Arbeiten im Team

Lehrstuhl für Softwaretechnik
Andreas Zeller

Arbeiten im Team

- Qualifikation
- Motivation
- Teambildung
- Effiziente Besprechungen
- Konfigurationsmanagement
- Problem Tracking

Allgemeine Qualifikationen

Ein Software-Entwickler benötigt folgende allgemeine Qualifikationen:

Abstraktionsvermögen. Nur mit Hilfe der Abstraktion können komplexe Systeme bewältigt werden.

Kommunikationsfähigkeit. Gute sprachliche Ausdrucksweise und Präsentation ist wichtig. Für die Dokumentation benötigt man eine gute Schriftform.

Teamfähigkeit. Mitarbeiter sollen Teamgeist besitzen und konstruktiv und kooperativ zum Teamergebnis beitragen.

Allgemeine Qualifikationen (2)

Wille zum lebenslangen Lernen. Das Wissen der Software-Technik verdoppelt sich alle vier Jahre.

Intellektuelle Flexibilität und Mobilität. Auch das gesamte Umfeld der Software-Technik ändert sich permanent (Beispiel: Internet, e-commerce, Mobile Anwendungen).

Kreativität. In der Software-Technik gibt es (noch?) kein breites Erfahrungspotential, aus dem man unbesehen schöpfen könnte.

Stellensituation

- Zahl der offenen Stellen für IT-Experten auf dem Arbeitsmarkt hat sich zwischen 2009 und 2011 fast verdoppelt
- 84% der befragten Unternehmen suchen Software-Entwickler
- Beste Aussichten für weibliche Fachkräfte

Stress

Stressverträglichkeit ist ein Muss:

- Immer mehr Projekte mit wenig Personal
- Teamleiter in Sandwich-Position
- Hohe Arbeitsverdichtung

Unternehmen halten dagegen:

→ Pausenrituale, Blockzeiten, flexible Arbeitszeit

→ Überlappende Verantwortlichkeiten

Quelle: Anja Gerlmaier vom Institut Arbeit und Qualifikation (IAQ) der Universität Duisburg-Essen

Das Zerrbild des Programmierers

The popular image of a programmer
is of a lone individual working far
into the night peering into a terminal
or poring over reams of paper
covered with arcane symbols.

Tatsächlich jedoch hoher Anteil (ca. 50%) an Gruppenarbeit und Kommunikation.

The lack of encouragement and the unattractive image may
explain why even the most mathematically able college women
are more likely to major in the humanities
than in a discipline like computer science.

Englisch

Sprachtest

Can you English?

London: Bestehen Sie vor der Sprachpolizei?

Wenn der Grund- oder Leistungskurs schon ein paar Jahre her ist, rostet das Englisch und holpert. Im Berufsleben kann das peinlich werden. Machen Sie hier den Test, wie weit Sie mit Ihrem Englisch kommen.

- Schnittstellen, Dokumentation und Code sind in der Regel auf Englisch
- CS lectures, too

Was motiviert Menschen bei der Arbeit?

Wir unterscheiden drei *Motivationstypen*:

- Aufgabenbezogener Typ
- Selbstbezogener Typ
- Interaktionsbezogener Typ

Aufgabenbezogener Typ

- motiviert durch
  intellektuelle Herausforderung der Arbeit
- Selbstcharakterisierung als unabhängig, einfallsreich, zurückhaltend, introvertiert, energisch, wetteifernd, selbständig
- Mehrheit der Softwareentwickler ist aufgabenbezogen

Selbstbezogener Typ

- motiviert durch *persönlichen Erfolg*
  (gemessen in Geld oder Statussymbolen)
- Selbstcharakterisierung als eklig, lästig, hartnäckig, dogmatisch, introvertiert, eifersüchtig, draufgängerisch, wetteifernd
- Zielerreichung vor allem im Management

Interaktionsbezogener Typ

- motiviert durch *Zusammenarbeit*
- Selbstcharakterisierung als friedfertig, hilfsbereit, rücksichtsvoll, besonnen, geringes Autonomie- und Statusbedürfnis
- durch Anwender-orientierte Projekte angezogen
- Frauen häufiger interaktionsbezogen (Gründe unklar!)

Gruppenzusammensetzung

Der Gruppenerfolg ist abhängig von der *Zusammensetzung:*

Gleichartig motivierte Gruppen erreichen ihr Ziel nur bei interaktionsbezogenen Mitgliedern.

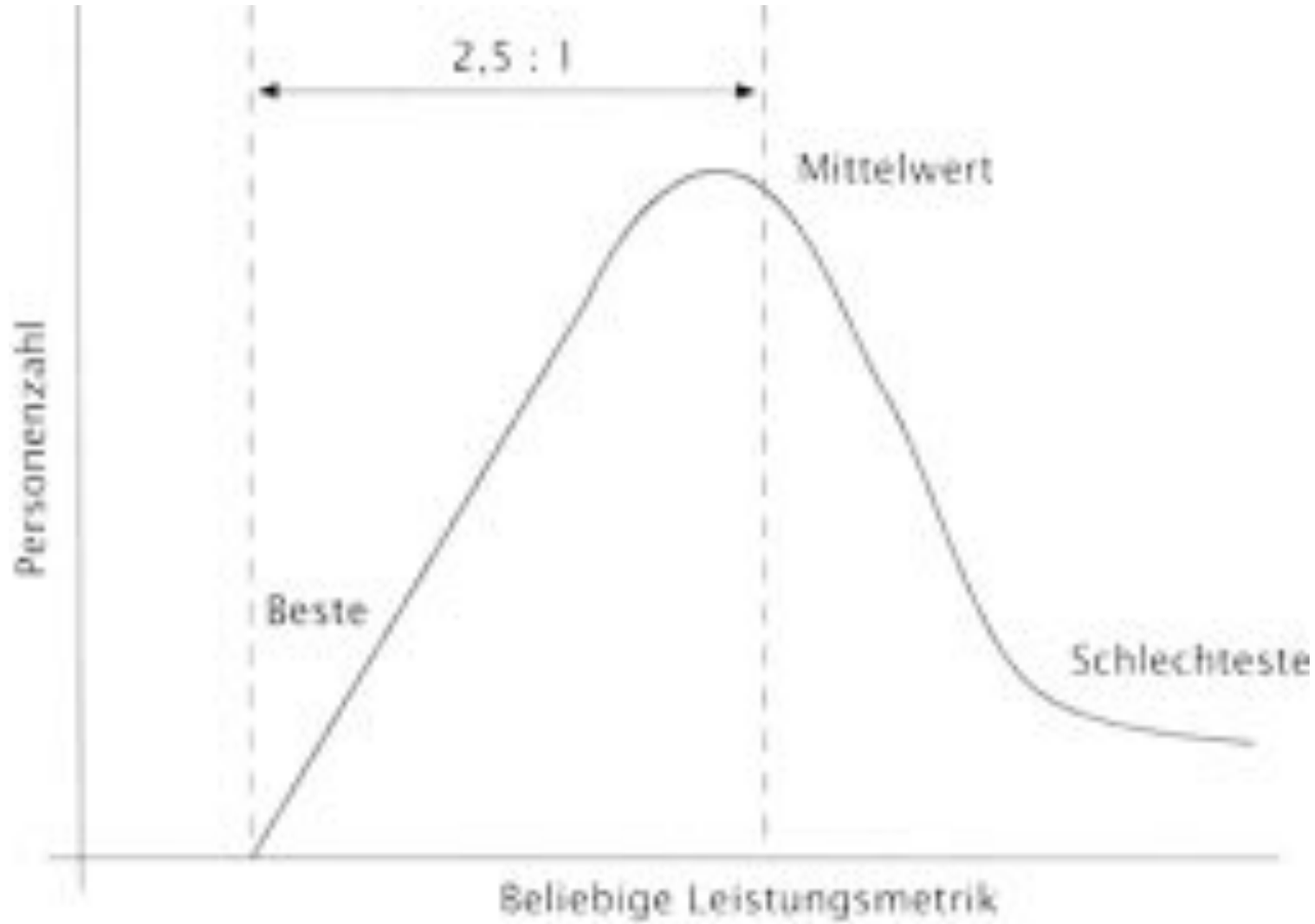
Folgerung:

- Gruppen sollten Mitglieder *aller Motivationskategorien* enthalten
- Der Gruppenleiter sollte *aufgabenbezogen* sein.

Vielfalt steigert die Kreativität!

Produktivität fördern

Große Produktivitätsunterschiede zwischen Software-Entwicklern:



Grundregeln Produktivität

- Die besten Mitarbeiter sind 10× besser als die schlechtesten
- Die besten Mitarbeiter sind 2,5× besser als der Durchschnitt
- Die überdurchschnittlichen Mitarbeiter übertreffen die unterdurchschnittlichen Mitarbeiter im Verhältnis 2:1.

Diese Regeln gelten für fast jede beliebige Leistungsmetrik (Zeit, Fehler…).

Die besten Mitarbeiter sind um ein Vielfaches produktiver, erhalten aber nur geringfügig mehr Gehalt.
Konsequenz: gute Leute einstellen!

Relevante Faktoren

Ein Mitarbeiter ist um so produktiver

- je ruhiger sein Arbeitsplatz (= je weniger er gestört wird)
- je besser die Privatsphäre ist
- je größer der Arbeitsplatz ist.

Übersicht Umfrageergebnisse:

Arbeitsplatzfaktoren		bestes Viertel der Teilnehmer	schlechtestes Viertel der Teilnehmer
1	Wieviel Arbeitsplatz steht Ihnen zur Verfügung?	7m^2	4,1 m^2
2	Ist es annehmbar ruhig?	57% ja	29% ja
3	Ist Ihre Privatsphäre gewahrt?	62% ja	19% ja
4	Können Sie Ihr Telefon abstellen?	52% ja	10% ja
5	Können Sie Ihr Telefon umleiten?	76% ja	19% ja
6	Werden Sie oft von anderen Personen grundlos gestört?	38% ja	76% ja

Hintergrund

Um produktive Ingenieurarbeit zu erledigen, muss man „in Fahrt“ *(in flow)* sein:

- „in Fahrt“: Zustand tiefer, fast meditativer Versunkenheit
- Man fühlt eine leichte Euphorie und verliert das Zeitgefühl

Für diesen Zustand benötigt man ca. 15 Minuten voller Konzentration

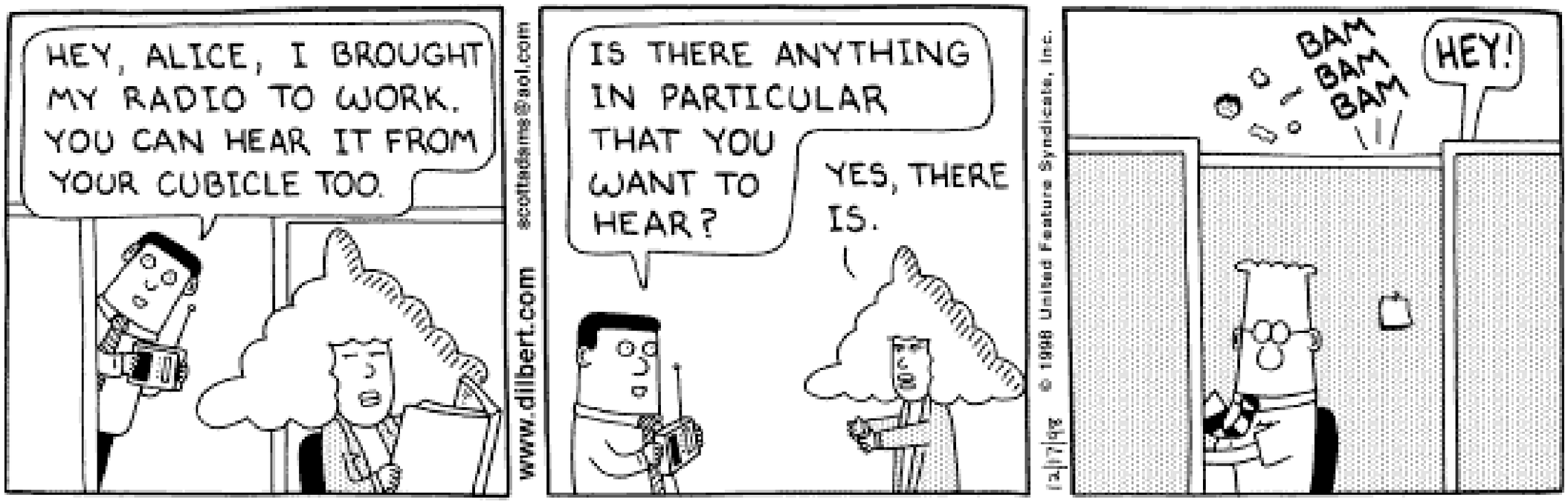
In diesem Zustand ist man besonders anfällig für Störungen

Ein Telefonanruf und die 15 Minuten Eintauchphase beginnen von vorne!

Konsequenzen

- Abstellbare (oder gar keine) Telefone – keine Lautsprecherdurchsagen
- Einzelbüros (mit verschließbarer Tür) – keine „Cubicles“
- Niedriger Geräuschpegel – keine Großraumbüros

Musik kann zwar einen Geräuschpegel übertönen, blockiert jedoch die rechte Gehirnhälfte, die für Kreativität zuständig ist



Teambildung fördern

Die folgenden Faktoren fördern die Teambildung:

Team zu Erfolgen verhelfen. Teams brauchen ein gemeinsames *konkretes* Ziel. Keine abstrakten Firmenziele *(mission statements)*, sondern messbare, prüfbare Ziele. Die Arbeit sollte so aufgeteilt werden, dass genügend oft Erfolgserlebnisse da sind.

Elite-Team. Mitarbeiter brauchen das Gefühl, *einzigartig* zu sein. Egal, worin sich die Einzigartigkeit ausdrückt, sie ist Grundlage für die Identität des Teams.

Rituale

Teambildung fördern (2)

Qualitätskult. „Nur das Beste ist gut genug für uns“. Jedes Team braucht eine *Herausforderung.* Mittelmäßige Aufgaben nehmen den Ehrgeiz, eine herausragende Leistung zu bringen. Niemand ist stolz, an einem „Schundprojekt“ zu arbeiten.

Kein Overengineering. Vergolden Sie keine Funktionen, nur weil es dem Team Spaß macht. Perfektionieren Sie das Notwendige.

Vielfalt. Größere Erfolgschancen, wenn Team vielfältig zusammengesetzt ist (z.B. Männer und Frauen, Endanwender und Entwickler…)

Persistenz. „Never change a winning team“.

Teambildung fördern (3)

Vertrauen statt Kontrolle. Der Manager soll sich beschränken, *Strategien* vorzugeben, sich aber nicht in die Taktik einmischen. Der Manager muss dem Team *Vertrauen* entgegenbringen; das Team muss die Möglichkeit haben, autonom sein Vorgehen zu wählen.

Bürokratie vermeiden. Im richtigen Umfang ist Dokumentation notwendig. Aber: Es darf nicht der Eindruck entstehen, das Management sei nur auf „Planerfüllung“ aus. Das Team muss spüren, dass auch das Management an sein Ziel glaubt.

Räumliche Nähe. Räumliche Trennung behindert das Gefühl der Zusammengehörigkeit.

Teambildung fördern (4)

Ein Team pro Nase. Eingeschworene Teams entstehen nur, wenn die Mitglieder den größten Teil ihrer Zeit darin verbringen. Mitgliedschaft in mehreren Teams erschwert die Teambildung – und die Effizienz.

Echte Termine. Das Management darf nur Termine vorgeben, die auch einzuhalten sind. Alles andere zerstört Glaubwürdigkeit und Vertrauensbasis.

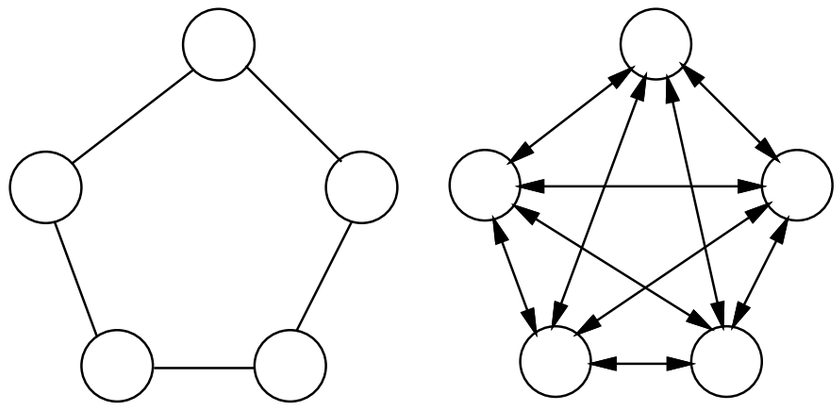
Merkmale eines eingeschworenen Teams

- Niedrige Fluktuationsrate
- Ausgeprägtes Identifikationsbewusstsein
- Freude an der Arbeit
- Bewusstsein einer Elitemannschaft

Egoless Programming

egoless programming – demokratische, dezentralisierte Organisation:

- Alle Mitglieder haben gleiche Befugnisse
- Teamleitung variiert mit Kompetenz
- Zielfindung durch Konsens
- *Code exchange*: Programme sind „Allgemeingut“ und werden zur Fehlersuche und Begutachtung ausgetauscht (vergl. *extreme programming*)



(a) Organisationsstruktur (b) Kommunikationsstruktur

Egoless Programming (2)

Vorteile

- geeignet für schwere Aufgaben
- einheitlicher Programmier- und Dokumentationsstil
- unempfindlich gegen Personalwechsel
- hohe Arbeitszufriedenheit

Nachteile

- Kommunikationsoverhead
- ineffizient bei Standardaufgaben
- häufig Terminprobleme
- hohe Risikotoleranz kann zum Misserfolg führen
- neue Ideen können unterdrückt werden

Effiziente Besprechungen

Viele Besprechungen kommen nur schlecht zum Ergebnis; sie gleichen mehr

Laberrunden: alle reden durcheinander

Selbstfindungssitzungen (ohne Ergebnis, „gut, dass wir darüber geredet haben“) oder

Haifischbecken (alle hacken aufeinander ein).

Tips für effiziente Besprechungen

Nur dann tagen, wenn es keine Alternative gibt. Gibt es nichts zu besprechen, muss man auch keine Zeit darauf verwenden.

Moderator bestimmen. Vor jeder Sitzung sollte ein Moderator bestimmt werden, der sich um Raum, Einladungen, Tagesordnung kümmert.
Der Moderator muss nicht der Gruppenleiter sein.

Pünktlich anfangen. Nicht warten, bis alle da sind – sonst gehen die ersten wieder und müssen später eingesammelt werden.

Störungen vermeiden. Die Sitzung sollte nicht gestört werden (auch nicht durch Zuspätkommende). Telefone und Handys ausschalten.

Tips für effiziente Besprechungen (2)

Tagesordnung. Vor jeder Sitzung muss klar sein, worüber geredet werden soll – damit sich die Teilnehmer vorbereiten können.

Eine generische Tagesordnung sieht so aus:

1. Protokoll der letzten Sitzung
2. Stand der Dinge
3. Ziele
4. Umsetzung
5. Nächste Schritte
6. Verschiedenes

Der Moderator stellt die Tagesordnung zu Beginn der Sitzung vor. Sie kann auch während der Sitzung geändert werden.

Tips für effiziente Besprechungen (3)

Stand der Dinge. Eine Sitzung beginnt gewöhnlich mit einer Zusammenfassung über den Stand der Dinge (z.B. die Ziele der letzten Sitzung und deren Umsetzung)

Ziele setzen. Die Ablaufstruktur einer Sitzung muss *zielorientiert* sein. (Für reine Informationen benötigt man keine Sitzung; an der Vergangenheit kann man nichts mehr ändern.)

Nach dem Stand der Dinge soll man deshalb *Ziele festlegen und erkennen* – möglichst spezifisch, messbar und auf ein konkretes Datum bezogen („Am 01.03. erwartet unser Kunde eine Präsentation“)

Tips für effiziente Besprechungen (4)

Problembearbeitung mit Methode. Jedes Problem (= jedes Ziel) lässt sich wie folgt behandeln:

1. Was ist das konkrete Problem?
2. Was sind die Ursachen für das Problem?
3. Was sind mögliche Lösungen?
4. Was ist die beste Lösung für das Problem?

Umsetzung planen. Wie kann man die beste Lösung erreichen? Hier ist Diskussion gefragt. Der Moderator achtet darauf, dass zielgerichtet diskutiert wird – also die geplanten Aktivitäten auch zu den Zielen passen.

Themen, die später auf der Tagesordnung stehen, kommen auch erst später dran.

Tips für effiziente Besprechungen (6)

Diskussionsregeln.

Die wichtigsten Regeln für gutes Diskutieren:

- Bis zum Schluss zuhören und andere aussprechen lassen
- Wortbeiträge nur mit vorheriger Wortmeldung
- Der Moderator erteilt und entzieht das Wort
- Einhaltung der Zeitvorgaben, z. B. Pausen
- Wir bleiben immer beim Thema
- Neue Themen und Gedanken werden notiert
- Fragen sind jederzeit zugelassen
- Fasse Dich kurz

Dies verlangt viel Disziplin, steigert aber die Effizienz.

Tips für effiziente Besprechungen (7)

Zusammenfassen. Treffen während der Diskussion verschiedene Ansichten aufeinander, ist es hilfreich und effizient, die Standpunkte *zusammenzufassen*.

Rechtzeitiges Zusammenfassen ist Aufgabe des *Moderators*.

Es ist aber auch eine gute Übung für die *Kontrahenten*, vor den eigenen Argumenten die des Vorredners zusammenzufassen. *(und ggf. neu zu interpretieren :-)*

Tips für effiziente Besprechungen (8)

Nächste Schritte. Hier werden wiederum *konkrete, messbare* Aktivitäten entschieden, die später (z.B. in der nächsten Sitzung) geprüft werden können.

Ergebnisse niederschreiben. Vergessen Sie nicht, die Ergebnisse (= Stand der Dinge und nächste Schritte) in einem *Protokoll* festzuhalten.

Dies ist gewöhnlich Aufgabe des Protokollführers (der auch identisch mit dem Moderator sein kann).

Der Moderator sorgt dafür, dass alle Teilnehmer das Protokoll erhalten (um ggf. später Einspruch einzulegen).

Kreativität fördern

Viele Aktivitäten der Software-Entwicklung erfordern ein hohes Maß an Kreativität – insbesondere der *Entwurf* und die *Fehlerbeseitigung.*

Kreativität ist die Fähigkeit, Wissen und Erfahrung aus verschiedenen Bereichen zu neuen Lösungen und Ideen zu verschmelzen, wobei verfestigte Denkmuster überwunden werden.

Klassisches Brainstorming

Brainstorming ist eine spezielle Form einer Gruppensitzung, mit dem Ziel, Ideen und Gedanken einer Gruppe frei fließen zu lassen und sie zu neuem zu kombinieren.

Brainstorming – Regeln

1. *Freies und ungehemmtes Aussprechen von Gedanken.* Auch sinnlos erscheinende und phantastische Einfälle sind erwünscht, da sie andere Teilnehmer inspirieren können.

   Alle Vorschläge an Tafel oder Flipchart schreiben.

2. Die gemachten Vorschläge sind als *Anregungen* aufzunehmen und assoziativ weiterzuentwickeln.

   Voraussetzung: Zuhören und inhaltlich offen sein.

3. *Kritik und Bewertung* sind während der Sitzung verboten.

   Keine *Killerphrasen* wie „Das haben wir noch nie gemacht“, „Das hat noch keiner geschafft“.

4. *Quantität* geht vor Qualität; Vernunft und Logik sind nicht gefragt.

Brainstorming – Voraussetzungen

- Erfahrener Moderator
- Disziplinierte Teilnehmer
- 4–7 Teilnehmer
- Maximale Dauer: 30 Minuten

Alternative: *Brainwriting* (Ideen werden auf Karten geschrieben, die zum Nachbarn weitergereicht werden)

Weniger spontan, aber geringeres Risiko, dass rhetorisch begabte Teilnehmer dominieren.

Zusammenfassung Teamarbeit

- Gut gestaltete Arbeitsplätze fördern die Produktivität
- Es gibt zahlreiche Faktoren, die die *Teambildung* fördern (Messbare Erfolge, Qualitätskult, …)
- Teams sollten möglichst divers zusammengesetzt sein (Motivation, Hintergrund)
- Besprechungen sollen *effizient* gestaltet werden
- Brainstorming fördert die Kreativität

Technische Aspekte der Teamarbeit

Neben den *organisatorischen* Aspekten gibt es auch *technische* Hilfsmittel, die die Teamarbeit unterstützen.

Wir betrachten

- Konfigurationsmanagement
- Problem Tracking

Konfigurationsmanagement

Ein großes Software-Produkt besteht aus

- Tausenden von Komponenten,
- die von Hunderten oder gar Tausenden Personen entwickelt und gewartet werden,
- die oftmals auch noch auf viele Orte verteilt sind,

und an all diesen Komponenten werden von all diesen Personen *Änderungen* vorgenommen.

Die Aufgabe, solche Änderungen zu organisieren und zu kontrollieren, heißt *Software-Konfigurationsmanagement*.

Konfigurationsmanagement: Ursprung

Konfigurationsmanagement: ursprünglich von der US-Raumfahrtindustrie in den 50er Jahren eingeführt

Problem zu dieser Zeit: Raumfahrzeuge unterlagen während ihrer Entwicklung zahlreichen undokumentierten *Änderungen.*

Erschwerend: Raumfahrzeuge wurden im Test normalerweise *vernichtet*

Folge: Nach einem erfolgreichen Test waren Hersteller nicht in der Lage waren, eine Serienfertigung aufzunehmen oder auch nur einen Nachbau durchzuführen.

Konfigurationsmanagement soll solchen *Informationsverlust* verhindern.

Ziele des Konfigurationsmanagements

Rekonstruktion: *Konfigurationen* müssen zu jedem Zeitpunkt wiederhergestellt werden können; eine Konfiguration ist dabei eine Menge von Software-Komponenten in bestimmten Versionen.

Koordination: Es muß sichergestellt werden, daß Änderungen von Entwicklern nicht versehentlich verlorengehen. Dies bedeutet insbesondere das Auflösen von *Konflikten.*

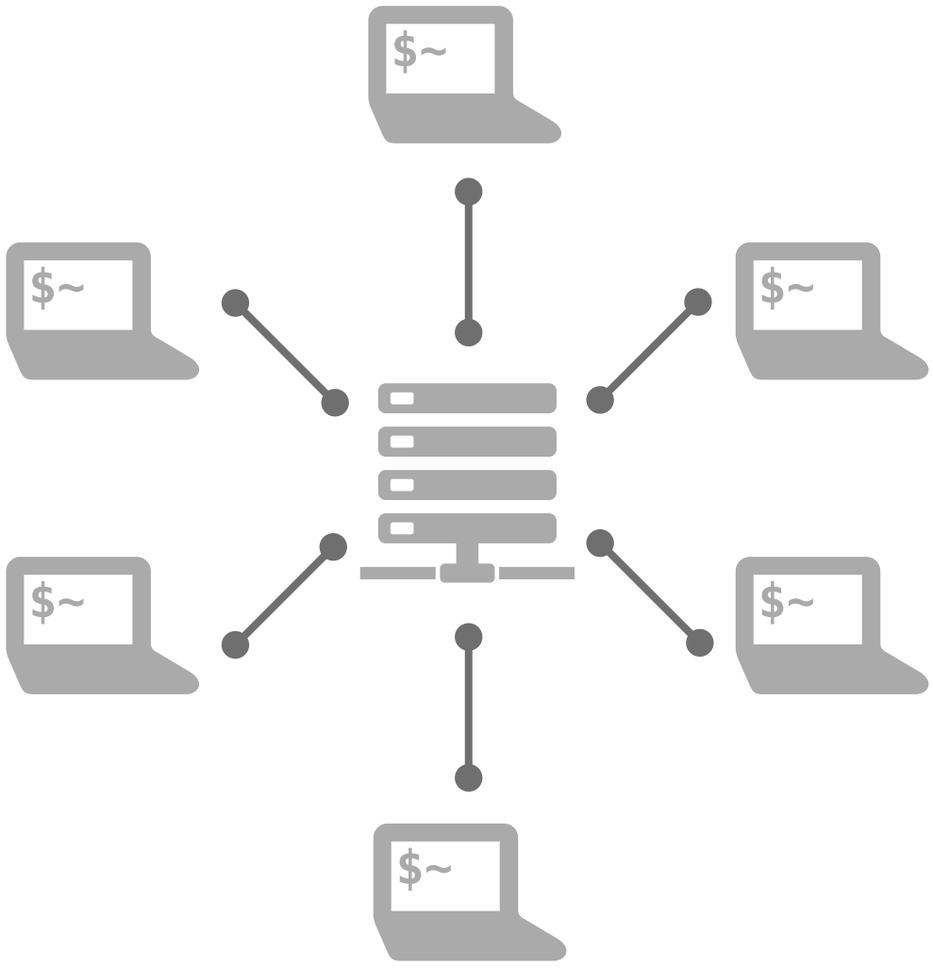
Identifikation: Es muß stets möglich sein, einzelne Versionen und Komponenten eindeutig zu identifizieren, um die jeweils angewandten Änderungen erkennen zu können.

Software-Systeme verwalten mit Git

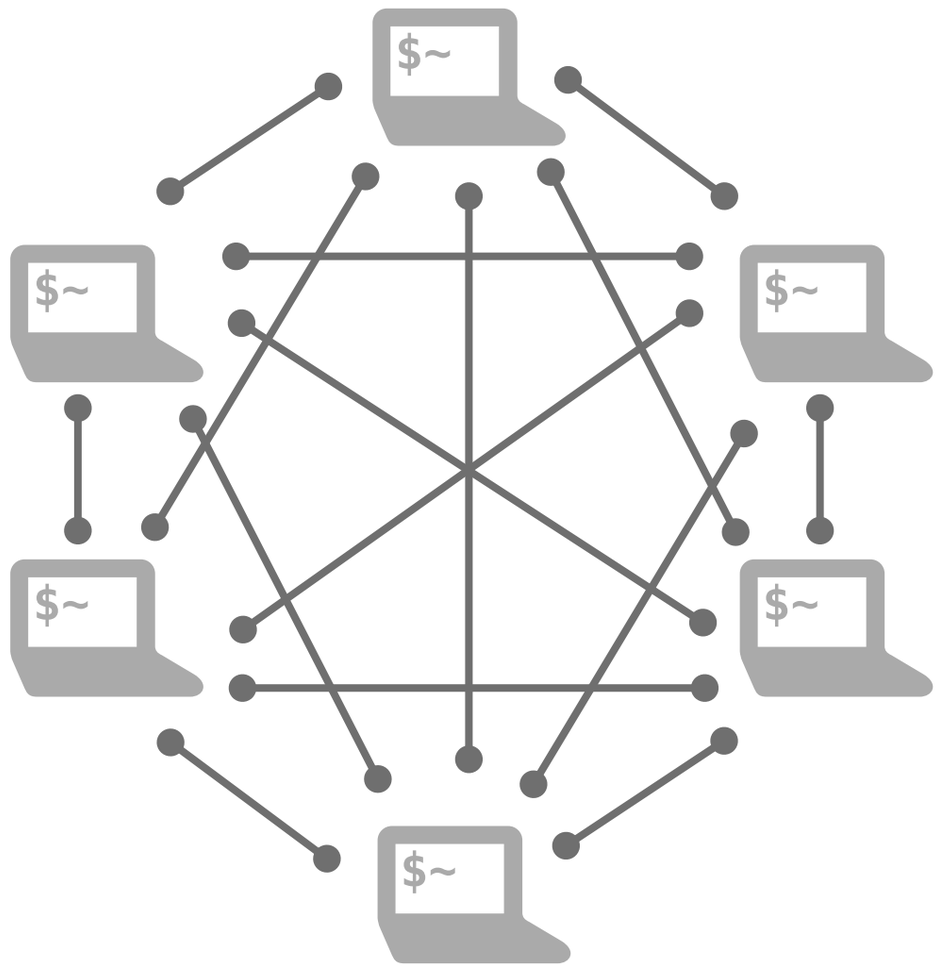
Entworfen und entwickelt von Linus Torvalds, um die Entwicklung des Linux Kernels zu unterstützen

Standard im Software-Praktikum

Zentral vs. Dezentral

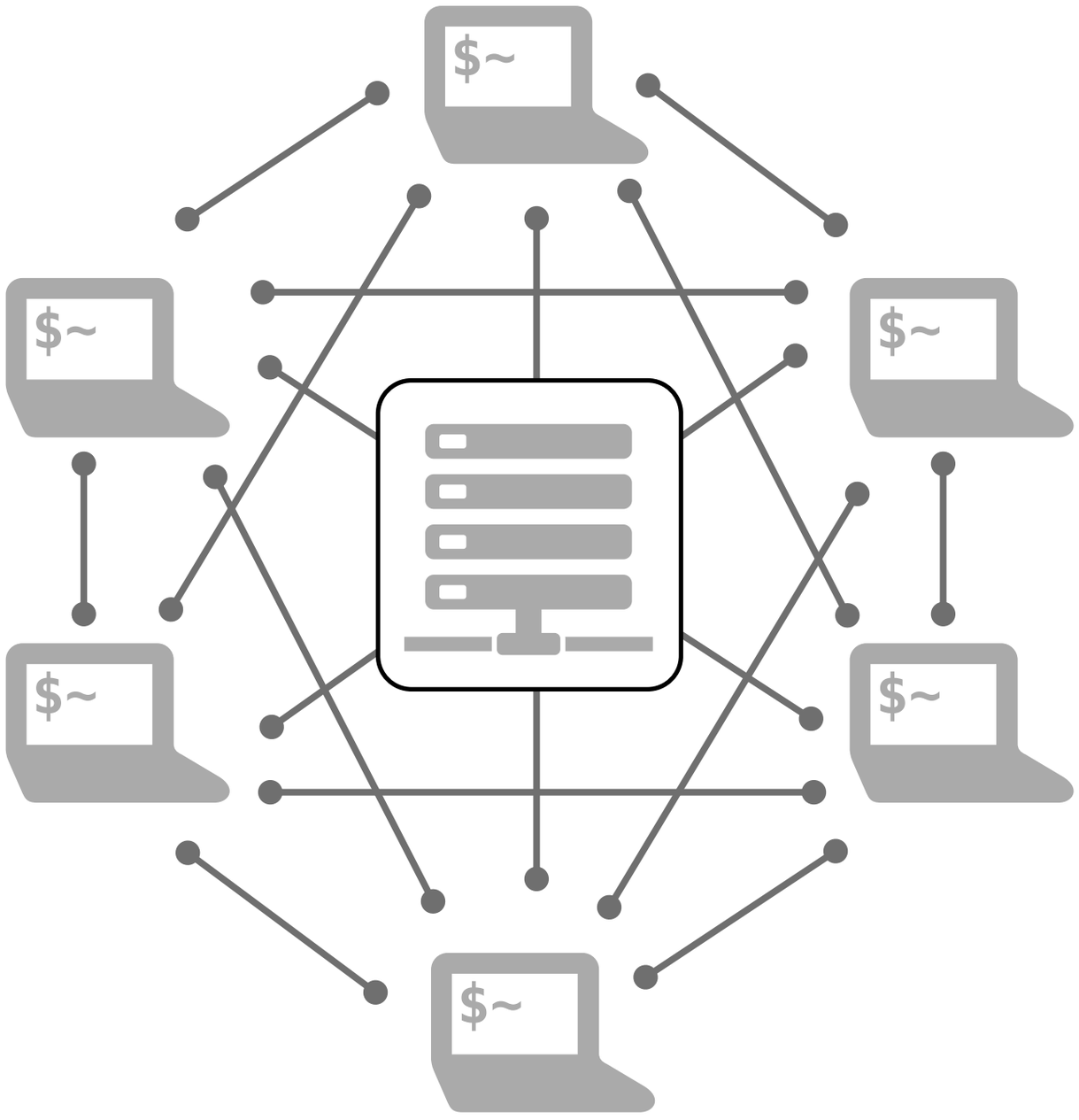


Subversion, Perforce, CVS, ...



Git, Mercurial, Darcs, ...

Die Infrastruktur im Software-Praktikum



Dezentrales Versionsmanagement

Keinen zentraler Server notwendig, der die Daten vorhält

Jeder Benutzer hält sich lokal die komplette Entwicklungs-Historie vor

Verliert ein Benutzer die Daten, kann er das komplette Archiv von einem anderen Benutzer kopieren

Offline-Arbeiten möglich

Besonders effiziente Speicherung der Daten notwendig

Git-Repository

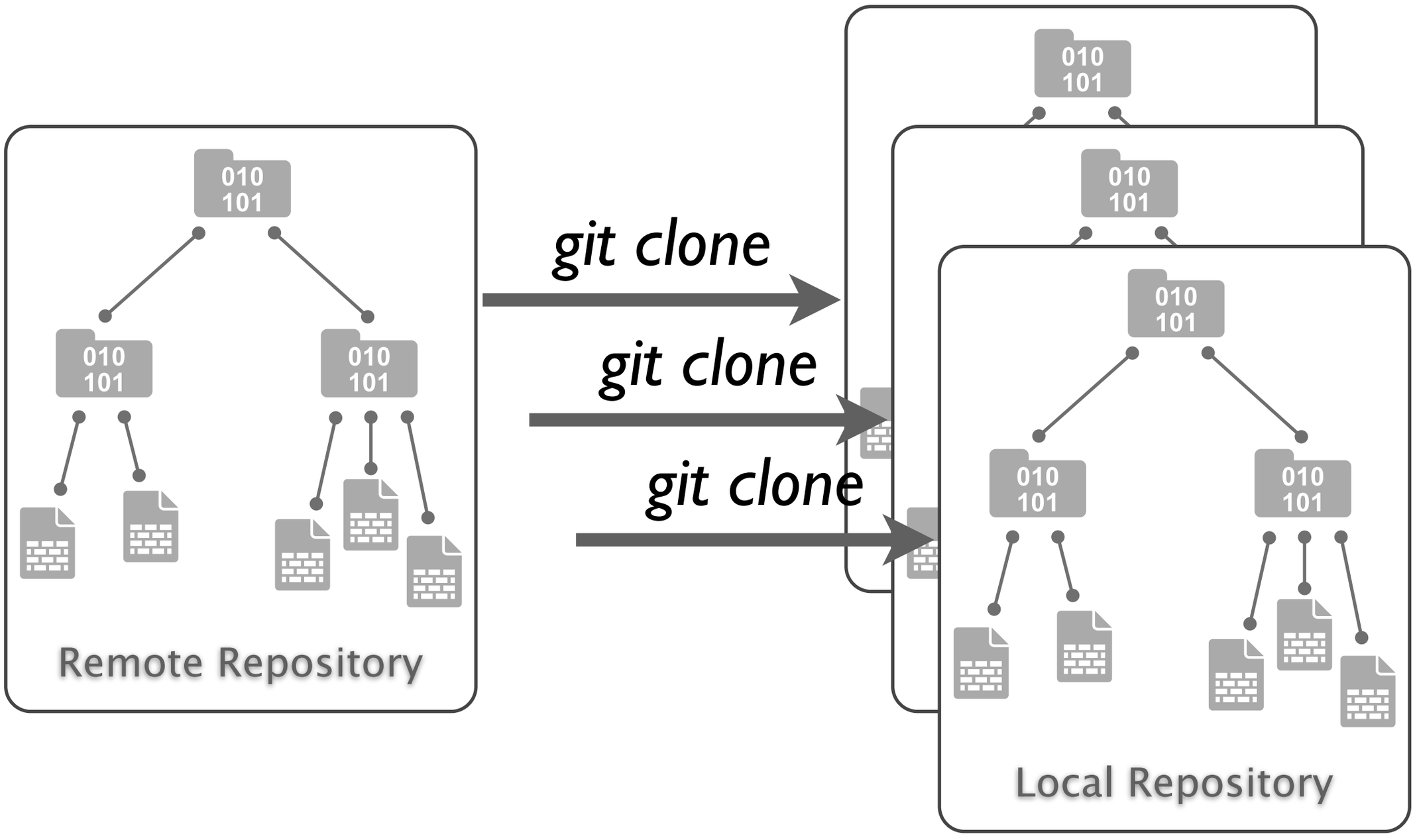
Git speichert auf jedem Rechner ein komplettes *Repository*

Repository = Verzeichnis der kompletten Projekt-Historie

Will man z.B. auf die Version einer Datei oder eines Verzeichnisses vom 27. August zugreifen, ist dies problemlos möglich

Auf diese Weise kann der Zustand von jedem beliebigen Zeitpunkt wiederhergestellt werden

Projekt einrichten: **clone**

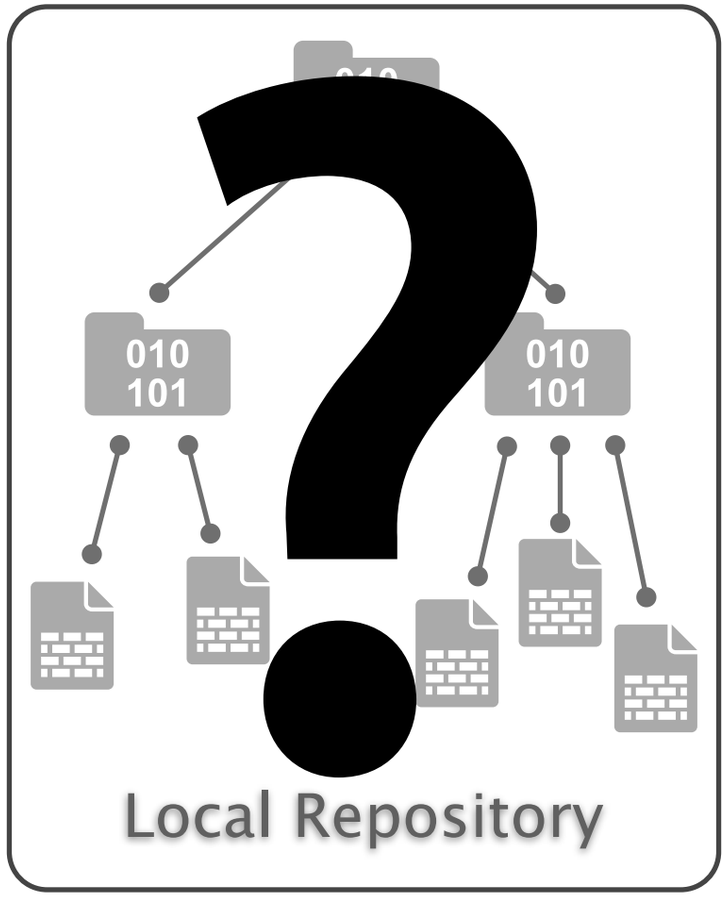


Projekt einrichten: **clone** (2)

Beispiel: Klonen des SoPra Repositories

```
$ git clone sopra:gruppe13 sopra
Cloning into sopra...
remote: Counting objects: 2171, done.
remote: Compressing objects: 100% (1104/1104), done.
remote: Total 2171 (delta 459), reused 1092 (delta 158)
Receiving objects: 100% (2171/2171), 4.65 MiB, done.
Resolving deltas: 100% (459/459), done.
$ _
```

Status anzeigen: **status**

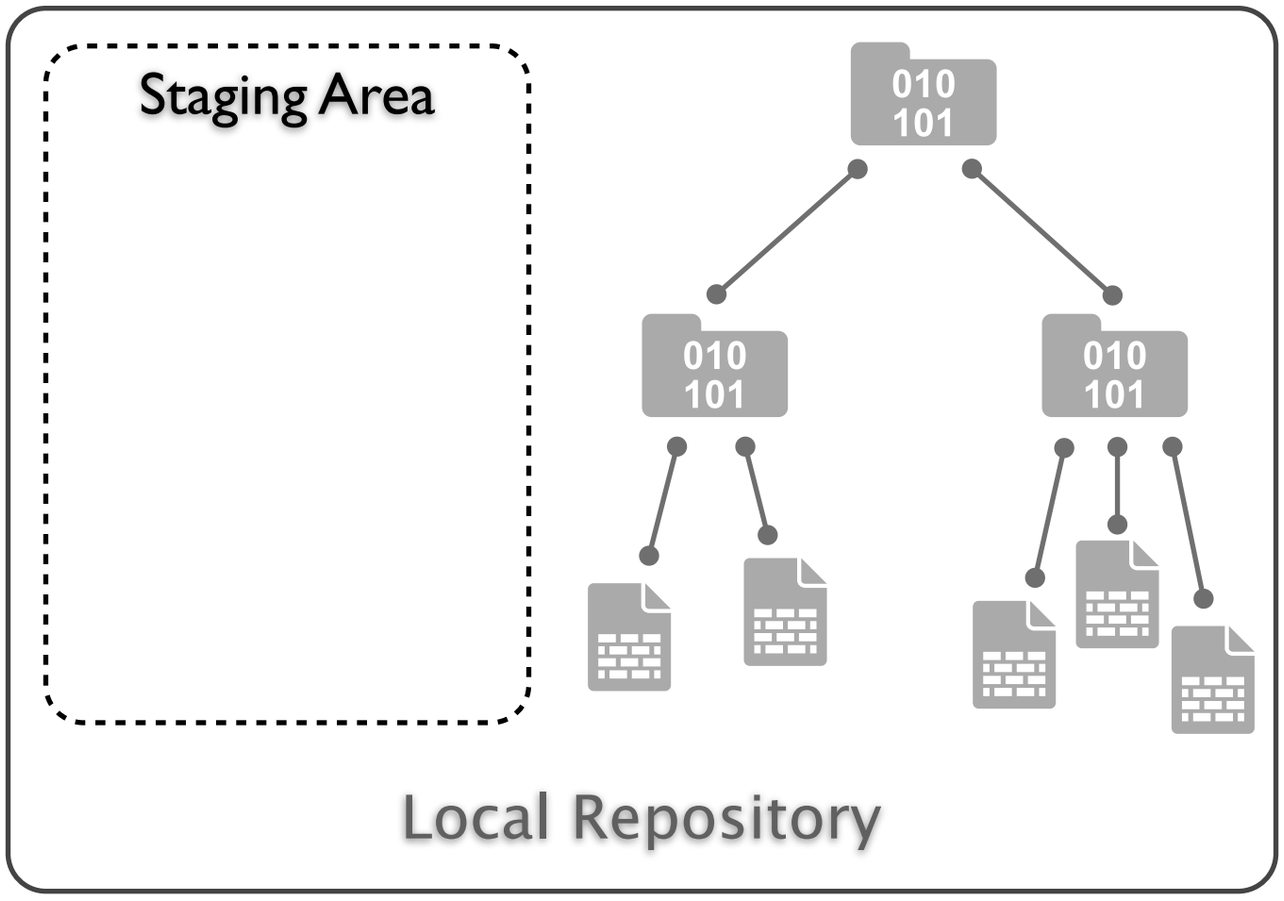


Status anzeigen: **status** (2)

Beispiel: Status des eben geklonten Repositories

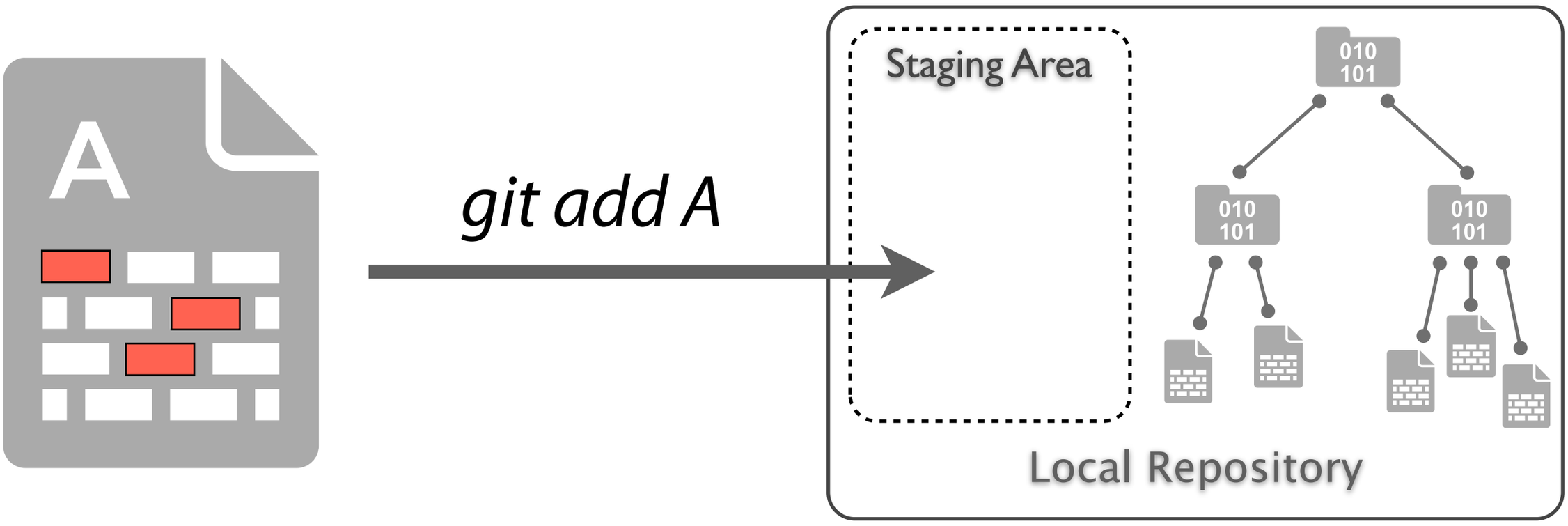
```
$ git status
# On branch master
nothing to commit (working directory clean)
$ _
```

Änderungen erfassen



In der *Staging Area* (Bereitstellungsraum)
werden Änderungen zum Übertragen vorgemerkt

Änderungen erfassen: **add**



Änderungen erfassen: **add** (2)

Schritt 1: Status prüfen

```
$ 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 directory)
#
# modified:   implementation/pom.xml
#
no changes added to commit
    (use "git add" and/or "git commit -a")
$ _
```

Änderungen erfassen: **add** (3)

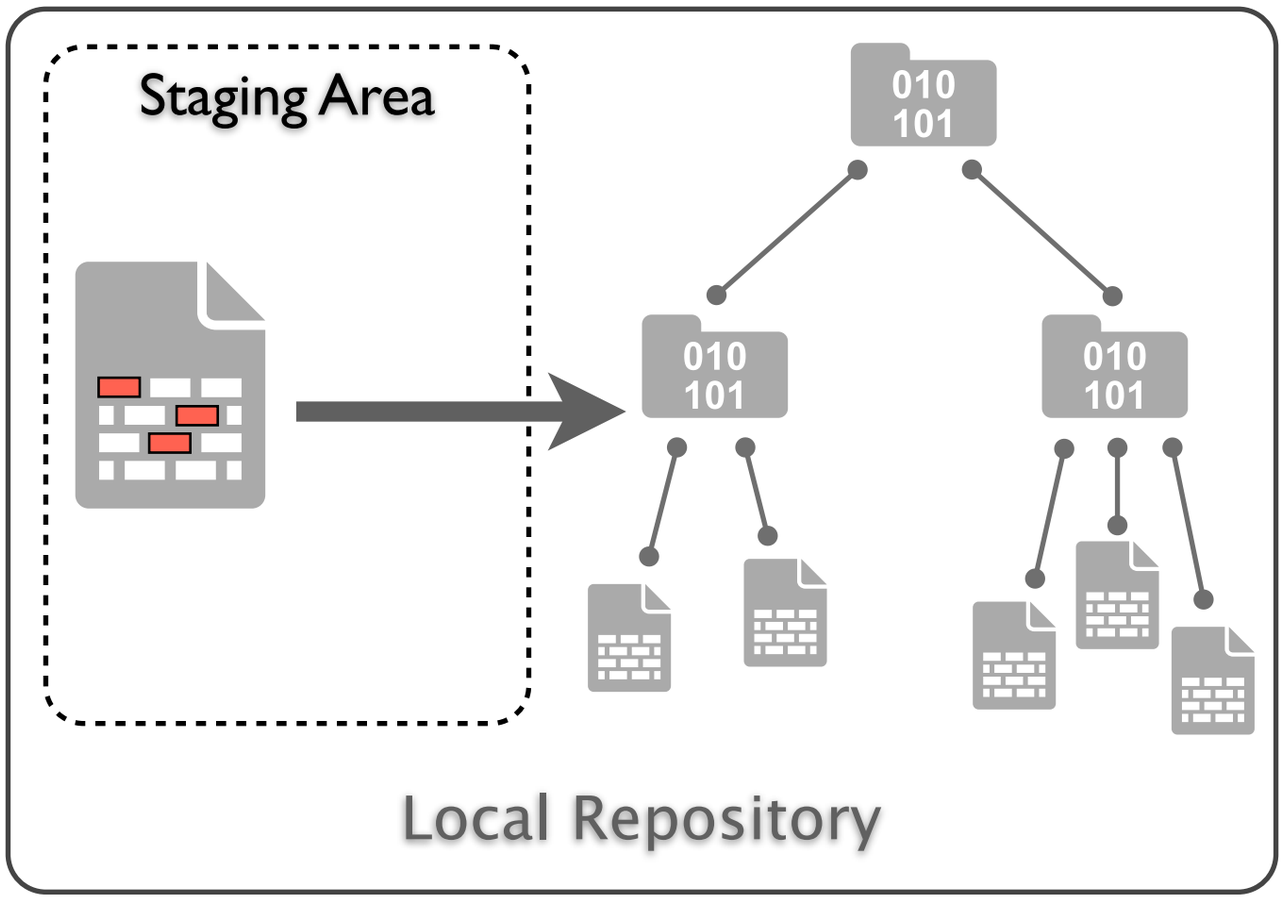
Schritt 2: Änderung zum Übertragen vormerken

```
$ git add implementation/pom.xml
$ _
```

Schritt 3: Status prüfen

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   implementation/pom.xml
#
$ _
```

Änderungen lokal übertragen: **commit**



git commit -m “pom.xml geändert”

Änderungen lokal übertragen: **commit** (2)

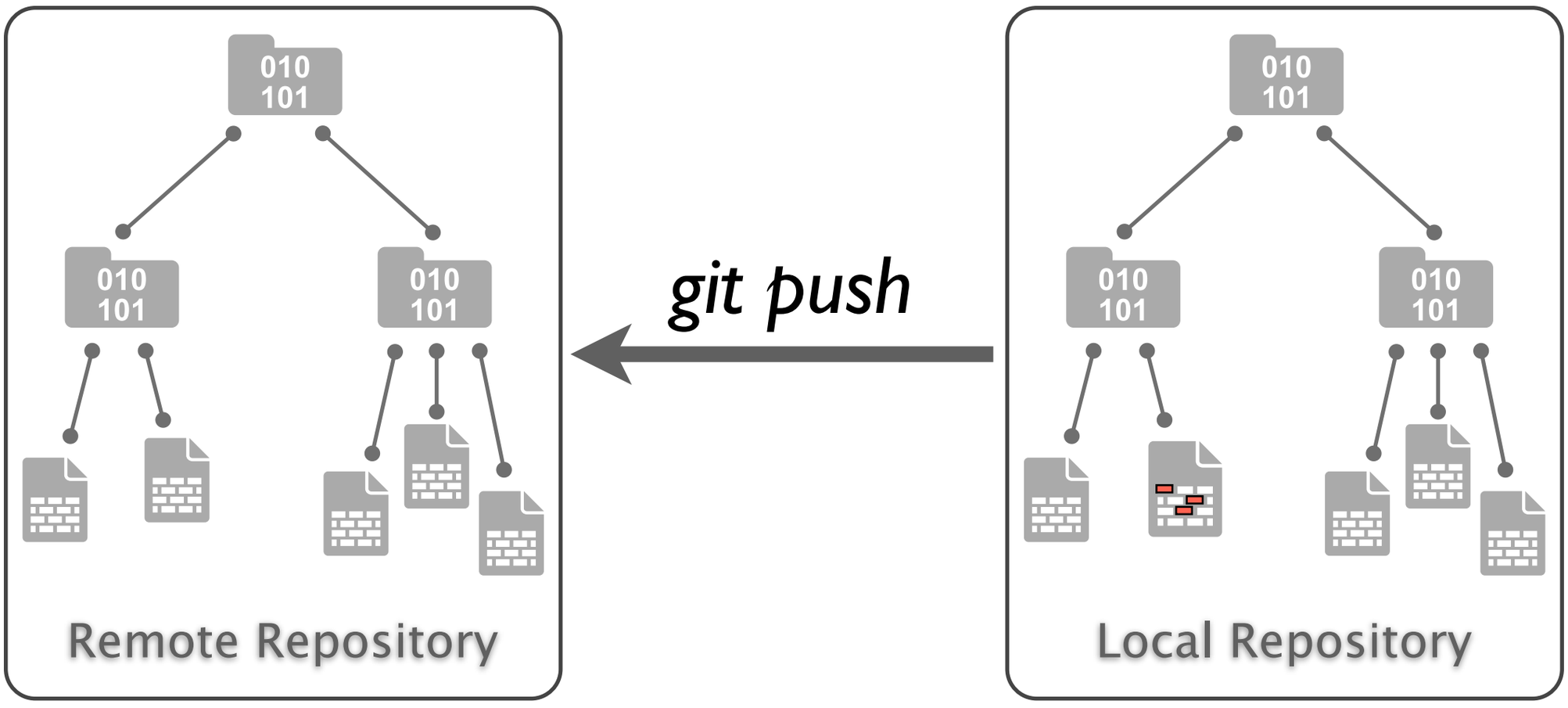
Schritt 1: Änderung übertragen

```
$ git commit –m "pom.xml geändert"
[master e480281] pom.xml geändert
 1 files changed, 1 insertions(+), 1 deletions(–)
$ _
```

Schritt 2: Status prüfen

```
$ git status
# On branch master
#  Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
$ _
```

Änderungen zum Server übertragen: **push**

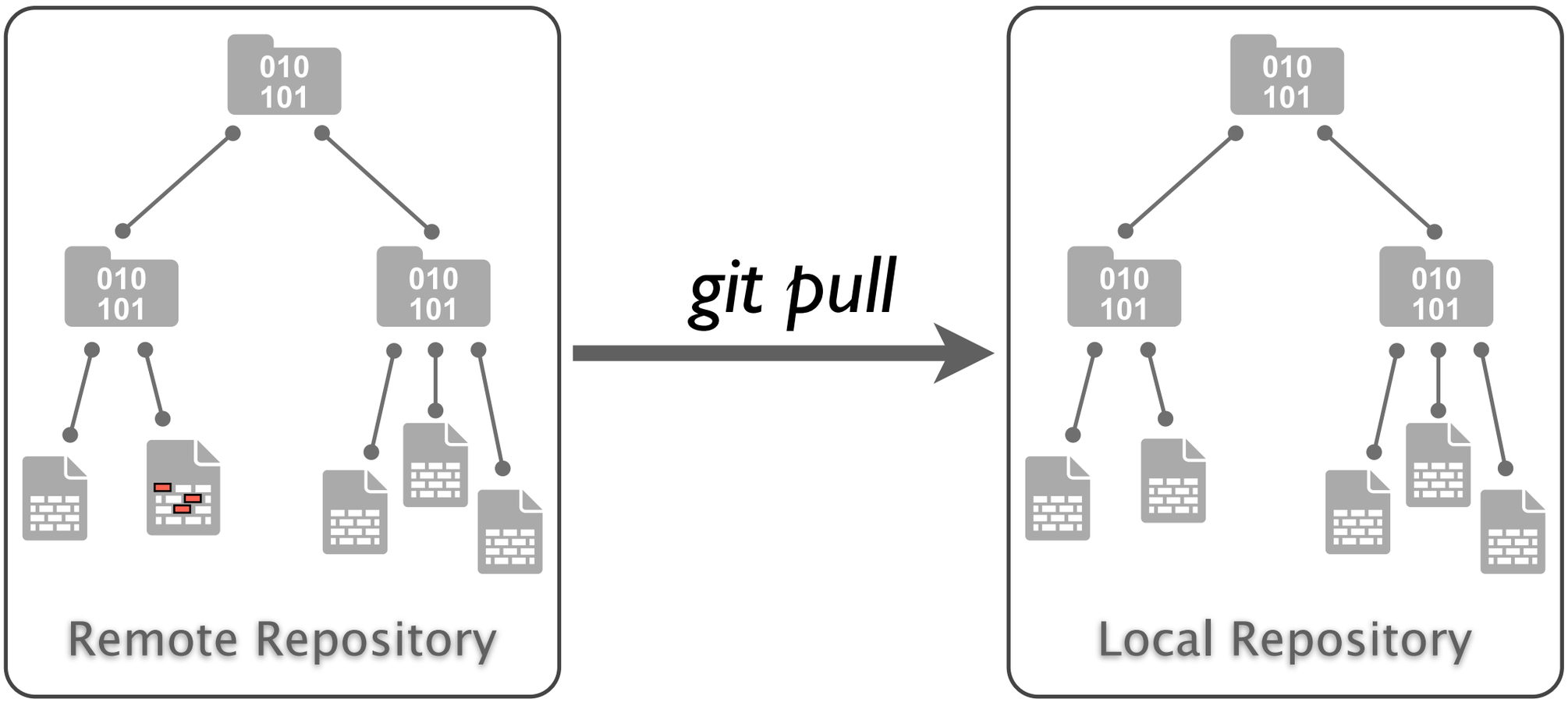


Änderungen zum Server übertragen: **push** (2)

Schritt 1: Änderung übertragen

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 369 bytes, done.
Total 4 (delta 2), reused 0 (delta 0)
To sopra:gruppe13
   be62268..65ce354  master -> master
$ _
```

Änderungen vom Server übernehmen: **pull**



Änderungen vom Server übernehmen: **pull** (2)

```
$ git pull
remote: Counting objects: 93, done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 43 (delta 15), reused 27 (delta 9)
Unpacking objects: 100% (43/43), done.
From sopra:code
   65ce354..51ed0b6  master     -> origin/master
First, rewinding head to replay your work on top of it...
Fast-forwarded master to
51ed0b62727d28ddb3a73283d228fe713cf5c147.
$ _
```

Integration / Merge

Sind mehrere Änderungen gleichzeitig vorgenommen worden, kommt es zum *Merge* (mischen).

Vor jedem *push* überprüft Git, ob die Dateien im Ziel-Repository unverändert sind:

- Wenn ja, gibt es kein Problem und die Änderungen werden übertragen.
- Wenn nein, muss der Benutzer zuerst die Änderungen übernehmen.

Integration / Merge (2)

git pull übernimmt die Änderungen aus einem entfernten Repository in das lokale Repository.

Hat sich sowohl die Version im entfernten Repository verändert als auch die lokale, kommt es zu *Konflikten*.

Konflikte können entweder automatisch aufgelöst werden, oder manuelle Interaktion erfordern.

```
Terminplan:
Das Dokument muss bis zum
01.09. abgegeben werden.
```

Nutzer A

```
Terminplan:
Das Dokument muss bis zum
01.10. abgegeben werden.
```

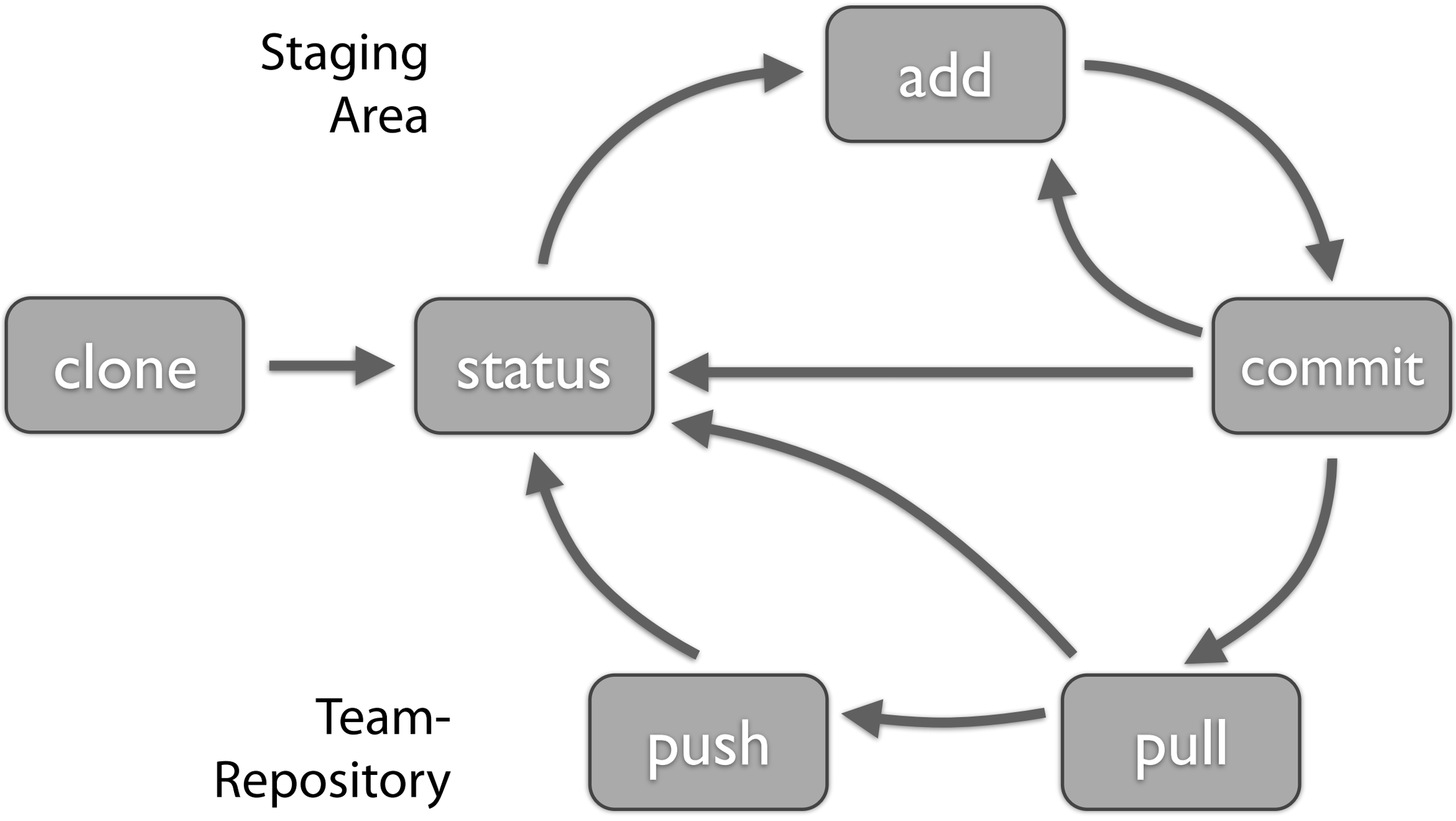
Nutzer B

```
Terminplan:
Wir müssen das Dokument bis
zum 01.09. abgeben.
```

Konflikt

```
Terminplan:
<<< Nutzer A
Das Dokument muss bis zum
01.10. abgegeben werden.
=== Nutzer B
Wir müssen das Dokument bis
zum 01.09. abgeben.
>>>
```

Übersicht: Typischer Arbeitsablauf mit Git



Problem Tracking

Ein Anwender (oder ein Entwickler) hat ein Problem. Wie kann der Entwickler das Problem reproduzieren, um es zu beheben?

Lösung – *Alle relevanten Informationen* über das Problem erheben.

Relevante Informationen

Explizite *Richtlinien* aufsetzen, was erhoben wird. Etwa:

- *Produkt-Version*
- *Einsatzumgebung* (z.B. Betriebssystem)
- *Problem-Geschichte*
- Beschreibung des *erwarteten* Verhaltens
- Beschreibung des *beobachteten* Verhaltens (typischerweise mit Verweis auf das Pflichtenheft)
- Wünschenswert: *Testfall*, der das Problem automatisch reproduziert

Diese Informationen enden in einem *Problembericht* (problem report, bug report)

Probleme verfolgen

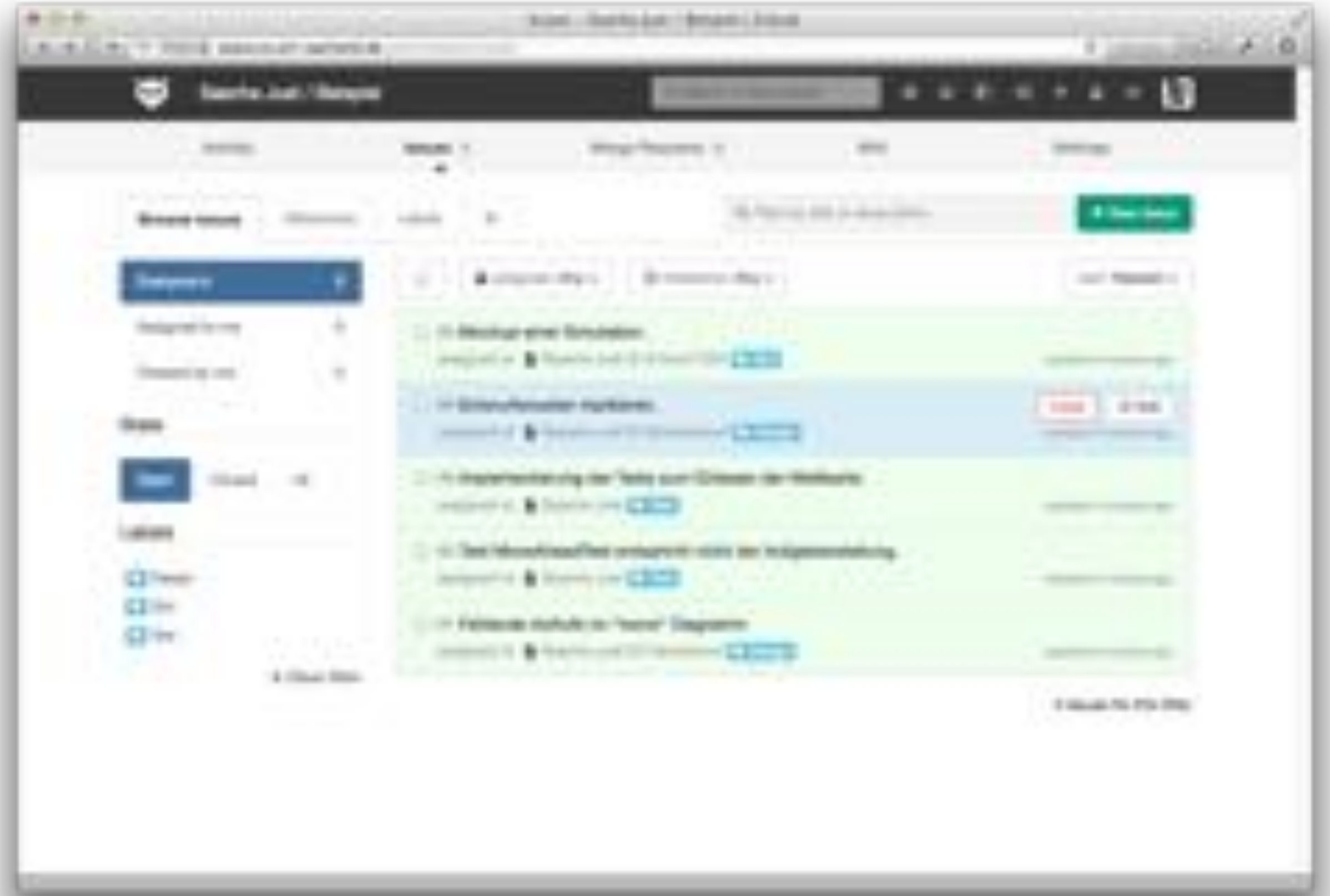
Ist ein Problembericht erhoben, muss er gespeichert werden.

Mögliche Speicherorte:

Ein ‘PROBLEMS.txt’-Dokument. Einfach zu installieren, skaliert aber nicht.

Eine *Problem-Datenbank*. Speichert alle Problemberichte.


Gitlab



Test MoveAheadTest entspricht nicht der Aufgabenstellung.

Problem-Identifikation

Jedes Problem hat einen eindeutigen *Bezeichner* (auch bekannt als *Ticket number*, *PR number* oder *CR number* – von PR = problem report, CR = change request).

Entwickler können sich darauf beziehen in

- E-mails
- Änderungsmeldungen
- Status-Berichten

Schwere des Problems

Blocker Blocks development and/or testing work. This highest level of severity is also known as *Showstopper.*

Critical Crashes, loss of data, severe memory leak.

Major Major loss of function.

Normal This is the “standard” problem.

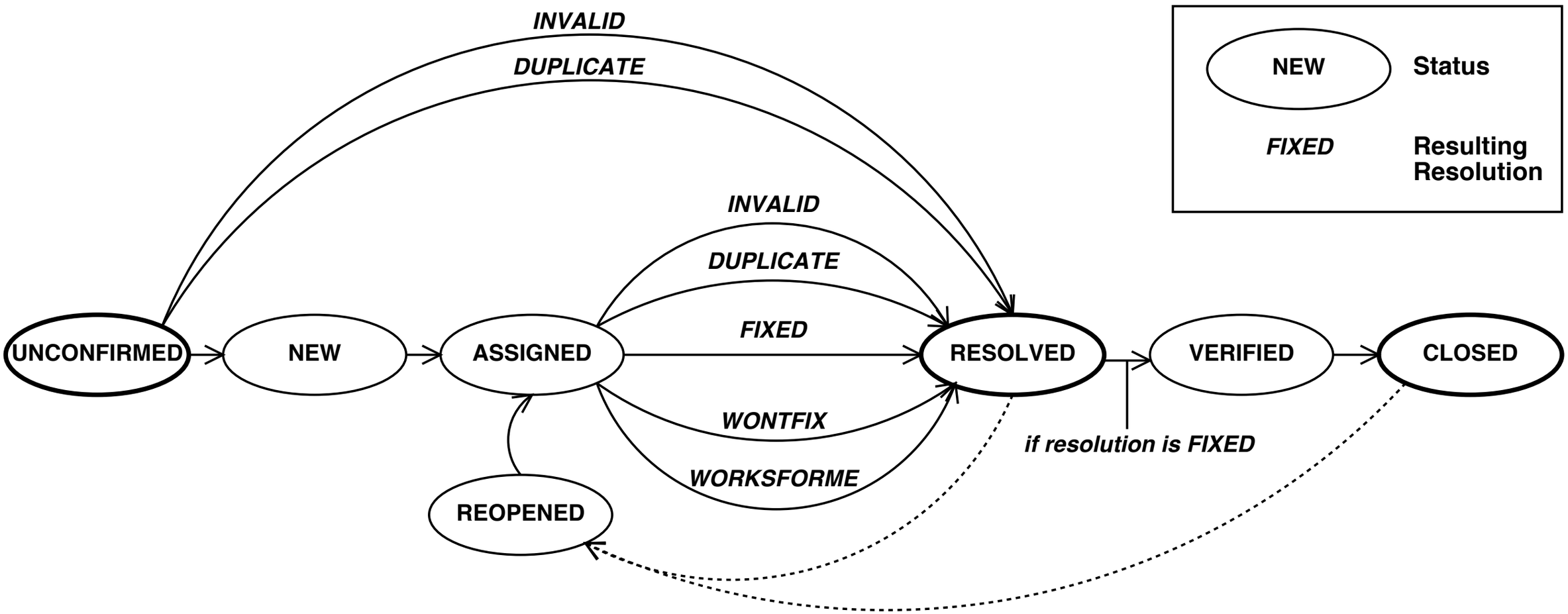
Minor Minor loss of function, or other problem where an easy workaround is present.

Trivial Cosmetic problem like misspelled words or misaligned text.

Enhancement Request for enhancement.

Problem-Lebenszyklus

am Beispiel Mozilla



Problem-Status und Auflösung

FIXED This problem has been fixed and tested.

INVALID The problem described is not a problem.

WONTFIX The problem described is a problem which will never be fixed.

LATER The problem will be fixed in a later version.

REMIND Like LATER, but might still be fixed earlier.

DUPLICATE The problem is a duplicate of an existing problem.

WORKSFORME All attempts at reproducing this problem were futile. If more information appears later, the problem will be re-assigned.

Vorrang des Problems

Der *Vorrang* (Priority) bestimmt, in welcher Reihenfolge Probleme bearbeitet werden.

Der Vorrang ist das wichtigste Mittel, um die Entwicklung zu steuern!

In der Praxis gibt es eigene Komitees (aus 3–5 Entwicklern/Testern/Managern), die einzelnen Problemen einen Vorrang einräumen.

Nicht Spielprobleme, sondern Kernprobleme lösen!

Problembezogene Teamarbeit

Grundidee: *Das Team arbeitet, bis alle Probleme gelöst sind.*

Das Projekt beginnt mit dem Ursprungsproblem

Das Produkt ist nicht da

Man teilt dieses Problem auf und weist Aufgaben zu.

Probleme können nicht nur im Produkt auftreten, sondern auch in allen Dokumenten – also von Beginn an.

Der Status *aller Probleme* wird über die Problem-Datenbank verwaltet (und ist jederzeit einsehbar).

Ist das letzte Problem gelöst, ist die Arbeit beendet :-)

Und nun: Viel Spaß bei der Teamarbeit!