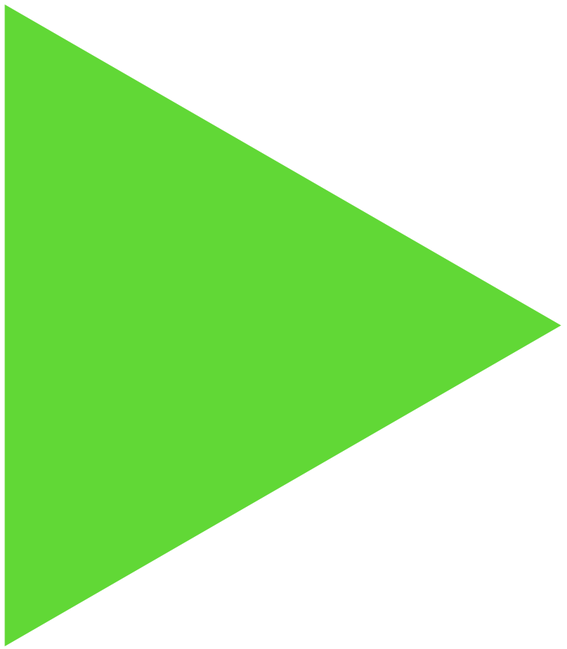
make, version control, git

Sven Köhler
Hasso-Plattner-Institut
2017-11-28

Wie übersetzt ihr bisher eure Programme?

- [ ] `$ cc ggt.c`
- [ ] `$ cc ggt.c -o ggt`
- [ ] `$ make ggt`
- [ ] <anders>

Sobald ein Projekt aus mehr als einer Datei besteht lohnt es sich die Übersetzung zu automatisieren.
Anforderung: Was sich nicht ändert, muss nicht neu übersetzt werden.



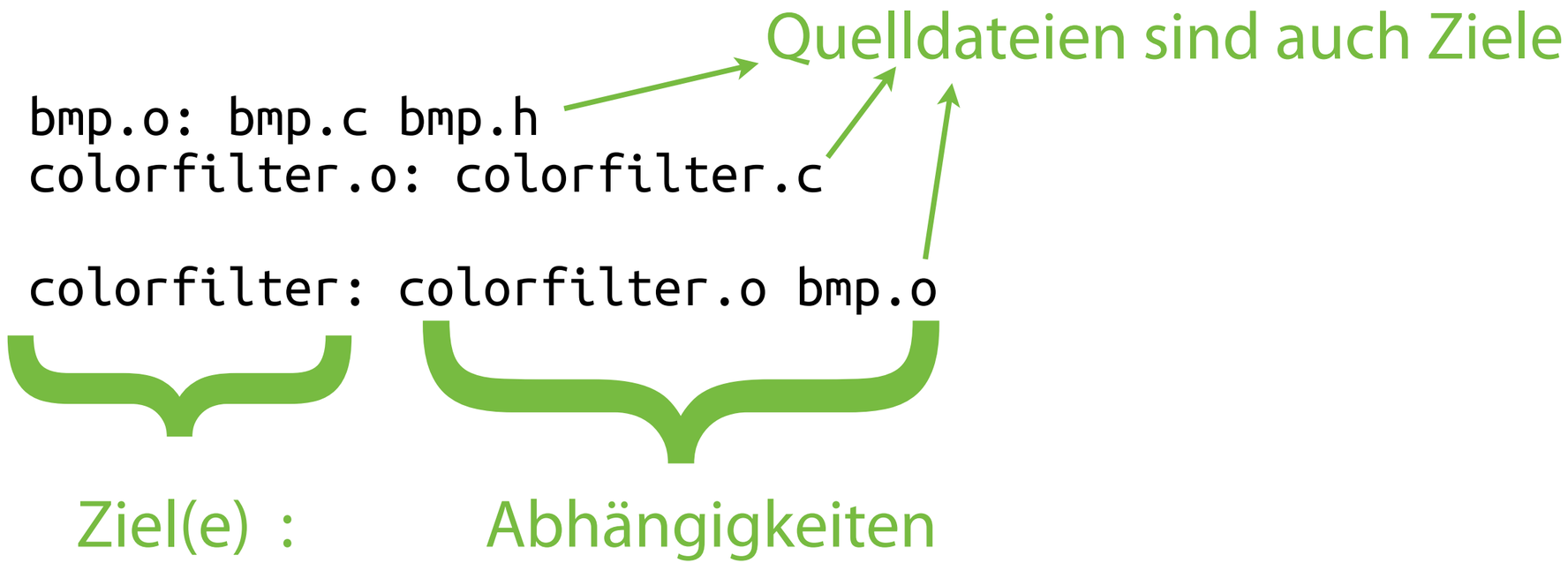
1

make

make ::

ist ein Technologieprogramm, dass z.B. Programme und Bibliotheken aus Quelldateien erstellt. Hierzu werden Befehle (**rules**) ausgeführt um (Zwischen-)Bedingungen (**targets, objects**) zu erfüllen.
Regeln und Ziele werden in einem **Makefile** notiert.

Ziele können von Dateien, oder anderen Zielen **abhängen**.
Make überprüft transitiv deren Erfüllung.



Dateiziele sind erfüllt, sobald **die Datei existiert.**
Ziele sind **veraltet** oder **unerfüllt** wenn ihr Zeitstempel älter ist als mindestens eine ihrer Abhängigkeiten.

```
$ make ziel
```

Führt die notwendigen Befehle aus um `ziel` zu erstellen, wenn es noch nicht vorhanden ist.

Wenn man make ohne ein zu bauendes Ziel ruft,
wird **das Erste Ziel** im Makefile erzeugt.
Übliche Praxis: Ein Standardziel "`all`" wird am Anfang notiert.

```
all: colorfilter

bmp.o: bmp.c bmp.h
colorfilter.o: colorfilter.c

colorfilter: colorfilter.o bmp.o
```

Es können Makros (Variablen) definiert werden:

oder Optionen für den Kompiler (hier z.B. Debuginfos) aktiviert werden:

```
CFLAGS = -g
OBJ = bmp.o colorfilter.o
colorfilter: $(OBJ)
```

Einige vordefinierte Variablen und Makros

CC	C-Kompilerbefehl
CFLAGS	C-Kompileroptionen
CXXFLAGS	C++-Kompileroptionen
LDFLAGS	Linkeroptionen
LOADLIBES	Einzubindende Bibliotheken
SHELL	Shell zur Verwendung

Wofür wird eine Shell gebraucht?

<30sec brainstorming>

Variablen und Regel können Kommandozeilenbefehle enthalten

```
SRC = $(shell find . -iname "*.c")
OBJ = $(SRC:%.c=%.o)

all: colorfilter # Falls nichts abgegeben wurde

colorfilter: $(OBJ)
clean:
	rm -f $(OBJ) colorfilter
```

Musterersetzung in Makro

Kommentar

Ein Kommando pro Zeile

beginnt mit TAB, mit \ über Zeilengrenzen verlängert

Advanced

Vordefinierte Variablen stehen bei Regeldefinition zur Verfügung

$@	Name des Ziels
$<	Die erste Abhängigkeit
$+	Alle Abhängigkeiten
$^	Alle Abhängigkeiten, ohne Dopplungen
$?	Alle ungültigen Abhängigkeiten

Beispiel:

```
test: test.c
    $(CC) $< -o $@

%.o: %.c                # Vordefiniert für alle C-Dateien (siehe make -p)
    $(CC) -Wall -g -c $< -o $@
```

[https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html]

Okay, das war ein Ritt.
Nehmen wir wieder ein einfaches Beispiel …

Ich hab folgende, einfache Regel:

```
colorfilter.o: colorfilter.c
```

Was passiert, wenn ich die eingebundene `bmp.h` ändere?

Nichts!

Reicht es, wenn ich einfach `bmp.h` in die Abhängigkeiten aufnehme?

```
colorfilter.o: colorfilter.c bmp.h
```

<20sec brainstorming>

Die Kompiler GCC und Clang haben eine praktische Option,
die alle Abhängigkeiten für eine Quelltextdatei ermittelt:

```
$ cc -MM colorfilter.c
colorfilter.o: colorfilter.c bmp.h
```

Und wie verwende ich das im Makefile?

<??sec brainstorming>

… okay, dass muss man wissen.

```
SRC = $(shell find . -iname "*.c")
OBJ = $(SRC:%.c=%.o)
DEP = $(SRC:%.c=%.d)

%.d: %.c
    $(CC) -MM $(CFLAGS) $< > $@

-include $(DEP)

colorfilter: $(OBJ)

clean:
    rm -f $(OBJ) $(DEP)
    rm -f colorfilter
```

Kann mir das jemand decodieren?

Was passiert wenn jemand eine Datei namens “clean” anlegt?

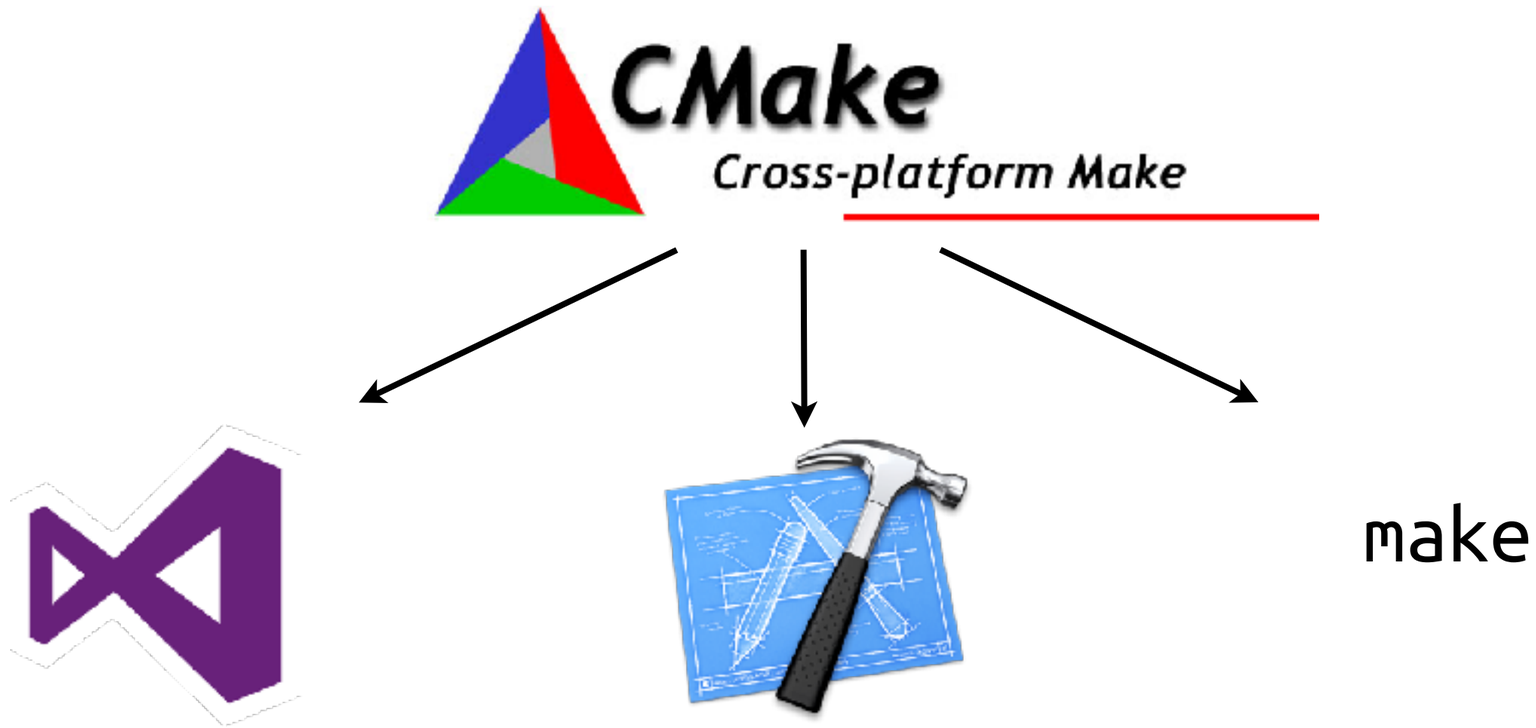
<20sec brainstorming>

```
SRC = $(shell find . -iname "*.c")
OBJ = $(SRC:%.c=%.o)
DEP = $(SRC:%.c=%.d)

%.d: %.c
    $(CC) -MM $(CFLAGS) $< > $@

-include $(DEP)

colorfilter: $(OBJ)

clean:
    rm -f $(OBJ) $(DEP)
    rm -f colorfilter

.PHONY: clean
```

Phony Ziele sind immer veraltet, wenden also **immer neu** gebaut.

[https://www.gnu.org/software/make/manual/html_node/Special-Targets.html]

Make ist von 1977 und hat inzwischen verschiedene Implementierungen und Nachfolger (Autotools, QMake, Ant, SCons, CMake, …)



Alle erben das Konzept voneinander abhängiger Ziele.

Aber make findet sich immer noch in IDEs und kann mehr als nur C

```
SRC = $(shell egrep -l '^[^%]*\\begin\{document\}' *.tex)
PDF = $(SRC:%.tex=%.pdf)
DEP = $(SRC:%.tex=%.d)
SVG = $(shell find . -iname "*.svg")
PDFFIG = $(SVG:%.svg=%.pdf)

all: $(PDFFIG) $(PDF)

$(PDF): %.pdf: %.tex
    pdflatex $<

$(PDFFIG): %.pdf : %.svg
    inkscape $< -E $@

$(DEP):  %.d : %.tex
    export FIGPATH=$(FIGPATH); \
    $(get_dependencies) ; echo $$deps ; \
    $(getpdf) ; echo $$pdfes ; \
    echo "$*.pdf $@ : $< $$deps $$pnges $$pdfes" > $@

-include $(DEP)
```

(aus meinem MOD2-Projekt)

```
# ...
SRCFILES=$(shell find . -name "*.c")
SRCFILES=$(shell find . -name "*.h")
OBJFILES=$(SRCFILES:%.c=%.o)
PRJFILES = $(HDRFILES) $(SRCFILES) boot/dts.s boot/loader.s Makefile link.ld
BACKUPTMP = ../backup-tmp
BACKUPDIR = ../backups
# ...
```

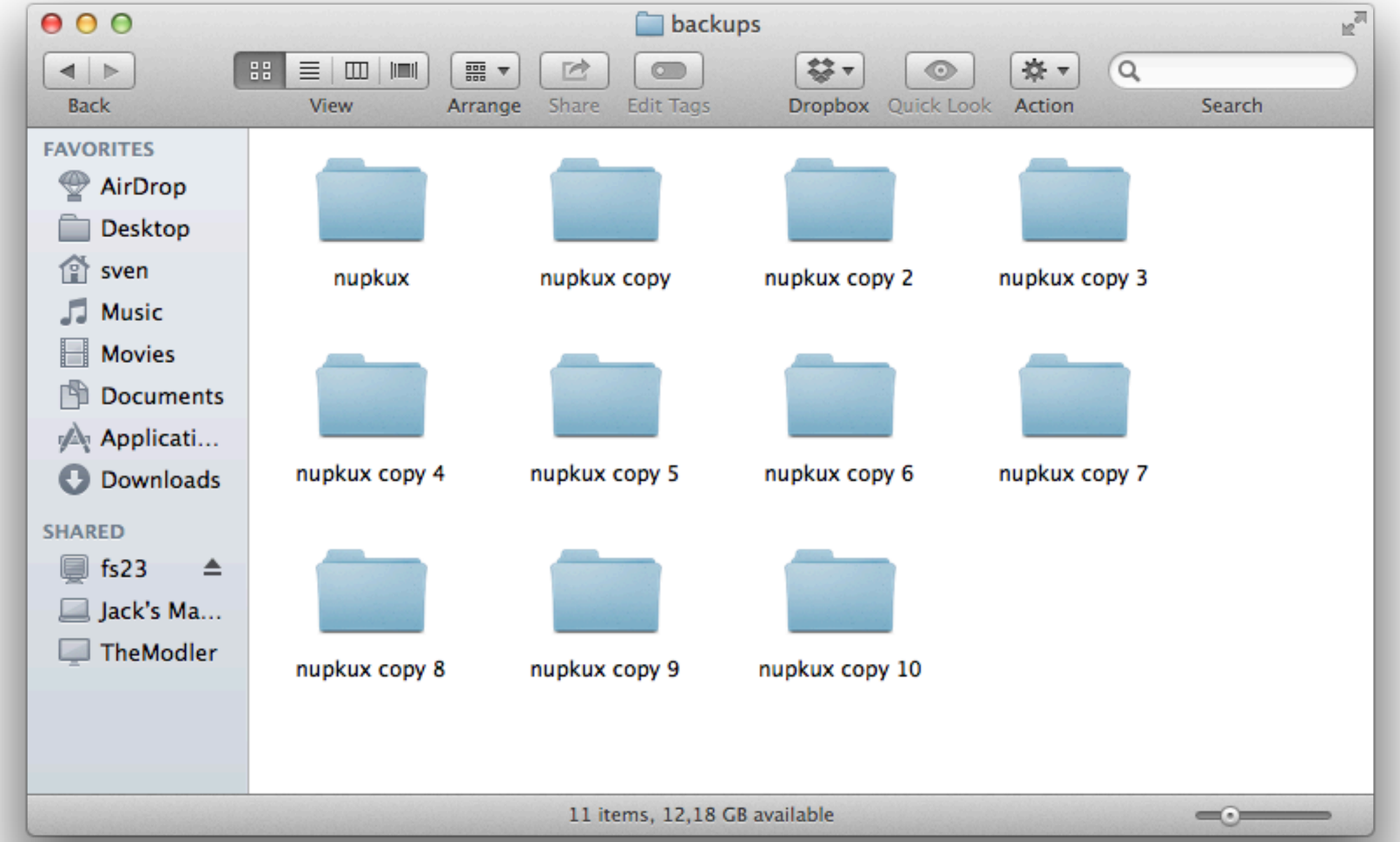
(z.B. Backups)

```
backup: clean
   cp -a . $(BACKUPTMP)
   for file in $(PRJFILES); do $(RM) $(BACKUPTMP)/$$file~; done
   (cd $(BACKUPTMP); tar -c * | gzip > nupkux.tar.gz)
   cp $(BACKUPTMP)/nupkux.tar.gz $(BACKUPDIR)/nupkux-$(shell date +%y-%m-%d).tar.gz
   $(RM) -r $(BACKUPTMP)
```

Was sind die Nachteile dieser Backupmethode?

<30sec brainstorming>

Sie unterscheidet sich nicht davon den Ordner hin und wieder zu kopieren



damit erfüllt sie nur wenige Ansprüche von …

Revisionsverwaltung

2

Wer tat was wann und warum?

Software Configuration Management ::

Verfolgt Zeit, Autor, Zweck and Inhalt von Quelländerungen.
Kann im Fehlerfall helfen die Ursache zu finden.
Unterstützt **gleichzeitige** Arbeit mehrer Entwickler an verschiedenen Orten.

Ziele:

- Projektgeschichte und Prozess verfolgbar, auch wenn Entwickler gehen
- Ältere Versionen können für Analyse/Vergleich wiederhergestellt werden.

[Disclaimer: Gekürzte Definition]

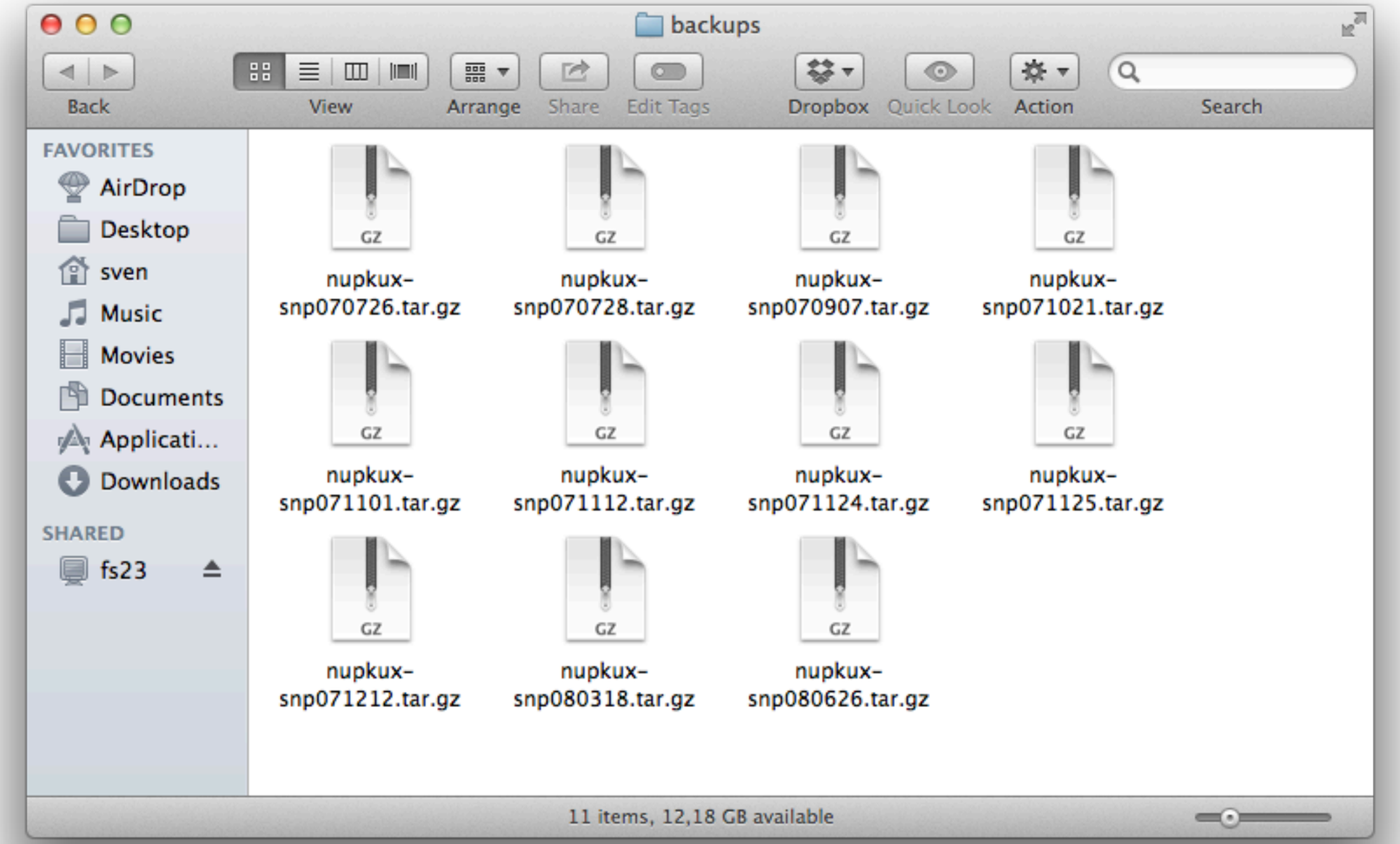
Versionsverwaltung ::

Änderungen an Dokumenten verwalten.
Ein Aspekt von SCM, wird of synonym verwendet.

Versionsverwaltung speichert verschiedene Versionen in einem **Repository**.
Entwickler haben eine Kopie einer Version (+eigene Änderungen) in ihrer **Working Copy**, die sich mit einem **Checkout** laden.
Die Menge aller zusammengehörigen Änderungen ist das **Change Set**.
Mit einem **Commit** (**Check-in**) übertragen sie die Änderungen ins Repo.
Sie können ein Dokumenten vor Veränderungen durch andere sperren (**lock**) oder verschiedene Änderungen kombinieren (**merge**).

Entwickler können mit einem **Tag** einen bestimmt Versionsstand benennen, oder die Entwicklung in einem Seitenzwei fortsetzen (**branch**), der später wieder gemerged werden kann.

Terminologie



✓ Zeitpunkt
X Autor
X Zweck (Notizen)
? Was geändert?

X Nebenläufig
? Kann Verlauf zeigen
✓ Wiederherstellung

Behandelt nur das gesamte Projekt, nicht einzelne Zeilen davon.

Eignen sich Dropbox als Versionskontrollsystem für Softwareprojekte?

Nein. Stellt nur einzelne Dateien wieder her.
Der Kontext der Projektrevision geht verloren.

Bazaar (Canonical/GNU-Projekt, Martin Pool u.a.)
BitKeeper (BitMover, Larry McVoy)
ClearCase (IBM, ehemals Rational, ehemals Atria, ehemals Apollo)
CVS (Brian Berliner, Jeff Polk u.a.)
git (Linus Torvalds u.a.)
Mercurial (Matt Mackall)
Perforce (Perforce Software)
PVCS (Silver Lake Partners, ehemals Serena, Merant, Intersolv, ...)
RCS (Walter F. Tichy)
SCCS (AT&T Bell Labs)
Subversion (CollabNet)
TeamWare (Sun, Larry McVoy)
Visual SourceSafe (Microsoft)
Visual Studio Team Foundation Server (Microsoft)

3

git

Git ::

ein verteiltes Versionsverwaltungssystem, bei dem alle Nutzer
eine lokale Kopie des gesamten Repositories
(einschl. Geschichte) haben.

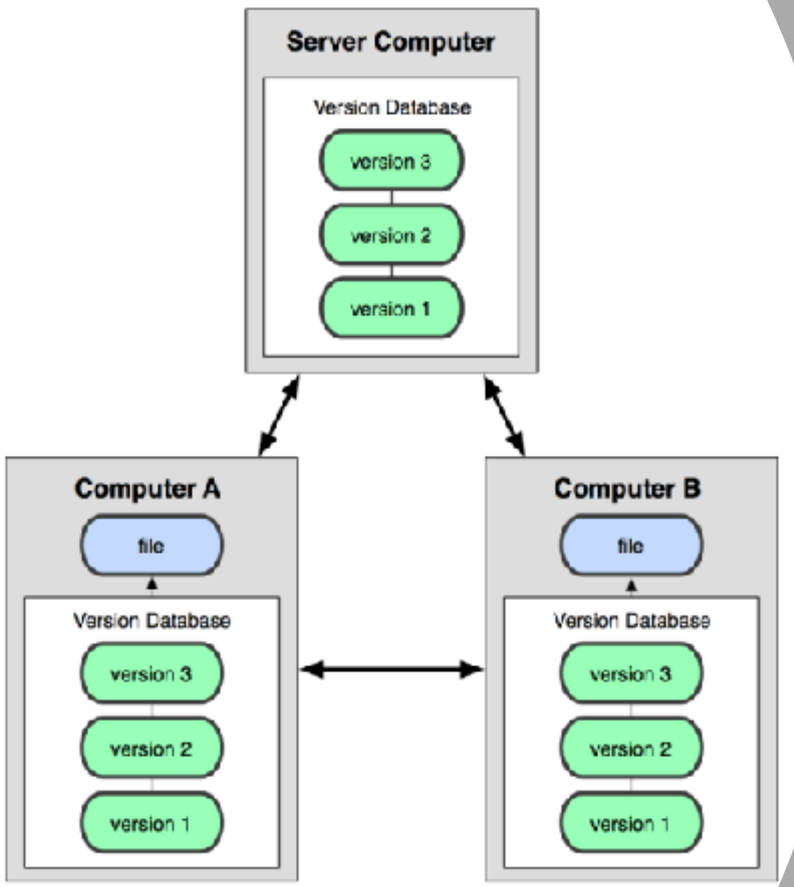
(unterscheidet sich von Subversion, CVS, …
bei dem es einen zentralen Server gibt)

Ursprünglich von Linus Torvalds entwickelt für die
Verwaltung des Linux-Kernels entwickelt.

Aber für einfacheren Austausch kann ein gemeinsamer Server verwendet werden.
Verschiedene Anbieter wie GitHub, BitBucket, …

HPI hat eine Infrastruktur dafür:
https://gitlab.hpi.de

Es reicht auch ein SSH-Server oder ein Netzwerkshare.



< DEMO >

Ein neues Repository anlegen:

```
$ git init
Initialized empty Git repository in colorfilter/.git/
```

Den aktuellen Zustand anzeigen und eine Datei hinzufügen:

```
$ git status
Untracked files:
    apples.bmp
    colorfilter.c
$ git add colorfilter.c
$ git status
Changes to be committed:
    new file:   colorfilter.c
Untracked files:
    apples.bmp
```

Nun wird diese Datei von Git verwaltet (**tracked**).

Alle anderen Dateien in diesem Order werden erstmal **ignoriert**.

Nun wird eine neue Version angelegt (Commit).

```
$ git commit -am"Added skeleton code"
[master (root-commit) 73096bd] Added skeleton code
 1 file changed, 218 insertions(+)
 create mode 100644 colorfilter.c
```

Das `-m` übergibt eine Notiz. Wenn die Option fehlt, öffnet Git den Standardeditor und lässt eine eingeben.

Beschreibt **immer** was ihr getan habt!
Eure Kollegen und Zukünftiges Ich danken!

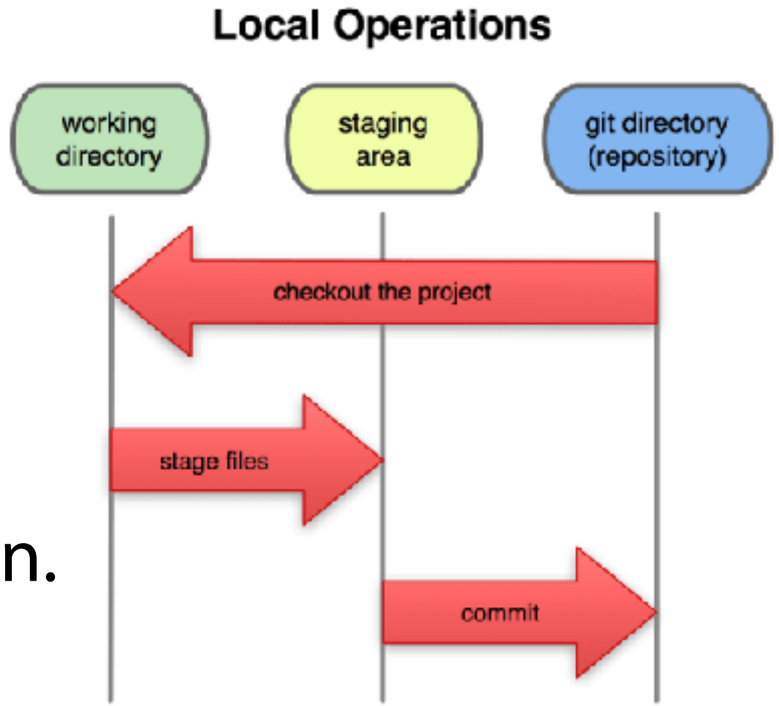
AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

xkcd.com/1296/

Die drei Zustände einer Datei in Git:

(untracked :: Git kümmert sich nicht.)
commited :: Im Repository gespeicherte Dateien.
modified :: Dateien mit ungespeicherten Änderungen.
staged :: Änderungen, die gespeichert werden sollen (nach einem add).

Quell häufiger Verwirrung



Mit dem Aufruf `git commit -a`, werden alle modifizierten Dateien hinzugefügt und committed.
Für Experten: Nutzt "`git add`" um den Zustand einzelner Dateien und Zeilen zu ändern.
Für Einsteiger: Nutzt einfach `-a`

Ich habe das in meiner `~./gitconfig`, um die Zustände zu unterscheiden:

```
[color "status"]
      added = green
      changed = yellow
      untracked = cyan
```

Die eigene Arbeit zu einer anderen Person bekommen:

Beide definieren ein **remote:**

Das ist eine SSH url

```
$ git remote add hpi git@gitlab.hpi.de:repo.git
```

Name ist frei wählbar

Zum Senden wird das Repository auf den Server ge**push**t.

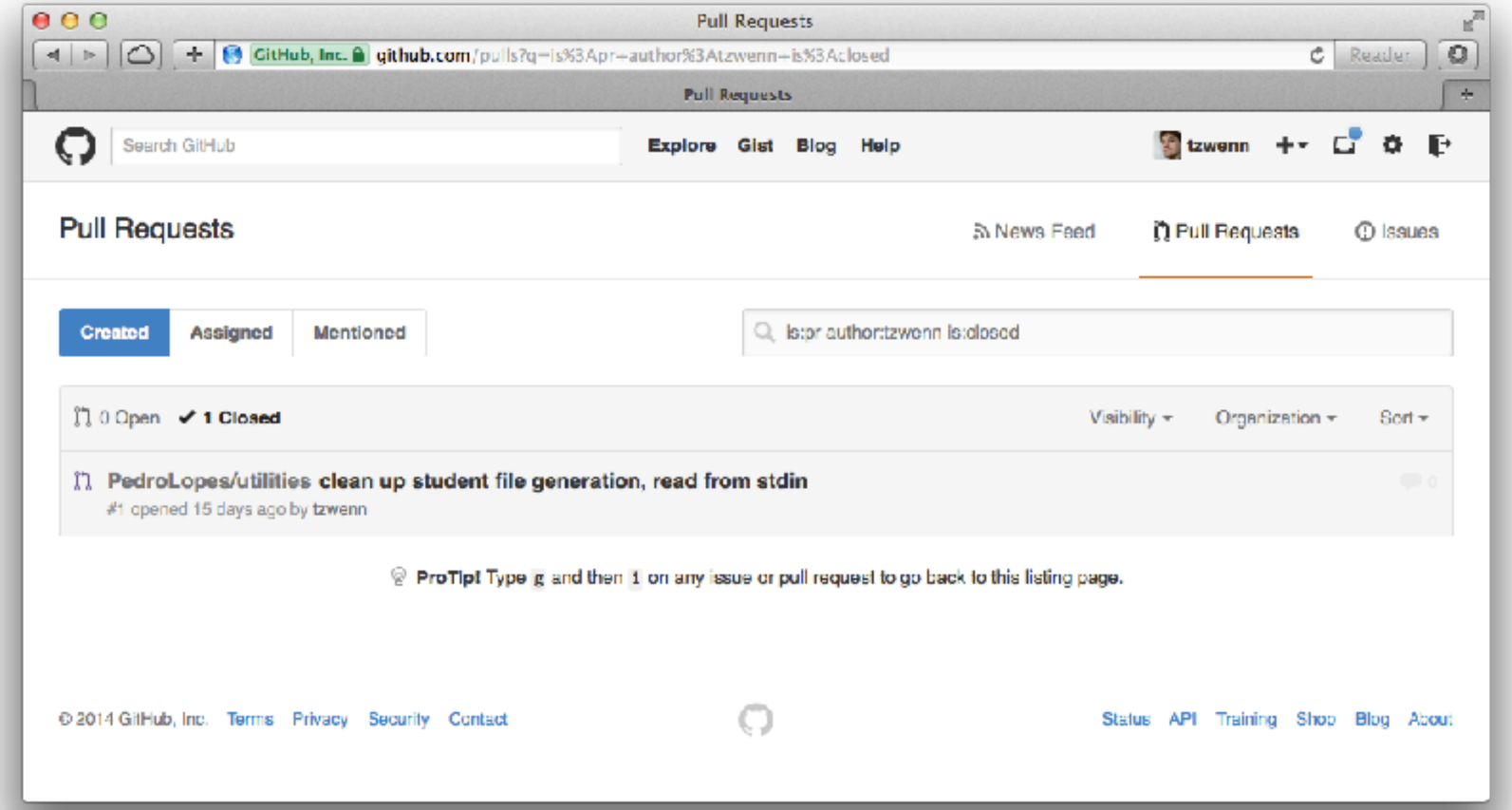
```
$ git push hpi master
```

Welcher Branch soll übertragen werden

Zum Entfangen, wird vom Remote ge**pull**t:

```
$ git pull hpi master
```

Angedachter Workflow:
Erteile anderen Personen Lesezugriff auf dein Repository und lass sie die vorgeschlagenen Änderungen übernehmen.

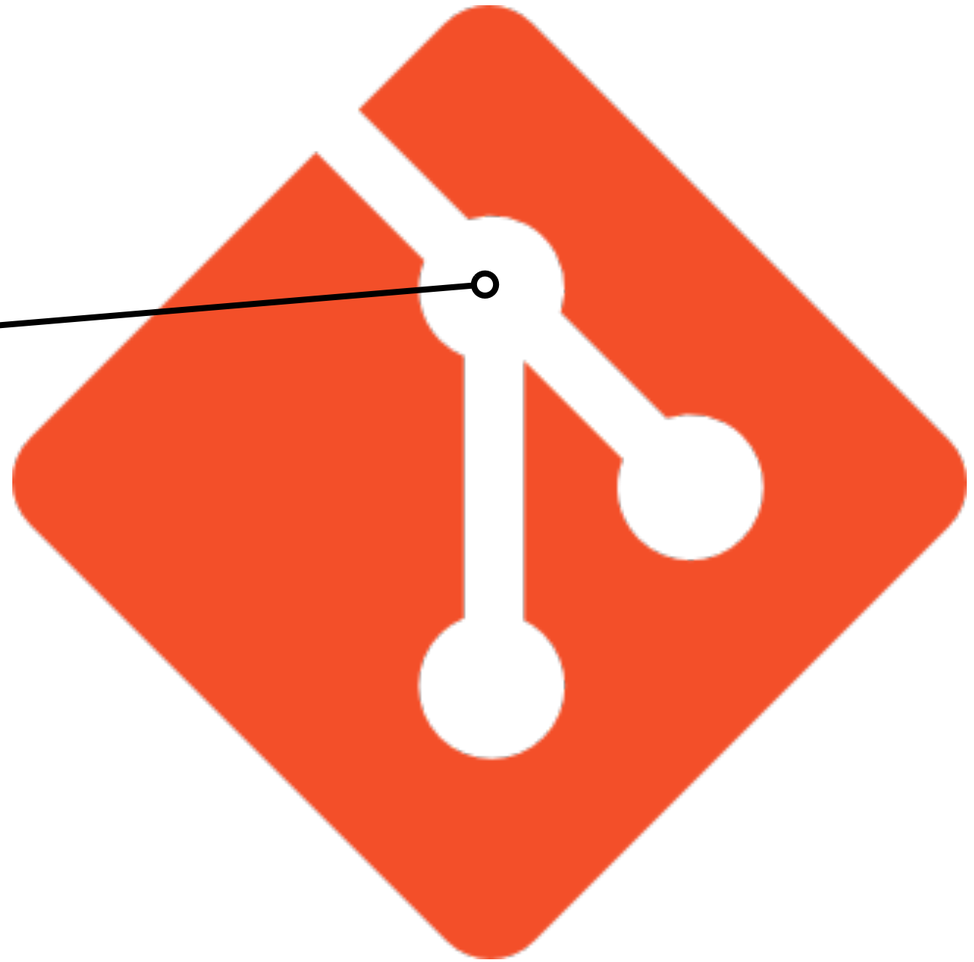


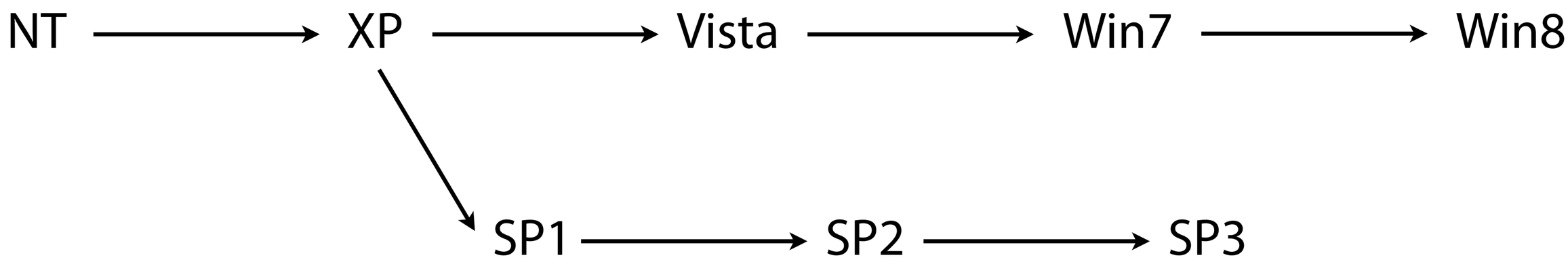
Wie bekommt man ein bereits bestehendes Projekt?

```
$ git clone <url>
```

wobei `url` eine SSH-Adresse (letztes Beispiel), HTTP-WebDAV,
oder sogar ein anderes Projekt auf einem USB-Stick ist.
Diese URL wird automatisch zum remote namens “origin”.

Git’s real power: Einfach Branchen

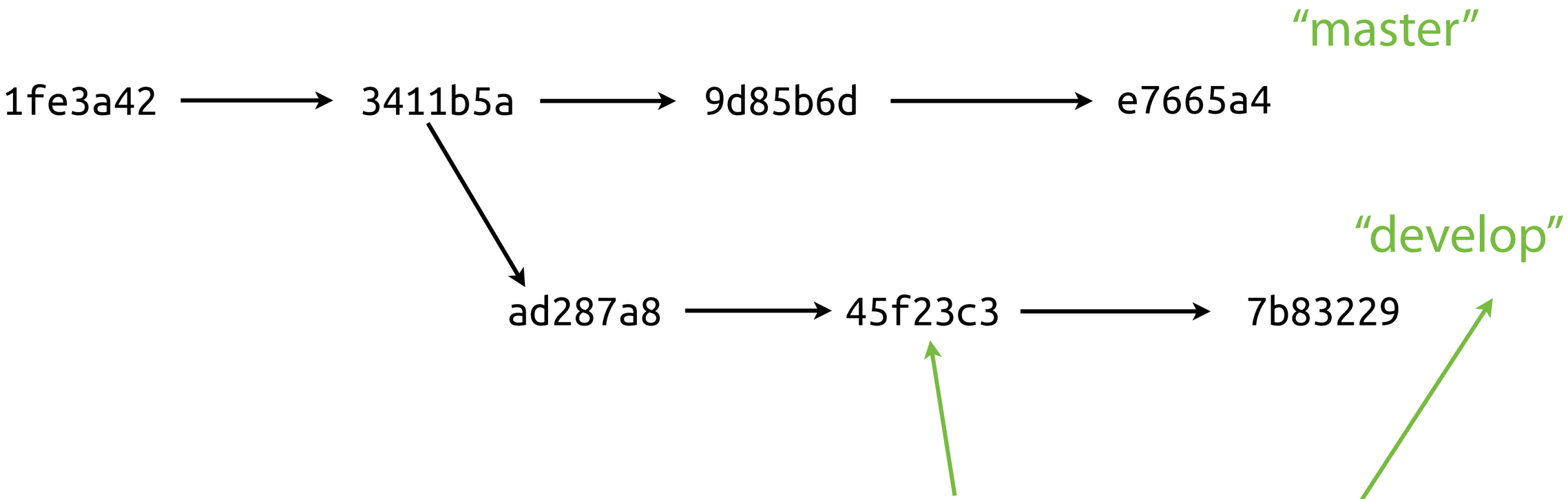




(Revision history not to scale)

Erinnerung:

Branch :: (nebenläufiger) Strang von Änderungen an der gleichen Quelle



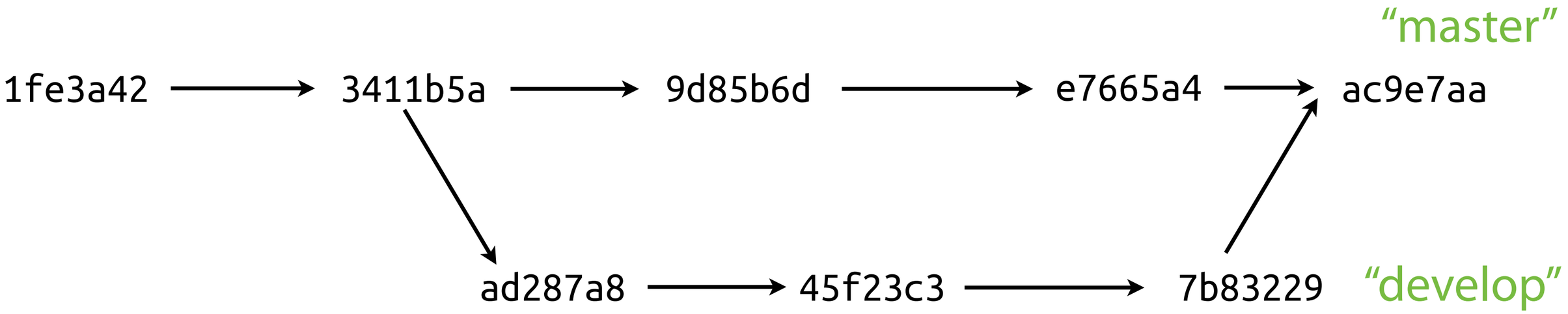
Git erstellt pro Commit eine Prüfsumme (**hash**) von Inhalt, Autor, Datum …

Ein Referenzname zeigt auf die Spitze (**tip**) jedes Branches.

```
$ git branch develop     # Erzeugt einen neuen Branch am aktuellen Commit
$ git checkout develop   # Wechselt zum "develop" Branch
$ git commit -am"Erzeuge einen neuen Commit auf develop Branch"
$ git checkout master    # Wechselt zurück zum "master" Branch
```

Den Hash zu einem Branch finden: Entweder `git log` oder

```
$ git rev-parse master
e7665a48437c24d49c0552b868662263a477719b
$ cat .git/refs/heads/master
e7665a48437c24d49c0552b868662263a477719b
```

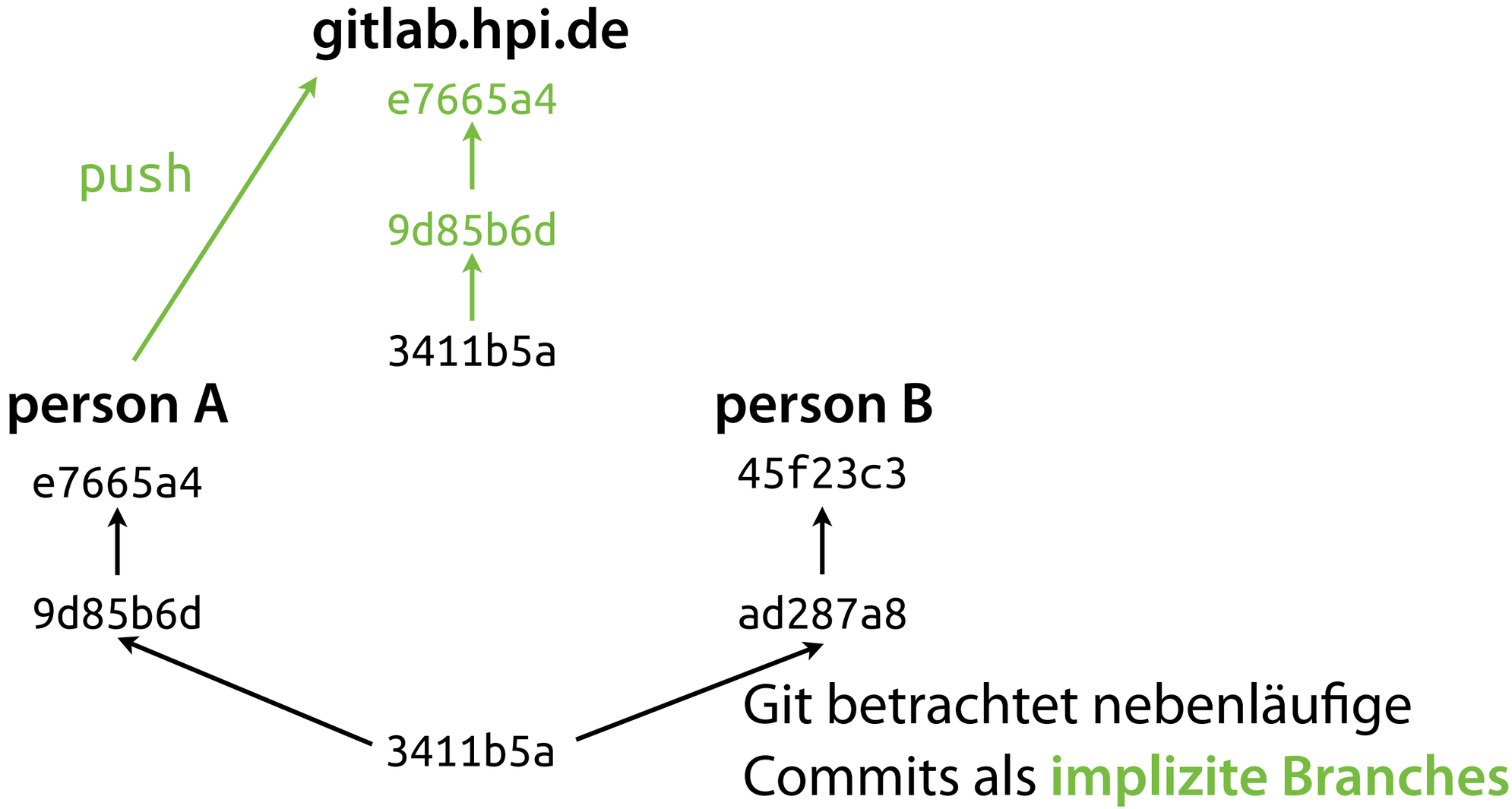


Die Arbeit zurück auf den Master-Branch führen
(und dabei einen neuen Commit erzeugen):

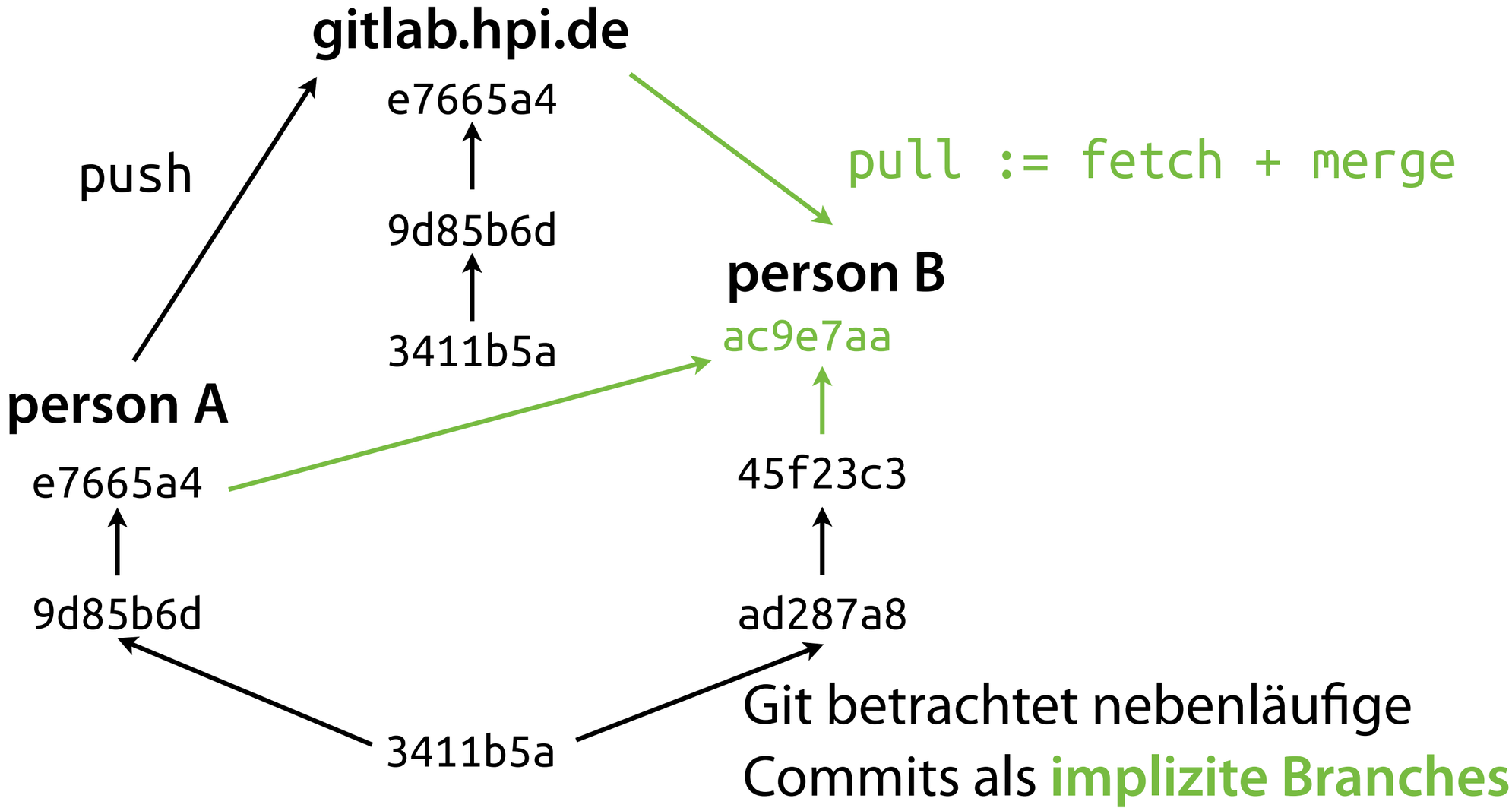
Git ist gut darin diese zu verhindern

```
$ git checkout master
$ git merge develop
$ git mergetool          # Im Falle von Konflikten (unvereinbare Änderungen)
```

Synchronisation von nebenläufiger Arbeit über zentralen Server.



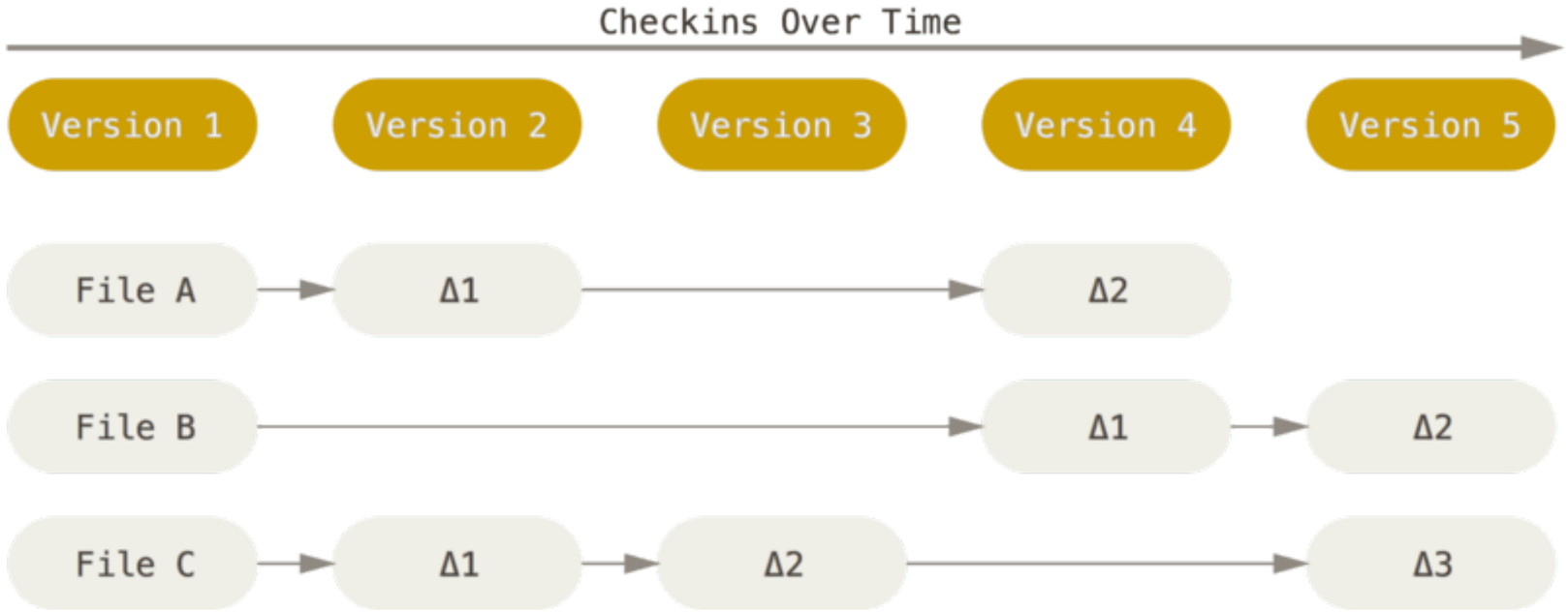
Synchronisation von nebenläufiger Arbeit über zentralen Server.



Gits Interne Datenstrukturen

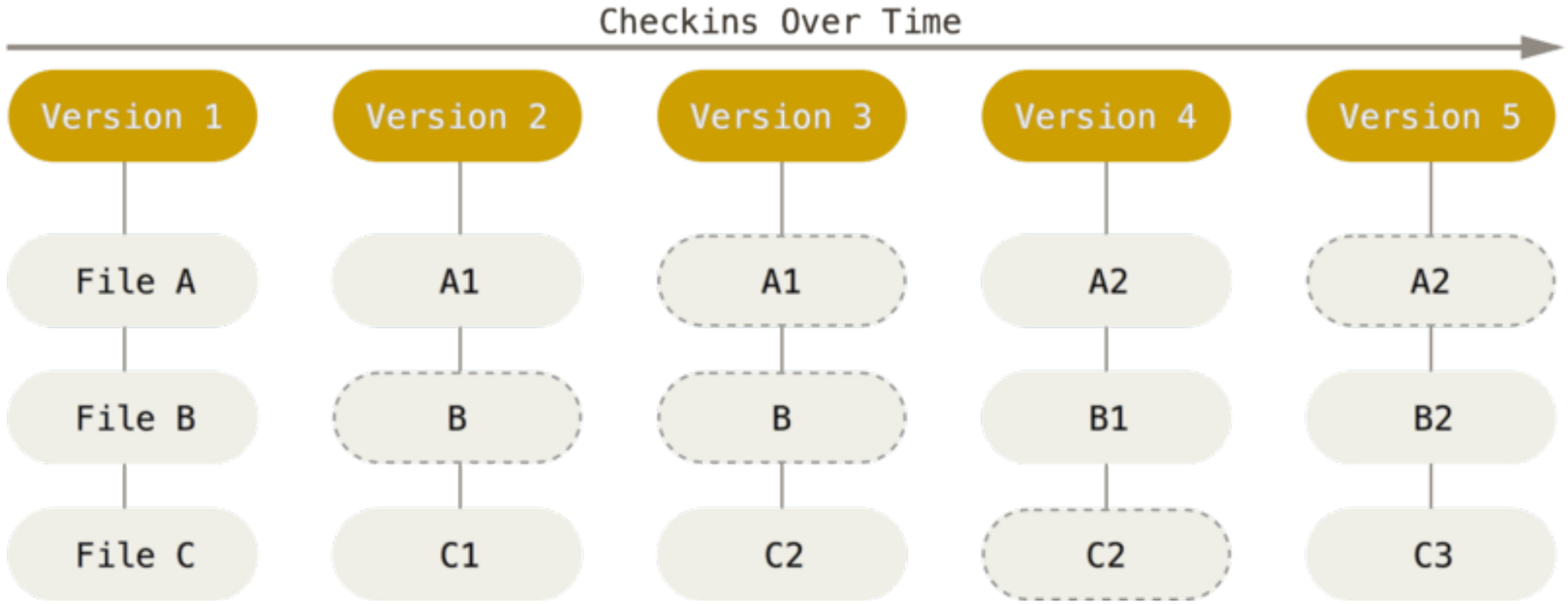
http://git-scm.com/book/

Subversion, CVS, …
speichern fortlaufende Änderungen an Dateien (**diff**s, **delta**s)



Git speichert den Zustand (**snapshot**s) aller Dateien und Referenzen darauf.

Integrität über SHA-1 Prüfsummen festgestellt.



Git-Objekt ::

In Git gespeicherte Daten (Inhalt einer Datei (Blob), Baum, ein Commit), adressierbar über einen Namen.

Der Name ist die SHA-1 Summe (160 Bit) des Inhalts als Hexadezimalstring. Ermöglicht schnelles Finden und Vergleichen anhand des Namens.

Gespeichert in `.git/objects/<erstes Byte>/<Restliche Bytes>`

Für Blobs wird `sha1("blob " <länge als text> "\0" <daten>)` berechnet.

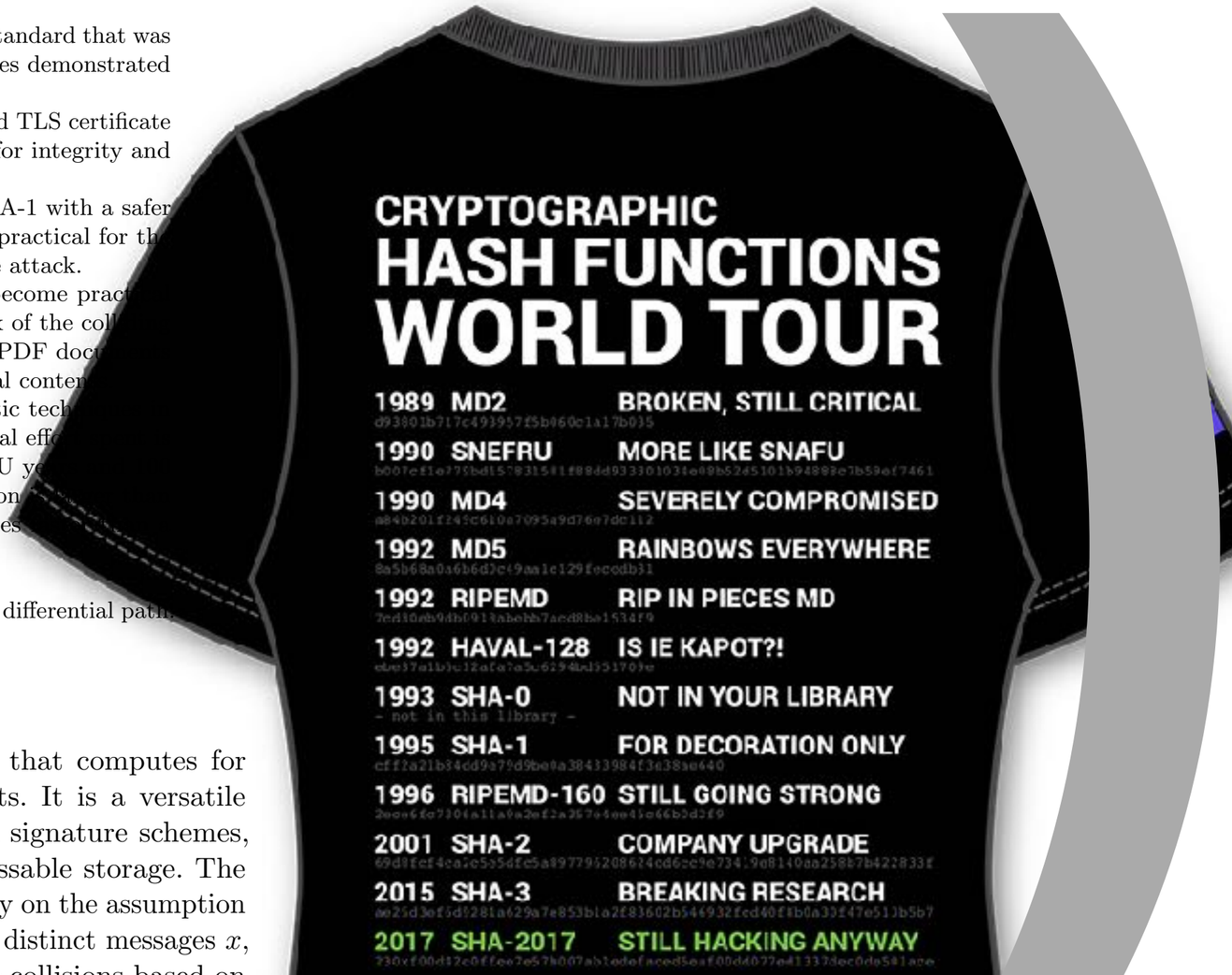
The first collision for full SHA-1

Marc Stevens[1], Elie Bursztein[2], Pierre Karpman[1], Ange Albertini[2], Yarik Markov[2]

[1] CWI Amsterdam
[2] Google Research
info@shattered.io
https://shattered.io

Abstract. SHA-1 is a widely used 1995 NIST cryptographic hash function standard that was officially deprecated by NIST in 2011 due to fundamental security weaknesses demonstrated in various analyses and theoretical attacks.
Despite its deprecation, SHA-1 remains widely used in 2017 for document and TLS certificate signatures, and also in many software such as the GIT versioning system for integrity and backup purposes.
A key reason behind the reluctance of many industry players to replace SHA-1 with a safer alternative is the fact that finding an actual collision has seemed to be impractical for th past eleven years due to the high complexity and computational cost of the attack.
In this paper, we demonstrate that SHA-1 collision attacks have finally become pract by providing the first known instance of a collision. Furthermore, the prefix of the col messages was carefully chosen so that they allow an attacker to forge two PDF doc with the same SHA-1 hash yet that display arbitrarily-chosen distinct visual conte
We were able to find this collision by combining many special cryptanalytic tech complex ways and improving upon previous work. In total the computational eff equivalent to $2^{63.1}$ SHA-1 compressions and took approximately 6 500 CPU y GPU years. As a result while the computational power spent on this collision other public cryptanalytic computations, it is still more than 100 000 times brute force search.

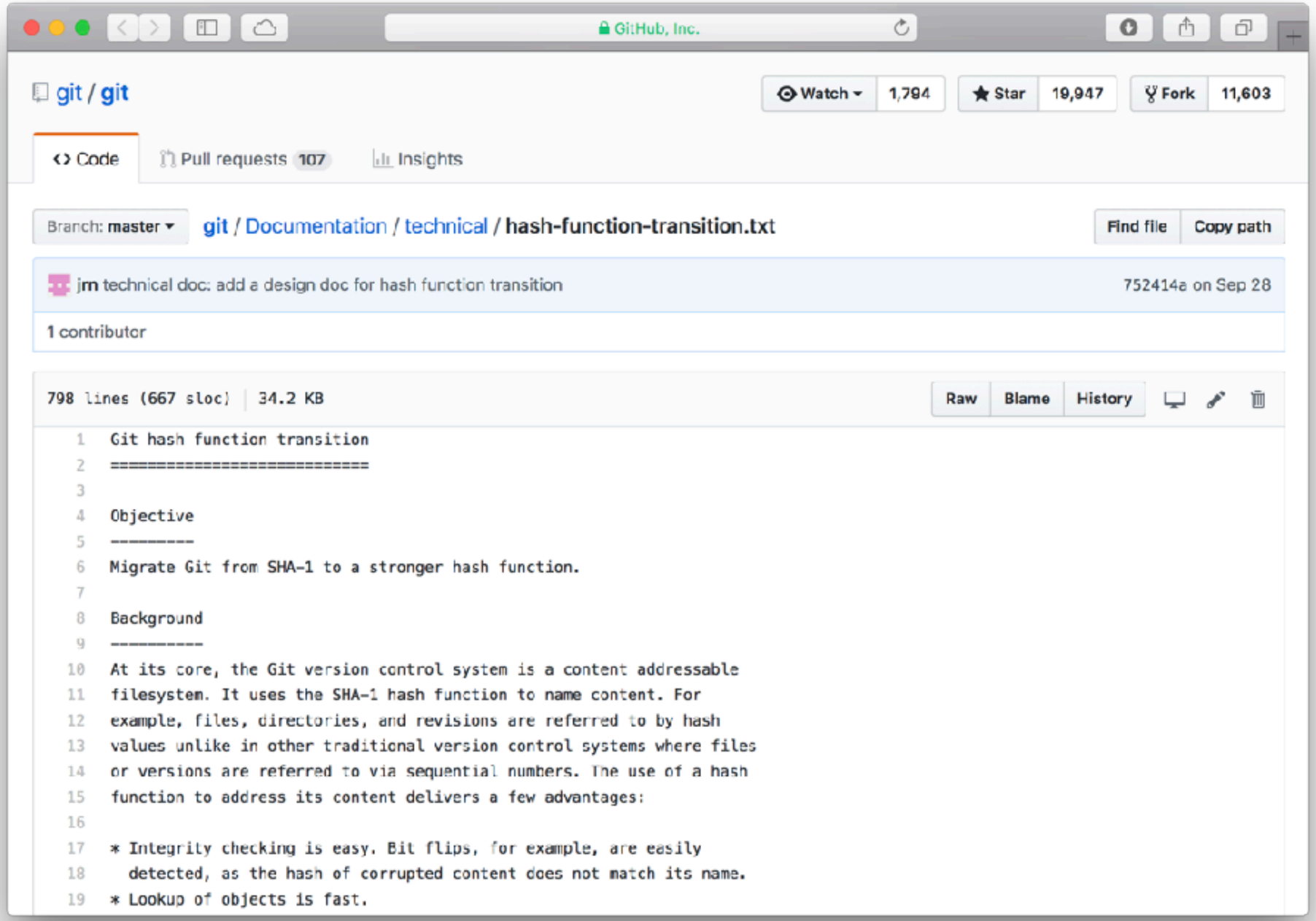
Keywords: hash function, cryptanalysis, collision attack, collision example, differential path



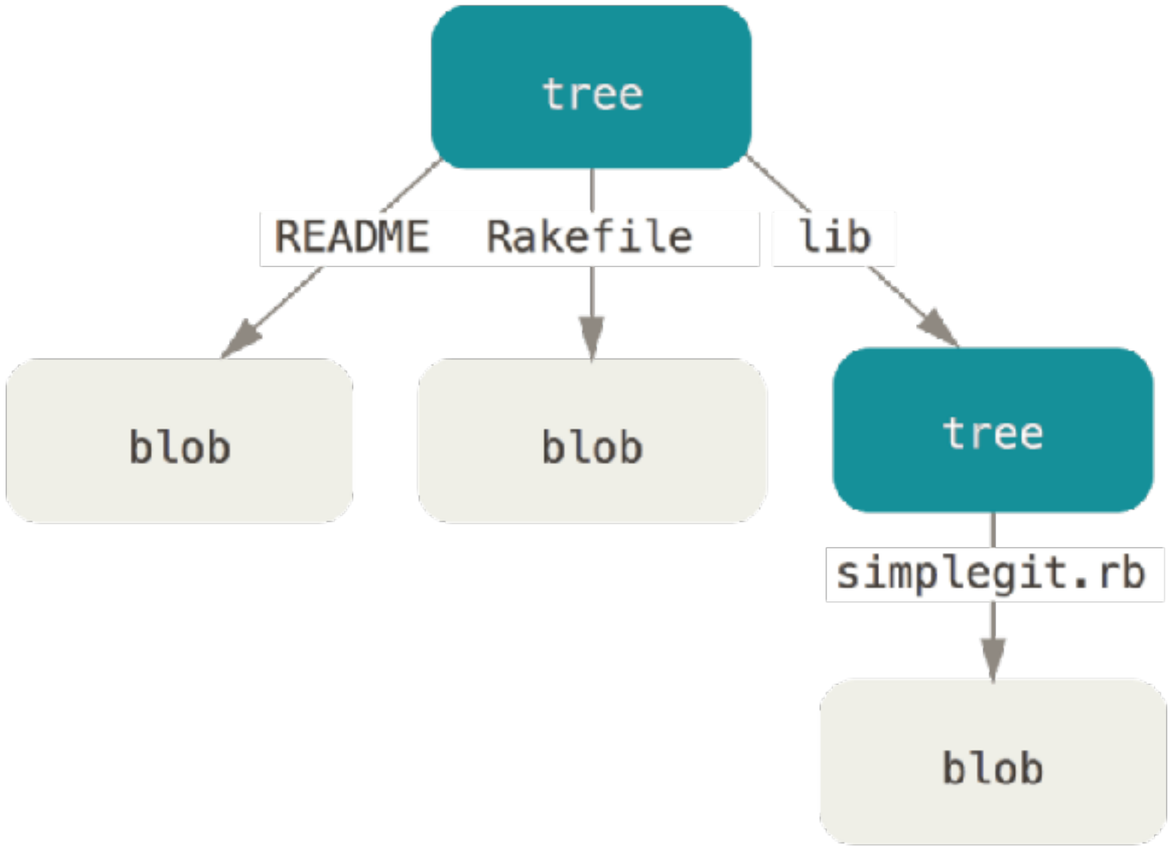
1 Introduction

A cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^n$ is a function that computes for any arbitrarily long message M a fixed-length hash value of n bits. It is a versatile cryptographic primitive used in many applications including digital signature schemes, message authentication codes, password hashing and content-addressable storage. The security or even the proper functioning of many of these applications rely on the assumption that it is practically impossible to find collisions. A collision being two distinct messages x,

[23.02.2017]


GitHub, Inc.
git / git
Watch 1,794
Star 19,947
Fork 11,603
Code
Pull requests 107
Insights
Branch: master
git / Documentation / technical / hash-function-transition.txt
Find file
Copy path
jrn technical doc: add a design doc for hash function transition
752414a on Sep 28
1 contributor
798 lines (667 sloc) 34.2 KB
Raw
Blame
History
Git hash function transition
Objective
Migrate Git from SHA-1 to a stronger hash function.
Background
At its core, the Git version control system is a content addressable
filesystem. It uses the SHA-1 hash function to name content. For
example, files, directories, and revisions are referred to by hash
values unlike in other traditional version control systems where files
or versions are referred to via sequential numbers. The use of a hash
function to address its content delivers a few advantages:
* Integrity checking is easy. Bit flips, for example, are easily
detected, as the hash of corrupted content does not match its name.
* Lookup of objects is fast.

[28.09.2017]

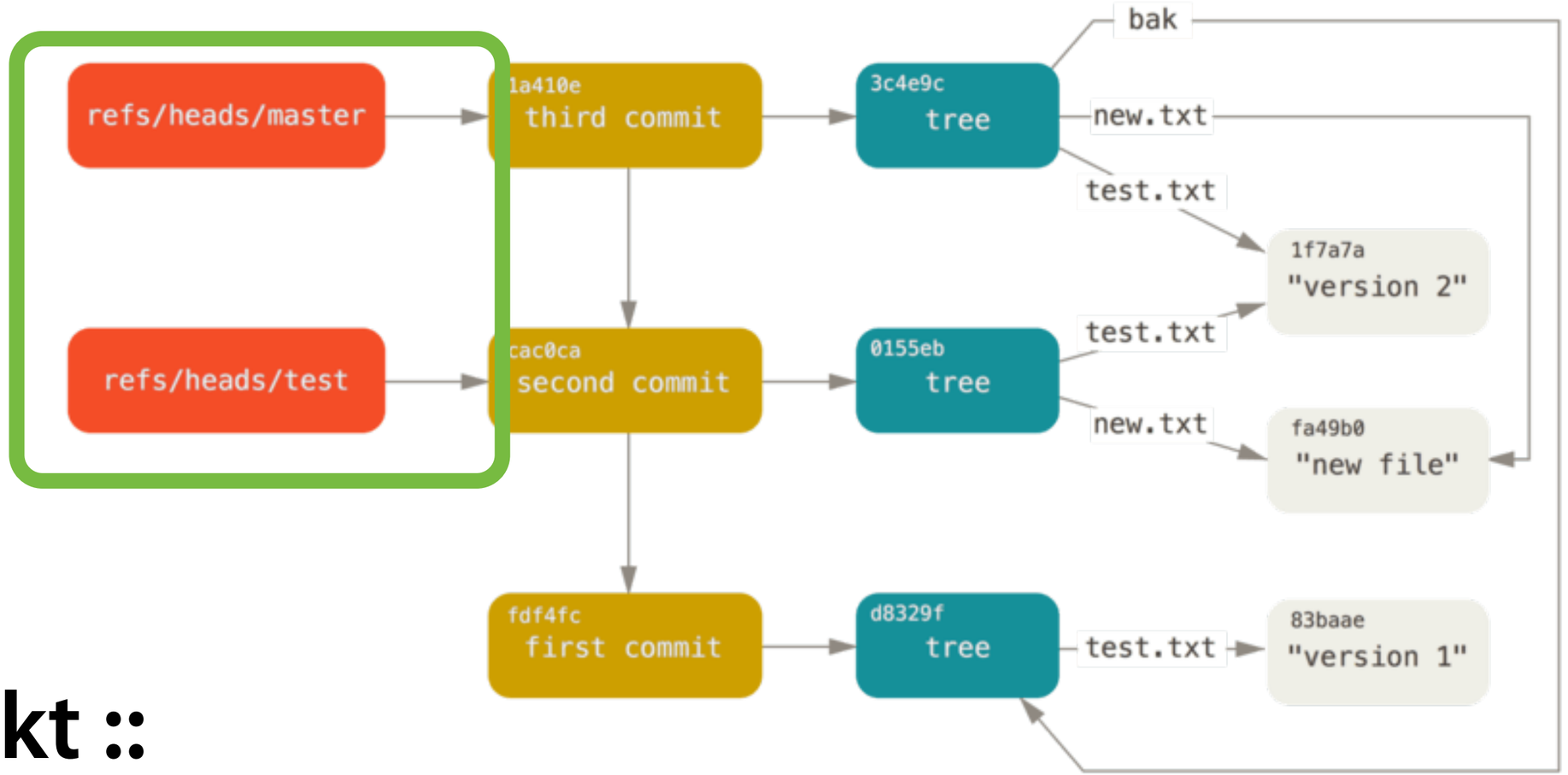


Tree-Objekt ::

Besteht aus einer Sammlung weiterer Objekte (Blobs, Trees).

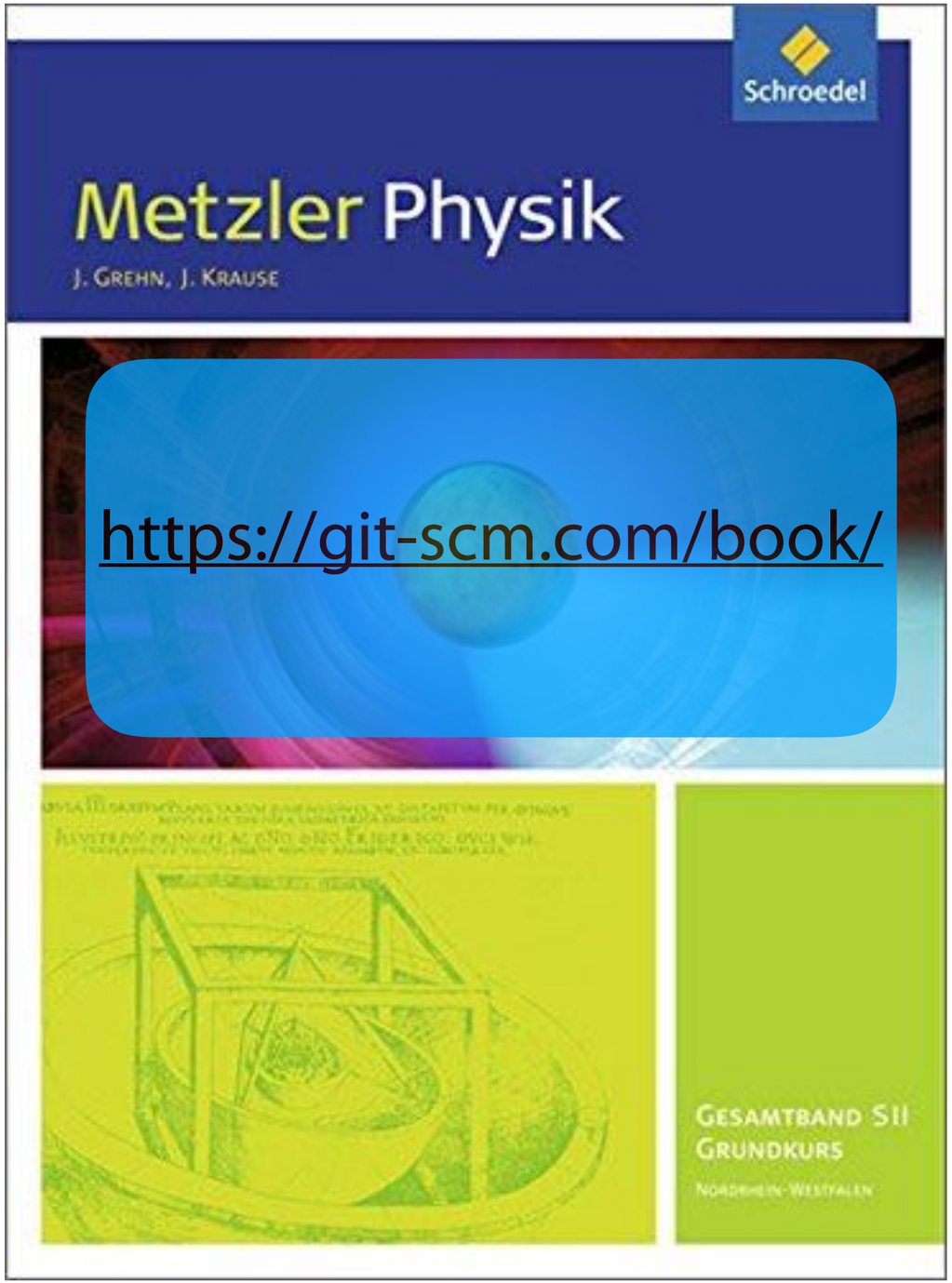
Kann ein Verzeichnis mit Dateinamen, -inhalt, -rechten, Symlinks, … abbilden.

Referenzen sind lesbarere, selbst-definierte Namen.



Commit-Objekt ::

Speichert den Verzeichnisbaum des aktuellen Projektstands.
Enthält zusätzlich den Ersteller des Commits, den Autor, die Zeit, eine Notiz und den Elterncommit.
Bildet die Geschichte als Hash-Kette ab. Änderungen an der Geschichte ändern alle folgenden Hashs.



Nicht nur über die Welt lesen um mit Ihr zu interagieren. Rausgehen!

=

Für jedes Projekt, jede Hausaufgabe, die mehr als 20 Minuten dauert:
Macht ein Git-Repo!

ende