



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Anhang 0 Teil 1

TERMINAL- UND CLI- GRUNDLAGEN

Heutige Lernziele

Am Ende dieser Sitzung werden Sie:

1. **Verstehen**, was ein Terminal ist und warum Entwickler es nutzen
2. **Kennen**, den Unterschied zwischen Terminal, Shell und Betriebssystem
3. **Navigieren** können im Dateisystem mit Textbefehlen
4. **Lesen** können von Befehlssyntax wie `git commit -m "message"`
5. **Verstehen**, was PATH ist und "command not found"-Fehler beheben
6. **Erkennen**, warum Automatisierung durch Scripting eine Kernkompetenz ist

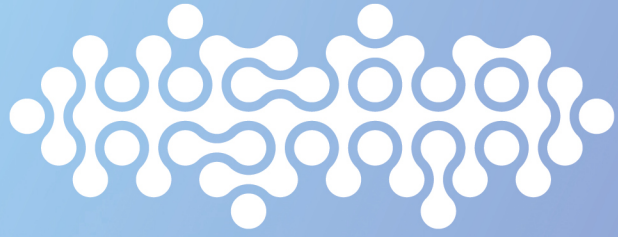
Warum das wichtig ist

Alles in diesem Kurs läuft über das Terminal:

- Installation von Git, Python und Werkzeugen
- Repositories klonen und Code verwalten
- Tests, Linter und Formatter ausführen
- CI/CD-Pipelines und Deployment

Wenn "öffne ein Terminal und tippe diesen Befehl" einschüchternd klingt -- diese Sitzung ist für dich.

Viele professionelle Entwickler haben genau dort angefangen, wo du jetzt bist.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Was ist ein Terminal?

1

Eine Textkonversation mit deinem Computer

Stell es dir wie einen Chat vor:

```
Du:      "Zeig mir alle Dateien in diesem Ordner"  
Computer: "document.txt  photo.jpg  notes.md"  
  
Du:      "Erstelle einen neuen Ordner namens 'project'"  
Computer: (Stille bedeutet Erfolg)
```

In der Praxis:

```
$ ls  
document.txt  photo.jpg  notes.md  
  
$ mkdir project
```

Das `$` ist der **Prompt** -- es bedeutet "bereit für deinen nächsten Befehl." Nicht eintippen!

Kurze Umfrage

Hast du schon einmal ein Terminal benutzt?

- **A)** Noch nie -- das ist alles neu für mich
- **B)** Ein paar Mal, aber ich fühle mich nicht sicher
- **C)** Ich benutze es gelegentlich
- **D)** Ich bin bereits sicher im Umgang damit

Keine falsche Antwort! Diese Sitzung bringt alle auf den gleichen Stand.

Von physischen Geräten zu Software

1960er-70er: Ein "Terminal" war ein physisches Gerät -- Tastatur + Bildschirm an einem Großrechner

Heute: Terminal-Emulatoren -- Software, die diese Textkonversation simuliert

Immer noch das Fundament moderner IT:

- Server, die Websites betreiben
- Cloud-Infrastruktur
- Automatisierte Test-Pipelines
- Deployment-Systeme

Jeder professionelle Entwickler, Sysadmin und DevOps-Ingenieur nutzt das Terminal täglich.

Warum Entwickler immer noch das Terminal nutzen

Praktische Gründe:

- **Geschwindigkeit** -- 3 Sekunden statt 30
- **Automatisierung** -- Befehle in Skripten speichern
- **Präzision** -- keine versteckten GUI-Annahmen

Professionelle Gründe:

- **Reproduzierbarkeit** -- per Text teilen
- **Remote-Zugriff** -- Server haben keine GUI
- **Tool-Ökosystem** -- Git, pip, pytest, ruff

Wichtige Erkenntnis: Die GUI-Buttons in VS Code sind Wrapper um Terminal-Befehle. Terminal verstehen = die gesamte Pipeline verstehen.

GUI vs. Terminal: Ein Vergleich

Einen Projektordner mit README erstellen:

GUI: Explorer öffnen → Navigieren → Rechtsklick → Neuer Ordner → Benennen → Öffnen → Rechtsklick → Neue Datei → Benennen → **7 Schritte**

Terminal:

```
mkdir project && cd project && touch README.md
```

1 Zeile. Fertig.

Und du kannst diese Zeile speichern und für immer wiederverwenden.

Was ist mit KI-Assistenten?

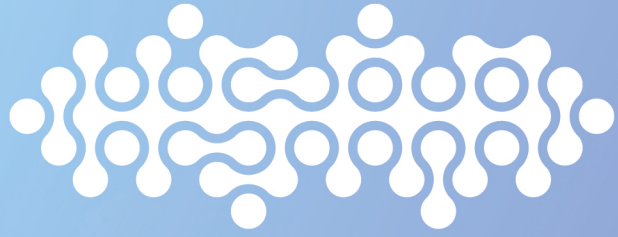
Tools wie **GitHub Copilot CLI** können natürliche Sprache in Terminal-Befehle übersetzen.

Warum also Terminal-Befehle lernen?

Aus dem gleichen Grund, warum man Mathematik lernt, obwohl es Taschenrechner gibt:

- Du musst **verstehen**, was die KI vorschlägt, bevor du es ausführst
- Du musst **überprüfen und debuggen**, wenn etwas schiefgeht
- Du brauchst **mentale Modelle**, um gute Fragen zu stellen

KI verstärkt Verständnis -- sie ersetzt es nicht.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Terminals, Shells und Betriebssysteme

2

Verschiedene Betriebssysteme, verschiedene Standards

Betriebssystem	Standard-Terminal	Standard-Shell	Prompt
Windows	Windows Terminal	PowerShell	PS C:\Users\du>
macOS	Terminal.app	zsh	du@mac ~ %
Linux	GNOME Terminal	bash	du@pc:~\$

Sogar das **Prompt-Symbol** unterscheidet sich: >, %, \$

Alle haben den gleichen Zweck: **"Ich bin bereit für deinen nächsten Befehl."**

Terminal vs. Shell -- Der wichtige Unterschied

- **Terminal** = das **Fenster** (Anzeige, Farben, Schriften, Kopieren/Einfügen)
- **Shell** = die **Sprache**, die darin läuft (interpretiert deine Befehle)

Analogie:

Das Terminal ist das **Telefon**. Die Shell ist die **Sprache**, die du am Telefon sprichst.

Gleiches Telefon, verschiedene Sprachen. Gleiches Terminal-Fenster, verschiedene Shells.

Warum verwirrend? "PowerShell" ist sowohl eine Shell ALS AUCH eine Terminal-App. "Git Bash" ist ein Terminal, das bash ausführt. Die Namen sind inkonsistent -- das Konzept zählt.



Kurze Frage

Was meint jemand, wenn er sagt "öffne bash"?

- **A)** Eine Terminal-Anwendung öffnen, die die bash-Shell ausführt
- **B)** Ein neues Programm namens "bash" installieren
- **C)** Einen Webbrowser öffnen
- **D)** Die Windows-Eingabeaufforderung öffnen

Denke an den Unterschied zwischen Terminal und Shell!

► Antwort

Windows: Viele Terminal-Optionen

Terminal	Was es ist	Für diesen Kurs?
Eingabeaufforderung	Ältestes Windows-Terminal (1980er)	Selten nötig
PowerShell	Moderner Windows-Standard	Funktioniert, aber andere Syntax
Git Bash	Bash-Shell unter Windows (mit Git installiert)	Empfohlen
WSL	Vollständiges Linux in Windows	Fortgeschrittene Option

Unsere Empfehlung: Verwende **Git Bash** oder das **VS Code Terminal** mit Git Bash. So funktionieren alle Kursbeispiele auf jedem Betriebssystem.

Das VS Code Terminal

Öffnen: ``Ctrl+``` (Backtick) oder **Ansicht → Terminal**

Das VS Code Terminal ist eine Terminal-Anwendung mit **deiner Wahl der Shell** darin.

Klicke auf den Dropdown (▼) neben + um verfügbare Shells zu sehen:

- PowerShell (Standard unter Windows)
- Git Bash
- Eingabeaufforderung
- JavaScript Debug Terminal

Git Bash als Standard setzen:

`Ctrl+Shift+P` → "Terminal: Select Default Profile" → Git Bash

Bash vs. Zsh

bash:

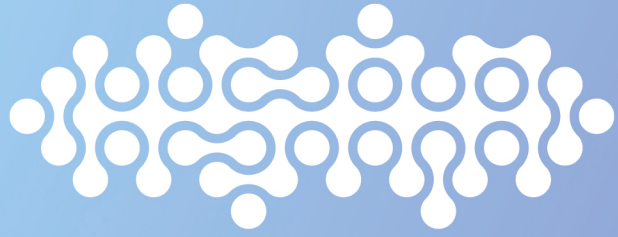
- Standard auf Linux, Git Bash
- Einfache Tab-Vervollständigung
- Der Scripting-Standard
- Konfiguration: `~/ .bashrc`

zsh:

- Standard auf macOS (seit 2019)
- Erweiterte Tab-Vervollständigung
- Rechtschreibkorrektur
- Konfiguration: `~/ .zshrc`

Beide verstehen die gleichen Kernbefehle: `cd`, `ls`, `mkdir`, `rm`, `cat`, `echo`, `grep` ...

Fazit: Benutze, was dein System mitbringt. Nicht zu viel nachdenken.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Im Dateisystem navigieren

3

Den Prompt lesen

```
user@computer:~/Documents$
```

Das sagt dir:

- **Wer du bist:** user
- **Wo du bist:** ~/Documents (~ = Home-Ordner)
- **Bereit für Eingabe:** \$

Unter Windows PowerShell:

```
PS C:\Users\du\Documents>
```

Wenn Tutorials \$ am Anfang zeigen -- nicht eintippen!

```
$ git status    # ← du tippst nur: git status
```

Wichtige Navigationsbefehle

Was du willst	Befehl	Was er tut
Wo bin ich?	<code>pwd</code>	Aktuelles Verzeichnis anzeigen
Was ist hier?	<code>ls</code>	Dateien und Ordner auflisten
In Ordner gehen	<code>cd ordnername</code>	Verzeichnis wechseln
Eine Ebene hoch	<code>cd ..</code>	Zum übergeordneten Verzeichnis
Nach Hause	<code>cd ~</code>	Zum Home-Verzeichnis
Ordner erstellen	<code>mkdir name</code>	Neues Verzeichnis anlegen
Datei erstellen	<code>touch name.txt</code>	Leere Datei erstellen

Diese funktionieren in bash, zsh, Git Bash **und** PowerShell (über Aliase).

Weitere wichtige Befehle

Was du willst	Befehl	Was er tut
Dateiinhalte anzeigen	<code>cat datei.txt</code>	Dateiinhalte ausgeben
Datei kopieren	<code>cp quelle ziel</code>	Datei kopieren
Verschieben/Umbenennen	<code>mv alt neu</code>	Verschieben oder umbenennen
Datei löschen	<code>rm datei.txt</code>	Entfernen (kein Undo!)
Bildschirm leeren	<code>clear</code>	Anzeige aufräumen

Warnung: `rm` hat **keinen Papierkorb**. Gelöschte Dateien sind weg. Vorsicht!

Dateipfade verstehen

Absoluter Pfad (vollständige Adresse):

```
# Linux/macOS
/home/user/Documents/README.md

# Windows
C:\Users\user\Documents\README.md
```

Relativer Pfad (vom aktuellen Ort):

```
project/README.md
./project/README.md
../Downloads/file.zip
```

Spezielle Symbole: `.` = aktuelles Verz., `..` = übergeordnetes Verz., `~` = Home, `/` = Root

Tipp: Git Bash unter Windows akzeptiert Schrägstriche `/` -- benutze sie!

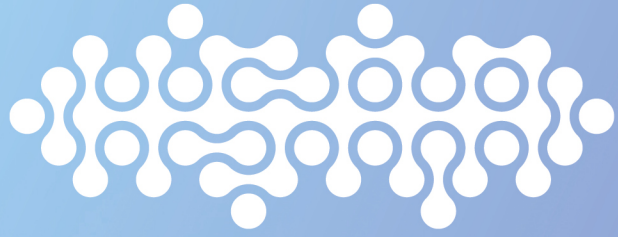


Praxis: Im Dateisystem navigieren

Mach mit in deinem Terminal:

```
pwd          # Wo bin ich?
ls           # Was ist in diesem Ordner?
mkdir terminal-practice # Neuen Ordner erstellen
cd terminal-practice  # Hineingehen
pwd             # Standort bestätigen
touch hello.txt      # Leere Datei erstellen
echo "Hello Terminal" > hello.txt  # Text hineinschreiben
cat hello.txt        # Datei lesen
cd ..              # Eine Ebene zurück
```

Nimm dir 3 Minuten. Probiere jeden Befehl. Frag, wenn etwas unklar ist!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Befehlssyntax und Hilfe

4

Anatomie eines Befehls

```
git commit -m "Add README file"
```

| | | └─ Argument: der Wert für -m
| | └─ Flag: ändert das Verhalten
| └─ Unterbefehl: die Aktion
└─ Programm: das Werkzeug

Nicht jeder Befehl hat alle Teile:

```
git status      # Programm + Unterbefehl  
ls              # nur ein Programm  
ls -la          # Programm + Flags  
cd Documents    # Programm + Argument
```

Die --help Ausgabe lesen

Jedes gute CLI-Tool hat eingebaute Dokumentation:

```
git --help          # alle Unterbefehle auflisten
git commit --help   # detaillierte Hilfe für commit
```

So liest du die Synopsis:

```
git commit [-a | --all] [-m <msg>] [<pathspec>...]
```

Notation	Bedeutung
[...]	Optional
<...>	Durch eigenen Wert ersetzen
	Eines auswählen
...	Mehrere Werte möglich

Gemeinsame Muster aller CLI-Tools

Diese Muster wiederholen sich überall:

```
# Hilfe bekommen
git --help          python --help          ruff --help

# Version prüfen
git --version        python --version        uv --version
```

Kurze vs. lange Flags:

```
git commit -m "message"      # kurz: -m
git commit --message "message" # lang: --message
```

Kurze Flags kombinieren:

```
ls -l -a      # zwei separate Flags
ls -la        # kombiniert -- gleiches Ergebnis
```



Interaktiv: Diesen Befehl analysieren

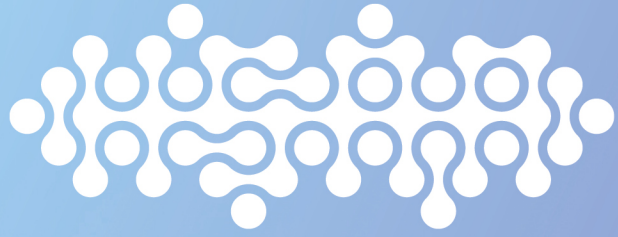
Identifiziere die Teile:

```
git clone --depth 1 https://github.com/user/repo.git
```

Besprich mit deinem Nachbarn:

1. Was ist das **Programm**?
2. Was ist der **Unterbefehl**?
3. Was ist das **Flag** und was könnte es bewirken?
4. Was ist das **Argument**?

Nimm dir 2 Minuten, dann besprechen wir es.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

PATH und Umgebungsvariablen

5

Das "Command Not Found"-Problem

Irgendwann **wirst** du das sehen:

```
$ ruff check  
bash: ruff: command not found
```

Oder in PowerShell:

```
ruff: The term 'ruff' is not recognized as the name of  
a cmdlet, function, script file, or operable program.
```

Das bedeutet NICHT, dass das Programm kaputt ist.

Meist weiß die Shell einfach nicht, **wo sie das Programm finden** kann.

Was ist der PATH?

PATH = eine Liste von Ordnern, die die Shell durchsucht, wenn du einen Befehl eingibst

Stell es dir wie ein **Telefonbuch** vor: die Shell schlägt den Befehlsnamen in einer Liste bekannter Adressen nach.

```
# bash / Git Bash (durch Doppelpunkte getrennt)
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/home/user/.local/bin
```

```
# PowerShell (durch Semikolons getrennt)
PS> echo $env:PATH
C:\Windows\System32;C:\Program Files\Git\bin
```

Wenn du `git` tippst, prüft die Shell jeden Ordner **der Reihe nach**, bis sie `git` findet.

"Command Not Found" beheben

Schritt 1: Ist das Programm installiert? Wenn nicht, installiere es.

Schritt 2: Ist sein Ordner im PATH?

```
which git          # bash / Git Bash
where.exe git      # PowerShell (nicht nur "where"!)
Get-Command git    # PowerShell-native Alternative
```

Hinweis: `where` allein in PowerShell ist ein Alias für `where-object` -- nicht das Richtige!

Schritt 3: Hast du das Terminal nach der Installation neu gestartet?

Viele Installer ändern den PATH, aber Änderungen gelten nur für **neue** Terminal-Fenster.

Schritt 4 (fortgeschritten): Ordner manuell zum PATH hinzufügen. (Frag um Hilfe.)

In 90% der Fälle löst Schritt 3 das Problem!

Weitere Umgebungsvariablen

PATH ist die wichtigste, aber es gibt weitere:

Variable	Was sie speichert
HOME	Dein Home-Verzeichnis
USER	Dein Benutzername
LANG	Spracheinstellung
EDITOR	Bevorzugter Texteditor (wird von git genutzt)

```
echo $HOME      # bash / Git Bash
echo $env:USERPROFILE # PowerShell
```

Umgebungsvariablen = **globale Einstellungen** für deine Terminal-Sitzung.



Praxis: Deinen PATH erkunden

Probiere diese Befehle in deinem Terminal:

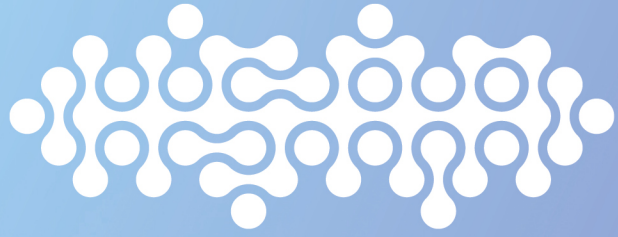
```
# Deinen PATH anzeigen
echo $PATH

# Herausfinden, wo git liegt
which git

# Verzeichnisse in deinem PATH zählen
# (Tipp: zähle die Doppelpunkte oder Semikolons)
```

Fragen zur Diskussion:

- Wie viele Verzeichnisse durchsucht deine Shell?
- Wo ist git auf deinem Rechner installiert?
- Was passiert, wenn zwei Ordner im PATH beide `git` enthalten?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Was ist ein Programm?

6

Programme sind einfach Dateien

Ein Programm ist eine Datei mit Anweisungen, die dein Computer ausführen kann. **Das ist alles.**

Ein Programm "installieren" = Dateien auf die Festplatte kopieren + den Speicherort zum PATH hinzufügen

Ein Programm "ausführen" = das Betriebssystem liest die Datei und folgt den Anweisungen

Aspekt	Windows	Linux / macOS
Erkennt Executables durch	Dateierweiterung (<code>.exe</code>)	Dateiberechtigungen
Typische Programmdatei	<code>git.exe</code>	<code>git</code> (keine Erweiterung)
Ausführbar machen	<code>.exe</code> -Erweiterung geben	<code>chmod +x script.sh</code>

Kompilierte vs. interpretierte Programme

Kompiliert (z.B. `git`, C-Programme):

- Quellcode → Maschinencode (einmalig)
- Läuft direkt, keine Extra-Software
- Eigenständige `.exe` oder Binary

Interpretiert (z.B. Python):

- Wird Zeile für Zeile gelesen und ausgeführt
- Braucht den Interpreter installiert
- `python script.py`

Deshalb muss Python installiert sein, bevor du `.py`-Dateien ausführen kannst.

Python ist der **Interpreter**, der deine Skripte liest.

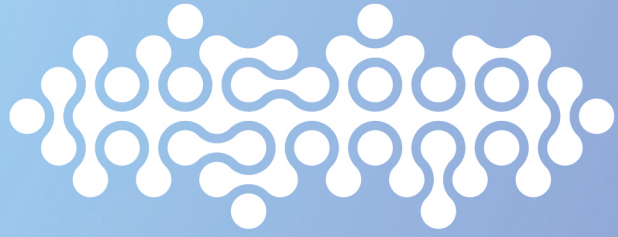
Programme vs. IDE-Erweiterungen

Warum Python UND die Python VS Code Extension installieren?

Installation	Wo	Was es tut
Python (Programm)	Auf deinem Computer	Führt <code>.py</code> -Dateien tatsächlich aus
Python (Extension)	In VS Code	Syntax-Highlighting, Autocomplete
Ruff (Programm)	Auf deinem Computer	Prüft Code-Stil im Terminal
Ruff (Extension)	In VS Code	Zeigt Ruff-Warnungen als Unterstreichungen

Die Extension ruft das Programm auf -- sie ersetzt es nicht.

Das Programm ist der **Motor**. Die Extension ist das **Armaturenbrett**.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Automatisierung und Scripting

7

Von einzelnen Befehlen zur Automatisierung

Die Progression:

Einzelbefehl → **Verkettete Befehle** → **Shell-Skript** → **CI/CD-Pipeline**

```
# 1. Einzeln
mkdir project

# 2. Verkettet (&& bedeutet "wenn erfolgreich, dann auch...")
mkdir project && cd project && git init

# 3. Skript (in setup.sh speichern, ausführen mit: bash setup.sh)

# 4. CI/CD (läuft automatisch wenn du Code pushst)
```

Jede Stufe baut auf der vorherigen auf. **Das Terminal ist der Ort der Automatisierung.**

Ein echtes Skript-Beispiel

setup.sh -- Projekt erstellen und initialisieren:

```
#!/bin/bash
PROJECT_NAME="my-project"

mkdir "$PROJECT_NAME"
cd "$PROJECT_NAME"
git init
echo "# $PROJECT_NAME" > README.md
git add .
git commit -m "Initial commit"
echo "Project $PROJECT_NAME created!"
```

Ausführen: `bash setup.sh`

Schneller, keine Tippfehler, wiederholbar. 10 Projekte nötig? 10 Mal ausführen.

Automatisierung in diesem Kurs

All das wirst du kennenlernen -- alles basiert auf Terminal-Automatisierung:

- **Pre-commit Hooks** -- Skripte, die Code vor jedem Commit prüfen
- **GitHub Actions** -- automatisierte Pipelines beim Push
- **Pytest** -- findet und führt alle Tests automatisch aus
- **Ruff** -- prüft deinen Code-Stil automatisch

Das Muster ist immer gleich: manuelle Schritte automatisieren.



Praxis: Dein erstes Skript schreiben

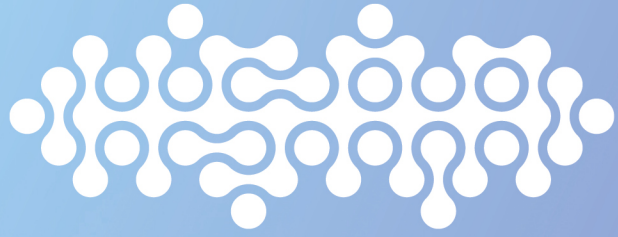
1. Erstelle eine Datei namens `setup.sh` (mit VS Code oder einem Editor)
2. Füge das Skript von der vorherigen Folie ein
3. Öffne ein Terminal und navigiere zur Datei
4. Führe es aus:

```
bash setup.sh
```

5. Überprüfe:

```
ls my-project  
cd my-project && git log
```

Nimm dir 5 Minuten. Heb die Hand, wenn du den Commit siehst!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kurzreferenz und Zusammenfassung

8

Tastenkürzel für das Terminal

Tastenkürzel	Was es tut
Pfeiltasten hoch/runter	Durch vorherige Befehle blättern
Tab	Dateinamen und Befehle vervollständigen
Ctrl+C	Aktuellen Befehl abbrechen
Ctrl+L	Bildschirm leeren
Ctrl+R	Befehlshistorie durchsuchen

Tab-Vervollständigung ist dein bester Freund -- anfangen zu tippen und Tab drücken!

Ctrl+C ist deine Notbremse -- benutze es, wenn etwas hängt.

Die wichtigsten Erkenntnisse

1. Ein **Terminal** ist eine Textschnittstelle; eine **Shell** ist die Sprache darin
2. Verschiedene Betriebssysteme = verschiedene Standards, aber **Kernbefehle sind gleich**
3. Befehle folgen: `Programm Unterbefehl --Flag Argument`
4. **PATH** sagt der Shell, wo Programme zu finden sind
5. Programme sind nur Dateien -- Erweiterungen sind nur Konventionen
6. **Automatisierung** durch Scripting ist eine professionelle Kernkompetenz

Du hast jetzt das Fundament, um sicher im Terminal zu arbeiten!

Nächste Schritte

Bereit, diese Fähigkeiten einzusetzen? Gehe zu:

- **Tools installieren:** [Vorlesung 1: Git installieren](#)
- **VS Code einrichten:** [Vorlesung 1: VS Code Setup](#)
- **Git-Workflows:** [Anhang 1: Fortgeschrittenes Git](#)

Vollständige Vorlesungsreferenz: [Anhang 0 Teil 1: Terminal- und CLI-Grundlagen](#)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

VIELEN DANK