Unix & Shell-Programmierung SS21

Vorlesungswoche 9

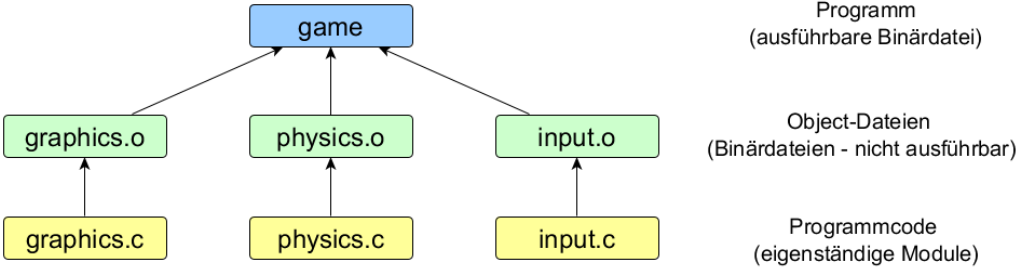
Helga Karafiat

FH Wedel

Automatisierte Erzeugungsprozesse (Motivation)

- Unix-Systeme werden häufig zur Softwareentwicklung eingesetzt
- Größere Softwareprojekte bestehen aus einer Vielzahl an kleinen Teilprojekten, die dann zu einem großen Ganzen zusammengesetzt werden
- Die Erzeugung von einzelnen Projektteilen kann sehr aufwändig / zeitintensiv sein

Fiktives Beispielprojekt:

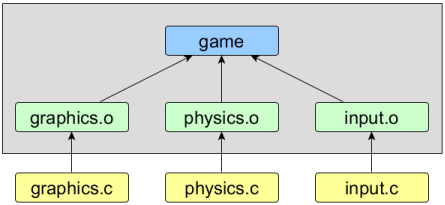


- Erzeugung des ausführbaren Programms ist mehrstufiger Prozess, abbildbar z.B. als Shellskript:

```
#!/bin/sh

# Erzeugung der Object-Dateien
cc -c -g -O input.c
cc -c -g -O physics.c
cc -c -g -O graphics.c

# Erzeugung des ausfuehrbaren Programms
cc -o game graphics.o physics.o input.o -lm
```



- Nachteile:
  - Die Module `input.c`, `physics.c` und `graphics.c` sind nicht voneinander abhängig, ihre Object-Dateien werden aber bei jedem Aufruf des Skripts erzeugt - egal was sich geändert hat
  - Skript weder sonderlich modular noch sonderlich anpassungs- und erweiterungsfreundlich
  - Hierarchische Struktur des Projekts (Abhängigkeiten) schwer erkennbar

make

- `maintain, update and regenerate groups of programs`
- Im POSIX-Standard definiertes Tool zur Abbildung von automatischen Erzeugungsprozessen
- Verwendungszweck: Verwaltung von Software-Builds
- verwaltet Abhängigkeiten in Form von Baumstrukturen, dabei
  - Erzeugung von Zielen aus Quellen,
    also z.B. ausführbares Programm aus Sourcecode und Bibliotheken
  - Ermittlung ob ein Ziel (neu) erzeugt werden muss, anhand eines Vergleichs der Zeitstempel von Quelle(n) und Ziel
- sucht Quellen und Ziele stets im aktuellen Verzeichnis (nicht rekursiv)
- Nützlich für alle Konstellationen in denen verschiedene Quellen (auch über mehrere Stufen / Zwischenziele) zu einer oder mehreren Ausgaben (Zielen) kombiniert werden

GNU make

- Das im POSIX-Standard definierte make ist exakt auf die Bedürfnisse bei der Erzeugung von Software-Builds zugeschnitten
- Das Prinzip von make kann aber viele Aufgaben / Erzeugungsprozesse deutlich vereinfachen (auch außerhalb von Software-Builds)
- GNU make ist an einigen Stellen deutlich eleganter und flexibler als POSIX make und bietet sich somit für Projekte an, die nicht auf maximale Portabilität zwischen verschiedenen Unix-Systemen ausgelegt sein müssen
- GNU make unterstützt auch vollständig POSIX-konforme Makefiles
- Alle Infos zu GNU make: https://www.gnu.org/software/make/manual/

Verwendung von make

- Syntax: `make [-f MAKEFILE] [OPTIONS]... [TARGETS]...`
  Wenn kein `MAKEFILE` mit `-f` angegeben ist, nimmt `make` die Datei mit dem Namen `Makefile` im aktuellen Verzeichnis
- Einige nützliche Optionen
  - `-k` (`--keep-going`)
    Bricht die Verarbeitung nicht ab, wenn ein Fehler aufgetreten ist, sondern versucht so viel wie möglich ist zu bauen
  - `-S` (`--no-keep-going`, `--stop`)
    Verarbeitung bricht ab, sobald ein Fehler beim Abarbeiten der Regeln aufgetreten ist
    default-Verhalten, Angabe ggfs. bei rekursiven Aufrufen sinnvoll
  - `-n` (`--just-print`, `--dry-run`)
    gibt die Befehle an, die ausgeführt werden würden, führt sie aber nicht aus

Makefile

- “Kochbuch” für make
- Datei, die die Abhängigkeiten und Anweisungen zur Erstellung des gewünschten Ziels / der gewünschten Ziele beschreibt (Regeln bzw. Rezepte)
- befindet sich üblicherweise immer im gleichen Verzeichnis wie die Quellen
- verfügt bereits über diverse vorgefertigte “Rezepte” aus dem Software-Build Bereich (z.B. für die Erzeugung von C-Programmen)
- Makefiles sind keine Shellskripte, enthalten aber üblicherweise sog. Befehlszeilen mit Shellcode zur Erzeugung von Zielen der dann jeweils von der Shell interpretiert und ausgeführt wird
- Syntax von `Makefiles`
  - Kommentare beginnen mit # und gehen bis ans Zeilenende
  - Leerzeilen werden ignoriert
  - Befehlszeilen müssen mit einem Tabulatorzeichen eingerückt werden

Regeln

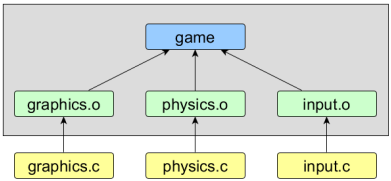
- Definition von Zielen über Abhängigkeiten in Form von:

  `target [target...]: [prerequisite...]`

  - bilden die Baumstruktur des Projekts ab
  - für das fiktive game-Beispiel

    ```
    game: graphics.o physics.o input.o

    graphics.o: graphics.c
    physics.o: physics.c
    input.o: input.c
    ```



- Erzeugung von Zielen aus Quellen über Befehlszeilen (Rezept):

  ```
  target [target...]: [prerequisite...]
    [recipe]
  ```

- Hinweise zu Befehlszeilen:
  - Befehlszeilen **müssen** mit einem Tabulatorzeichen beginnen und können beliebige und beliebig viele Shell-Befehle enthalten (Maskierung von Zeilenumbrüchen mit \ ist möglich)
  - Jede Befehlszeile wird in einer eigenen Shell ausgeführt, d.h. wenn Kommandos sich beeinflussen sollen, müssen sie in einer Zeile stehen (z.B. `cd ; tue_irgendwas`)
  - Wenn eine Befehlszeile nicht ausgeführt werden kann, bricht `make` die komplette Ausführung an dieser Stelle ab (siehe auch Optionen)
  - Jede Befehlszeile wird standardmäßig bei der Ausführung ausgegeben, das Präfix `@` am Beginn der Zeile unterdrückt diese Ausgabe (z.B. bei Ausgaben mit `echo`)

- “Naives” Makefile

```
game: graphics.o physics.o input.o
  cc -o game graphics.o physics.o input.o -lm

graphics.o: graphics.c
  cc -c -g -O graphics.c

physics.o: physics.c
  cc -c -g -O physics.c

input.o: input.c
  cc -c -g -O input.c
```

- Nur wenn eine der Quellen (Abhängigkeiten) neuer als das entsprechende Ziel ist oder das Ziel noch nicht vorhanden ist, wird es beim Aufruf von `make` neu erstellt (ggfs. rekursiv)
- Internes Vorgehen von `make` (Fallunterscheidung):
  - Ziel und Quelle existieren
    $\Rightarrow$ Zeitstempel des letzten Schreibens vergleichen, wenn Quelle neuer als Ziel, Ziel erzeugen
  - Ziel(datei) existiert nicht
    $\Rightarrow$ Behandeln wie Quelle neuer als Ziel
  - Ziel existiert aber keine Regel für das Ziel
    $\Rightarrow$ Ziel als aktuell annehmen
  - Ziel existiert nicht und keine Regel für das Ziel vorhanden
    $\Rightarrow$ Fehler

- Aufruf-Beispiele:
  - `make game`
    erzeugt das Programm game anhand seiner Abhängigkeiten, und nur dann wenn sich mindestens eine der Dateien `input.o`, `graphics.o`, `physics.o`, `input.c`, `graphics.c` oder `physics.c` seit dem letzten Erzeugen verändert hat oder `game` noch nicht existiert
  - `make input.o`
    erzeugt die Objektdatei `input.o` aus der Datei `input.c`, falls `input.c` neuer ist oder `input.o` noch nicht exisitiert
  - `make`
    erzeugt das Ziel game, da dieses als erstes Ziel im Makefile steht
    (Konvention: sinnvolles Standard-Target als erstes Ziel im Makefile definieren)

- Abhängigkeiten können beliebig tief geschachtelt sein, z.B.:

```
foo: bar
  ...
bar: batz
  ...
batz: blubb
  ...
```

Vorgehen bei Aufruf von `make foo`

- `bar` vorhanden und neuer als `batz` und `blubb`
  → erzeuge `foo` aus `bar`
- `bar` nicht vorhanden oder älter als `batz`, `batz` vorhanden und neuer als `blubb`
  → erzeuge erst `bar` aus `batz`, dann `foo` aus `bar`
- `bar` nicht vorhanden oder älter als `batz`, `batz` nicht vorhanden oder älter als `blubb`
  → erzeuge erst `batz` aus `blubb`, dann `bar` aus `batz`, dann `foo` aus `bar`

Muster-Regeln (pattern rules)

- Spezielle Form von Regeln zur Abstraktion und Zusammenfassung von mehreren Zielen
- werden verwendet um Dateien, die einem bestimmten Muster entsprechen (z.B. bestimmte Dateiendung) immer nach dem gleichen Schema zu erzeugen
- Schreibweise analog zu den bereits bekannten normalen Regeln, also: "Ziel(e): Quelle(n)" aber mit genau einem %-Zeichen auf der linken Seite (Ziel)
- %-Zeichen:
  - werden als Muster (ähnlich einer Wildcard) zur Dateiangabe genutzt
  - erweitern zu einer beliebigen (nicht-leeren) Zeichenkette
  - %-Zeichen in der Quellabgabe gibt an, wie sich die Namen von Quelle und Ziel zueinander verhalten (% steht in beiden Angaben für die selbe Zeichenkette)
- Beispiel (Object-Dateien aus C-Dateien erzeugen):

```
%.o: %c
    ...
```

- Spezielle Variablen (automatic variables / internal macros) um Quellen und Ziele innerhalb von Regeln zu adressieren

Name	Bedeutung
`$@`	aktuelles Ziel
`$*`	aktuelles Ziel ohne Dateiendung
`$<`	erste Quelle / aktuelle Quelle
`$?`	Liste von Quellen, die neuer sind als das Ziel
$%	*nur für das Erzeugen von Bibliotheken von Interesse*
`$^`	alle Quellen
`$+`	wie `$^`, aber explizite Listung mehrfach vorhandener Abhängigkeiten
...	...

- Verbessertes Beispiel (mit Muster-Regel)

```
game: graphics.o physics.o input.o
    cc -o game graphics.o physics.o input.o -lm

%.o: %.c
    cc -c -g -O $<
```

- Muster-Regeln - Fortsetzung:
  - %-Zeichen können von einem optionalem Präfix und einem optionalen Suffix umgeben sein, welche den Namen weiter spezifizieren
  - GNU `make` bevorzugt bei Auswahl zwischen mehreren passenden Regeln, stets die, die zur kürzesten Zeichenkette im % führt (also die speziellere) und bei weiterem Konfikt die Regel, die weiter oben im Makefile definiert wird (die "ältere")
  - Praktisch z.B. zur Organisation von Dateien in Unterordnern oder wenn sich Ziele durch ein anderes Kriterium als die Dateiendung unterscheiden

- Beispiel:

```
%.o: %.c
  ...
%.o: %.f
  ...
lib/%.o: lib/%.c
  ...
```

Auswertung (unter Annahme, dass Ziele noch nicht existieren):

- `make bar.o`
  - `bar.c` und `bar.f` existieren → `bar.o` wird aus `bar.c` erzeugt ("ältere" Regel)
  - nur `bar.f` existiert → `bar.o` wird aus `bar.f` erzeugt
- `make lib/bar.o`
  - nur `bar.c` existiert → `lib/bar.o` kann nicht erzeugt werden (kein passendes % für beide Seiten)
  - `lib/bar.f` und `lib/bar.c` existieren → `lib/bar.o` wird aus `lib/bar.c` erzeugt ("kürzeres %")
  - nur `lib/bar.f` exisistiert → `lib/bar.o` wird aus `lib/bar.f` erzeugt (nach zweiter Regel)

Variablen in Makefiles

- ähnlich zu Variablen in der Shell können Variablen (Makros) in Makefiles definiert werden um eine größere Flexibilisierung zu erreichen
- Variablen sind einfache Textersetzung (Zeichenketten)
- Erzeugung: `NAME = Inhalt` (Leerzeichen sind erlaubt)
- Referenzierung über `$(NAME)`
- Nicht gesetzte Variablen dürfen verwendet werden und werten (analog zu Shellvariablen) zu einer leeren Zeichenkette aus
- Variablen können Listen von Zeichenketten enthalten, Einträge werden durch Leerzeichen voneinander getrennt

- Verbessertes Beispiel (jetzt mit anpassbarem Compiler und Optionen):

```
CC     = cc
CFLAGS = -g -O
LDLIBS = -lm

game: graphics.o physics.o input.o
    $(CC) $(LDFLAGS) -o game graphics.o physics.o input.o $(LDLIBS)

%.o: %.c
    $(CC) -c $(CFLAGS) $<
```

- Variablen können beim Aufruf von make gesetzt / überschrieben werden, z.B.

```
make CC='clang' CFLAGS='-O3 -march=native' LDFLAGS='-S'
```

- Macro Expansion zur Ersetzung von Dateiendungen innerhalb von Variablen (Makros)
  - Schreibweise: `$(MAKRO:subst1=subst2)`
  - Vorsicht: Wenn `MAKRO` dabei selbst eine Macro Expansion beinhaltet, ist Verhalten undefiniert
  - Beispiel:

    ```
    SOURCES = foo.c bar.c batz.c
    OBJECTS = $(SOURCES:.c=.o)
    ```

    Erzeugt die Namen der Object-Dateien zu den gegebenen Source-Dateien
    (entspricht `OBJECTS = foo.o bar.o batz.o`)

- Beliebige Ersetzungen auf Zeichenketten durch `subst` und `patsubst`
  - `subst` kann nur reine Textersetzung, `patsubst` kann auch Muster (`%`)
  - Syntax: `$(subst from,to,text)` bzw. `$(patsubst pattern,replacement,text)`
  - Beispiele:
    `$(subst ee,EE,Feet on the Street)` ergibt `FEEt on the StrEEt`
    `$(patsubst %.c,%.o,foo.c.c bar.c)` ergibt `foo.c.o bar.o`
  - Macro Expansion ist Teil der Kurzschreibweise von `patsubst`:
    `$(var:pattern=replacement)` entspricht `$(patsubst pattern,replacement,$(var))`

- Verbessertes Beispiel
  (Angabe der Sourcedateien und automatische Erzeugung der Liste von Zwischenzielen):

```
CC      = cc
CFLAGS  = -g -O
LDLIBS  = -lm

SOURCES = graphics.c physics.c input.c
OBJECTS = $(SOURCES:.c=.o)

game: $(OBJECTS)
    $(CC) $(LDFLAGS) -o game $(OBJECTS) $(LDLIBS)

%.o: %.c
    $(CC) -c $(CFLAGS) $<
```

- Automatisches Generieren von Eingabedatei-Listen durch `wildcard`-Funktion
  - Automatische Erweiterung (Wildcards durch die Shell, Muster durch make) findet nur innerhalb von Befehlszeilen (Shell) oder Muster-Regeln (%) statt
  - Automatische Erweiterung für Variablen mit `wildcard`-Funktion möglich
  - Beispiel:

    ```
    SOURCES = $(wildcard *.c)
    ```

    findet alle Dateien mir der Endung `.c` im aktuellen Verzeichnis
  - Sinnvoll in sehr großen Projekten (viele Eingabedateien) oder wenn die Namen der Eingabedateien nicht feststehen oder sich häufig ändern

Häufige zusätzliche Ziele (Allgemeine Konventionen)

- `all`: macht alles, also baut alle Ziele
- `clean`: räumt auf, also entfernt alle generierten Dateien
- Andere gängige Ziele (meistens bei größeren Projekten):
  - `doc`: Erzeugt die Dokumentation aus den Quelldateien
  - `test` oder `check`: Lässt eine Testsuite laufen
  - `install`: installiert das Projekt: Verzeichnisse, Binaries, Bibliotheken, manpages, etc.
  - `uninstall`: das Gegenteil von install
  - `mostlyclean`: wie `clean` allerdings werden aufwändig zu bauende Ziele nicht gelöscht
  - `dist`: Generiert ein Package (Installationspaket)
  - `distclean`: weitreichendes `clean`, löscht alles aus einem entpackten Package, was zum Zeitpunkt des Entpackens nicht da war

Einige zusätzliche vordefinierte spezielle Ziele (ohne Befehlszeilen)

- `.IGNORE`
  - Fehlersignalisierung der als Abhängigkeiten genannten Ziele wird ignoriert (kein Abbruch des make-Prozesses bei Fehlschlagen des Ziels)
  - wenn keine Abhängigkeiten definiert sind, so gilt `.IGNORE` für alle Ziele im Makefile
  - Verhalten kann auch nur für einzelne Befehlszeilen aktiviert werden, indem das Präfix `-` für die entsprechenden Zeile verwendet wird
  - das Verhalten von `.IGNORE` entspricht der Kommandozeilen-Option `-i`
- `.SILENT`
  - für die als Abhängigkeiten genannten Ziele erfolgt keine Ausgabe des Befehls vor der Ausführung
  - wenn keine Abhängigkeiten definiert sind, so gilt `.SILENT` für alle Ziele im Makefile
  - Verhalten kann auch nur für einzelne Befehlszeilen aktiviert werden, indem das Präfix `@` für die entsprechenden Zeilen verwendet wird
  - das Verhalten von `.SILENT` entspricht der Kommandozeilen-Option `-s`

- `.PRECIOUS`
  - im Falle einer asynchronen Benachrichtigung durch das System (`SIGHUP`, `SIGTERM`, `SIGINT` oder `SIGQUIT`) wird üblicherweise das aktuelle Ziel gelöscht, es sei denn es ist als Abhängigkeit von `.PRECIOUS` definiert
  - wenn keine Abhängigkeiten definiert sind, so gilt `.PRECIOUS` für alle Ziele im Makefile
  - wird üblicherweise nur in größeren / sehr aufwändigen `build`-Projekten eingesetzt
- `.PHONY`
  - Abhängigkeiten beschreiben Ziele, die selbst keine Dateien repräsentieren und werden von `make` immer gebaut (auch wenn Datei mit gleichem Namen vorhanden ist)
  - Angabe von `.PHONY` ohne Abhängigkeiten hat keine Auswirkung
  - sollte nicht für Ziele definiert werden, die eine Datei repräsentieren, da diese ansonsten ungeachtet der Aktualität ihrer Abhängigkeiten immer neu erzeugt wird;
    klassische Anwendungsfälle: `all`, `clean`, ...

- `.INTERMEDIATE`
  - Abhängigkeiten beschreiben Ziele, die von `make` explizit als Zwischenziele behandelt werden sollen, also automatisch wieder gelöscht werden, wenn das Hauptziel gebaut wurde
  - Ein `.INTERMEDIATE` ohne Angabe von Abhängigkeiten hat keinen Effekt
- `.SECONDARY`
  - Abhängigkeiten beschreiben Ziele, die explizit nicht als Zwischenziele angenommen werden, also nicht automatisch gelöscht werden sollen
  - Ein `.SECONDARY` ohne Abhängigkeiten deaktiviert das automatische Löschen von Zwischenzielen
- `.DELETE_ON_ERROR`
  - Wenn `DELETE_ON_ERROR` definiert ist, werden Zieldateien, die durch ein letztendlich fehlgeschlagenes Rezept vor dessen Fehlschlagen bereits erzeugt / verändert wurden, gelöscht

- `.POSIX`
  - stellt sicher, dass das Makefile anhand der im POSIX-Standard definierten Regeln interpretiert wird
  - muss, falls benötigt, in der ersten Zeile des Makefiles stehen
  - darf weder über Abhängigkeiten noch Befehlszeilen verfügen
  - stellt meistens leider nicht sicher, dass nicht POSIX-konforme Erweiterungen ausgeschaltet werden
- Sonderfall: `.DEFAULT`
  - darf keine Abhängigkeiten haben
  - definiert eine oder mehrere Befehlszeilen, die für Ziele, die nicht im Makefile definiert sind, ausgeführt werden
  - sollte nur verwendet werden, wenn es ein sinnvolles Default-Verhalten gibt

Komplettes Beispielmakefile (GNU make)

```
CC      = cc
CFLAGS  = -g -O
LDLIBS  = -lm
SOURCES = input.c physics.c graphics.c # Alternativ $(wildcard *.c)
OBJECTS = $(SOURCES:.c=.o)
BINARY  = game

.PHONY: all clean

all: $(BINARY)

$(BINARY): $(OBJECTS)
  $(CC) $(LDFLAGS) -o $(BINARY) $(OBJECTS) $(LDLIBS)

clean:
  rm -f $(BINARY) $(OBJECTS)

%.o: %.c
  $(CC) -c $(CFLAGS) $<
```

POSIX-konformes make

- An Dateiendungen gebundene Inferenz-Regeln (keine Muster-Regeln) für generalisierte Ziele
  - Schreibweise als Ziel mit Angabe der Quellendung links und der Zielendung rechts, z.B.: `.c.o:`
  - Inferenz-Regeln dürfen nur über Befehlszeilen verfügen (keine Quellangabe nach `:`)
  - Um eigene Inferenzregeln zu verwenden, muss `make` darüber informiert werden, dass die Endungen eine Inferenzregel haben, dazu dient das spezielle Ziel `.SUFFIXES:`
  - `make` kennt bereits diverse Inferenz-Regeln (z.B. um aus .c-Dateien .o-Dateien zu erzeugen)
- Interne Makros beschränkt auf: `$@`, `$*`, `$?`, `$%`, `$<`
  Sonderfall `$<`: nur innerhalb Inferenz-Regeln gültig
- Ersetzung von Dateiendungen in Variablen (Makros) nur in Form von Macro Expansion, keine zusätzliche Verarbeitung von Zeichenketten, keine Patterns
- Keine `wildcard`-Funktionalität
- Vordefinierte Ziele beschränkt auf: `.POSIX`, `.IGNORE`, `.SILENT`, `.PRECIOUS`, `.DEFAULT`, `.SUFFIXES` (siehe Inferenz-Regeln), `.SCCS_GET` (nur für Versionsverwaltungssystem SCCS)

“Komplettes” POSIX-konformes Makefile

```
.POSIX:
.SUFFIXES:

CC     = cc
CFLAGS = -g -O
LDLIBS = -lm

SOURCES = graphics.c physics.c input.c
OBJECTS = $(SOURCES:.c=.o)
BINARY  = game

all: $(BINARY)

$(BINARY): $(OBJECTS)
    $(CC) $(LDFLAGS) -o $(BINARY) $(OBJECTS) $(LDLIBS)

clean:
  rm -f $(BINARY) $(OBJECTS)

.SUFFIXES: .c .o
.c.o:
  $(CC) -c $(CFLAGS) $<
```

Zusammenfassung

- Buildfiles (z.B. Makefiles)
  - erleichtern und optimieren automatische Erstellungsprozesse erheblich
  - werden auch verwendet um gemeinsamen Nenner (Compilerflags etc.) in Projekten festzulegen
- POSIX-konformes `make`
  - kleinster gemeinsame Nenner von Build-Systemen, zugeschnitten auf Software-Builds
  - Vorteil: hohe Portabilität
  - Nachteil: relativ unflexibel / eingeschränkt
- GNU `make`
  - eher allgemein gehaltenes Build-System mit vielen Features
  - Vorteile: hohe Flexibilität, viel Komfort, viele Erweiterungen, verarbeitet problemlos POSIX konforme Makefiles
  - Nachteile: Funktionsumfang ggfs. versionsabhängig, eingeschränkt portabel

Ausblick: Build-Verwaltung in größeren Systemen

- "Magische Beschwörungsformel":

```
./configure
make
make install
```

- Bei großen Softwareprojekten Einsatz von speziellen Tools zu Erzeugung von passenden und auf das System angepassten Makefiles
  - `configure`:
    - Shellskript zur Erzeugung von Makefiles aus Makefile-Stubs (Makefile.in) mit Platzhaltern
    - erkennt System und bereits installierte Software und überprüft Abhängigkeiten
    - ...
  - Nächste Stufe `autoconf`:
    - erzeugt ein `configure`-Skript aus einem Stub (configure.ac) mit Variablen und Definitionen
  - Nächste Stufe `automake`:
    - selbst Makefile.in wird generiert aus noch kleinerem Makefile.am
  - ...

The End

- Evaluierung (Zeitraum wird über Moodle angekündigt)
  - bitte evaluieren, Feedback ist wichtig und wird ernst genommen
  - gerne auch ausführliche Kommentare: wie kam das Konzept an, was können wir verbessern, was war gut, was nicht so ...
  - Martin Dietze freut sich auch, wenn er über die Kommentare Feedback zu seinem Veranstaltungsteil bekommt
- Feedback außerhalb der Evaluierung natürlich auch immer willkommen
- Vielen Dank fürs Zuschauen!