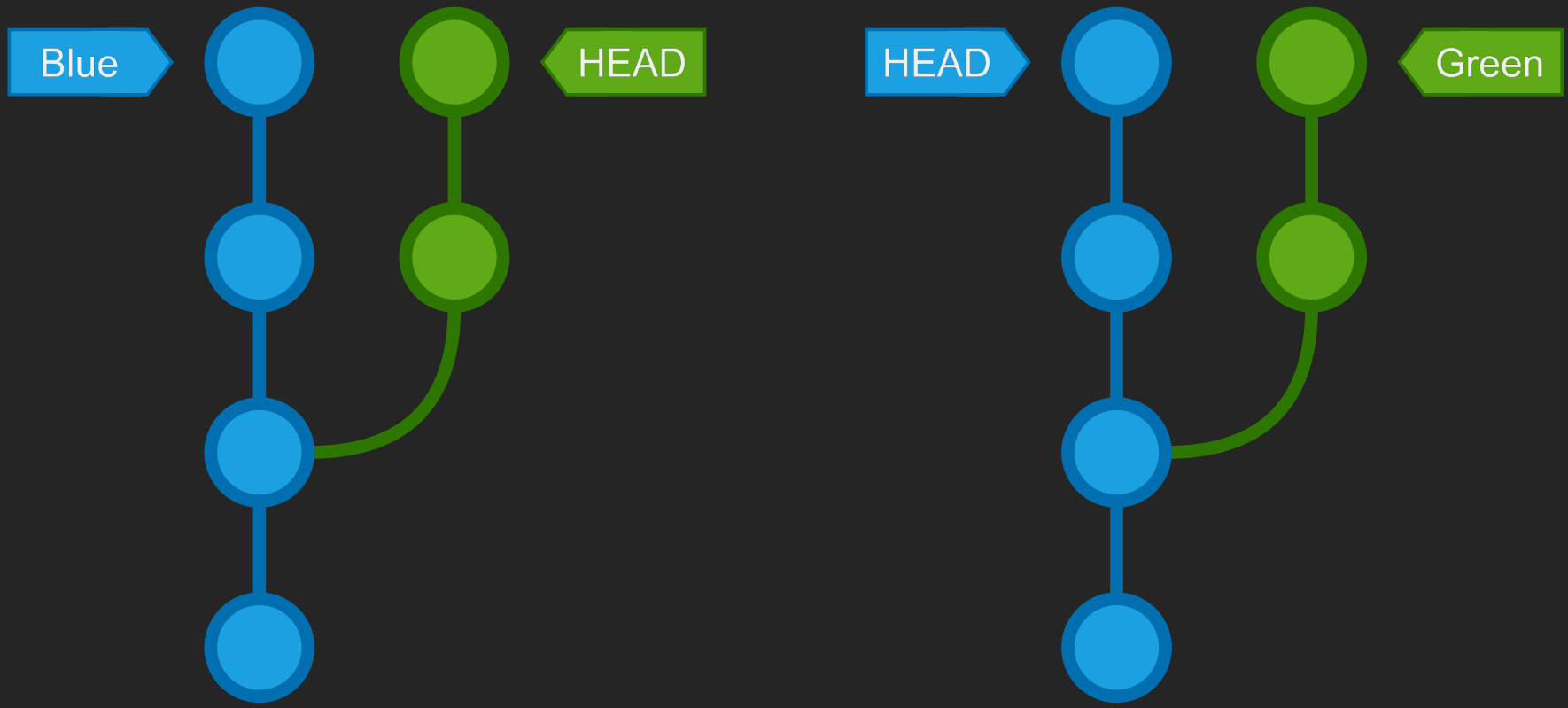
wechselt den aktuellen Branch

```
git checkout <sha>
```

springt direkt zum angegebenen Commit

In diesem Beispiel befinden wir uns zu Beginn auf Green, was durch den HEAD symbolisiert wird.

Durch `git checkout Blue` welchseln wir zu Blue, wodurch sich auch unser HEAD ändert



Merging

Wir haben nun auf mehreren verschiedenen Branches unsere jeweiligen Features implementiert, und wollten diese nun zusammenfügen. Dies geschieht mithilfe eines **Merges**

```
git merge <branchname>
```

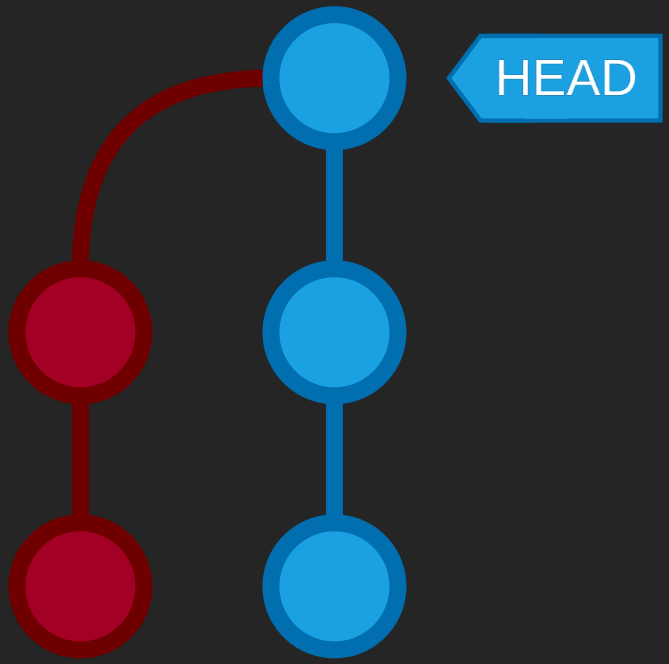
merged dabei einen **anderen** Branch in unseren **aktuellen** Branch

Nach dem Mergen kann der andere Branch gelöscht werden, ohne Daten zu verlieren, die Referenz wird duch den neuen aktuellen Branch behalten

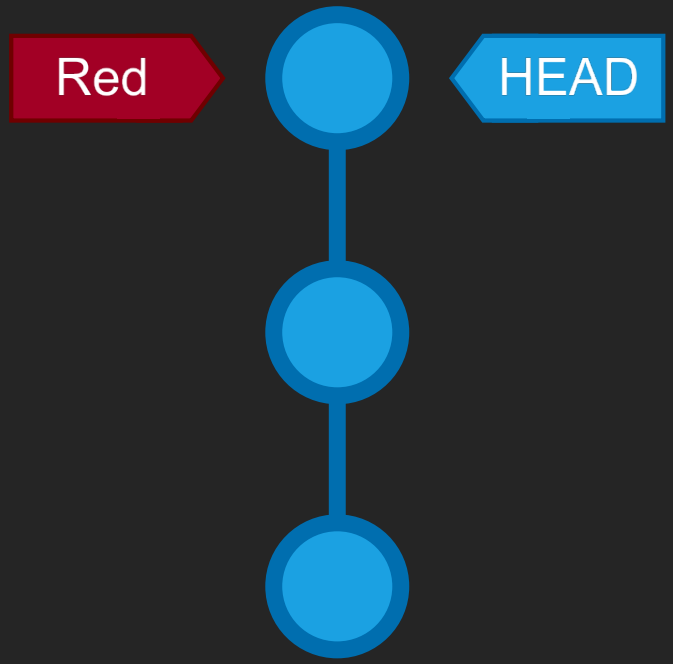
Arten von Merges

Grundsätzlich existieren 2 verschiedene Arten von Merges

3-Way-Merge



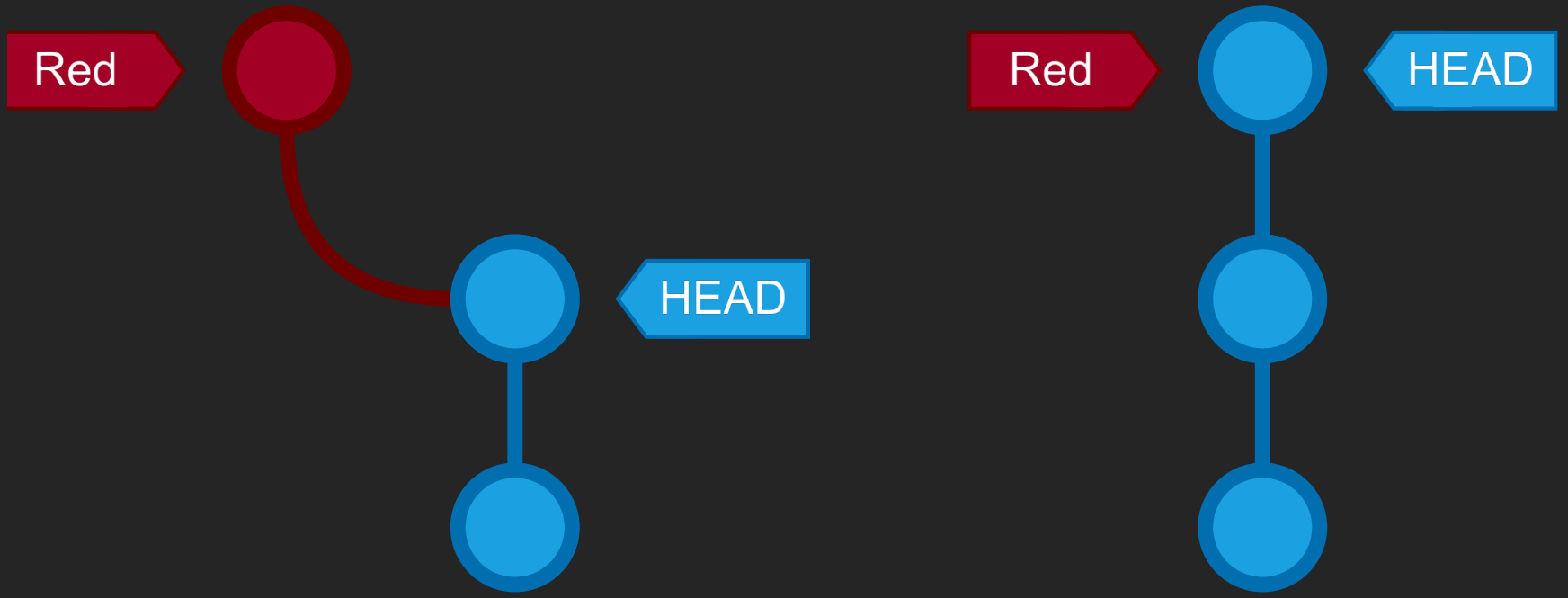
Fast-Forward-Merge



Fast-Forward-Merges

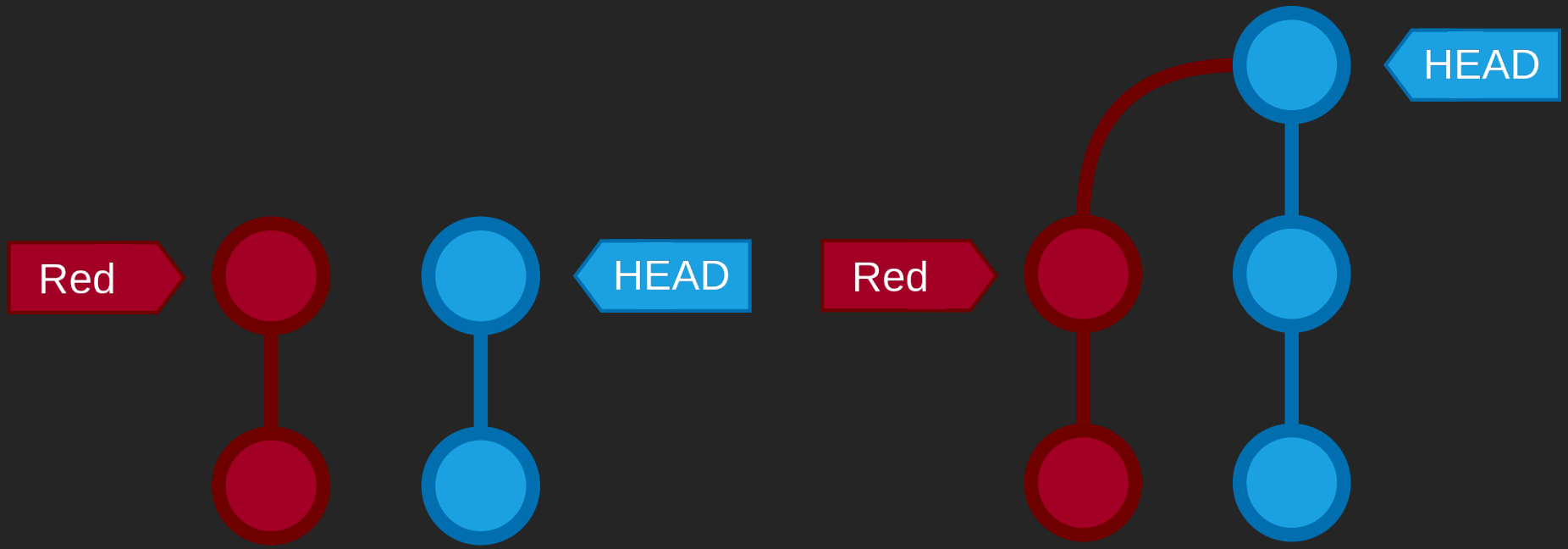
Ein Fast-Forward-Merge (FF-Merge) kann dann durchgeführt werden, wenn ein Branch B ein direkter Nachfahre eines Branches A ist, d.h. jeder Commit aus B ist direkt vom neuesten Commit auf A erreichbar.

Dabei wird einfach der Pointer A dem Pointer B gleich gesetzt



Three-Way-Merges

Ein three-way-Merge nimmt 2 Commits sowie deren neuesten gemeinsamen Vorgänger. Aus diesen 3 Commits wird dann ein neuer **Merge-Commit** gebildet



3-Way-Merge vs. FF-Merge

Standardmäßig versucht Git immer zuerst, einen FF-Merge durchzuführen, nur bei Fehlern wird ein 3-Way-Merge durchgeführt

Mit der Option `--no-ff` können wir direkt einen 3-Way-Merge durchführen

Alternativ können wir auch mit `--ff-only` nur FF-Merges erlauben

Merge Conflicts

Oft kann es vorkommen, dass eine Datei auf beiden Branches verändert wurde. In einem solchen Fall kann Git nicht wissen, welche Version jetzt die richtige ist. Ein **Merge Conflict** entsteht

Es ist nun die Aufgabe des Programmierers, diesen zu beheben!

Im Code sieht ein Merge conflict wie folgt aus:

```
<<<<<<< HEAD (Current Change)
Hello World!
=======
Goodbye World!
>>>>>>> sha (Incoming change)
```

Der Code oberhalb von `=======` stammt von unserem Branch, der Code darunter vom anderen Branch.

Eine mögliche Lösung für unseren Merge conflict sieht wie folgt aus:

Vorher

```
<<<<<<< HEAD (Current Change)
Hello World!
=======
Goodbye World!
>>>>>>> sha (Incoming change)
```

Nacher

```
Hello world and goodbye!
```

Merke: wir können auch eigene Änderungen hinzufügen!

Wir können nun mit `git merge --continue` den Merge-Prozess fortsetzen

Wenn wir lieber gar nicht erst mergen wollen, können wir den Prozess mit `git merge --abort` abbrechen

Hosting-Anbieter

Git ist ein unabhängiger Service, welcher bei verschiedenen Anbietern gehostet werden kann (ähnlich wie z.B. Emails). Ein paar populäre Anbieter sind:

- GitHub
- GitLab (service & self-hosted)
- BitBucket
- CodeBerg
- GiTea (self-hosted)
- Forgejo (self-hosted)

Aufgabe

1. Clont das folgende Beispiel-Repo:
   github.com/JaN0h4ck/Workshop-ColdBlooded
2. Wechselt auf den `dev` branch
3. Mergt `origin/merge-conflict-A` in `dev`
4. Behebt den dabei entstehenden Merge conflict
5. Setzt den Merge fort

Nützliche Commands:

- `git clone`
- `git checkout`
- `git merge`

Gitignore

Es gibt auch Dateien, welche man gar nicht erst in Git verwalten will/sollte. Dazu gehören z.B. lokale Projekt-Dateien, die von einem Programm erstellt werden & oft recht groß sind

Wir können optional eine Datei mit dem Namen **.gitignore** hinzufügen, in welcher die Dateien & directories stehen, die ignoriert werden.

Die Pfade in der .gitignore sind immer relativ, und es kann beliebig viele .gitignores an verschiedenen Stellen im Projekt geben

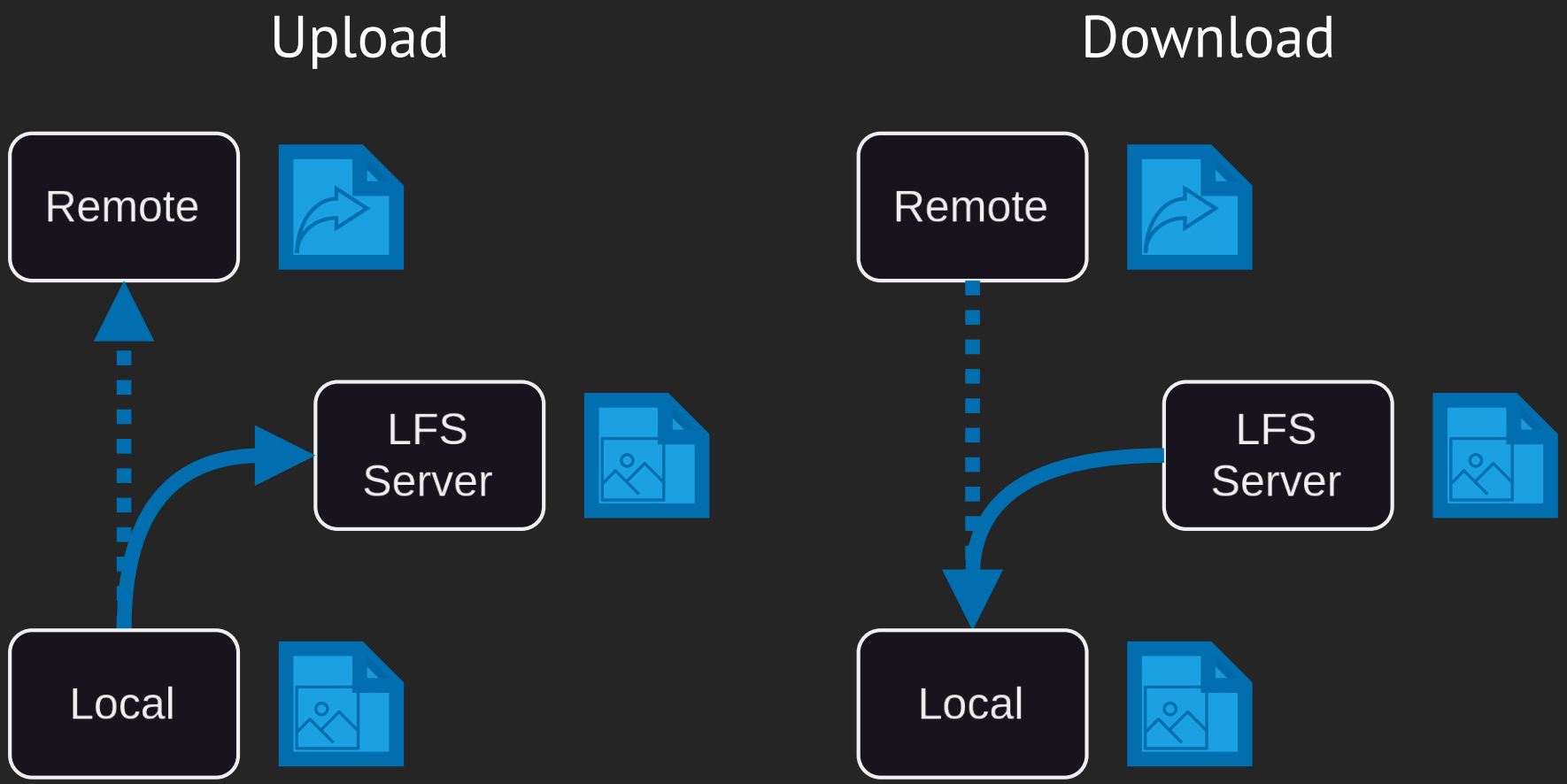
Large File System

Besonders bei der Spieleentwicklung stoßen wir oft an ein paar Grenzen von Git:

- Dateien dürfen maximal 100MB groß sein
- Je größer das Projekt ist, desto langsamer wird Git
- Die diff-tools funktionieren nur mit Text

Als Lösung hierfür gibt es als optionales Plugin das Git Large File System (**LFS**), welches große Dateien auf einem separaten Server speichert

Um LFS in einem Repo zu aktivieren, verwendet man (nach Installation des Plugins) `git lfs install`


Upload
Remote
LFS
Server
Local
Download
Remote
LFS
Server
Local

Damit LFS weiß, welche Dateien diese Sonderbehandlung bekommen, müssen wir diese LFS mitteilen.

```
git lfs track <file>
```

fügt eine einzelne Datei/Directory hinzu

```
git lfs track '*.<filetype>'
```

fügt alle Dateien mit einer bestimmten Dateiendung hinzu

z.B. führt `git lfs track '*.png'` dazu, dass alle PNG-Dateien von LFS verwaltet werden

Reset

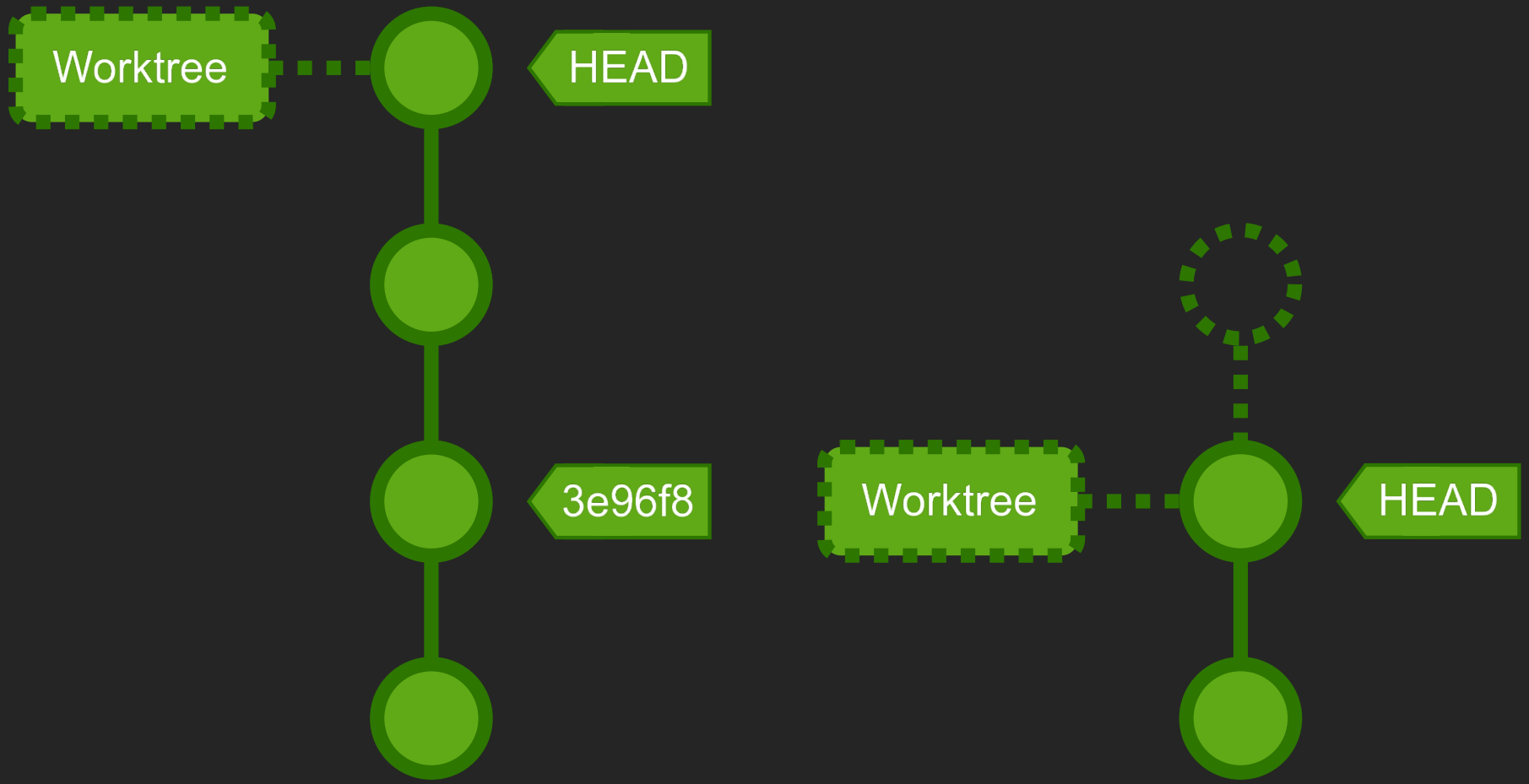
Manchmal wollen wir unsere Änderungen verwerfen und nochmal neu Anfangen. Mit **Resets** können wir die Inhalte unseres Indexes anpassen.

Es gibt 3 Arten von Resets:

- Soft Reset
- Hard Reset
- Mixed Reset

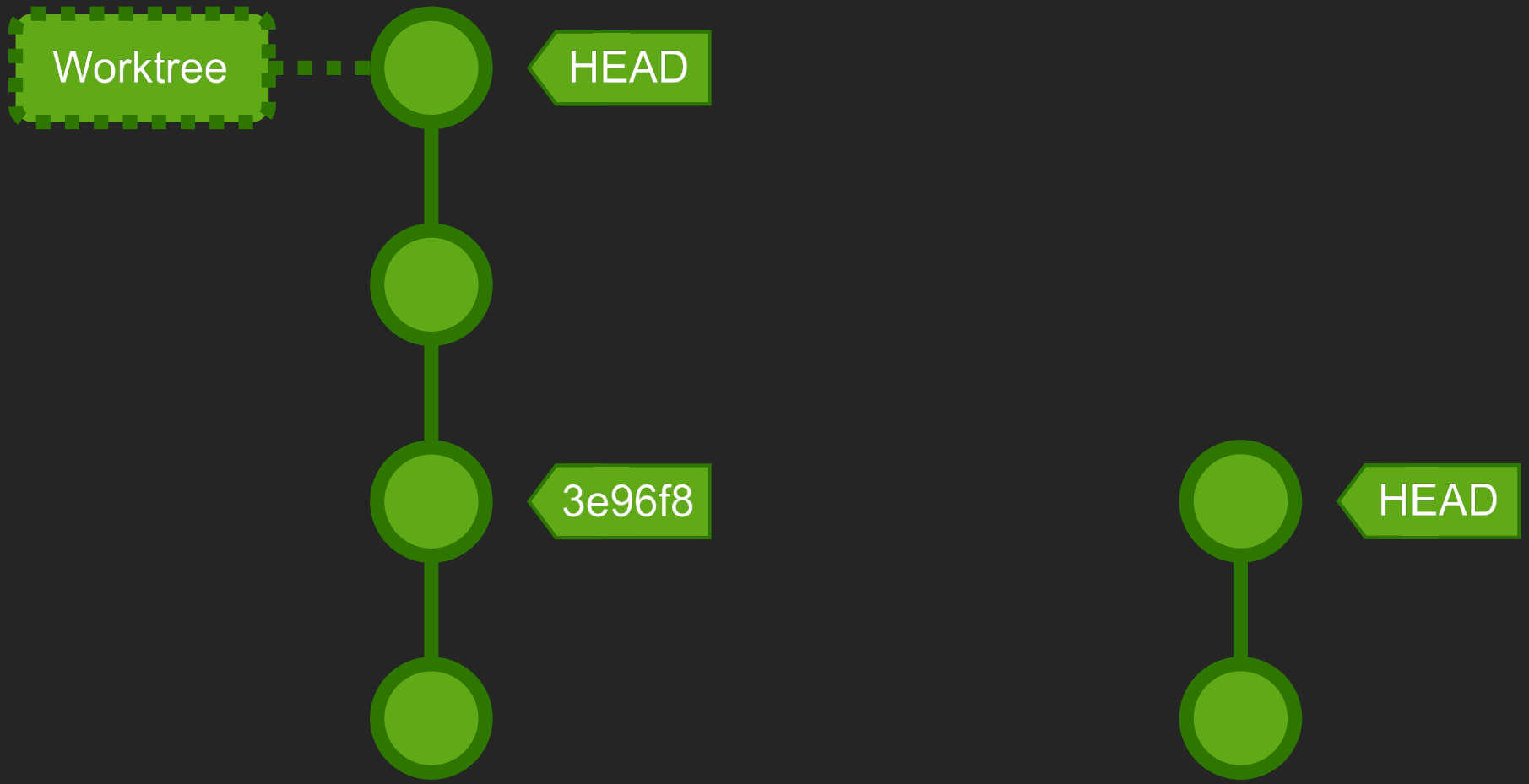
Soft Reset

Der HEAD wird auf 3e96f8 zurückgesetzt. Alle Changes aus den folgenden Commits werden in den Index verschoben, alle Changes im Worktree & Index bleiben erhalten



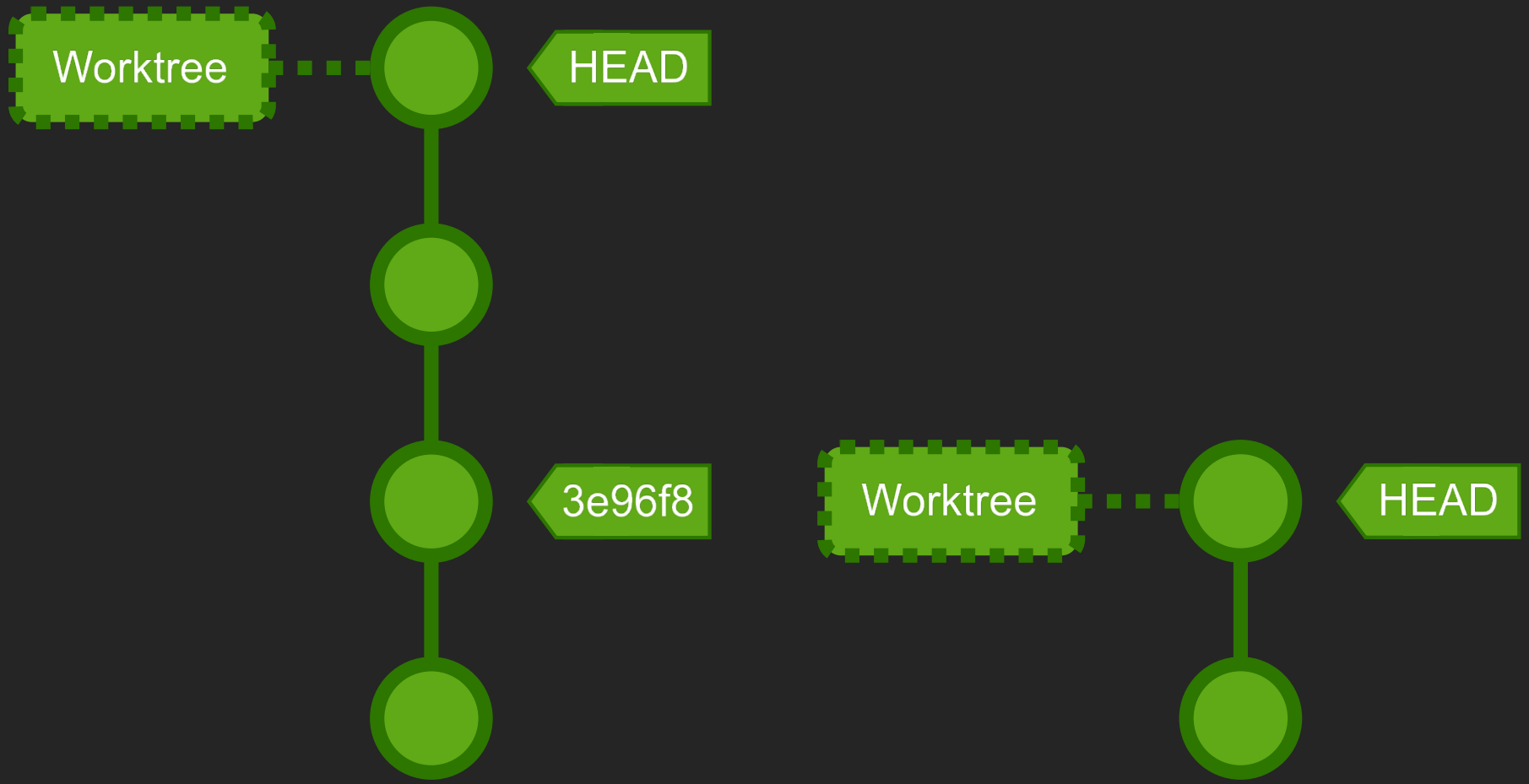
Hard Reset

Der HEAD wird auf 3e96f8 zurückgesetzt. Alle Changes im Worktree und Index, sowie alle Commits dazwischen werden verworfen



Mixed Reset

Der HEAD wird auf 3e96f8 zurückgesetzt. Alle Changes im Index, sowie alle Commits dazwischen werden verworfen, alle Changes im Worktree bleiben erhalten



Clean

Wenn wir unseren Worktree zurücksetzen wollen, ohne den Index zu beeinflussen, ist das mit Resets nicht möglich

Stattdessen verwenden wir dafür `git clean`, um unseren Worktree zu verwerfen. Der Index bleibt dabei erhalten

Als Vorsichtsmaßnahme muss man diese Aktion immer mit `--force` bestätigen, da hier die Änderungen wirklich unwiederbringlich verloren gehen

Optional kann man mit der Option `-i` eine interaktive Session starten, um für jede Datei einzeln zu entscheiden.

Standardmäßig beeinflusst Git clean nur Dateien, die Git auch „sehen“ kann (.gitignore). Dies kann jedoch mit `-x` umgangen werden, um z.B. Build results zu löschen

Aufgabe

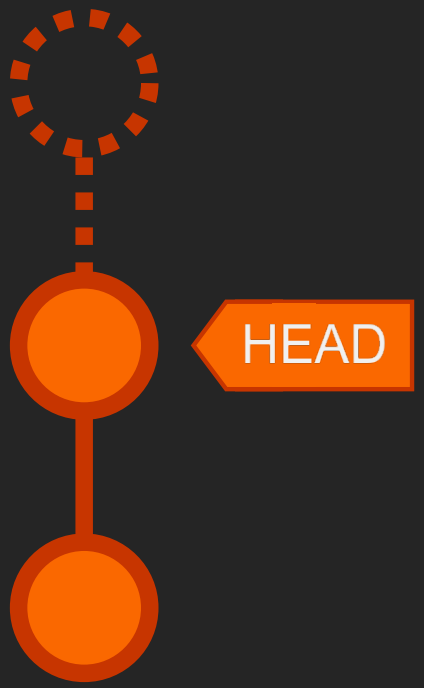
1. Erstellt ein neues Godot-Projekt
2. Fügt eine .gitignore hinzu
3. Bearbeitet diese, damit folgendes ignoriert wird:
   - das `.godot` directory
   - Dateien mit der Ending .import
4. Überprüft, ob eure .gitignore richtig funktioniert
5. Initialisiert Git LFS
6. Fügt ein LFS-Tracking-Pattern für PNG-Dateien hinzu

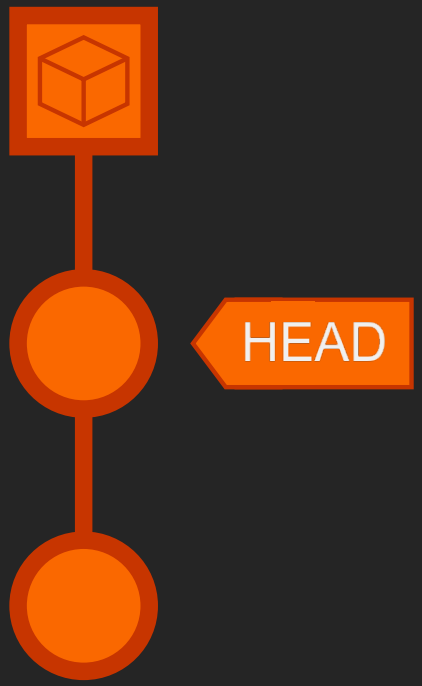
Nützliche Commands:

- `touch <filename>`
- `git lfs install`
- `git lfs track <pattern>`

Stashing

Manchmal will man von seinen Changes ein temporäres backup erstellen, ohne gleich einen Commit zu erstellen. Ein Stash speichert alle aktuellen Changes aus dem Worktree und dem Index (im Gegensatz zum normalen Commit).

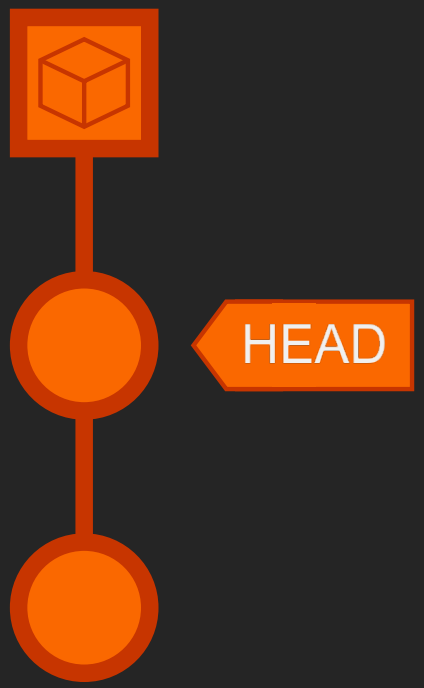


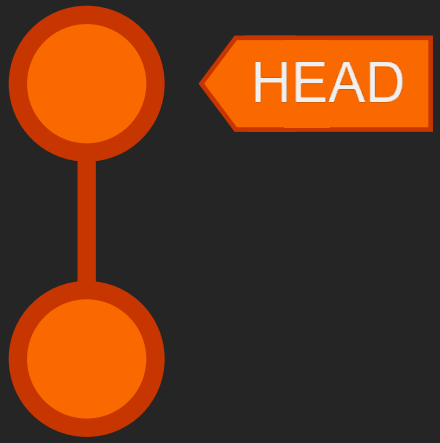


Ein Stash wird mit `git stash` erstellt.

Drop

Löscht den Stash & die darin enthaltenen Daten



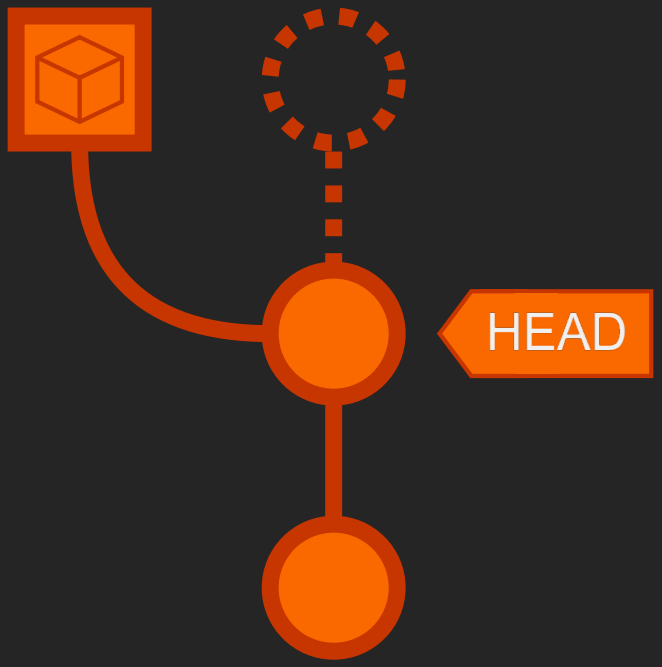


Clear

Droppt alle Stashes

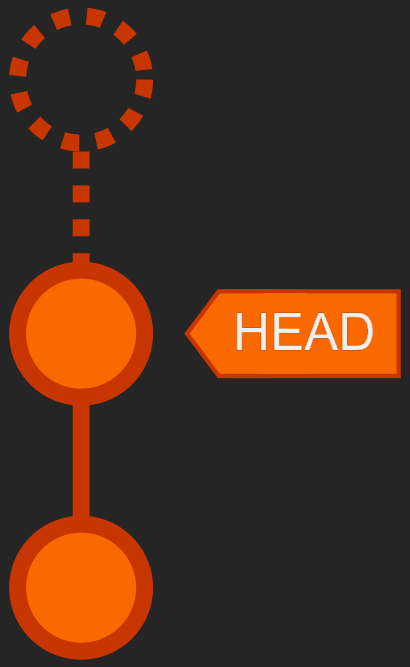
Apply

Kopiert die Daten aus dem Stash zurück in den Index. Der Stash selbst bleibt erhalten



Pop

Führt apply & danach drop aus



Aufgabe

1. Erstellt eine neue Datei mit dem Namen `LICENSE`
2. Speichert euren Worktree als einen Stash
3. Erstellt einen neuen Branch
4. Übertragt den Stash & löscht diesen

Nützliche Commands:

- `git stash`
- `git stash list`
- `git stash apply`
- `git stash pop`
- `git stash drop`

Git Flow

Um ein Repo sortiert zu halten gibt es viele verschiedene Workflows. Einer der bekanntesten dieser ist **Git-flow**

Hierbei sind alle Branches temporär, bis auf zwei permanente Branches:

- **main/master** enthält i.d.R. den neuesten Release
- **develop** enthält die aktuelle WIP-version

Dazu gibt es 3+ Arten von tempörären Branches: **feature**, **release** & **hotfix**

Sobald ein Branch seinen Zweck erfüllt hat, wird dieser gemergt und danach nicht mehr verwendet

I.d.R wird die passende Branch-Kategorie vorne an den Namen angehängt, z.B. **feature**/fireball

Um Git flow (nach Installation des Plugins) in einem Repo zu aktivieren, verwendet man `git flow init`

Um dann einen neuen typ-spezifischen Branch zu starten/ beenden, verwendet man

```
git flow <type> start/finish <name>
```

z.B.

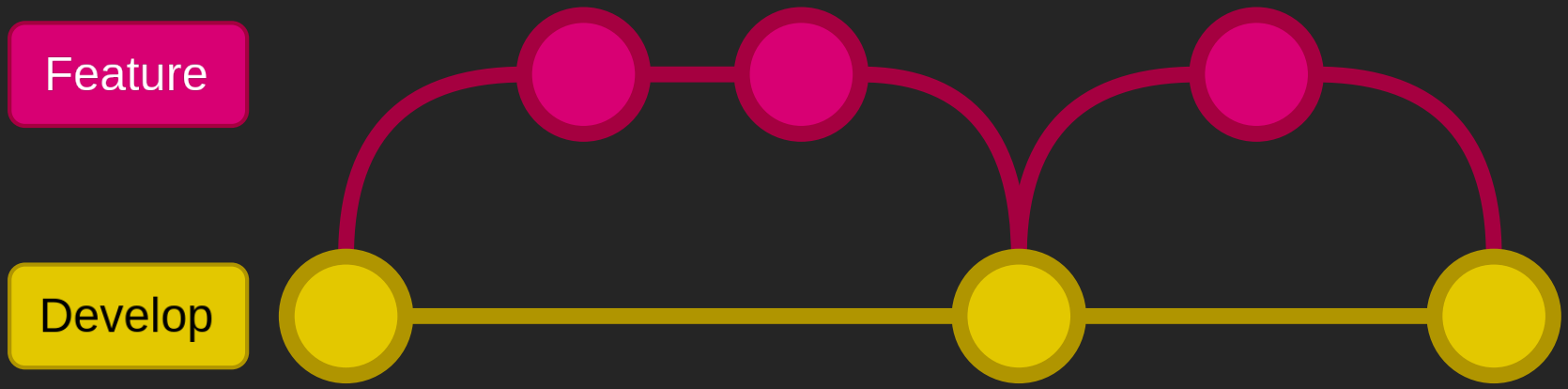
```
git flow feature start fireball
```

Git flow startet dann automatisch den jeweiligen Branch/Merge vom korrekten Branch aus & räumt danach auf

Mit `git flow <type> publish <name>` kann ein Branch gepusht werden, z.B. um den Feature-namen zu „reservieren“ oder zusammen auf dem Branch zu arbeiten

Feature Branches

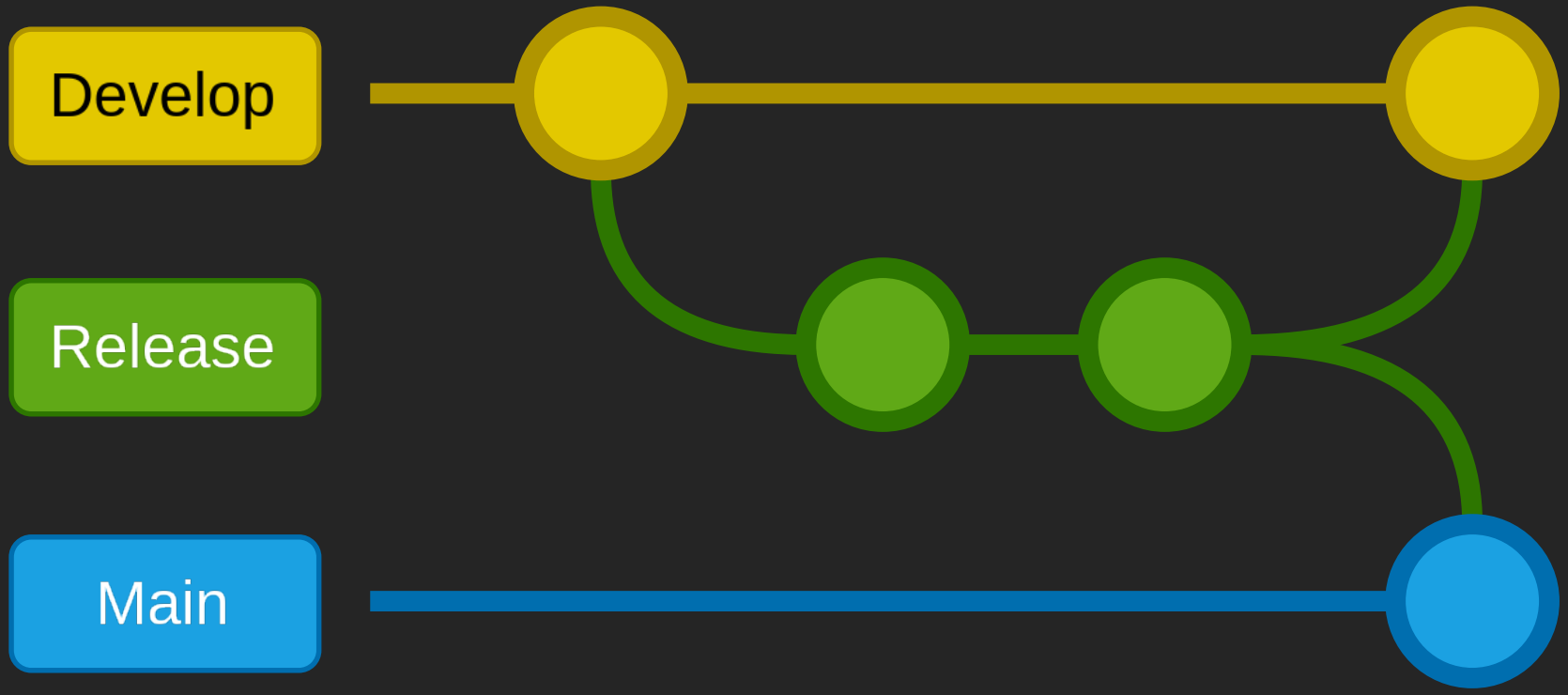
Feature Branches spalten sich von **develop** ab, und werden wieder in **develop** gemergt



Diese Branches sind nur dazu da, jeweils ein neues Feature zu implementieren

Release Branches

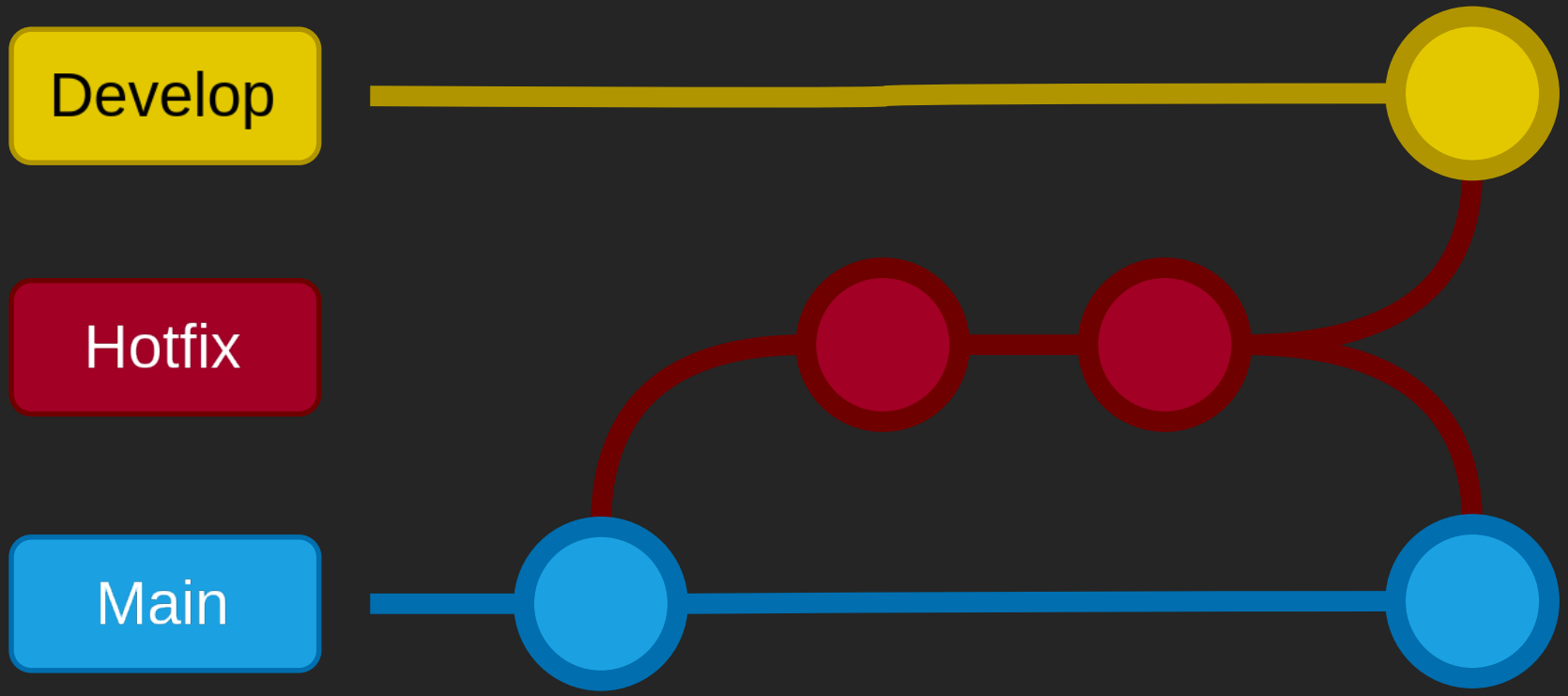
Release Branches spalten sich von **develop** ab, und werden in **develop** & **main** gemergt



Hier wird nur noch aufgeräumt (Bugfixing, Print-Statements löschen, etc.)

Hotfix Branches

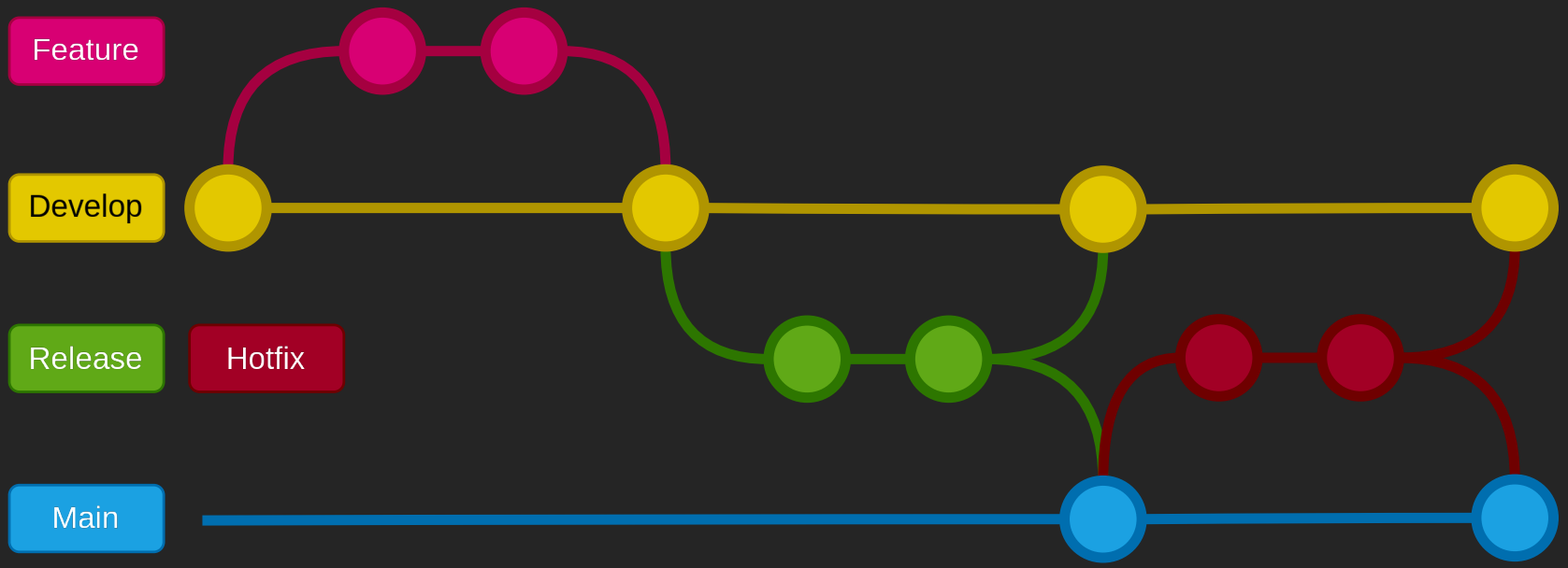
Hotfix Branches spalten sich von **main** ab, und werden in **main** & **develop** gemergt



Hotfixes sind für kritische Bugs, die nicht bis zum nächsten Release warten können

Flow overview

Insgesamt ergibt sich daraus ein **Graph**, der in etwa so aussieht:



Dadurch bestehen Branches nur kurzzeitig, was Merge conflicts reduziert

Je nach Plugin/Frontend kann es auch weitere spezialisierte Branches geben!

Tags

Wir können bestimmte Commits als besonders markieren, indem wir einen **Tag** anhängen. Es gibt 2 Arten von Tags:

1. Lightweight Tags sind simple Pointer, die auf einen Commit zeigen
2. Annotated Tags sind Objekte ähnlich wie Commits mit einer zusätzlichen Message

Tags müssen mit gepusht werden, damit andere diese sehen können. Per Konvention pusht man nur annotated Tags, und behält lightweight Tags als lokale Marker

Hierfür verwendet man

```
git push --follow-tags
```

Branch-Archivierung

Tags können verwendet werden, um Branches zu „archivieren“, wodurch die Branch-liste kurz gehalten wird

1. `git tag <name> <branch>` erstellt einen neuen tag
2. `git branch -d -r origin/<branch>` löscht den Branch lokal
3. `git push -d origin/<branch>` löscht den Branch remote
4. `checkout -b <branch> <tag>` stellt den Branch wieder her

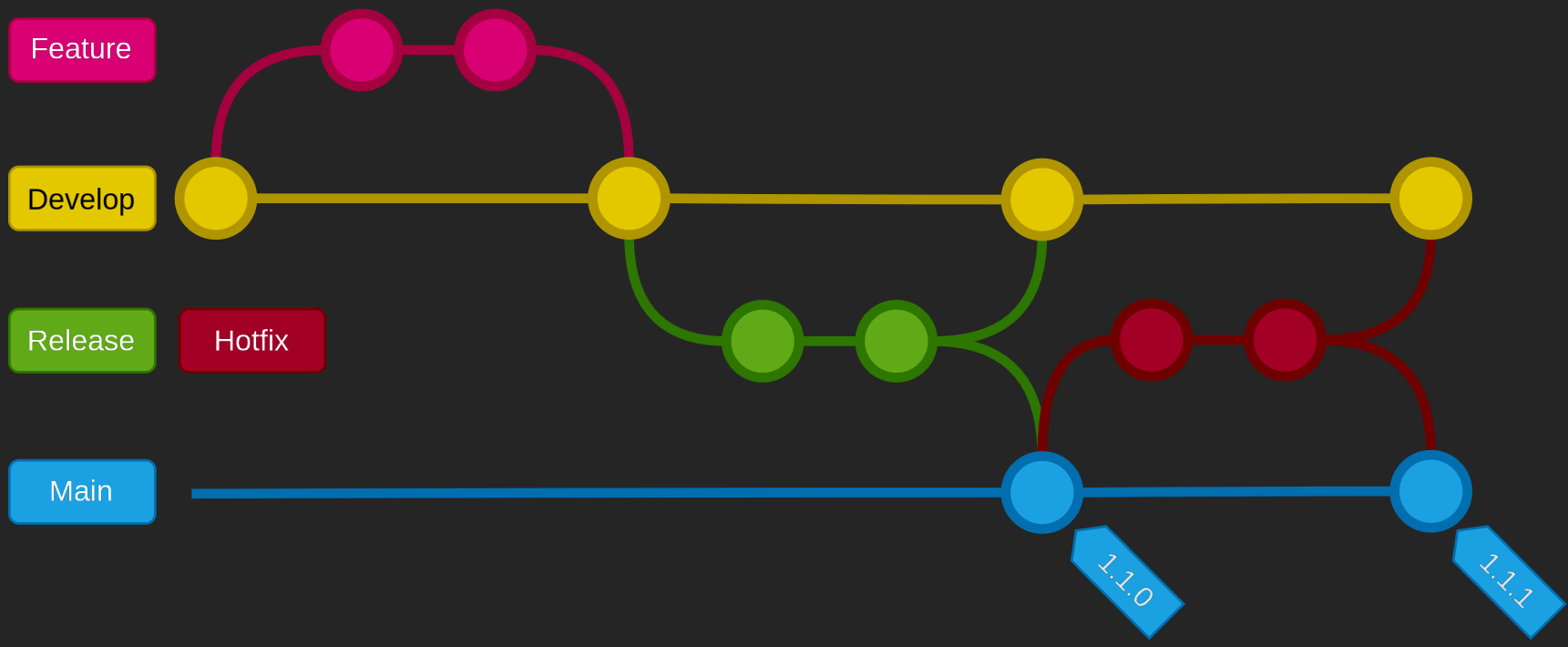
Vorsicht: Damit kann man sich auch leicht das Projekt zerschießen.

Wer mehr erfahren will, dem empfehle ich diesen Thread

Semantische Versionierung

Häufig werden Tags zur „Semantischen Versionierung“, d.h. Versionsnummern, verwendet.

Im Bezug auf Git-flow ergibt sich aus Main, Release & Hotfix braches die Versionsnummer vA.B.C



Forks & Pull Requests

In größeren Projekten (besonders bei Open-Source) will man in der Regel nicht, dass einfach jeder irgendwas ins Repo pushen kann. Stattdessen erstellt man einen **Fork**

Ein **Fork** ist quasi eine Kopie des Repos, die euch selbst gehört, hier könnt ihr dann z.B. ein neues Feature entwickeln

Anschließend kann man einen "Antrag" stellen, dass Commits vom Fork ins originale Projekt übernommen werden. Das nennt man eine **Pull Request**

(Je nach Anbieter heißt das manchmal auch **Merge Request**, ist aber quasi das selbe)

Der Open-Source Workflow

1. Ein Issue wird geöffnet
2. Ein Fork wird erstellt
3. Im Fork wird ein Feature entwickelt/Bug gefixt
4. Eine Pull Request wird erstellt
5. Die Pull Request wird untersucht & diskutiert
6. Die Pull Request wird akzeptiert
7. Der Fork wird gelöscht

Aufgabe

1. Forkt das folgende Repo:
   github.com/JaN0h4ck/workshop-html-template
2. Verändert etwas im Code
3. Erstellt eine Pull-Request

1. Block Geschafft!

Damit seid ihr vorerst erlöst, wir sehen uns in einer Woche für die fortgeschrittenen Themen.

Gibt es noch Fragen?

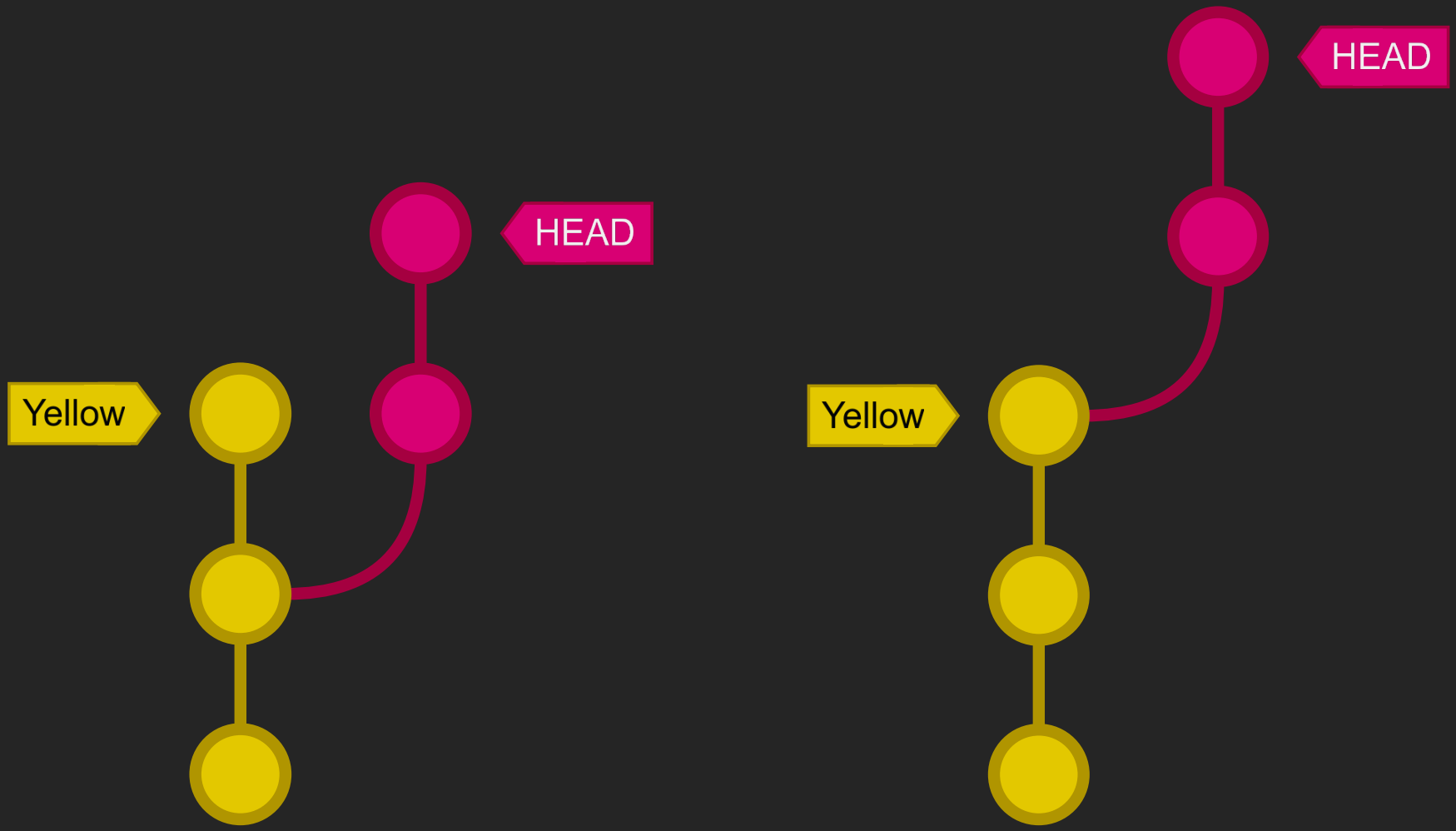
Willkommen zurück!

Heute geht es um fortgeschrittene Themen.

Vorab: gibt es noch Fragen?

Rebase

Ein Rebase ändert (wie der name schon sagt) die Basis eines Branches. Das Command hierfür lautet `git rebase <target>`



Der Hauptgrund zu rebasen, ist, dass wir danach einen garantierten FF-Merge durchführen können. Somit entsteht kein zusätzlicher Merge-Commit, und unsere History bleibt übersichtlicher. Außerdem können wir auf diese Weise Commits & Branches „verschieben"

Man kann z.B. seinen Feature Branch auf dem aktuellen develop/main rebasen, damit man die neuesten fremden Features hat

Da Commits immer von deren Vorfahren abhängig sind, können diese nicht einfach verschoben werden. Stattdessen werden die Commits kopiert & die Originale gelöscht

Das bedeutet, dass bei einem Rebase auch Merge conflicts entstehen können

Aufgabe

1. Wechselt zurück zum ColdBlooded Projekt von vorhin
2. Wechselt auf den Branch `rebase-example`
3. Rebased den Branch auf `dev`

Nützliche Commands:

- `git rebase <branch/commit>`
- `git rebase --continue`
- `git rebase --abort`

History rewrites

Wenn ihr `rebase-example` jetzt pushen wollt, ist das nur mit einem Force-Push möglich

Da Rebases unsere Commits nicht wirklich verschieben, sondern neu aufsetzen, ändert sich dabei zwangsweise immer unsere Commit History!

Das kann auch Probleme bereiten, insbesondere dann, wenn die betroffenen Commits bereits auf dem remote vorliegen.

In einem solchen Fall müsste man sicher stellen, dass jeder die neue version der History besitzt, und alle existierenden alten histories gelöscht wurden

Als Faustregel daher: History rewrites am besten nur bei rein lokalen Commits anwenden!

Interactive Rebase

Rebases bieten uns nicht nur die Möglichkeit, Commits zu verschieben. Mit `-i` können wir eine interactive rebase session starten, die uns einige weitere Werkzeuge zur Verfügung stellt

In einer interaktiven session wird eine Liste aus Commits bereitgestellt, welche wir bearbeiten können:

```
pick b03d907 Setup project
pick 561e3a0 Added player
pick 8b3a2ec Added enemy
pick 7dbf0dc Added fireball
```

Merke, dass die Reihenfolge hier umgekehrt ist, als wir es gewohnt sind! Die ältesten Commits sind oben, die neuesten unten

Wir können diese Liste nun bearbeiten, um unser gewünschtes Ergebnis zu erzielen:

```
pick b03d907 Setup project
pick 561e3a0 Added player
pick 8b3a2ec Added enemy
pick 7dbf0dc Added fireball
pick 7fbec96 Changed color
```

```
edit b03d907 Setup project
reword 561e3a0 Added hero
pick 7dbf0dc Added fireball
pick 8b3a2ec Added enemy
```

Was genau ist hier jetzt eigentlich passiert?

- Setup-Commit wird für Bearbeitung markiert
- Player-Commit wird umbenannt
- Enemy & Fireball-Commits werden vertauscht
- Farbänderungs-Commit wird gelöscht

Wir können die Datei nun speichern & schließen, um unseren rebase fortsetzen

- `edit` lässt uns einen Commit für die Bearbeitung markieren. Wenn der rebase durchläuft, wird an dieser Stelle pausiert, damit wir unsere Änderungen vornehmen können. Danach können wir mit `git rebase --continue` weiter machen
- `squash` & `fixup` lassen uns einen Commit mit seinem Vorgänger kombinieren.

Aufgabe

1. Wechselt zurück auf das HTML-Template Repo
2. Kombiniert alle commits zu einem einzigen

Nützliche Commands:

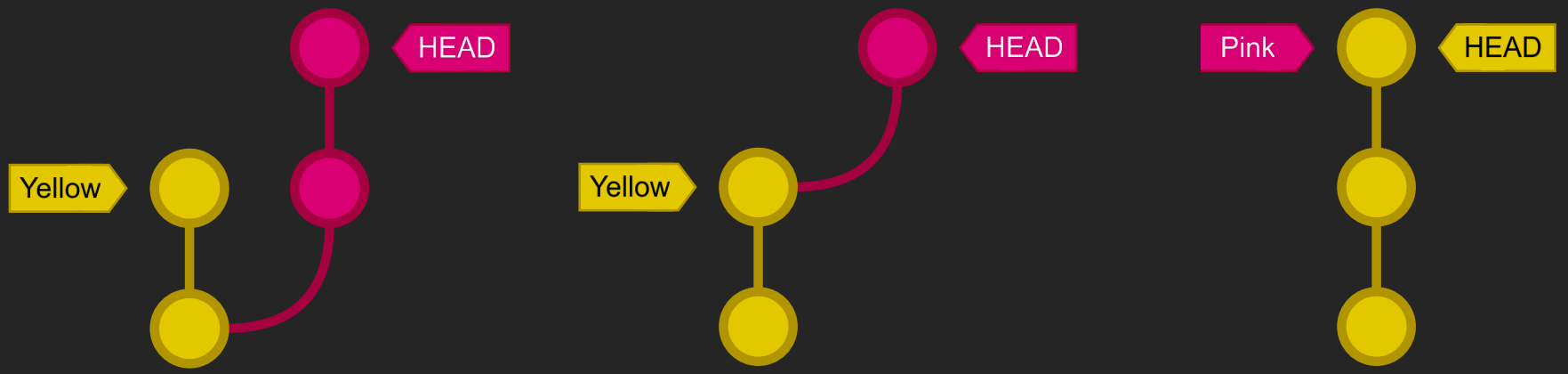
- `git rebase -i`

Hinweis: `HEAD~n` bedeutet so viel wie: „n Commits vor dem aktuellen HEAD“

Manche arbeiten gar nicht mit 3-Way-Merges, sondern nur mit Rebases & FF-Merges.

In diesem Beispiel wird der pinke Branch zu einem einzelnen Commit zusammengedrückt, und dieser dann rebased.

Danach kann man den pinken Branch FF-Mergen, und der Commit-Graph ist eine einzige Linie



Ob man diesen Weg nehmen will ist Geschmackssache

Maintenance

Mit der Zeit sammelt Git im Hintergrund Daten an, welche mit der Zeit unsere Prozesse verlangsamen.

Wir können Git anordnen, im Hintergrund aufzuräumen, um speicher wieder freizugeben & unsere Prozesse zu beschleunigen:

```
git maintenance run
```

Wir können Git auch anordnen, die maintenance regelmäßig (standardmäßig jede Stunde) auszuführen:

```
git maintenance start
```

Branch Protection

Es ist normal, dass Leichtsinnsfehler passieren, wie z.B. ein push direkt auf main. Wir können Scripts an sog. Hooks anhängen, um „Schranken“ einzubauen.

GitHub lässt uns dies Grafisch mittels sog. **Branch Protection Rules** erledigen

Wir finden diese unter `Account > Repo > Settings > Branches`

Beispiele für solche Regeln wären:

- keine Force-Pushes auf main
- Merges auf develop benötigen eine Pull Request mit 3 OKs
- Tests müssen vor Merges erfolgreich durchlaufen
- alle Commits müssen mit einer Signatur unterzeichnet sein

Auf diese Weise kann z.B. auch Git-flow erzwungen werden

Grep

Unsere IDEs erlauben uns i.d.R., alle unsere Code-Dateien nach einem pattern zu durchsuchen (z.B. `ctrl+F health`)

Git bietet dieses Feature auch an, um schnell ein gesamtes Repo zu durchsuchen:

```
git grep -En <regex>
```

durchsucht das Repo nach einem bestimmten regulären Ausdruck

(`-E` bietet den erweiterten regex-Syntax, `-n` zeigt Zeilennummern an)

Blame

Manchmal ist es sinnvoll zu wissen, **wer** ein Feature implementiert hat. Wenn man z.B. wissen will, warum etwas genau so implementiert wurde, macht es oft nicht so viel Sinn „Wer hat das gemacht?“ in den Gruppenchat zu schreiben

Mit `git blame <file>/<commit>` kann Git uns genau sagen, wer was implementiert hat. Dann können wir diese Person direkt kontaktieren

Blame erkennt standardmäßig auch whitespace-Changes & moves. Diese lassen sich jeweils mit `-w` für whitespaces, und `-M` für moves ignorieren

Mit `-L` kann nur ein bestimmter Bereich an Zeilen überprüft werden

Aufgabe

1. Wechselt zurück zum ColdBlooded Repo
2. Durchsucht das Repo nach `health`
3. In manchen Dateien ist die Health auf 1000 gesetzt. Findet heraus, wer diesen Commit erzeugt hat

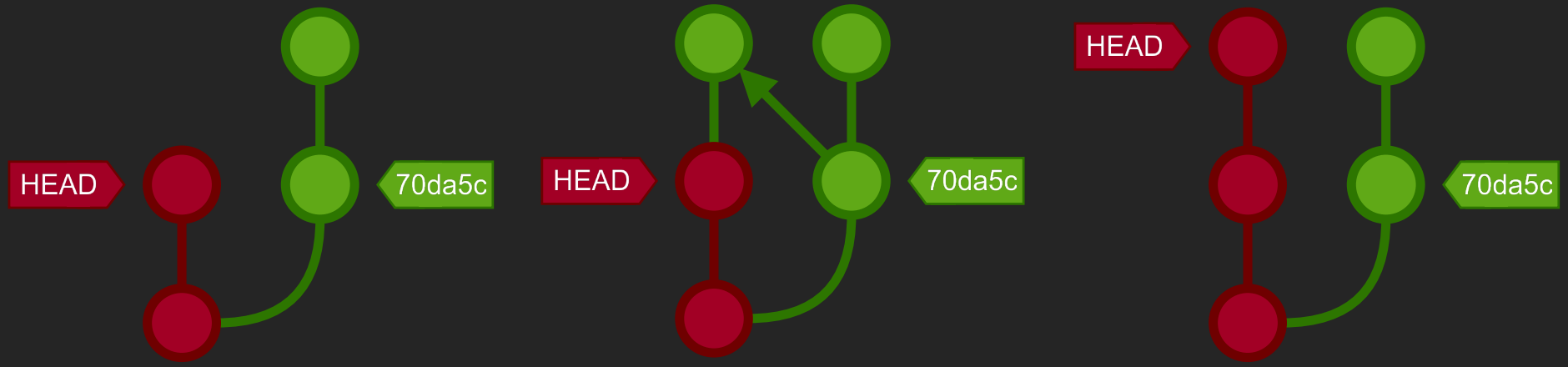
Nützliche Commands:

- `git grep -En <pattern>`
- `git blame <file/commit>`

Cherry Picking

Manchmal braucht man ein Feature von einem anderen Branch, dieser ist jedoch noch nicht bereit zum mergen

Mit `git cherry-pick <commit1> ...` können wir beliebige Commits auf unseren aktuellen Branch kopieren



Alternativ können wir mit `--no-commit` den Commit direkt in unseren Worktree kopieren, ohne einen neuen Commit zu erzeugen

Aufgabe

1. Wechlset auf den `cherry-pick-example` Branch
2. Übertragt den Commit `eb77156a` auf den aktuellen Branch
3. Überprüft dies mit logs

Nützliche Commands:

- `git cherry-pick <commit>`
- `git log`

Worktrees

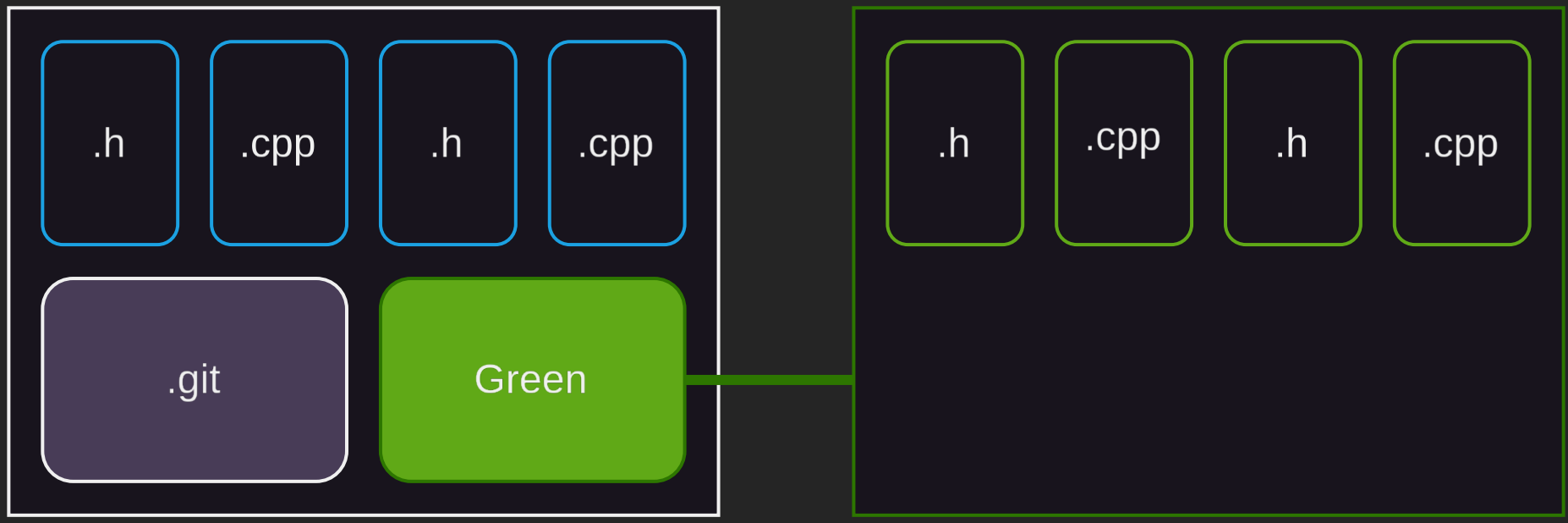
Bisher haben wir immer nur in einem Worktree gearbeitet. Git erlaubt uns jedoch, mehrere Worktrees gleichzeitig zu verwalten.

Im Endeffekt bedeutet das, dass wir an mehreren Branches gleichzeitig arbeiten können

`git worktree add <directory> <branch>` öffnet `<branch>` in einem neuen Directory. Innerhalb dieses directories können wir dann wie gewohnt arbeiten

Mit der Option `-b` wird ein neuer Branch erzeugt

Mit `git worktree prune` wird ein worktree entfernt, nachdem das Directory gelöscht wurde (interner clean-up)


.h
.cpp
.h
.cpp
.git
Green
.h
.cpp
.h
.cpp

Aufgabe

1. Erstellt einen neuen worktree für `<branch>`
2. Nehmt eine Änderung im worktree vor
3. Entfernt den worktree

Nützliche Commands:

- `git worktree add <directory> <branch>`
- `git worktree prune`

Submodules

Wir wollen in unserem Projekt ein Plugin/Library einbinden, welches durch ein eigenes Git Repo verwaltet wird.

Anstatt die Dateien manuell zu kopieren & zu updaten, können wir das externe Repo als ein sog. **Submodule** integrieren

Ein Submodule enthält immer die Dateien eines bestimmten Commits

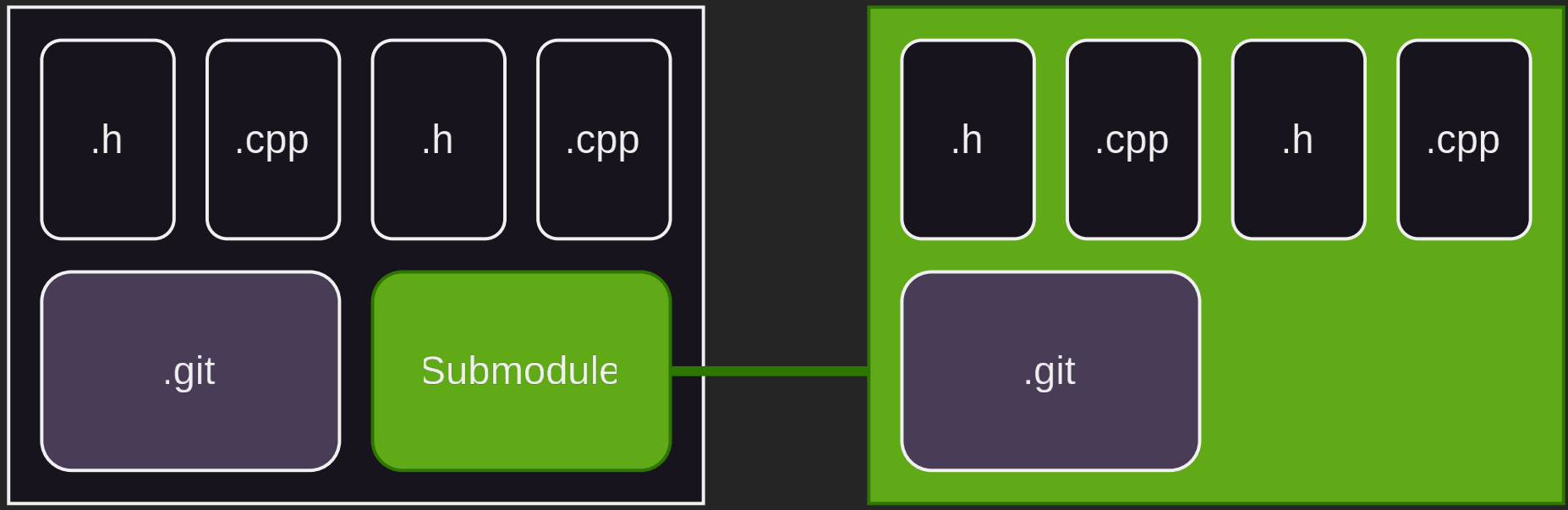
`git submodule add <url>` fügt ein submodule hinzu, und speichert eine Referenz darauf in `.gitmodules`

`git submodule init <path>` initialisiert ein submodule.

Wird ein Repo mit submodules geclont, so muss dieser Schritt jeweils neu durchgeführt werden! Um ein submodule zu aktualisieren, verwenden wir
`git submodule update --remote --merge <path>`

Vorsicht: `git submodule update` synchronisiert nur das submodule mit der Konfigurations-Datei, ohne `--remote` werden keine neuen Daten heruntergeladen!

Wollen wir ein submodule deaktivieren, so können wir es mit `git submodule deinit <path>` deinitialisieren


.h
.cpp
.h
.cpp
.git
Submodule
.h
.cpp
.h
.cpp
.git

Bisect

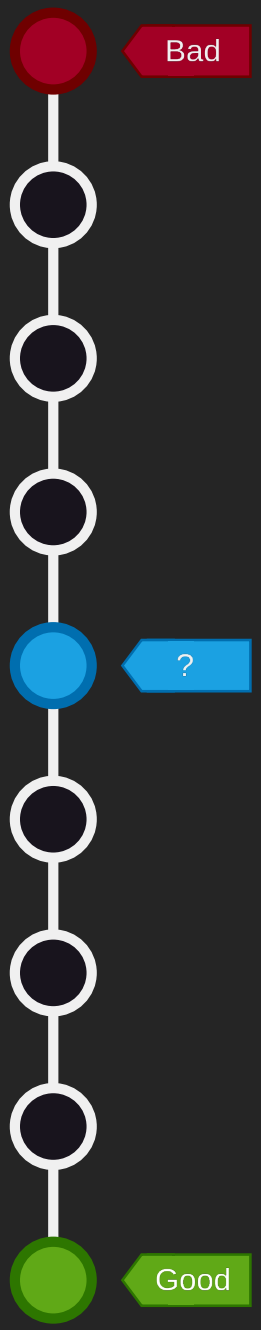
Manchmal ist es wichtig zu wissen, wann sich etwas verändert hat, z.B. wann ein Bug entstanden ist.

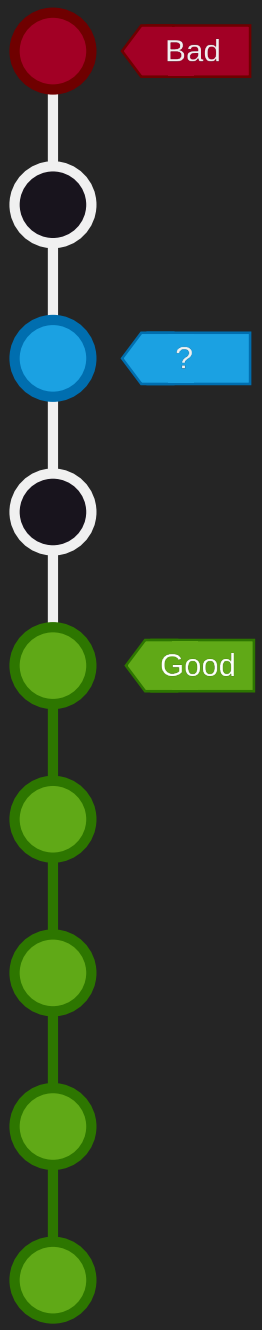
Bisects helfen uns, mithilfe einer binären Suche den betroffenen Commit zu finden

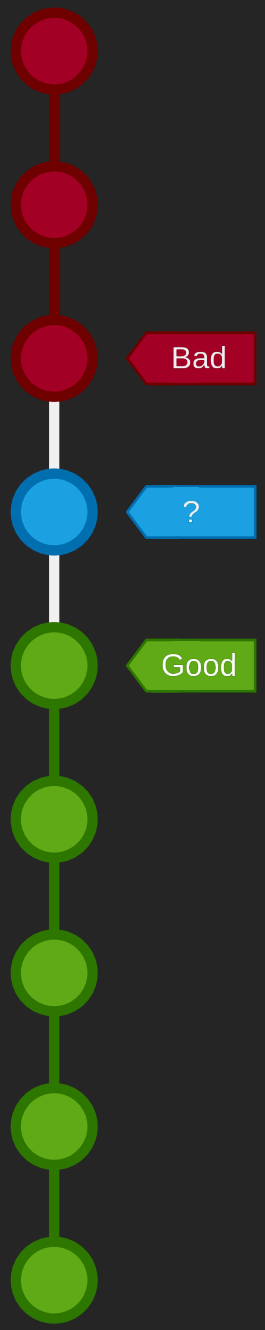
`git bisect start <bad_commit> <good_commit>` started einen Bisect zwischen den angegebenen Commits

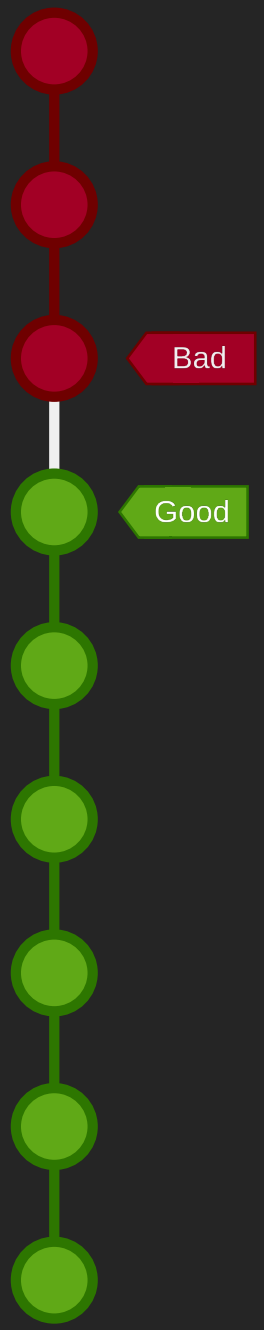
Git wählt einen Commit aus, welchen wir dann überprüfen müssen. Mit `git bisect good/bad` können wir Git mitteilen, was der Status des aktuellen Commits ist

Dies wiederholt sich, bis der korrekte Commit gefunden ist. Danach können wir mit `git bisect reset` zu unserem Ausgangspunkt zurückkehren


Bad
?
Good


Bad
?
Good


Bad
?
Good


Bad
Good

Aufgabe

1. Wechselt auf den `example-bisect` Branch
2. Aktuell wird beim Sprung kein Ton mehr abgespielt. Findet mit Bisect den Commit, in dem diese Änderung eingefügt wurde

Nützliche Commands:

- `git bisect start <bad_commit> <good_commit>`
- `git bisect good`
- `git bisect bad`
- `git bisect reset`

(Man kann auch `new` statt `bad` und `old` statt `good` verwenden)

Reflog

Wir kennen bereits `git log`, um uns die History eines Commits anzeigen zu lassen.

Git bietet uns außerdem das sog. **Reflog** (Reference Log)

Das Reflog stellt die History der Referenzen (d.h. Branches, Tags, etc) dar

Mit dem Reflog können wir somit aktionen wie Merges, Rebases, etc. einsehen

Das Reflog lässt sich mit `git reflog` einsehen

Über das Reflog lassen sich z.B. Branches & Commits wiederherstellen, und stellt die zugrundelegende Funktion hinter den „Undo“ buttons dar, die es in manchen Frontends gibt

CI/CD Pipeline

Um zu verhindern, dass fehlerhafter Code im Repository gespeichert wird, werden häufig vordefinierte Tests verwendet

Git Hosts erlauben uns oft, diesen & viele weitere Prozesse mithilfe der CI/CD-Pipeline zu automatisieren

Diese Pipeline besteht aus den folgenden Teilen:

- Continuous Integration
- Continuous Delivery/Deployment

Eines der Hauptziele dabei ist, sog. „Merge Days“ zu verhindern

Ein Beispiel wäre GitHub Pages, wobei Changes auf dem main Branch direkt das Deployment der page triggern

Continuous Integration

Anstatt mehrere große Branches zu haben, die Erst zum Ende eines Features gemergt werden, werden bereits kleine Arbeitsergebnisse gemergt, und somit möglichst regelmäßig vom neuesten main Branch gearbeitet

Gepushte Changes werden vom CI-System automatisch überprüft (Formatting, Linting, Building, Unit/Integration tests, etc.)

Continuous Delivery/Deployment

Sofern die CI-Tests alle erfolgreich abgeschlossen sind, mergt das Continuous Delivery System die Änderungen in den sog. **Trunk** (aka main), und bereitet automatisch den Build & Release vor

Sobald die Änderungen abgesegnet wurden, werden diese dann veröffentlicht. Somit ist kaum weitere Arbeit dafür notwendig

Continuous Deployment geht noch einen Schritt weiter: Hier sind alle Schritte automatisiert, und es ist kein menschlicher Eingriff mehr gefordert

Geschafft

Nun solltet ihr auch einige fortgeschrittene Techniken kennen, und somit gut für das GE-Lab oder andere Projekte vorbereitet sein!

Gibt es noch Fragen?

Weiterer Lesestoff

Wer noch mehr Features von Git entdecken will, dem lasse ich hier ein paar Pointer dar:

- aliases
- archive
- autocorrect
- autosquash
- bundles
- filter branch/repo
- hooks
- patches
- rerere

Für einen wirklichen Deep-Dive kann ich „Pro Git“ empfehlen: https://git-scm.com/book/en/v2