Programmierkurs
1.12.2017

Ole Klein Steffen Müthing

Interdisciplinary Center for Scientific Computing, Heidelberg University

November 17, 2017

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?

Warum Versionskontrolle?

Wer hat schon einmal. . .

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?
- Dropbox benutzt, um Dateien auf mehreren Computern zu synchronisieren?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?
- Dropbox benutzt, um Dateien auf mehreren Computern zu synchronisieren?
- Dropbox mit mehreren Leuten benutzt?

Warum Versionskontrolle?

Wer hat schon einmal...

- Berge von Kopien einer Datei angelegt, um zu einer alten Version zurückgehen zu können?
- Trotzdem die eine wichtige Version verloren?
- von Hand mehrere dieser Dateien verglichen, um herauszufinden, was geändert wurde?
- Dateien per USB-Stick von Rechner zu Rechner transportiert?
- Dropbox benutzt, um Dateien auf mehreren Computern zu synchronisieren?
- Dropbox mit mehreren Leuten benutzt?
- Dabei Dateien verloren, weil zwei Leute gleichzeitig gespeichert haben?

git - Distributed Version Control System

- Kommandozeilenprogramm zum Verwalten von (ursprünglich) text-basierten Dateien
- Entwickelt 2005 von Linus Torvalds für die Quellen des Linux-Kernels
- Extrem schnell
- Verwaltet einige der grössten Codebasen weltweit:
  - Linux-Kernel ($>$ 25 Mio. Zeilen, $>$ 10.000 Commits / Version)
  - Windows (300 GB, 3,5 Mio. Dateien)
- Kostenloses Repository-Hosting, z.B. GitHub, Bitbucket, GitLab
- Unterstützung für Bilder etc. mit git-lfs
- Inzwischen de-facto Industriestandard

Glossar

Repository Datenbank mit allen Informationen über Dateiversionen in einem Projekt, liegt im versteckten Verzeichnis `.git` im obersten Projektverzeichnis.

Commit Globaler Schnappschuss aller Projektdateien mit einer Beschreibung der Änderungen zur vorherigen Version.

Branch Eine Abfolge von Commits, die einen Entwicklungszweig abbilden. Ein Repository kann mehrere Branches enthalten. Der Standard-Branch heißt `master`.

Tag Ein dauerhafter Name für einen Commit, z.B. für ein Release.

Globale Konfiguration

Zuerst sollten wir zwei Dinge einrichten:

- git möchte wissen, wer wir sind:

```
git config --global user.name "Steffen Müthing"
git config --global user.email "steffen.muething@iwr.uni-heidelberg
```

- Für Git-Status in der Kommandozeile folgende Zeilen in `.bash_profile` einfügen:

```
export PS1='[\u@\h \W$(__git_ps1 " (%s)")]\$ '
export GIT_PS1_SHOWDIRTYSTATE=1
```

Getting Started I

Zum Anlegen des Repositories ins oberste Projektverzeichnis wechseln und dann:

```
[git-tutorial]$ git init
Initialized empty Git repository in /Users/smuething/tmp/git-tutorial/.
[git-tutorial (master #)]$
```

Damit existiert das Repository, aber es gibt noch keinen Commit:

```
[git-tutorial (master #)]$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        input.cc
        input.hh

nothing added to commit but untracked files present (use "git add" to t
[git-tutorial (master #)]$
```

Änderungen hinzufügen

git speichert nur Änderungen, von denen wir im explizit erzählen:

```
[git-tutorial (master #)]$ git add input.cc
[git-tutorial (master +)]$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   input.cc

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        input.hh

[git-tutorial (master +)]$
```

Änderungen committen

Jetzt können wir auch noch input.hh hinzufügen und einen Commit erzeugen:

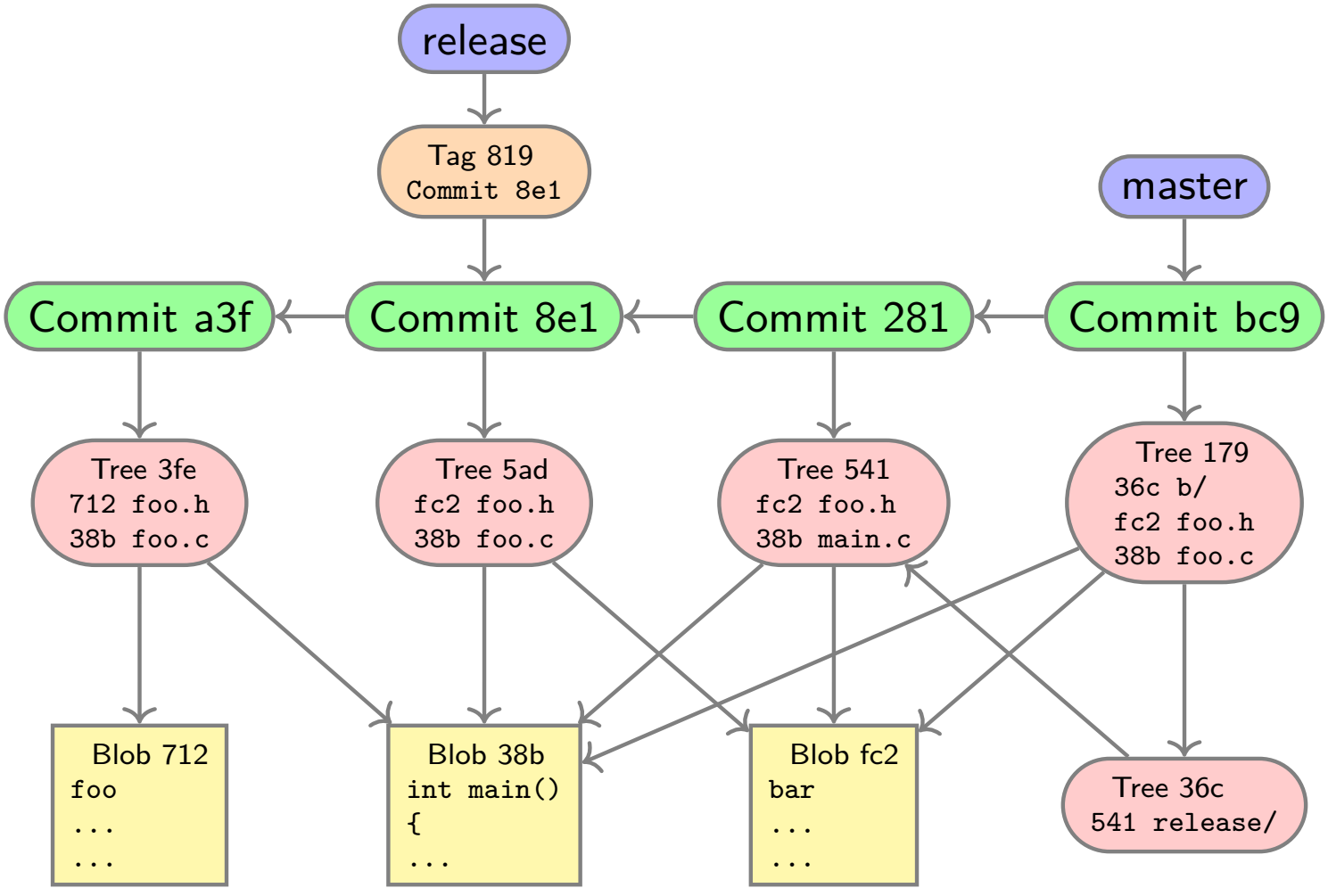
```
[git-tutorial (master +)]$ git add input.hh
[git-tutorial (master +)]$ git commit
[master (root-commit) 1bb9ef8] Added input files
 2 files changed, 25 insertions(+)
 create mode 100644 input.cc
 create mode 100644 input.hh
[git-tutorial (master)]$ git status
On branch master
nothing to commit, working tree clean
[smuething@muh109-191 git-tutorial (master)]$ git log
commit 1bb9ef87e4c235cc72e07009fc48cefb38df6154 (HEAD -> master)
Author: Steffen Müthing <muething@dune-project.org>
Date:   Fri Dec 1 10:55:17 2017 +0100

    Added input files
[smuething@muh109-191 git-tutorial (master)]$
```

Commits

- Commits enthalten
  - einen Snapshot aller Dateien,
  - den Erstellungszeitpunkt,
  - Namen und Inhalt vom Autor der Änderungen und von der Person, die den Commit erstellt hat,
  - eine Beschreibung der Änderungen (Changelog),
  - Eine Liste mit Verweisen auf die Eltern-Commits.
- Commits werden durch einen Hash (eine Prüsumme) ihres Inhalts identifiziert, z.B. `1bb9ef87e4c235cc72e07009fc48cefb38df6154`.
- Commit-Hashes können abgekürzt werden, solange sie eindeutig sind.
- Commits können nicht verändert werden:
  anderer Inhalt $\Rightarrow$ anderer Hash.

Hashes all the way down



Weiterarbeiten

Wir verändern die Datei `input.hh` und speichern die veränderte Datei:

```
[git-tutorial (master *)]$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working direct

        modified:   input.hh

no changes added to commit (use "git add" and/or "git commit -a")
[git-tutorial (master *)]$
```

Zur Erinnerung: Änderungen müssen wir git immer mitteilen!

```
[git-tutorial (master *)]$ git add input.hh
[git-tutorial (master +)]$ git commit
[master bab868d] Added comment
 1 file changed, 2 insertions(+)
[git-tutorial (master)]$
```

Unterschiede anzeigen

Anzeigen, was der aktuelle Commit verändert hat:

```
[git-tutorial (master)]$ git show master
commit bab868dd7e345f1b660157a8bd4519cad175733d (HEAD -> master)
Author: Steffen Müthing <muething@dune-project.org>
Date:   Fri Dec 1 11:06:27 2017 +0100

    Added comment

diff --git a/input.hh b/input.hh
index 3b74a4a..3bef25c 100644
--- a/input.hh
+++ b/input.hh
@@ -4,6 +4,8 @@
 #include <string>
 #include <istream>

+// Reads from input until EOF and returns the
+// result as a string.
 std::string read_stream(std::istream& input);

 #endif // INPUT_HH
[git-tutorial (master)]$
```

Branches

- Ein Branch ist ein Entwicklungszweig innerhalb eines Repositories.
- Repositories können beliebig viele Branches enthalten.
- `git status` sagt einem, auf welchem Branch man sich befindet.
- Ein Branch zeigt auf einen Commit.
- Wenn man einen neuen Commit erstellt, speichert er den aktuellen Commit als Vater und der Branch zeigt danach auf den neuen Commit.

Branches: Befehle

- Branch `playground` erstellen:

  ```
  git branch playground
  ```

- Branches auflisten:

  ```
  [git-tutorial (master)]$ git branch
  * master
    playground
  [git-tutorial (master)]$
  ```

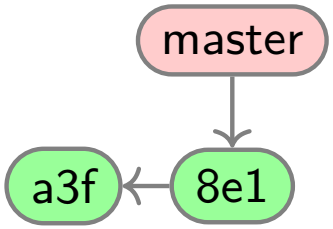
- Branch wechseln:

  ```
  [git-tutorial (master)]$ git checkout playground
  Switched to branch 'playground'
  [git-tutorial (playground)]$
  ```

- Änderungen von anderem Branch importieren (merge):

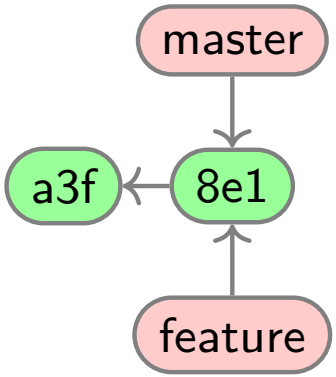
  ```
  git merge other-branch
  ```

Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

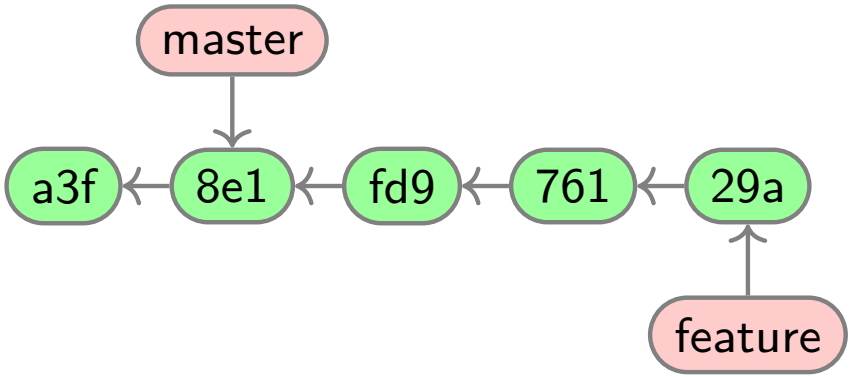
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

- Branch anlegen

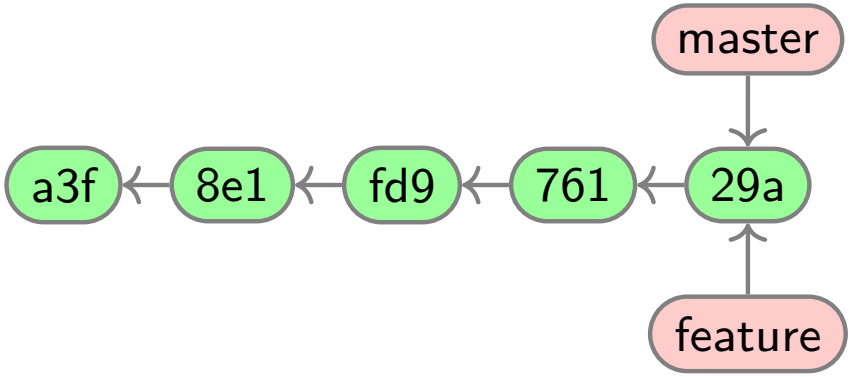
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

- Branch anlegen
- Auf Branch arbeiten

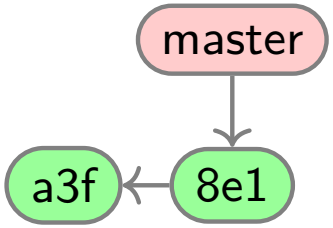
Merging



Der simple Fall: Es gibt nur eine Commit-Reihe

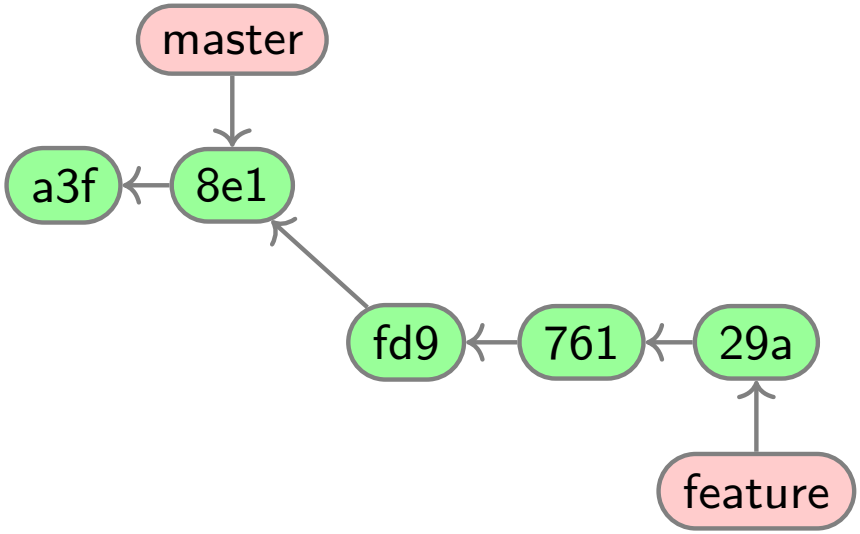
- Branch anlegen
- Auf Branch arbeiten
- Keine neuen Commits in master $\Rightarrow$ fast-forward

Merging II



Der realistische Fall: Gleichzeitige Änderungen

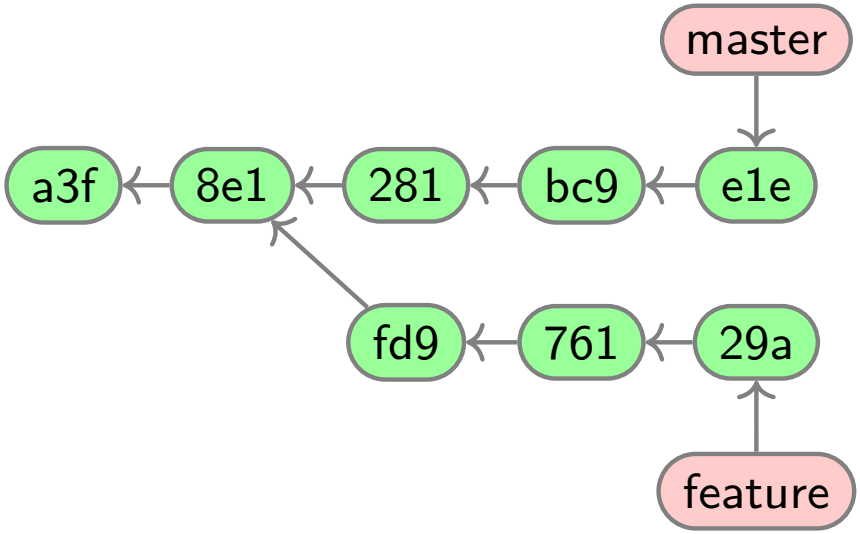
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten

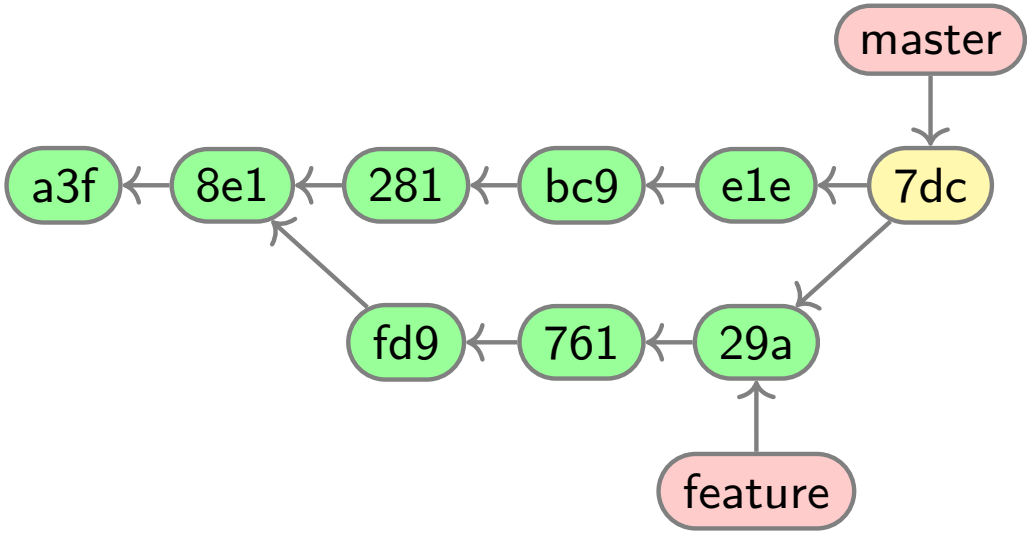
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten
- Master wird verändert

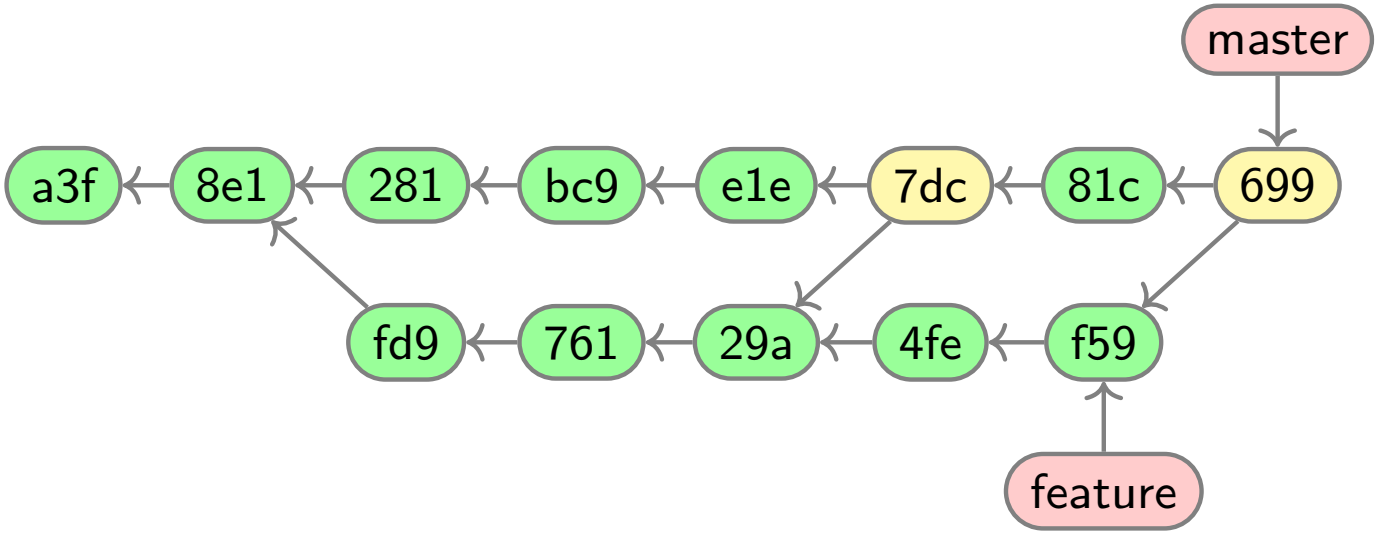
Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten
- Master wird verändert
- Änderungen nach master mergen
  - Erzeugt Merge-Commit
  - Bei Konflikten muss manuell nachgeholfen werden

Merging II



Der realistische Fall: Gleichzeitige Änderungen

- Branch erstellen und darauf arbeiten
- Master wird verändert
- Änderungen nach master mergen
  - Erzeugt Merge-Commit
  - Bei Konflikten muss manuell nachgeholfen werden

Mehrere Repositories I

- git kann Änderungen zwischen Repositories synchronisieren.
- Zusätzliche Repositories heißen remote.
- Branches aus einem Remote-Repository bekommen den Namen des Repositories vorangestellt.
- Man kann ein Remote-Repository klonen, um den Inhalt zu bekommen:

```
[folder]$ git clone https://gitlab.dune-project.org/core/dune-commo
Cloning into 'dune-common'...
remote: Counting objects: 54485, done.
remote: Compressing objects: 100% (13788/13788), done.
remote: Total 54485 (delta 40840), reused 54059 (delta 40531)
Receiving objects: 100% (54485/54485), 12.83 MiB | 19.15 MiB/s, don
Resolving deltas: 100% (40840/40840), done.
[folder]$
```

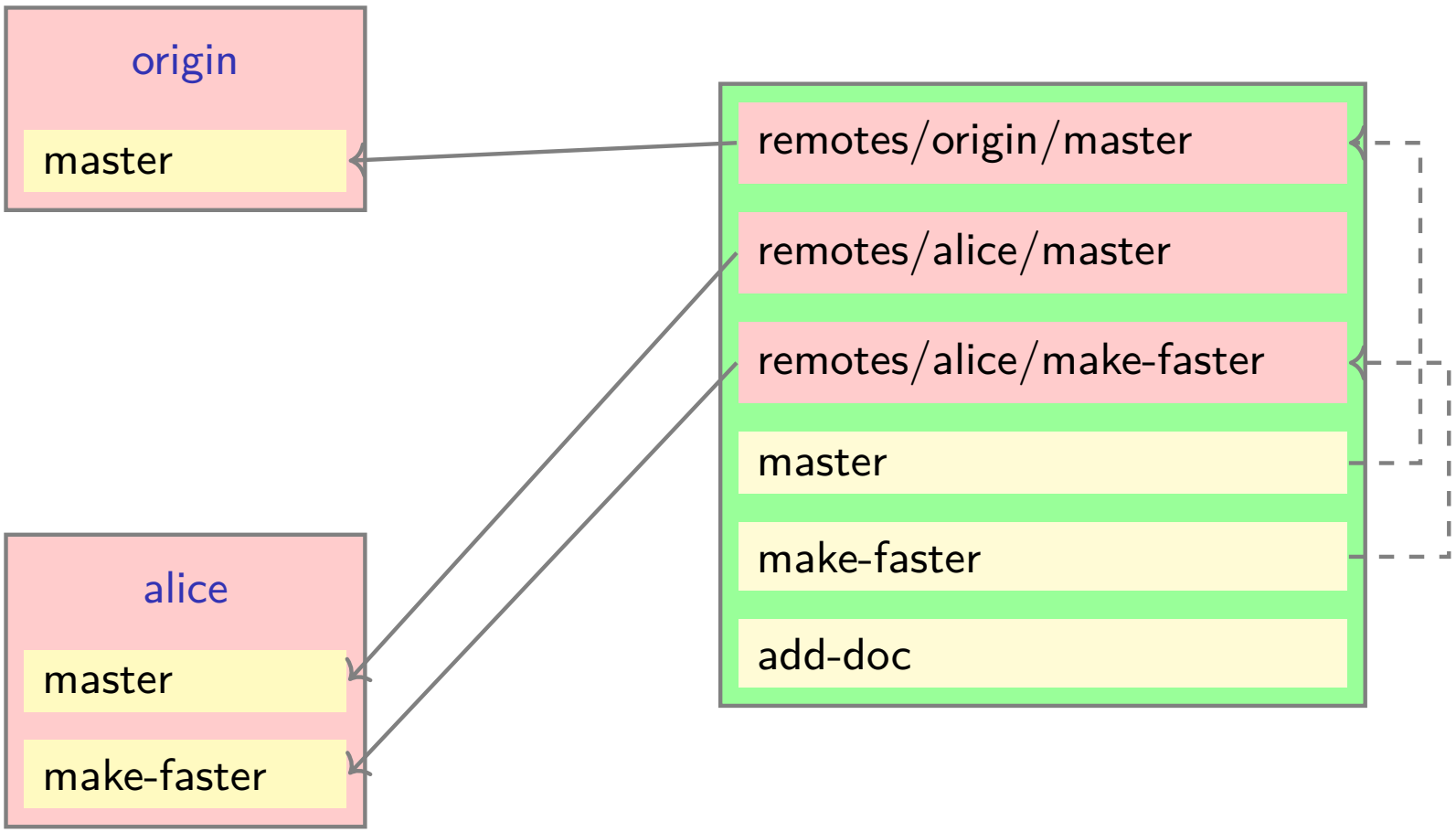
Mehrere Repositories II

- Man kann Branches von Remote-Repositories nicht direkt auschecken.
- Git legt beim ersten checkout einen tracking branch an:

```
[dune-common (master)]$ git checkout releases/2.6
Branch 'releases/2.6' set up to track remote branch 'releases/2.6'
Switched to a new branch 'releases/2.6'
[dune-common (releases/2.6)]$
```

- Neue Änderungen können mit `git pull` heruntergeladen und gemergt werden.
- Eigene Änderungen können mit `git push` hochgeladen werden.
  - Wenn jemand anderes vorher Änderungen hochgeladen hat: Fehler!
  - Lösung: Änderungen erst mit `git pull` herunterladen und integrieren, dann nochmal pushen.

Mehrere Repositories - Schema



Demo

Git - Wichtige Befehle

init	Leeres Repository anlegen
clone	Bestehendes Repository klonen
add	Änderungen für Commit registrieren
commit	Commit erstellen
log	Verlauf ansehen
diff	Änderungen ansehen
branch	Branches anlegen, auflisten, löschen
checkout	Branch wechseln
merge	Branches zusammenführen
pull	Neue Änderungen herunterladen
push	Neue Änderungen hochladen

`git help <Befehl> für Hilfe!`

Objektorientierte Programmierung

- Programme aus Objekten, die
  - einen internen Zustand haben (member variables)
  - Operationen ausführen können (member functions / Methoden)
- Jedes Objekt ist eine *Instanz* einer *Klasse*.
- Eine Klasse definiert das Verhalten all ihrer Instanzen.
- Wichtige Konzepte:
  - Kapselung
  - `const`ness
  - Komposition vs. Vererbung
  - Initialisierung und Cleanup

Kapselung

Klassen können die Sichtbarkeit von enthaltenen Variablen und Funktionen kontrollieren:

```
1  class Polygon {
2  // not visible outside Polygon
3  private:
4    std::vector<Point> _corners;
5  // only visible to Polygon and classes that inherit from it
6  protected:
7    const Point& corner(int i) const;
8  // visible to everyone
9  public:
10   double area() const;
11   void rotate(double angle);
12 };
```

- Die Standard-Sichtbarkeit in `class` ist `private`.
- Es ist sinnvoll, für private Member ein Namensschema einzuführen.

Kapselung - Richtlinien

- Wenn externer Zugriff auf private Variablen erforderlich ist: Accessor-Methoden

```
1  class Complex {
2    double _real, _imaginary;
3  public:
4    double real() const // or getReal()
5    {
6      return _real;
7    }
8
9    void setReal(double v)
10   {
11     _real = v;
12   }
13 ...};
```

- In Member-Funktionen direkt auf private Variablen / Funktionen zugreifen!

const und Klassen

```
1  const double x = 2.0;
2  std::cout << x << std::endl; // ok, x wird nur gelesen
3  x = x + 2; // Compile-Fehler
4
5  const Complex c(1.0,2.0);
6  std::cout << x.real() << std::endl; // ????
7  c.setReal(3.0); // darf nicht funktionieren
```

Methodenaufruf bei einer **const** Instanz

const und Klassen

```
1  const double x = 2.0;
2  std::cout << x << std::endl; // ok, x wird nur gelesen
3  x = x + 2; // Compile-Fehler
4
5  const Complex c(1.0,2.0);
6  std::cout << x.real() << std::endl; // ????
7  c.setReal(3.0); // darf nicht funktionieren
```

Methodenaufruf bei einer `const` Instanz

- Woher weiss der Compiler, dass die Methode die Instanz nicht verändert?
- Lösung: Funktionssignatur so verändern, dass die Instanz (und alle Member) in der Funktion `const` sind.

Initialisierung und Cleanup

- Objekte müssen vor Verwendung initialisiert werden (Speicher allokieren, Dateien öffnen etc.) und danach Ressourcen wieder freigeben.
- C++ macht hier strikte Garantien:
  - Für jedes Objekt wird ein Konstruktor aufgerufen, bevor der Programmierer Zugriff bekommt.
  - Das gilt auch für Objekte, die Member von anderen Objekten sind, und Basisklassen (Vererbung).
  - Für jedes Objekt, dessen Konstruktor erfolgreich beendet wurde, wird ein Destruktor aufgerufen, bevor das Objekt aufhört zu existieren.
- Ein Objekt hört auf zu existieren, wenn
  - Das Scope (-Paar) endet, in dem die Variable angelegt wurde (normale Variablen).
  - `delete` aufgerufen wird (Pointer).
- Strengere Garantien als viele andere Sprachen.

Konstruktor

```
1  class Triangle : public Shape {
2    Point _x1, _x2, _x3;
3  public:
4    Polygon(const Point& x1, const Point& x2, const Point& x3)
5      : Shape(), _x1(x1), _x2(x2), _x3(x3)
6    {}
7  };
```

- Konstruktoren sind Methoden, die genauso heissen wie die Klasse und keinen Rückgabewert haben.
- Es kann mehrere Konstruktoren mit unterschiedlichen Argumenten geben.
- Vor dem Body des Konstruktors kommt die constructor initializer list:
  - Liste von Konstruktor-Aufrufen für Basisklassen und Member-Variablen.
  - Wenn Basisklassen oder Variablen hier nicht aufgeführt werden, wird der Default-Konstruktor (ohne Argumente) aufgerufen.
  - Variablen immer hier initialisieren, nicht im Body!

Destruktor

```
1  class Pointer {
2    double* _p;
3  public:
4    Pointer(double v)
5      : _p(new double(v))
6    {}
7
8    ~Pointer()
9    {
10     delete _p;
11   }
12 };
```

- Destruktoren heissen wie die Klasse mit vorgestellter "~".
- Destruktoren haben nie Argumente $\Rightarrow$ es gibt nur einen pro Klasse.
- Cleanup-Aufgaben
  - Speicher freigeben
  - Dateien schliessen
  - Netzwerkverbindungen schliessen
  - ...

Ressourcenverwaltung

Programme müssen alle Ressourcen (Speicher etc.), die sie allokieren, auch wieder freigeben (sonst Bugs)!
Methoden:

Manuell Irgendwo `new` aufrufen und von Hand überlegen, wo `delete` erforderlich.

- aufwendig
- fehleranfällig

Garbage Collection Speicher wird speziell markiert, in periodischen Abständen wird im Hintergrund unbenutzter Speicher gesucht und freigegeben

- komfortabel
- kann zu Programm-Rucklern führen
- Funktioniert nicht für andere Ressourcen (Dateien etc.)

RAII Die C++-Lösung

Ressource Acquisition is Initialization (RAII)

C++ verwaltet Ressourcen mit dem RAII-Idiom:

- Klasse, die genau eine Ressource kapselt.
- Ressource wird im Konstruktor allokiert.
- Ressource wird im Destruktor freigegeben.
- C++ garantiert, dass der Destruktor aufgerufen wird, falls der Konstruktor erfolgreich beendet wurde.
- Funktioniert für beliebige Arten von Ressourcen.
- Der Programmierer muss die RAII-Klasse bewusst verwenden.
- Diverse Implementierungen in der standard library:
  - Speicher: `std::unique_ptr`, `std::shared_ptr`, ...
  - Dateien: `std::fstream`
  - Locks: `std::lock_guard`, ...

RAII: Beispiel

```
1  #include <fstream>
2  #include <iostream>
3  #include <string>
4
5  int main(int argc, char** argv)
6  {
7    {
8      std::ofstream outfile("test.txt");
9      outfile << "Hello, World" << std::endl;
10   } // file gets flushed and closed here
11
12   std::ifstream infile("test.txt");
13   std::string line;
14   std::getline(infile,line);
15   std::cout << line << std::endl;
16   return 0;
17 }
```

Default-Konstruktor

Der Default-Konstruktor ist der Konstruktor ohne Argumente:

```
1  class Empty {
2  public:
3    Empty()
4    {}
5  };
```

- Wenn eine Klasse keinen Konstruktor definiert, erzeugt der Compiler einen Default-Konstruktor.
- Ansonsten muss man ihn von Hand schreiben, wenn man ihn braucht.
- Der Compiler erzeugt auch keinen Default-Konstruktor wenn eine der Member-Variablen oder eine Basisklasse keinen Default-Konstruktor hat (entweder von Hand geschrieben oder default).