Häufige Anfängerfehler und Denkmodelle

Warum Denkmodelle wichtiger sind als Befehle

Viele Git-Probleme entstehen nicht, weil ein Befehl unbekannt ist, sondern weil das zugrunde liegende Modell fehlt. Wer Git nur als Sammlung von Kommandos lernt, merkt sich etwa:

```
git add .
git commit -m "Änderungen"
git push
```

Das funktioniert in einfachen Situationen. Sobald aber ein Konflikt, ein falscher Commit oder eine Änderung auf dem falschen Branch auftaucht, fehlt die Orientierung.

Ein belastbares Verständnis beginnt mit dieser Grundidee:

> Git verwaltet nicht einfach Dateien. Git verwaltet **Versionen von Projektzuständen** und die Beziehungen zwischen diesen Versionen.

Die folgenden Denkmodelle helfen dabei, Git-Situationen ruhig zu analysieren, statt Befehle auf Verdacht auszuprobieren.

Das wichtigste Modell: Git hat mehrere Zustände

Eine Datei kann in Git nicht nur „da“ oder „gespeichert“ sein. Für die tägliche Arbeit sind drei Bereiche entscheidend:

1. **Arbeitsverzeichnis**
   Die Dateien, die gerade im Editor liegen und bearbeitet werden.
2. **Staging Area** oder **Index**
   Die Auswahl der Änderungen, die im *nächsten* Commit landen soll.
3. **Lokales Repository**
   Die bereits erstellte, lokale Commit-Historie.

Später kommt häufig noch ein vierter Bezugspunkt hinzu:

4. **Entferntes Repository**
   Beispielsweise ein Repository auf GitHub.

Der typische Weg einer Änderung sieht so aus:

```
Datei bearbeiten
      ↓
Arbeitsverzeichnis
      ↓ git add
Staging Area
      ↓ git commit
lokale Historie
      ↓ git push
GitHub oder anderer Remote
```

Die zentrale Frage bei jedem Problem

Bevor du einen Git-Befehl eingibst, frage dich:

> **Wo befindet sich die Änderung gerade – und wohin soll sie?**

Diese Frage ist oft hilfreicher als die Suche nach einem vermeintlichen „Rückgängig-Befehl“.

Situation	Die Änderung befindet sich gerade	Typisches Ziel
Datei wurde bearbeitet, aber noch nicht vorgemerkt	Arbeitsverzeichnis	Staging Area oder verwerfen
Falsche Datei wurde für den Commit ausgewählt	Staging Area	Zurück ins Arbeitsverzeichnis

Situation	Die Änderung befindet sich gerade	Typisches Ziel
Commit wurde lokal erstellt, aber noch nicht veröffentlicht	Lokales Repository	Commit korrigieren oder Historie anpassen
Commit wurde bereits auf GitHub veröffentlicht	Remote und lokale Historie	Nachvollziehbar korrigieren, nicht unbedacht umschreiben

Der Befehl `git status` beantwortet genau diese Zustandsfrage und sollte daher zum Standard werden.

```
git status
```

Denkmodell: Ein Commit ist ein Snapshot, kein Änderungsprotokoll

Anfänger stellen sich einen Commit oft als Liste einzelner Änderungen vor, etwa:

> „In diesem Commit wurde Zeile 18 in Datei A angepasst.“

Das ist als Anzeige nicht falsch, aber als Modell unvollständig. Git speichert bei einem Commit grundsätzlich einen **vollständigen Snapshot des vorgemerkten Projektzustands**.

Ein Commit bedeutet eher:

> „So sah das gesamte Projekt zu diesem Zeitpunkt aus.“

Git kann daraus sehr effizient die Unterschiede zwischen zwei Snapshots berechnen. Deshalb wirken Commits in Tools und Diffs häufig wie Änderungslisten. Intern und gedanklich ist aber der Snapshot entscheidend.

Konsequenz für die Praxis

Ein Commit sollte einen **in sich stimmigen Zustand** festhalten:

- Die Anwendung lässt sich idealerweise ausführen.

- Tests sind möglichst grün.
- Die Änderung erfüllt genau einen nachvollziehbaren Zweck.
- Andere Teammitglieder können verstehen, *warum* der Snapshot entstanden ist.

Schlecht wäre ein Commit wie:

```
WIP
```

Oder:

```
Neue Funktion, CSS angepasst, Debug-Ausgaben, Composer aktualisiert
```

Besser wäre eine fachlich klare Trennung:

```
feat: E-Mail-Adresse bei der Registrierung validieren
```

```
test: Validierung ungültiger E-Mail-Adressen abdecken
```

```
style: Formularabstände auf der Registrierungsseite vereinheitlichen
```

Ein guter Commit ist keine Momentaufnahme der eigenen Arbeitszeit. Er ist eine verständliche Einheit in der gemeinsamen Projektgeschichte.

Denkmodell: Git speichert lokale Arbeit nicht automatisch auf GitHub

Ein besonders häufiger Irrtum lautet:

> „Ich habe einen Commit erstellt, also ist die Änderung jetzt auf GitHub.“

Das stimmt nicht. Ein Commit wird zunächst nur im **lokalen Repository** erstellt. Erst ein Push überträgt ihn an ein entferntes Repository.

```
git commit  →  lokale Historie
git push    →  entfernte Historie
```

Umgekehrt lädt `git pull` nicht einfach „alles Neue“ herunter. Es holt Änderungen vom Remote und integriert sie in den aktuellen lokalen Branch.

Vier Fragen zur Einordnung

Wenn du nicht sicher bist, ob etwas gesichert oder veröffentlicht ist, unterscheide präzise:

1. Wurde die Datei gespeichert?
2. Wurde die Änderung mit `git add` vorgemerkt?
3. Wurde ein lokaler Commit erstellt?
4. Wurde dieser Commit zu GitHub übertragen?

Diese vier Schritte sind unabhängig. Eine gespeicherte Datei ist noch kein Commit; ein lokaler Commit ist noch keine Veröffentlichung.

> **Merksatz:** Ein Commit sichert eine Version lokal. Ein Push teilt diese Version mit einem Remote.

Denkmodell: Git und GitHub sind verschiedene Systeme

Git und GitHub werden im Alltag oft zusammen genannt, erfüllen aber unterschiedliche Aufgaben.

- **Git** ist das lokale Versionskontrollsystem. Es verwaltet Commits, Branches, Historie und Änderungen auf dem eigenen Rechner.
- **GitHub** ist eine Plattform für gehostete Git-Repositories und Zusammenarbeit. Sie ergänzt Git unter anderem um Pull Requests, Issues, Reviews, Actions und Zugriffsverwaltung.

Du kannst Git vollständig ohne GitHub verwenden:

```
git init
git add .
git commit -m "Initialer Stand"
```

Und GitHub kann ohne ein lokales Repository kaum sinnvoll genutzt werden: GitHub speichert zwar Git-Repositories, ersetzt aber nicht die lokale Git-Arbeit.

Typischer Denkfehler

> „GitHub hat meinen Code verloren.“

Oft liegt die Änderung nur lokal vor, wurde auf einem anderen Branch erstellt oder noch nicht gepusht. Bevor du von Datenverlust ausgehst, prüfe deshalb:

```
git status
git log --oneline
git branch
```

Erst danach sollte nach Problemen bei GitHub, dem Netzwerk oder Berechtigungen gesucht werden.

Denkmodell: Ein Branch ist keine Projektkopie

Anfänger behandeln Branches häufig wie getrennte Projektordner:

> „Wenn ich einen Branch erstelle, wird das Projekt dupliziert.“

Ein Branch ist in Git vor allem ein **beweglicher Name für einen Commit**. Er markiert den aktuellen Endpunkt einer Entwicklungslinie.

```
main     → Commit A
feature  → Commit A
```

Sobald auf `feature` ein neuer Commit entsteht, bewegt sich nur dieser Branch weiter:

```
main     → Commit A
feature  → Commit B
```

Das Projekt wird nicht jedes Mal vollständig kopiert. Diese Leichtgewichtigkeit ist einer der Gründe, warum Branches in Git so günstig und alltäglich sind.

Praktische Konsequenz

Du darfst Branches häufig und gezielt einsetzen:

- für ein Feature,
- für einen Fehler,
- für ein Experiment,
- für eine Dokumentationsänderung,
- für einen Pull Request.

Ein Branch ist kein Zeichen dafür, dass eine Aufgabe riesig oder riskant sein muss. Im Gegenteil: Kleine, kurzlebige Branches machen Änderungen besser isolierbar und überprüfbar.

Denkmodell: HEAD zeigt, wo du gerade arbeitest

`HEAD` ist ein zentraler Begriff, der zunächst abstrakt wirken kann. Vereinfacht bedeutet er:

> **HEAD markiert den aktuell ausgecheckten Stand.**

Meist zeigt `HEAD` auf den aktuell aktiven Branch. Wenn du auf `main` arbeitest, zeigt `HEAD` auf `main`. Ein neuer Commit verschiebt dann den Branch und damit indirekt auch `HEAD`.

```
HEAD → main → letzter Commit
```

Wenn du den Branch wechselst, wechselt auch dein Arbeitskontext:

```
HEAD → feature/login → letzter Feature-Commit
```

Warum das wichtig ist

Bevor du commitest, mergest, zurücksetzt oder Dateien wiederherstellst, solltest du wissen:

- Auf welchem Branch befinde ich mich?
- Welcher Commit ist aktuell ausgecheckt?
- Soll die nächste Änderung wirklich hier entstehen?

Diese Informationen liefert:

```
git status
```

Auch PhpStorm zeigt den aktuellen Branch deutlich in der Statusleiste und im Branch-Menü an. Ein kurzer Blick vor einer größeren Aktion verhindert viele Fehler.

Häufiger Fehler: Direkt auf `main` arbeiten

Gerade bei Einzelprojekten wirkt es zunächst bequem, alle Änderungen direkt auf `main` vorzunehmen. In Teams und bei Pull-Request-Workflows führt das jedoch oft zu Problemen:

- Unfertige Arbeit vermischt sich mit stabilem Code.
- Mehrere Aufgaben werden schwer voneinander trennbar.
- Reviews werden unübersichtlich.
- Ein Fehler ist schwieriger isoliert zurückzunehmen.
- Synchronisierung mit anderen wird konfliktanfälliger.

Ein besseres Grundmuster ist:

```
main
  └── feature/registrierung-validieren
```

Die Arbeit findet auf dem Feature-Branch statt. Erst wenn sie fertig, getestet und überprüft ist, wird sie in den Hauptbranch integriert.

Ausnahme: Kleine persönliche Experimente

In einem lokalen Lernrepository oder bei einer sehr kleinen, isolierten Änderung kann direkte Arbeit auf `main` vertretbar sein. Entscheidend ist nicht ein starres Verbot, sondern die Frage:

> **Muss diese Änderung getrennt entwickelt, geprüft oder später nachvollzogen werden?**

Wenn die Antwort „ja“ lautet, ist ein eigener Branch sinnvoll.

Häufiger Fehler: Alles mit `git add .` vormerken

Der Befehl

```
git add .
```

ist nicht grundsätzlich falsch. Er nimmt jedoch viele Änderungen auf einmal in die Staging Area. Das kann versehentlich einschließen:

- Debug-Ausgaben,
- lokale Konfigurationsdateien,
- nicht fertiggestellte Änderungen,
- Zugangsdaten,
- IDE-Dateien,
- Änderungen einer anderen Aufgabe.

Besseres Denkmodell: Die Staging Area ist eine bewusste Auswahl

Die Staging Area ist kein lästiger Zwischenschritt. Sie ist eine **Zusammenstellung des nächsten Commits**.

Statt gedankenlos alles vorzumerken, prüfe zunächst:

```
git status
git diff
```

Danach kannst du gezielt einzelne Dateien hinzufügen:

```
git add src/Validator.php
git add tests/ValidatorTest.php
```

Später lernst du auch, einzelne Teile einer Datei vorzumerken. Damit lassen sich logisch getrennte Commits erstellen, selbst wenn mehrere Änderungen gleichzeitig im Arbeitsverzeichnis liegen.

> **Merksatz:** `git add` bedeutet nicht „speichern“, sondern „für den nächsten Snapshot auswählen“.

Häufiger Fehler: Den Status nicht prüfen

Viele riskante Git-Aktionen beginnen mit einer Vermutung:

- „Ich glaube, alles ist committed.“
- „Ich bin bestimmt auf dem richtigen Branch.“
- „Die Änderung müsste schon auf GitHub sein.“
- „Der Merge müsste fertig sein.“

Git verlangt jedoch keine Vermutungen. Es liefert Informationen.

Der sichere Mini-Check

Vor und nach wichtigen Schritten lohnt sich:

```
git status
```

Achte dabei auf:

- den aktuellen Branch,
- nicht verfolgte Dateien,
- geänderte, aber nicht vorgemerkte Dateien,
- vorgemerkte Änderungen,
- lokale Commits vor oder hinter dem Upstream-Branch,
- laufende Operationen wie Merge oder Rebase.

Ergänzend helfen:

```
git log --oneline --decorate -n 10
```

```
git diff
```

```
git diff --staged
```

Damit beantwortest du drei verschiedene Fragen:

Frage	Geeigneter Befehl
In welchem Zustand befindet sich das Repository?	`git status`
Was wurde noch nicht vorgemerkt?	`git diff`
Was landet im nächsten Commit?	`git diff --staged`
Welche Commits liegen zuletzt vor?	`git log --oneline`

Häufiger Fehler: Commit und Push verwechseln

Ein klassischer Ablauf sieht so aus:

1. Datei bearbeiten
2. Commit erstellen
3. Browser öffnen
4. Auf GitHub keine Änderung finden
5. Verunsicherung

Die Ursache ist meist einfach: Der Push fehlt.

```
git push
```

Umgekehrt kann auch ein Push scheitern, obwohl der Commit erfolgreich erstellt wurde. Das betrifft dann nicht die lokale Historie, sondern den Datentransfer zum Remote. Typische Ursachen sind:

- fehlende Anmeldung,
- falsche Berechtigung,
- ein anderer Stand auf dem Remote,
- Netzwerkprobleme,
- ein noch nicht eingerichteter Upstream-Branch.

Sicheres Vorgehen

Wenn ein Push fehlschlägt:

1. Lies die Fehlermeldung vollständig.
2. Prüfe mit `git status`, ob der lokale Commit vorhanden ist.
3. Prüfe den aktuellen Branch.
4. Prüfe erst dann Anmeldung, Remote-URL oder Synchronisationskonflikte.

Wiederhole nicht blind denselben Befehl und nutze insbesondere keinen erzwungenen Push, nur um eine Fehlermeldung verschwinden zu lassen.

Häufiger Fehler: `pull` als harmlose Aktualisierung betrachten

`git pull` wird oft als reiner Download verstanden. Tatsächlich besteht ein Pull vereinfacht aus zwei Schritten:

1. Änderungen vom Remote abrufen.
2. Diese Änderungen lokal integrieren.

Das Integrieren kann einen Merge oder Rebase auslösen und unter Umständen Konflikte erzeugen.

Besseres Denkmodell

Ein Pull verändert möglicherweise den lokalen Projektzustand. Deshalb ist es sinnvoll, vorher zu prüfen:

```
git status
```

Sind lokale Änderungen vorhanden, solltest du wissen, ob sie mit den eingehenden Änderungen kollidieren könnten.

Für mehr Kontrolle kann es hilfreich sein, gedanklich zwischen zwei Aufgaben zu unterscheiden:

```
git fetch
```

holt Informationen und neue Commits vom Remote, ohne den aktuellen Branch direkt zu verändern.

Danach kannst du untersuchen, was sich geändert hat, und die Integration bewusst durchführen. Die konkreten Varianten werden später ausführlich behandelt; wichtig ist hier vor allem das Verständnis:

> Ein Pull ist nicht nur Empfang, sondern auch Integration.

Häufiger Fehler: Konfliktmarker als Git-Fehler missverstehen

Wenn Git einen Merge-Konflikt meldet, bedeutet das nicht, dass Git beschädigt ist oder dass etwas „kaputtgegangen“ ist. Git kann zwei Änderungen lediglich nicht eindeutig automatisch kombinieren.

In einer Datei können dann Markierungen wie diese erscheinen:

```
<<<<<<< HEAD
Lokale Variante
=======
Eingehende Variante
>>>>>>> anderer-branch
```

Diese Markierungen sind eine Aufforderung zur Entscheidung:

- Welche Variante ist fachlich korrekt?
- Müssen beide Varianten kombiniert werden?
- Fehlt Kontext aus Tests, Anforderungen oder Gesprächen?

Der richtige Umgang mit Konflikten

1. Nicht in Panik geraten.
2. Den Konfliktbereich und den fachlichen Zweck verstehen.
3. Die gewünschte Endfassung schreiben.
4. Konfliktmarker vollständig entfernen.
5. Datei testen und prüfen.
6. Die Git-Operation kontrolliert fortsetzen.

Ein Konflikt ist kein Makel. Er zeigt, dass zwei Entwicklungslinien dieselbe Stelle unterschiedlich verändert haben und eine menschliche Entscheidung erforderlich ist.

Häufiger Fehler: Änderungen vor einem Branch-Wechsel ignorieren

Vor dem Wechsel des Branches sollten lokale Änderungen bewusst behandelt werden. Andernfalls entstehen Unsicherheiten:

- Gehört diese Änderung zum alten oder neuen Branch?
- Wurde sie unbeabsichtigt übernommen?
- Verhindert sie den Wechsel?
- Wird sie später versehentlich committet?

Praktische Regel

Vor einem Branch-Wechsel:

```
git status
```

Danach bewusst entscheiden:

- Änderungen committen, wenn sie einen sinnvollen Zwischenstand bilden.
- Änderungen gezielt verwerfen, wenn sie nicht gebraucht werden.
- Änderungen vorübergehend sicher ablegen, wenn sie später fortgesetzt werden sollen.

Wichtig ist nicht, immer sofort zu committen. Wichtig ist, den Arbeitszustand nicht unbemerkt mitzunehmen.

Häufiger Fehler: Unfertige oder gemischte Commits erstellen

Ein Commit mit vielen unabhängigen Änderungen erschwert fast alles:

- Reviews,

- Fehlersuche,
- Rücknahme einzelner Änderungen,
- Cherry-Picks,
- spätere Historienanalyse,
- Verständlichkeit für andere Personen.

Ein Beispiel für einen problematischen Commit:

```
Fixes und Änderungen
```

Er enthält vielleicht gleichzeitig:

- einen Bugfix,
- eine Umbenennung,
- Formatierungsänderungen,
- eine neue Funktion,
- aktualisierte Abhängigkeiten.

Besseres Denkmodell: Ein Commit beantwortet eine Frage

Ein guter Commit sollte idealerweise eine klare Frage beantworten können:

> **Welchen Zweck erfüllt diese Änderung?**

Beispiele:

- „Warum wurde die Passwortlänge angepasst?“
- „Warum wird diese Ausnahme jetzt abgefangen?“
- „Warum wurde die Abhängigkeit aktualisiert?“
- „Warum wurde die Datenbankabfrage umgestellt?“

Wenn ein Commit mehrere unabhängige Antworten benötigt, sollte er wahrscheinlich aufgeteilt werden.

Häufiger Fehler: Geheimnisse und lokale Dateien committen

Ein Repository ist kein privater Arbeitsordner. Alles, was committed wird, kann langfristig in der Historie verbleiben und nach einem Push für weitere Personen sichtbar werden.

Besonders kritisch sind:

- Passwörter,
- API-Schlüssel,
- Tokens,
- private Schlüssel,
- Datenbank-Zugangsdaten,
- produktive Konfigurationsdateien,
- personenbezogene Testdaten.

Auch wenn eine Datei später gelöscht wird, kann sie in älteren Commits weiterhin vorhanden sein.

Sicheres Denkmodell

> Was committed wird, gehört zur nachvollziehbaren Projektgeschichte.

Prüfe daher vor jedem Commit die vorgemerkten Änderungen:

```
git diff --staged
```

Für Konfigurationen eignet sich oft dieses Muster:

```
.env.example
```

wird versioniert und enthält nur Platzhalter, während die echte lokale Datei etwa über `.gitignore` ausgeschlossen wird:

```
.env
```

Die Regeln für Ignore-Dateien und die sichere Behandlung von Geheimnissen folgen in einem eigenen Kapitel. Die wichtigste Gewohnheit beginnt jedoch sofort: **Nie sensible Inhalte ungeprüft stagen oder committen.**

Häufiger Fehler: Dateilöschung mit Versionsverlust gleichsetzen

Wenn eine Datei im Arbeitsverzeichnis gelöscht wird, ist sie nicht automatisch für immer verloren. Wenn sie bereits committed war, kennt Git ältere Versionen weiterhin.

Das führt zu einem wichtigen Denkmodell:

> Die aktuelle Arbeitskopie ist nur ein sichtbarer Zustand, nicht die gesamte Historie.

Solange ein Commit erreichbar ist, lassen sich frühere Inhalte meist wieder anzeigen oder wiederherstellen. Dennoch gilt: Nicht jede ungespeicherte oder nie committete Änderung kann Git retten.

Was Git typischerweise retten kann

Situation	Rettungschance
Datei war in einem Commit enthalten	Sehr gut
Commit wurde lokal erstellt, aber noch nicht gepusht	Meist sehr gut
Branch wurde versehentlich gelöscht	Häufig gut
Datei wurde nur bearbeitet, nie gespeichert und nie committed	Git kann nicht helfen
Nicht vorgemerkte Änderung wurde bewusst überschrieben	Nur mit anderen Sicherungen eventuell möglich

Git ist eine starke Absicherung für versionierte Zustände, aber kein Ersatz für bewusstes Speichern, Backups oder vorsichtigen Umgang mit destruktiven Befehlen.

Häufiger Fehler: Befehle aus dem Internet blind kopieren

Bei Fehlermeldungen ist die Versuchung groß, einen gefundenen Befehl sofort auszuführen. Besonders gefährlich sind Befehle mit Optionen wie:

```
--hard
--force
--force-with-lease
```

Diese Optionen sind nicht grundsätzlich falsch. Sie haben wichtige, legitime Anwendungsfälle. Ohne Verständnis können sie jedoch lokale Arbeit entfernen oder gemeinsame Historien überschreiben.

Die Drei-Fragen-Regel vor risikoreichen Befehlen

Bevor du einen Befehl ausführst, frage:

1. **Was wird dadurch verändert?**
   Arbeitsverzeichnis, Staging Area, lokale Historie, Remote oder mehrere Bereiche?
2. **Welche Daten könnten verloren gehen?**
   Nicht gespeicherte, nicht vorgemerkte, lokale oder veröffentlichte Änderungen?
3. **Wie komme ich zurück, falls das Ergebnis falsch ist?**
   Gibt es einen Commit, einen Branch, einen Stash, ein Backup oder einen nachvollziehbaren Ausgangspunkt?

Wenn du diese Fragen nicht beantworten kannst, stoppe und untersuche den Zustand. Ein zusätzlicher Blick auf `git status` kostet Sekunden; eine beschädigte gemeinsame Historie kann dagegen viel Zeit kosten.

Häufiger Fehler: Git als Gegner betrachten

Git-Meldungen wirken am Anfang oft streng oder unverständlich. Häufig schützen sie aber vor einem riskanten Zustand.

Beispiele:

- Git verweigert einen Branch-Wechsel, damit lokale Änderungen nicht überschrieben werden.

- Git verweigert einen Push, damit fremde Commits nicht unbemerkt verloren gehen.
- Git meldet einen Konflikt, weil keine sichere automatische Entscheidung möglich ist.
- Git verweigert einen Commit, wenn nichts vorgemerkt wurde.

Statt die Meldung als Hindernis zu sehen, lies sie als Zustandsbeschreibung:

> „Git erklärt mir, welche Annahme gerade nicht erfüllt ist.“

Diese Haltung verändert die Fehlersuche grundlegend. Nicht „Wie zwinge ich Git dazu?“, sondern:

> **„Welchen Zustand schützt Git – und was möchte ich fachlich wirklich erreichen?“**

Ein systematisches Denkmodell für jede Git-Situation

Wenn etwas unerwartet passiert, hilft dieser Ablauf:

1. Anhalten

Keine weiteren Befehle auf Verdacht ausführen. Insbesondere keine Befehle, die Historie oder Arbeitsdateien überschreiben könnten.

2. Zustand erfassen

```
git status
```

Optional ergänzen:

```
git log --oneline --decorate -n 10
```

```
git branch --all
```

3. Ziel formulieren

Beschreibe ohne Git-Befehl, was erreicht werden soll:

- „Diese Datei soll nicht im nächsten Commit landen.“
- „Mein lokaler Branch soll die Änderungen vom Server enthalten.“
- „Der letzte Commit hat die falsche Nachricht.“
- „Ich möchte den Bugfix aus einem anderen Branch übernehmen.“

Eine klare Zielformulierung verhindert, dass ein unpassender Befehl nur zufällig plausibel wirkt.

4. Betroffenen Bereich bestimmen

Ziel	Hauptbereich
Datei bearbeiten oder verwerfen	Arbeitsverzeichnis
Auswahl für nächsten Commit ändern	Staging Area
Lokalen Commit korrigieren	Lokale Historie
Änderungen austauschen	Remote und Tracking-Branch
Entwicklung trennen	Branches

5. Kleinste sichere Aktion wählen

Bevorzuge eine gezielte, nachvollziehbare Aktion gegenüber einer globalen oder erzwungenen Aktion.

Beispiel: Wenn nur eine Datei versehentlich vorgemerkt wurde, ist es besser, genau diese Datei aus der Staging Area zu nehmen, als pauschal große Teile des Repository-Zustands zurückzusetzen.

6. Ergebnis prüfen

Nach jeder Veränderung:

```
git status
```

Und bei relevanten Änderungen:

```
git diff
```

Diese Schleife aus **analysieren → handeln → prüfen** ist eine der wichtigsten professionellen Git-Gewohnheiten.

Ein praktischer Vor-Commit-Check

Vor einem Commit kannst du diese kurze Checkliste verwenden:

- ☐ Bin ich auf dem richtigen Branch?
- ☐ Weiß ich, welchen Zweck dieser Commit erfüllt?
- ☐ Enthält der Commit nur zusammengehörige Änderungen?
- ☐ Habe ich die vorgemerkten Änderungen geprüft?
- ☐ Sind keine Zugangsdaten, Debug-Ausgaben oder lokalen Dateien enthalten?
- ☐ Ist die Commit-Nachricht konkret und verständlich?
- ☐ Wurde die relevante Anwendung oder der relevante Test ausgeführt?

Die technische Prüfung beginnt mit:

```
git status
```

Dann:

```
git diff --staged
```

Erst danach folgt der Commit.

Ein praktischer Vor-Push-Check

Ein Push veröffentlicht Arbeit für andere Personen und häufig auch für Automatisierungen. Prüfe daher kurz:

- ☐ Liegen die richtigen Commits auf dem aktuellen Branch?
- ☐ Sind Tests oder Qualitätsprüfungen erfolgreich?
- ☐ Enthält die Historie keine versehentlichen oder geheimen Inhalte?
- ☐ Ist klar, in welches Remote-Repository und auf welchen Branch gepusht wird?
- ☐ Wurden aktuelle Änderungen des Teams berücksichtigt?

Danach ist ein Push keine blinde Routine, sondern eine bewusste Veröffentlichung.

Die wichtigsten Merksätze

> **Git verwaltet Zustände und Historien, nicht nur Dateien.**

> **Ein Commit ist ein lokaler Snapshot des vorgemerkten Projektzustands.**

> **`git add` wählt Inhalte für den nächsten Commit aus.**

> **Ein Push veröffentlicht lokale Commits auf einem Remote; ein Commit allein tut das nicht.**

> **Ein Branch ist eine leichte Entwicklungslinie, keine vollständige Projektkopie.**

> **Ein Konflikt ist eine notwendige fachliche Entscheidung, kein Git-Defekt.**

> **Vor jeder riskanten Aktion zuerst den Zustand prüfen.**

> **`git status` ist kein Notfallwerkzeug, sondern ein alltäglicher Kompass.**

Mit diesen Denkmodellen werden die folgenden Git-Befehle nicht mehr wie isolierte Zauberformeln wirken. Sie werden zu gezielten Werkzeugen, mit denen du Änderungen sicher zwischen klar erkennbaren Zuständen bewegst.

Revision #1
Created 2026-07-25 11:13:41 UTC by art10m
Updated 2026-07-25 11:13:41 UTC by art10m