Objektorientierte Programmierung

Hochschule Bochum
WS 19/20
Dr.-Ing. Darius Malysiak

Dozent

- Darius Malysiak
- Studium:
  - Elektrotechnik HS-Bochum (Dipl.-Ing. FH)
  - Angewandte Informatik RUB (M.Sc.)
  - Mathematik RUB (B.Sc.)
  - Neuroinformatik RUB (Dr.-Ing.)

- Lehre:
  - Seit 2007 kontinuierlich RUB, HS RW, HS Bochum (Naturwissenschaften, Informatik)

- Industrie:
  - Unternehmensberatung
    - Kryptographie
    - Bildverarbeitung
  - Projektleitung
  - DevOps Cloud-Software
  - Softwareentwickler C++/Java

- Forschung:
  - GPU basiertes maschinelles Lernen
  - Bildverarbeitung
  - Verteilte Systeme

Inhalt

I. Formalitäten

II. Java Wiederholung / Versionsmanagement

III. Vertiefung Java 7/8

- Aktuelle Position von Java
- Generics
- Annotations
- Lambdafunktionen
- Streams
- Performance
- Tests

IV. Design Patterns

- UML
- *-Varianz
- MVC revisited
- Dependency Injection
- Builder
- Proxy

V. OOP Software Engineering

- Principle of Least Surprise
- Domänen Modell
- Softwarestruktur
- Codestyle
- SOLID Kriterien
- …

I. Formalitäten

- Benötigte Software: Eclipse, JDK8, Git, (Lombok, Maven)
- Vorlesung (H8): Montag 16:00-17:45
- Übung (C5-06): Montag 18:00-19:30
- Modulabschluss: Erfolgreiche Teilnahme an Klausur

I. Formalitäten

- Inhalt der Übung: Aufarbeitung / Vertiefung der Vorlesungsinhalte.
- Eigenes Notebook bevorzugt (Lerneffekt der Einrichtung), alternativ: Nutzung der Rechner an der HS verfügbar.
- Mitschreiben ist sehr hilfreich!
- Übungsblätter wöchentlich; Bearbeitung während der Übung + evtl. Nachbesprechung in der darauf folgenden Übung.
- Kontakt: darius.malysiak@hs-bochum.de, darius.malysiak@secunet.com, jederzeit auch bzgl. angrenzenden Themen zur Vorlesung.

II. Java Wiederholung

Klassen (Classes):

- Schablonen zur Erstellung von Containern für Logik und Daten.
- Logik und Daten eines Containerns können Zugriffsbeschränkungen erhalten (**public**, **protected**, **private**).
- Schablonen können auf Basis von bereits existierenden Schablonen erstellt werden (**Vererbung**).
- Ein Container C wird durch „*new C()*“ **instanziiert.**

II. Java Wiederholung

Klassen: Beispiel

```
public class Container {
    public Container(String[] strings){this.strings = strings;}
    public void print(){…}
    private String[] strings;
}


public class Main {
    public static void main(String[] args) {
        String[] strings = {"das,ist","ein,test,!"};
        Container c = new Container(strings);
    }
}
```

II. Java Wiederholung

Klassen: Beispiel

```
public abstract class Containers {
    public abstract void print();
}


public class Container extends Containers{
    public Container(String[] strings){this.strings = strings;}
    public void print(){…}
    private String[] strings;
}


public class Main {
    public static void main(String[] args) {
        String[] strings = {"das,ist","ein,test,!"};
        Container c = new Container(strings);
    }
}
```

II. Java Wiederholung

Schnittstellen (Interfaces):

- Definieren Funktionen, welche eine Klasse implementieren muss.
- Schnittstellen werden implementiert, hierbei kann eine Klasse mehrere Schnittstellen implementieren.
- Können auch Datencontainer für Klassen definieren.
- Können bereits Logik enthalten.
- Ein Container C wird durch „*new C()*“ **instanziiert**

II. Java Wiederholung

Interfaces: Beispiel

```
public interface Containers {
    public void print();
}


public class Container implements Containers{
    public Container(String[] strings){this.strings = strings;}
    public void print(){…}
    private String[] strings;
}


public class Main {
    public static void main(String[] args) {
        String[] strings = {"das,ist","ein,test,!"};
        Container c = new Container(strings);
    }
}
```

II. Java Wiederholung

Schnittstellen / Klassen:

Wofür ist Vererbung eigentlich nützlich?

- Redundanz wird vermieden (keine Neuerfindung des Rads).
- Abstraktion durch Eltenklassen möglich (konkrete Ausprägung kann bel. gewählt werden).

Wofür eigentlich überhaupt zwischen Klassen und Schnittstellen unterscheiden?

Warum nicht einfach Mehrfachvererbung nutzen? C++ funktioniert doch auch damit!

II. Java Wiederholung

Diamond-Problem:

```
public abstract class Containers {
    public abstract void print();
}
```

```
public class Container1 extends
Containers {
    public void print(){…}
}
```

```
public class Container2 extends
Containers {
    public void print(){…}
}
```

```
public class Container extends Container1,Container2{
    public Container(String[] strings){this.strings = strings;}
    private String[] strings;
}
```

```
public class Main {
    public static void main(String[] args) {
        String[] strings = {"das,ist","ein,test,!"};
        Container c = new Container(strings);
        c.print();}
}
```

II. Java Wiederholung

Diamond-Problem Lösung:

```
public interface Container1 {
    public void print();
}
```

```
public interface Container2 {
    public void print();
}
```

```
public class Container implements Container1,Container2{
    public Container(String[] strings){this.strings = strings;}
    public void print(){…}
    private String[] strings;
}
```

```
public class Main {
    public static void main(String[] args) {
        String[] strings = {"das,ist","ein,test,!"};
        Container c = new Container(strings);
        c.print();}
}
```

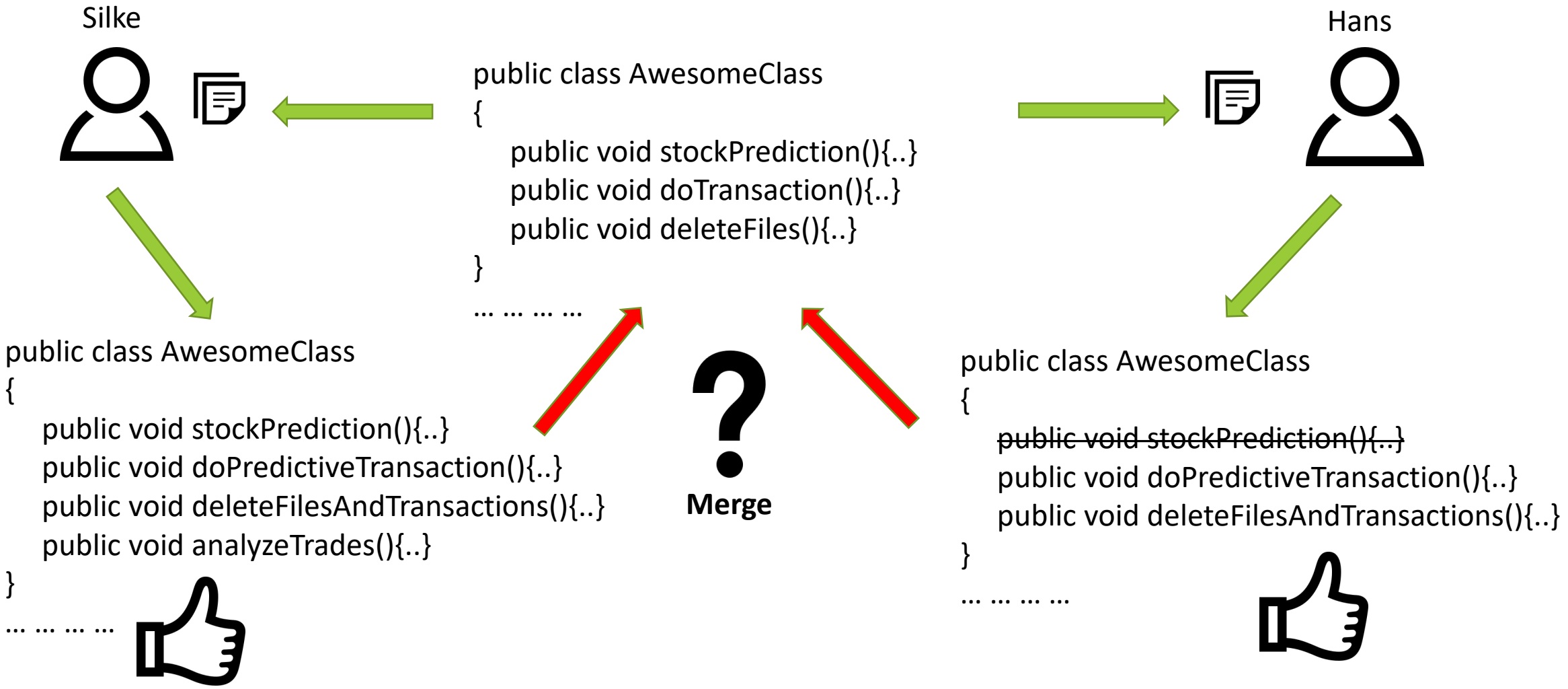
II. Versionsmanagement

Problem:

- Viele Entwickler arbeiten parallel an einem Projekt
- Häufig werden von diesen Entwicklern gemeinsam genutzte Komponenten modifiziert
- Komplexe Modifikationen erfordern Dokumentation (Vermeidung des menschlichen Single-Point-of-Failure)
- Feingranulare Dokumentation hilfreicher als nur grobe Dokumentation

Idee: Jeder Entwickler hat eine lokale Kopie der Codebasis auf welcher er arbeitet!

II. Versionsmanagement



II. Versionsmanagement

Funktional „ok“

```
public class AwesomeClass
{
    //2.8.2018 (Hans): added new algorithm
    //01.9.2018 (Lars): removed some vars
    public void doPredictiveTransaction(){..}
    //08.11.2018 (Peter): why do we use this?
    public void deleteFilesAndTransactions(){..}
}
… … … …
```

- Die **Wartbarkeit** der Software leidet mit der **Qualität des Quellcodes**:
  - Mangelhafte Dokumentation
  - Keine Verfolgung von Änderungen
  - Rollback von Änderungen nicht möglich (mit atomaren Deltas)

- Modularer Aufbau
- Testing
- Design Patterns
- Code Styles

Später mehr dazu

II. Versionsmanagement

Funktional „ok“

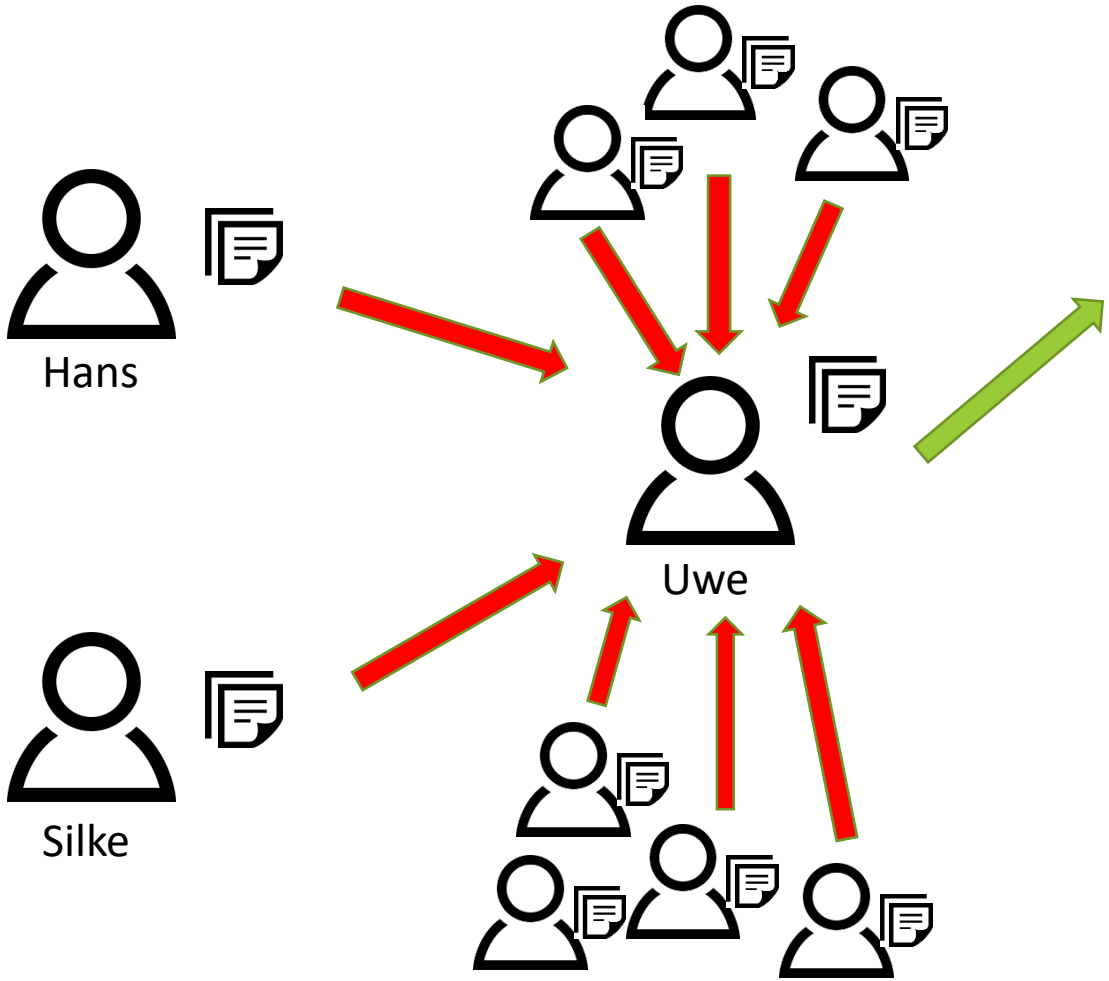
```
public class AwesomeClass
{
    public int doPredictiveTransaction(){return 1;}
    public void doXForYThingies(){..}
}
… … … …
```

- Die **Wartbarkeit** der Software leidet mit der **Qualität des Quellcodes**:
  - Mangelhafte Dokumentation
  - Keine Verfolgung von Änderungen
  - Rollback von Änderungen nicht möglich (mit atomaren Deltas)

- Modularer Aufbau
- Testing
- Design Patterns
- Code Styles

Später mehr dazu

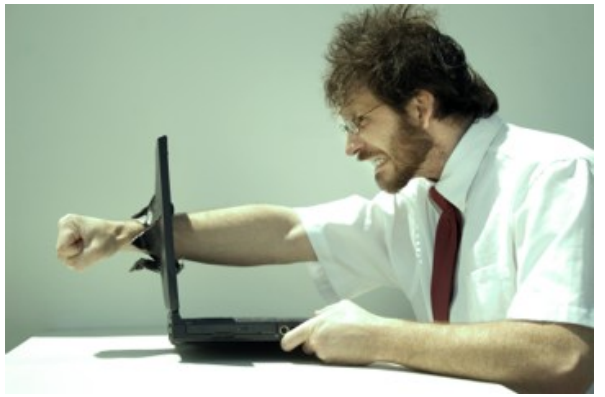
II. Versionsmanagement




```
public class AwesomeClass
{
    public void doPredictiveTransaction(){..}
    public void deleteFiles(){..}
    public void analyzeTrades(){..}
}
… … … …
```

Probleme:

- menschlicher Single-Point-of-Failure „Uwe“
- Domänenübergreifendes Wissen bei Uwe nötig
- Workload bei vielen Merges schwer zu bewältigen



https://imgflip.com/memetemplate/61706815/Angry-At-Computer

II. Versionsmanagement

Entwicklung = Funktionalität + Robustheit + Wartbarkeit

Lösungsversuche:

1. Jeder **Entwickler merged seine lokale Kopie** am Ende seines Arbeitstages **selber** in die Codebasis **ohne andere Funktionen** der Software **zu beschädigen**

3. Jeder **Entwickler arbeitet** auf seiner lokalen Kopie **bis** sein Feature / Fix **fertig** ist und **merged** erst dann **selbst** in die Codebasis

Bemerkungen:

- In einer idealen Welt mit perfekten Entwicklern und perfekter Hardware funktionieren beide Ansätze wunderbar.

II. Versionsmanagement

Lösungsversuche:

1. Jeder **Entwickler merged seine lokale Kopie** am Ende seines Arbeitstages **selber** in die Codebasis **ohne andere Funktionen** der Software **zu beschädigen**

Bemerkungen:

- Zeitdruck während der täglichen Entwicklung durch Merge-Pflicht
- Auch erfahrene Entwickler machen Fehler; **Zeitdruck** ⇔ **Komplexität** ⇔ **neue Technologiestacks**
- Auch mit Testabdeckung und automatisierten Analysen kann Software unbemerkt Fehler entwickeln (für welche es noch keine Tests gab)
- Eine defekte Codebasis muss repariert werden, hierfür sind in der Regel neben dem verantwortlichen Entwickler noch weitere Personen nötig, welche gemeinsam das nötige Wissen besitzen.

II. Versionsmanagement

Lösungsversuche:

2. Jeder **Entwickler arbeitet** auf seiner lokalen Kopie **bis** sein Feature / Fix **fertig** ist und **merged** erst dann **selbst** in die Codebasis.

Bemerkungen:

✓ Weniger Zeitdruck während der täglichen Entwicklung.

✓ Codebasis zeitlich weniger variabel

- Codebasis wächst mit der Zeit und ändert ihre Struktur, beim Mergen muss der eigene Code refaktorisiert werden. => **Unnötige Verzögerung / Kosten / Fehlerquellen / Zeitdruck**

- … Auch erfahrene Entwickler machen Fehler; **Zeitdruck** ⇔ **Komplexität** ⇔ **neue Technologiestacks**

- Auch mit Testabdeckung und automatisierten Analysen kann Software unbemerkt Fehler entwickeln (für welche es noch keine Tests gab)

II. Versionsmanagement

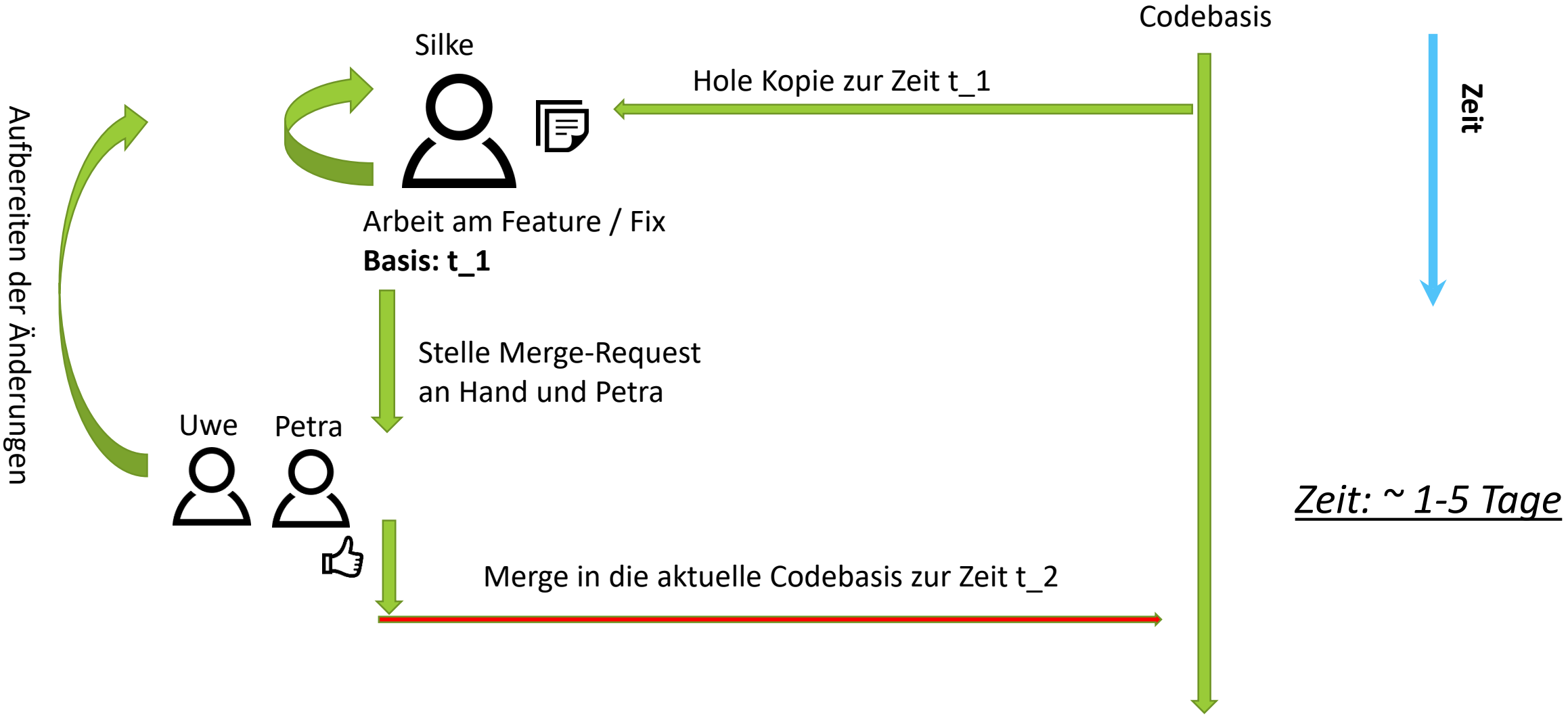
Lösungsversuche (ein letztes Mal):

3. Jeder **Entwickler arbeitet** auf seiner lokalen Kopie **bis** sein Feature / Fix **fertig** ist und **merged** erst dann **selbst** in die Codebasis, wird jedoch von 2 erfahrenen Entwicklern unterstützt. Hierbei werden die **Aufgaben zuvor feiner granularisiert**.

Bemerkungen:

- ✓ Weniger Zeitdruck während der täglichen Entwicklung.
- ✓ Codebasis zeitlich weniger variabel (kleine Änderungen unter allen Entwicklern)
- ✓ Änderungen in der Codebasis werden besser verteilt.
- ✓ 3 Entwickler machen durch gemeinsame Arbeit an kleinen Arbeitspaketen statistisch weniger Fehler.
- ***Auch mit Testabdeckung und automatisierten Analysen kann Software unbemerkt Fehler entwickeln (für welche es noch keine Tests gab)***

II. Versionsmanagement



II. Versionsmanagement

- Initial entwickelt von Linux Torvalds (2005)
- Verteiltes Versionsmanagement
- Geschrieben in C, Shell, Perl, TCL, Python
- De facto Standard in der Industrie

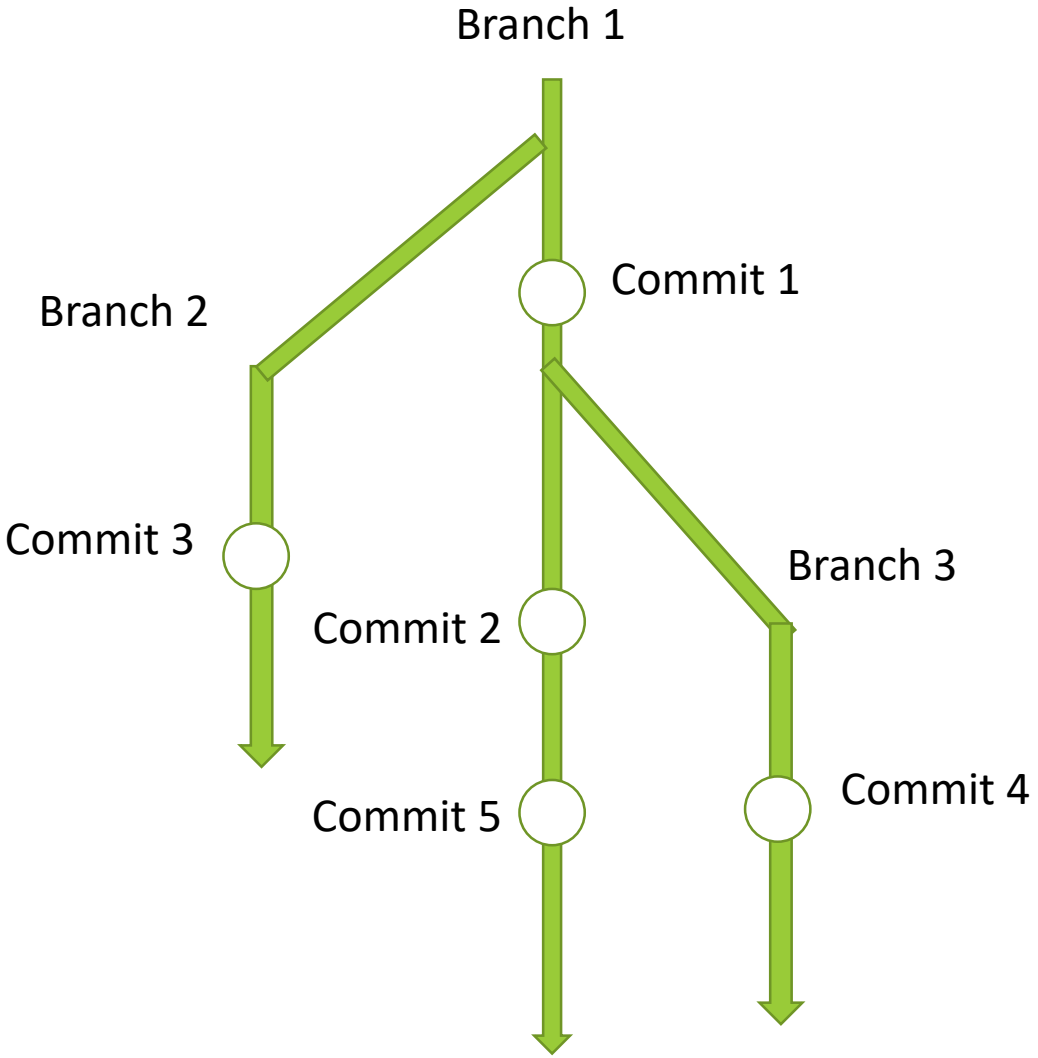
- **Trivia**: Microsoft verwaltet Sourcecode (~ 100GB) mit Git, entwickelte eigenes Dateisystem „Virtual Filesystem for Git“ dafür (OpenSource ☺)

https://upload.wikimedia.org/wikipedia/commons/thumb/6/69/Linus_Torvalds.jpeg/220px-Linus_Torvalds.jpeg

“I may be a huge computer nerd, but even so I don't think education should be about computers. Not as a subject, and not as a classroom resource either.”
Linux Torvalds, 2014
https://www.itwire.com/business-it-news/open-source/65402-torvalds-says-he-has-no-strong-opinions-on-systemd

II. Versionsmanagement

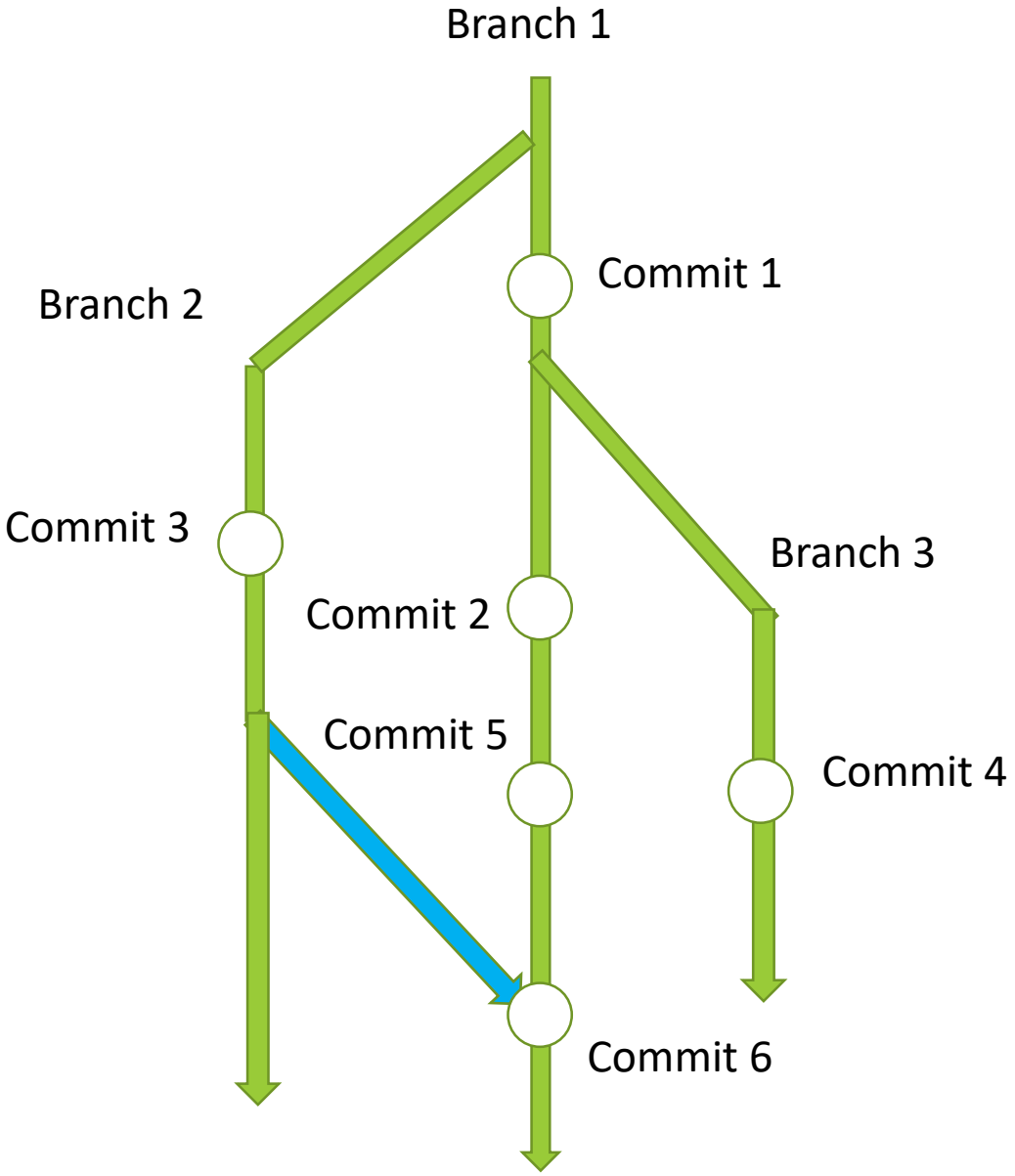
- Git arbeitet in Branches (Zweigen), ein **Branch** ist eine separate Codebasis unabhängig von ihrem Ursprung. Der erste Branch heißt **Master**.
- Änderungen an Dateien werden in Form von **Commits** in den Branch gepflegt, dazu ist eine kurze Commit-Nachricht Pflicht.
- Ein Commit kann beliebig viele Dateien enthalten.
- Git speichert keine Deltas, d.h. keine relativen Änderungen zur vorherigen Versionen. Wird eine Datei Datei geändert und Commited so wird die gesamte Datei als Kopie abgelegt



II. Versionsmanagement

- Zwei Branches können gemerged, d.h. ineinander überführt werden.
- Bei gleichen Dateien in beiden Branches kann git die Änderungen häufig erkennen und korrekt ineinander überführen.
- Die Branches können nach einem **Merge** weiterhin koexistieren.

Frage: Was passiert wenn Branch 2 nach einiger Zeit nochmals in Branch 1 gemerged wird? Probleme?

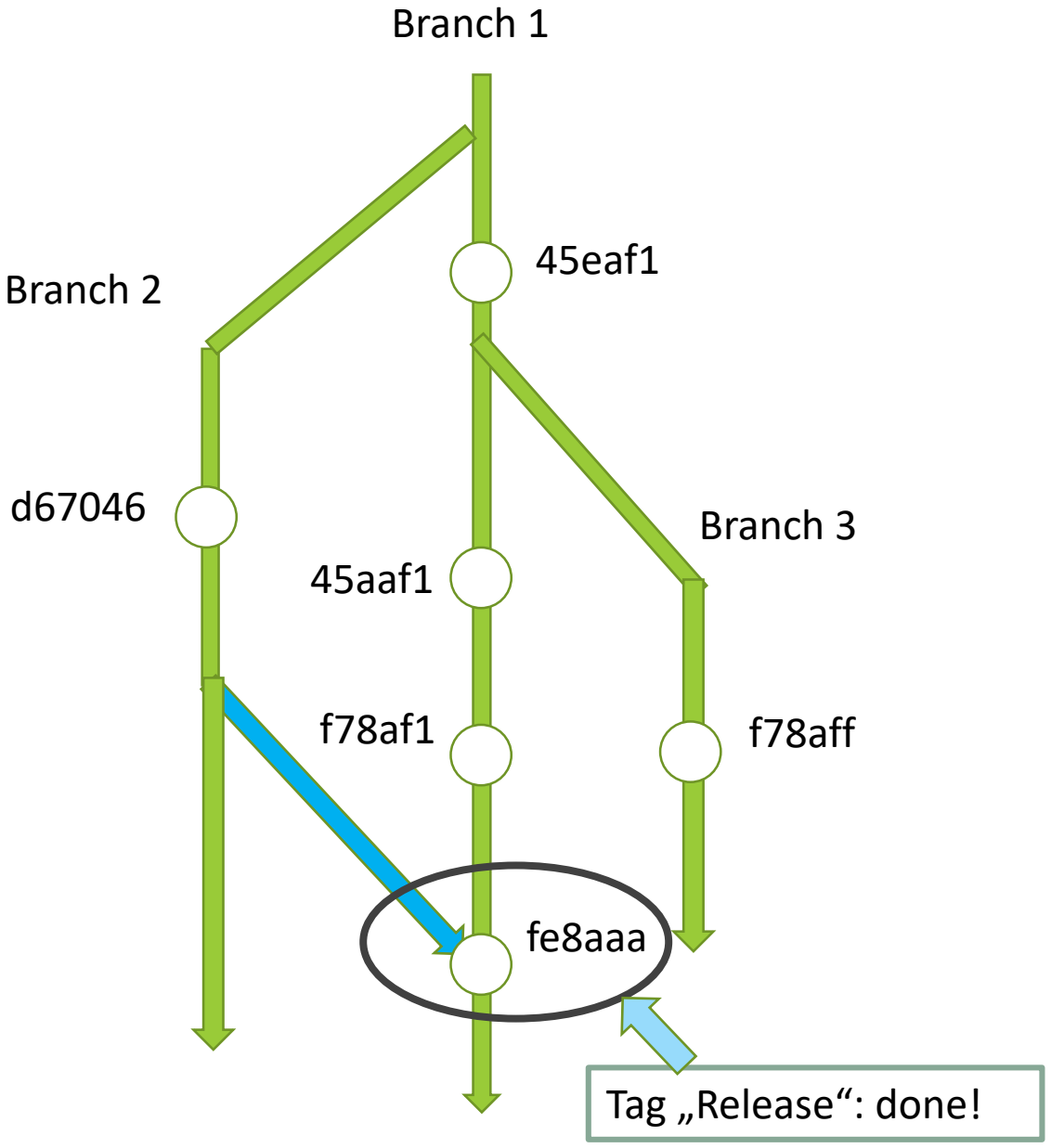


II. Versionsmanagement

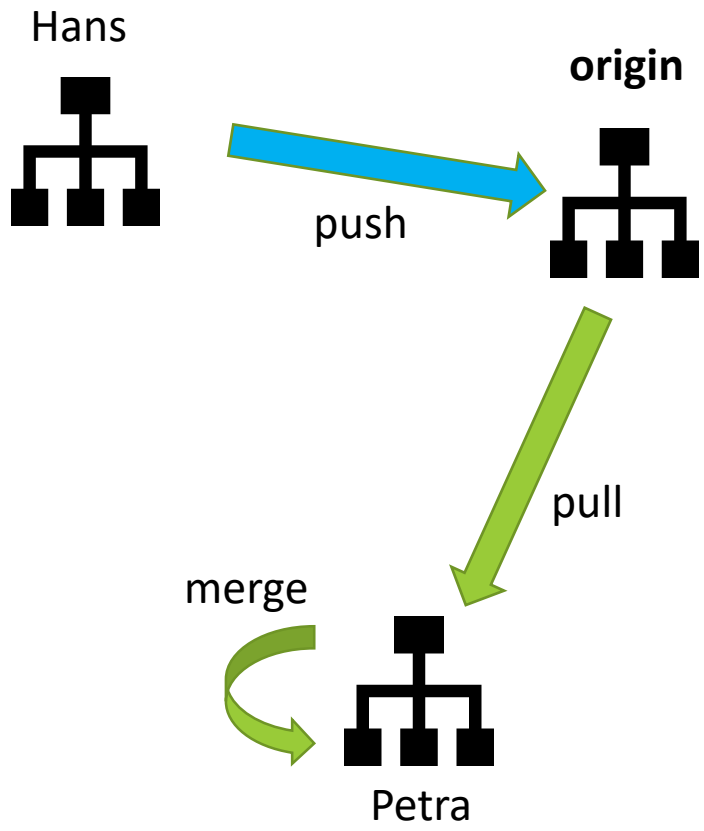
- Jeder Commit erhält einen eindeutigen SHA-1 Hash als Namen / Identifier
- git erlaubt es Commit zu Taggen, ein **Tag** ist ein Verweis (mit leserlichen Namen) samt kurzer Nachricht auf einen Tag
- Da SHA-1 Hashsummen 40 Zeichen enthalten, werden sie bei git gekürzt ***angezeigt***.

d670460b4b4aece5915caf5c68d12f560a9fe3e4

d67046

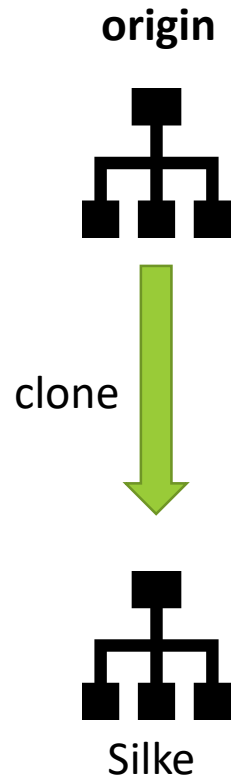


II. Versionsmanagement



- Jeder Entwickler kann lokal eine Kopie aller Branches haben.
- Alle Informationen der Git Branches (inklusive aller Dateien, aller Commits, …) liegen in einem Ordner namens „.git“. Dies ist das **Repository**.
- Dieser Ordner liegt üblicherweise im gleichen Verzeichnis wie die Projektdateien.
- Es besteht die Möglichkeit die Dateien auf einen zentralen Server (**origin**) zu **pushen**.
- Andere Entwickler können diese Dateien **pullen** und ihre lokale Kopie aktualisieren. Hierbei werden Änderungen mit der lokalen Kopie gemerged.

II. Versionsmanagement



- Falls ein Entwickler noch keine lokale Kopie eines Repositorys vom origin hat so muss er dieses zuvor **clonen**.

- Falls für lange Zeit kein pull auf einem lokalen Repository ausgeführt wurde und lokal keine Änderungen seit dem letzten push gemacht wurden, ist es häufig einfacher die lokale Kopie zu löschen und origin zu clonen.
  Warum?

- „Origin-Anbieter“ sind z.B. Gitlab oder Github.

- Diese Plattformen bieten weit mehr als einfache Origin-Funktionalität:
  - Wiki
  - Merge-Forum
  - Email-Benachrichtigunen
  - Build-Services

II. Versionsmanagement

- Je nach zu entwickelnder Software wird git unterschiedlich bezüglich der Branches genutzt.
- Häufig existieren 2+n Branches: Master, Development und verschiedene Fix/Feature Branches.
- Im Master befindet sich Sourcecode der veröffentlichten Software
- Im Development liegt der Code mit den getesteten neusten Features
- In den übrigen Zweigen werden Fehler behoben oder Features entwickelt und anschliessend in den Master gemerged. Die Branches werden danach gelöscht.
- Für ein Release wird der Development in den Master gemerged.

II. Versionsmanagement

- Erstellen eines neuen Repositorys in einem Ordner mit einer Java Datei

```
[darius@vm1 temp]# cat Main.java
public class Main
{
   public static void main(String [] args)
   {
      System.out.println("Hello");
   }
}
```

```
[darius@vm1 temp]# git init
Leeres Git-Repository in /home/darius/temp/.git/ initialisiert
```

II. Versionsmanagement

- Bevor eine neue / geänderte Datei commited werden kann muss sie **staged** werden
- Nur Dateien auf der Stage werden von git überhaupt berücksichtigt.
- Sind Dateien auf der Stage nicht commited so ist ein wechsel in einen anderen Branch des lokalen Repositorys nicht möglich.

```
[darius@vm1 temp]# git status
Auf Branch master

Noch keine Commits

Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum
Commit vorzumerken)

        Main.java

nichts zum Commit vorgemerkt, aber es gibt unversionierte
Dateien
```

II. Versionsmanagement


```
[darius@vm1 temp]# git status
Auf Branch master

Noch keine Commits

Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum
Commit vorzumerken)

        Main.java

nichts zum Commit vorgemerkt, aber es gibt unversionierte
Dateien
```

```
[darius@vm1 temp]# git add Main.java
```

II. Versionsmanagement


```
[darius@vm1 temp]# git status
Auf Branch master

Noch keine Commits

zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git rm --cached <Datei>..." zum Entfernen aus
der Staging-Area)

        neue Datei:     Main.java
```

```
[darius@vm1 temp]# git commit -m "first commit"
[master (Basis-Commit) 9e0dfec] first commit
 1 file changed, 7 insertions(+)
 create mode 100644 Main.java
```

II. Versionsmanagement


```
[darius@vm1 temp]# git branch feat/ticket-4
```

```
[darius@vm1 temp]# git checkout feat/ticket-4
Zu Branch 'feat/ticket-4' gewechselt
```

```
[darius@vm1 temp]# cat Main.java
public class Main
{
   public static void main(String [] args)
   {
      System.out.println("Hello World");
      //git should merge that without issues
   }
}
```

II. Versionsmanagement


```
[darius@vm1 temp]# git add Main.java
[darius@vm1 temp]# git commit -m "extend message"
[feat/ticket-4 8cfceed] extend message
 1 file changed, 2 insertions(+), 1 deletion(-)
[darius@vm1 temp]# git checkout master
Zu Branch 'master' gewechselt
```

```
[darius@vm1 temp]# git merge feat/ticket-4
Aktualisiere 9e0dfec..8cfceed
Fast-forward
 Main.java | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

I. Versionsmanagement


```
[darius@vm1 temp]# git log
commit 8cfceed58b8eab2e15568c2a6b00cb4629d3d5e1
(HEAD -> master, feat/ticket-4)
Author: d <d@d.d>
Date:   Fri Aug 3 00:16:23 2018 +0200

    extend message

commit 9e0dfec411a2f545e3d9b6d5953cd8d801019935
Author: d <d@d.d>
Date:   Fri Aug 3 00:10:13 2018 +0200

    first commit
```

II. Versionsmanagement


```
[darius@vm1 temp]# git log
commit 8cfceed58b8eab2e15568c2a6b00cb4629d3d5e1
(HEAD -> master, feat/ticket-4)
Author: d <d@d.d>
Date:   Fri Aug 3 00:16:23 2018 +0200

    extend message

commit 9e0dfec411a2f545e3d9b6d5953cd8d801019935
Author: d <d@d.d>
Date:   Fri Aug 3 00:10:13 2018 +0200

    first commit
```

```
[darius@vm1 temp]# git branch -d feat/ticket-4
Branch feat/ticket-4 entfernt (war 8cfceed).
```

II. Versionsmanagement

Nun kann das Repository zu einem origin gepushed werden

```
[darius@vm1 temp]# git push origin master
```

Bemerkung:

- Ein origin kann den Push ablehnen wenn er über aktualisierte Daten verfügt, welche lokal noch nicht verfügbar sind
- Hier müssen zunächst über einen pull jene Änderungen des origins gemerged werden.
- Es ist möglich zu beliebig vielen origins zu pushen, jedoch wird immer nur der letzte als aktiver origin im lokalen Repository vermerkt.