Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

Seminar Effiziente Programmierung - Buildsysteme

vorgelegt von

Yannic Köster

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Studiengang: Software-System-Entwicklung
Matrikelnummer: 6770103

Betreuer: Michael Kuhn

Hamburg, 2018-02-28

Abstract

Diese Ausarbeitung beschäftigt sich mit Buildsystemen und legt dar, was diese sind und in welche Gruppen die behandelten Systeme unterteilt werden können und wie diese miteinander agieren. Die behandelten 3 Buildsystemen sind Make, CMake und Meson und es soll eine grundlegende Aussicht auf diese und dessen Aufbau liefern. Es werden die Konfigurationsdateien der Systeme erläutert und es werden die Unterschiede der Systeme dargelegt.

Inhalt

1 Buildsysteme

Der Begriff Buildsystem umfasst Systeme die dazu dienen, den Bauprozess von Software zu unterstützen. Hierbei haben alle Systeme gemein, dass diese grundlegend den vom Programmierer geschriebenen Code kompilieren, linken und verpacken und diesen Vorgang aufs möglichste automatisieren.
Darüber hinaus bieten viele Systeme noch weitere Funktionen an wie zum Beispiel den automatischen Download von Abhängigkeiten.

Hierbei kann man unterscheiden zwischen Buildtools, welche den Bau tatsächlich selbst durchführen, und jenen welche ein anderes Buildtool hierfür als Abhängigkeit nutzen um diese Aufgabe zu erfüllen. Diese Tools steuern also die eigentlichen Buildtools und generieren für diese Konfigurationsdateien. Um die Übersicht zu wahren, werden diese Systeme innerhalb dieser Ausarbeitung als "Metabuildsysteme" bezeichnet. Auf die hier behandelten Tools bezogen vollführt Make den Bau selbständig, während CMake und Meson auf Make und/oder Ninja zurückgreifen.

Generell bietet das Nutzen dieser Systeme einige Vorteile. Zunächst einmal ist es sinnvoll eine konsistente Umgebung zu haben, in welcher der Bauprozess immer auf gleiche Weise ausgeführt wird. Hierdurch entstehen keine ungewollten Variationen im Bauprozess die eventuell zu Fehlern führen, welche dann mühselig ausfindig gemacht werden müssen.
Diese Konsistenz kann auch über unterschiedliche Rechner gewährt werden dank Überprüfungen von Abhängigkeiten, sodass Fälle von "Es funktioniert auf Rechner A, aber nicht auf Rechner B" nicht auf den Bauprozess zurückzuführen sind.
Da dieser Prozess automatisiert ist, verringert sich auch die Rate von Fehlern, welche von dem Benutzer versehentlich durch Fahrlässigkeit ausgelöst werden können. Ein sehr wichtiger Punkt, den Buildsysteme mit sich bringen, ist das Sparen von Zeit. Ist der Prozess einmal eingerichtet, so fallen im Laufe der Entwicklung zwar immer wieder Änderungen an, jedoch bedarf nicht jeder einzelne Bau der Software einen großen Zeitaufwand, sondern lediglich eine Ausführung des Buildtools.
Auf jeden Bauvorgang an sich bezogen ist ein Buildtool sinnvoll, da unveränderter Code nicht neu gebaut werden muss, sondern nur jener, an welchem wirklich gearbeitet wurde. So verringert sich die tatsächliche Zeit, welche der Computer für den Bau braucht. Dies ist vor allem sinnvoll bei großen Projekten welche aus vielen einzelnen Dateien bestehen.

2 Make

Make ist ein Buildtool welches den Softwarebau eigenständig durchführt, ohne Einbindung eines anderen Buildtools. Für die Kommunikation zum Nutzer für Instruktionen zum Bau, nutzt Make eine Datei namens “makefile”. Diese Datei enthält alle Anweisungen zum Bau der Software inklusive Angabe aller Quelldateien und Abhängigkeiten. Für den Bau von Software benötigt Make also den entsprechenden Quelltext, als auch ein auf den Quelltext angepasstes Makefile. Im Folgenden wird behandelt, was solch ein Makefile im Grundlegenden beinhalten sollte.

2.1 Makefile

Bevor mit der Konstruktion gestartet wird, ist es ratsam einige Variablen zu erstellen, auf dass diese leicht änderbar sind und der Quelltext leichter zu schreiben / übersichtlicher ist. In einem Makefile sehen Variablenbelegung und Variablenaufruf wie folgt aus:

```
1 CC = g++      //Variablenbelegung
2 $(CC)         //Variablenaufruf
```

Listing 2.1: Make - Variablen

Ein Makefile besteht aus sogenannten “Regeln”:

```
1 Ziel : Vorbedingungen
2 Anweisungen
```

Listing 2.2: Make - Regeln

Jede Regel besitzt ein Ziel, z.B. eine Datei die hierbei erstellt wird, Vorbedingungen, welche Dateien darstellen, die hierfür benötigt werden, und Anweisungen, wie das besagte Ziel wirklich konstruiert wird.

Minimal muss zunächst einmal ein Ziel angegeben werden, welches im Endeffekt gebaut werden soll. Wenn Make ohne Argumente in der Konsole aufgerufen wird, so wird das erste Ziel gebaut. Von daher ist es sinnvoll, zunächst ein Ziel zu erstellen, welches unser gewünschtes Ziel als Vorbedingung nutzt. Auf diese Weise kann dieses leicht verändert werden, ohne die Struktur des Makefile ändern zu müssen.

```
1 default : Demo
```

Listing 2.3: Make - Angabe eines Ziels

In diesem Beispiel ist nun “Demo” der Name des eigentlichen Ziels, welches gebaut werden soll. Nun muss das dieses auch tatsächlich konstruiert werden. Hierfür definieren wir folgende Regel. In diesem Beispiel besteht das Projekt aus 2 Klassen:

```
1 Demo : main.o Klasse2.o
2 $(CC) $(CFLAGS) -o Dateiname main.o Klasse2.o
```

Listing 2.4: Make - Definition eines Ziels

“main.o” und “Klasse2.o”, welche direkt nach dem Demo folgen beschreiben die Abhängigkeiten, welche benötigt werden, um das Ziel zu bauen. Grundlegend sind dies immer mindestens die Objektdateien der einzelnen Klassen, welche ebenfalls noch innerhalb des Makefile beschreiben werden müssen, damit diese erstellt werden können. Des weiteren geben wir den genutzten Compiler an und eventuelle Flags für diesen. Sinnvoll ist hier das Nutzen von Variablen. Zudem sollte ein Dateiname für das Ziel bestimmt werden. Dies tun wir mit dem Flag -o. Zuletzt ist eine Angabe nötig aus welchen Objektdateien letztendlich das Programm besteht. Die Syntax für die Regeln dieser Objektdateien verfolgt auf gleiche Weise, wie bei dem Ziel:

```
1 main.o : main.cpp header.h
2 $ ( CC ) $ ( CFLAGS ) -c main.cpp
3 Klasse2.o : Klasse2.cpp header.h
4 $ ( CC ) $ ( CFLAGS ) -c Klasse2.cpp
```

Listing 2.5: Make - Objektdateien erstellen

Direkt nach der Deklaration der Objektdatei werden wieder die Abhängigkeiten deklariert wie Klassendateien oder auch Headerdateien. Anschließend wieder der Compiler und eventuelle Flags von diesem und zum Schluss die Quelldateien, aus denen das Objekt gebaut wird.

Um sicherzugehen, dass auf verschiedenen Rechnern das Programm einwandfrei funktioniert, ist das Überprüfen von externen Abhängigkeit ein essenzieller Bestandteil. Diese Abhängigkeiten können als Argument dem Bau der Zieldatei übergeben werden:

```
1 dependency1 = -lboost_date_time
2 Demo : main.o Klasse2.o
3 $(CC) $(CFLAGS) $(dependency1)  -o Dateiname main.o
    ↪ Klasse2.o
```

Listing 2.6: Make - Abhängigkeiten erstellen

Für die einzelnen Klassenobjekte wird keine extra Angabe der Abhängigkeit benötigt, lediglich bei dem Erstellen des Ziels.

3 CMake

CMake ist ein Metabuildsystem, welches den Bau der Software nicht selbst ausführt, sondern stattdessen ein anderes Tool wie Make oder Ninja hierfür nutzt und für dieses eine Konfigurationsdatei erstellt.
Für die Verwendung wird also wie zuvor auch schon bei Make der Quellcode benötigt, und eine Konfigurationsdatei für CMake, eine "CMakeLists.txt"-Datei. Führen wir nun CMake aus, so erstellt CMake für den Nutzer die entsprechende Konfigurationsdatei des tiefer liegenden Systems, im Falle von Make das "Makefile" und im Falle von Ninja "build.ninja". Von hier aus wird Make bzw. Ninja genutzt um das Programm zu bauen. Sollten Änderungen an "CMakeLists.txt" anfallen, so wird beim Ausführen von Make bzw. Ninja ebenfalls die entsprechende Konfigurationsdatei automatisch angepasst.

Standartmäßig nutzt CMake Make auf unterer Ebene. Um ein Projekt mittels Ninja zu erstellen, wird das Argument "-GNinja" bei dem Erstellen verwendet.

3.1 CMakeLists.txt

Innerhalb der CMakeLists.txt definiert der Benutzter sein Projekt anhand von bereits vorhandenen, oder aber auch vom Nutzer selbst definierten Funktionen.
Die Belegung von Variablen erfolgt ebenfalls über eine Funktion namens "set()". Hierfür wird die Funktion aufgerufen und als ersten Parameter der Name der Variable übergeben, und im Anschluss darauf die zugehörige Belegung. Das folgende Beispiel zeigt eine Variable für die Quelldateien eines Projektes.

```
1 set(src main.cpp Klasse2.cpp)
```

Listing 3.1: CMake - Variablenbelegung

Wollen wir nun eine Variable aufrufen, um sie beispielsweise als Parameter zu nutzen, so tun wir dies mit folgender Syntax:

```
1 ${src}
```

Listing 3.2: CMake - Variablenaufruf

CMake benötigt zunächst einige Informationen über das Projekt.

```
1 Project ( Name Sprachen )
2 cmake_minimum_required( VERSION 3.0)
```

Listing 3.3: CMake - Project

In der Funktion "Project()" muss der Name des Projektes angegeben werden. Hierbei müssen auch noch die Sprachen des Programmes angegeben werden, damit diese für das Projekt aktiviert werden. C und C++ müssen nicht spezifisch angegeben werden, da diese Sprachen als Standard gelten.
Des weiteren sollte angegeben werden, welche CMake Version minimal für das Projekt genutzt werden soll, um Kompatibilitätsprobleme zu vermeiden.
Soll das Programm ausführbar sein, so muss ebenfalls hierfür folgender Code hinzugefügt werden:

```
1 add_executable(Name Quelldateien)
```

Listing 3.4: CMake - executable

Hierbei wird ein Name der Datei bestimmt und die Quelldateien angegeben, aus welcher diese erstellt wird, wobei dieser Name hierbei innerhalb des Projektes einzigartig sein muss. Die Datei wird ohne weitere Angaben beim ausführen von Make/Ninja in dem Projektverzeichnis erstellt.
Für Abhängigkeiten in CMake muss zunächst einmal das entsprechende Paket gesucht, und anschließend in das Projekt eingebunden werden. Anschließend wird dies mit der ausführbaren Datei gelinkt.

```
1 find_package ([Paket])
2 include_directories ( $ {[Paket]_INCLUDE_DIRS })
3 target_link_libraries ([Ausfuehrbare_Datei]
    ↪ ${[Paket]_LIBRARIES})
```

Listing 3.5: CMake - Abhaegigkeiten

Durch "find_ package()" wird hierbei ein externes Projekt gesucht und gibt unserem Projekt Rückmeldung, ob dieses existiert und liefert gegebenenfalls Informationen über dieses.
Durch "include_ directories()" wird nun dem Compiler Anweisung gegeben, dieses beim Bau zu berücksichtigen.
"target_ link_ libraries()" spezifiziert nun, dass diese Abhängigkeit bei dem Linken der ausführbaren Datei mitbenutzt wird.

4 Meson

Meson ist ein Metabuildtool welches Ninja für den eigentlichen Bau einbindet. Hierfür nutzt Meson eine Konfigurationsdatei namens “meson.build” aufgrund welcher die Konfigurationsdatei für Ninja, “build.ninja” erstellt wird. Im Gegensatz zu CMake implementiert Meson nicht Make, sondern nur Ninja. Dies hat den Grund, dass Meson Make als zu langsam empfindet, und dies auch nicht durch Codeverbesserungen zu beheben ist[7].
Dies stützt Meson mit einigen Tests innerhalb ihrer Dokumentation. Des weiteren ist Meson in Python geschrieben und benötigt von daher min. Python 3.4 auf dem entsprechenden Rechner, als auch Ninja 1.5 oder neuer.
Der Ablauf erfolgt hier auf gleiche Weise wie bei CMake mit Ausnahme, dass wir nicht innerhalb des gleichen Verzeichnisses, in welchem sich Quellcode und meson.build nefomdem, die Ausgabe erzeugen können. Hierfür wird ein neues Verzeichnis benötigt. Zunächst wird also Meson ausgeführt, wofür wir ein Verzeichnis für das Projekt erstellen bzw. ein vorhandenes nutzen. Hierdurch wird dann die build.ninja-Datei erstellt und der eigentliche Bau erfolgt in Zukunft durch Ninja. Hierbei ist anzumerken, dass Ninja darauf ausgelegt ist, von anderen Tools wie z.B. Meson genutzt zu werden. Es ist nicht dazu gedacht von Menschenhand geschrieben zu werden so wie Make.[8]. Von daher sind die build.ninja-Dateien nicht auf Übersichtlichkeit ausgelegt, da die Geschwindigkeit im Vordergrund steht.
Werden nun Änderungen an der meson.build-Datei vorgenommen, so wird beim ausführen von Ninja die build.ninja-Datei angepasst, um dem neuen Projekt zu entsprechen.

4.1 meson.build

Innerhalb der meson.build Datei werden vom Nutzer hauptsächlich Variablen deklariert und Funktionen von Meson aufgerufen. Hierbei können vom Nutzer keine eigenen Funktionen generiert werden. Dies ist eine bewusste Entscheidung von Meson, da vermieden werden soll, dass Meson Turing-Vollständig ist. Dies ermöglicht eine wesentlich einfachere Implementierung [9]. Des weiteren soll sich die Möglichkeiten offen gehalten werde, von Python umzusteigen, sollte es sich in der Zukunft als limitierender Faktor für Meson erweisen.[10]

Variablen werden innerhalb von Meson wie folgt angelegt:

```
1 src = ['main.cpp', 'Klasse2.cpp']
```

Listing 4.1: Meson - Variablenbelegung

In diesem Beispiel werden 2 Quelldateien and die Variable “src” gebunden.
Strings werden mit einem Apostroph ‘Ausdruck‘ eingeklammert, während Variablen keinerlei Klammern erhalten.
Als Parameter aufgerufen werden können diese Variablen ohne weitere spezielle Notationen, sie müssen lediglich an die entsprechende Parameterstelle einer Funktion geschrieben werden entsprechend ihres gewählten Namens. Eine Projektdeklaration in Meson benötigt zunächst einen Namen und die verwendeten Programmiersprachen. Der Name kann hierbei ein String nach eigenem Belieben sein.

```
1 project ('Showprogramm' , 'cpp')
```

Listing 4.2: Meson - Project

Für die Erstellung einer ausführbaren Datei nutzen wir folgendes Kommando:

```
1 executable ( 'Showprogramm' , src )
```

Listing 4.3: Meson - Project

Hier geben wir wieder einen Namen der Datei an, welche im Projektordner erstellt wird, und anschließend die Quelldateien.
Für die Nutzung von Abhängigkeiten muss in Meson ist die “dependency()” Funktion notwendig, welche dem Erstellen der ausführbaren Datei hinzugefügt werden muss:

```
1 dep = dependency('Name', version : '>=x.y')
2 executable ('Showprogramm' ,src , dependencies : dep)
```

Listing 4.4: Meson - Abhaengigkeiten

Mithilfe von “dependencies :” und im Anschluss Abhängigkeit als Argument können neue Abhängigkeiten hinzugefügt werde. Hierbei muss in der Funktion “dependency()” der Name und wahlweise eine Mindestversion übergeben werden. Der Übersichtlichkeit halber ist es ratsam diesen Code in eine Variable auszulagern.

5 Vergleich

5.1 Performance

Um die Performance zu vergleichen, habe ich den Linuxbefehl “time” zusammen mit meinem Beispielcode aus meinem Vortrag verwendet. Da es sich bei dem Beispielprogramm des Vortrages um kein großes Programm handelt und daher die Bauzeit nicht lang sein wird, habe ich die Bauvorgänge mehrere Male durchgeführt und das beste Ergebnis aufgelistet, um Variationen bestmöglich zu beachten. Zunächst die Zeiten zum erstellen der Projektstruktur. Im Falle von Make ist keine Zeit angegeben, da hier dieser Schritt nicht existiert / notwendig ist.

Buildtool	Zeit in Sekunden
make	-
CMake mit Make	0,97
CMake mit Ninja	0,78
Meson	0,38

Table 5.1: Zeit zum Bau der Projektstruktur

Die Bauzeiten haben einige Variationen, mit Meson weit vorn, allerdings ist dieser Schritt nur ein einziges mal zu Beginn des Projektes notwendig und von daher nicht der hauptsächliche Punkt, um die Performance zu vergleichen.

Die tatsächlichen Bauzeiten des Programmen ist stattdessen viel wichtiger, da dieser Prozess immer wieder stattfindet:

Buildtool	Zeit in Sekunden
make	1,99
CMake mit Make	1,93
CMake mit Ninja	1,88
Meson	1,98

Table 5.2: Bauzeiten der Builsdysteme

Es ist zu sehen, dass alle Systeme eine ähnliche Geschwindigkeit beim Bau aufweisen.

5.2 Syntax

Von Seiten der Syntax ähneln sich Meson und CMake sehr, während Make sich hierbei absondert. Hierbei ist das Makefile wesentlich weniger kompakt und weniger lesefreundlich, da mit dieser Datei direkt der Bau vorgenommen wird, während CMake und Meson lediglich zunächst ein Makefile bzw. eine build.ninja-Datei erstellen müssen.
CMake und Meson bieten hierbei ihre eigene Syntax, welche ähnlich zueinander aufgebaut ist, wobei beide größtenteils über Funktionsaufrufe arbeiten. Hierbei stellt sich Meson als um einiges kompakter als CMake heraus, und vollführt beispielsweise die Handhabung von Abhängigkeiten wesentlich kompakter als CMake und um einiges klarer.

5.3 Verbreitung

Bei Make handelt es sich um das älteste aller 3 Systeme, wurde 1976 veröffentlicht, und ist damit hierdurch weit verbreitet und selbst wenn es nicht direkt genutzt wird, wird es von anderen Metabuildtools wie z.B. CMake oder auch in Kombination anderer Programme als Teil der GNU Autotools genutzt. Eine direkter Anwendung von Make findet beispielsweise bei dem Linux Kernel statt.

CMake wurde im Jahr 2000 veröffentlicht. Wenn auch nicht so alt wie Make, ist es dennoch schon lang verfügbar und wird von Programmen wie Netflix, Blender oder MySQL verwendet.

Meson ist im Vergleich zu den beiden anderen Systeme relativ jung und wurde erst 2013 veröffentlicht. Aufgrund des geringen Alters weist es keine so große Verbreitung auf, wie die anderen beiden Systeme, wurde jedoch beispielsweise schon als alternatives Metabuildsystem des Xorg Servers eingeführt (neben der Nutzung von Autotools).

6 Fazit

Nachdem ich einige Erfahrung mit allen 3 Buildsystemen gesammelt habe bin ich zu dem Entschluss gekommen, dass die Arbeit mit Meson und CMake sich um ein wesentliches einfacher gestaltet, als die direkte Arbeit mit Make. Make allein benötigt im Vergleich mehr Code als CMake und Meson, je größer der Code wird. Beispielsweise das Hinzufügen einer neuen Klasse benötigt Codeänderungen an mehreren Stellen, während in CMake/Meson lediglich eine Änderung vonnöten ist, indem der Name der Datei hinzugefügt wird. Des weiteren ist es einfacher, mit Cmake/Meson ein Projekt für verschiedene Betriebssystemen aufrecht zu erhalten, da hierfür weniger Anpassungen nötig sind.

Im Vergleich von CMake und Meson ist das Ergebnis weniger eindeutig. Die beiden Metabuildsysteme binden beide auf unterer Ebene ein anderes Buildsystem ein und nutzen dies für den Bauvorgang. Bei der Wahl dieses Systems ist CMake vielseitiger, welches Make oder auch Ninja nutzen kann, während Meson, auch wenn es eine bewusste Entscheidung ist, auf Ninja beschränkt ist.
Syntax-technisch ist Meson im Vergleich zu CMake um einiges kompakter, und vollführt beispielsweise die Handhabung von Abhängigkeiten wesentlich kompakter als Make. Dadurch war das grundlegende Erlernen von Meson um einiges einfacher.
Zur Dokumentation ist zu sagen, dass Meson zwar eine gute Einführung zur Funktionalität bietet, jedoch die genauerer Dokumentation aller einzelnen Funktionen lediglich auf einer URL zusammen mit kurzen Beschreibungen aufzufinden ist. Daher ist es ein wenig umständlich, Informationen über eine bestimmte Funktion zu finden, da es keine dedizierte Suche für diese gibt. Bei CMake hingegen gibt es eine strukturierte Dokumentation, mit dedizierten Seiten zu einzelnen Funktionen. Zu diesem Punkt ist jedoch anzumerken, dass Meson im Vergleich zu CMake noch relativ jung ist und von daher gut zu vermuten ist, dass hier in Zukunft noch strukturelle Verbesserungen zutage kommen.
Von Seiten der Performance gibt es keine großen Unterschiede und sind zum größten Teil recht ähnlich.

Im Endeffekt läuft es darauf hinaus, welches der Systeme man in der Handhabung und wessen Syntax man bevorzugt. Alle Systeme haben sich bewährt und finden Anwendung.

6.1 Anhang - Builddateien des Vortrags

Buiddateien des Vortrags für 2 Klassen und einen Header.

```
1  CC = g++ //Compiler Variable
2  CFLAGS  = -g -Wall //Compiler Flags Variable
3  LDFLAGS = -lboost_date_time //Abhängigkeit Variable
4
5  default: Demo        //Standardziel
6
7  Demo:  main.o Klasse2.o  //Ziel definieren
8      $(CC) $(CFLAGS) $(LDFLAGS) -o Demo main.o Klasse2.o
9  main.o:  main.cpp header.h  //Objektdatei definieren
10     $(CC) $(CFLAGS) -c main.cpp
11 Klasse2.o:  Klasse2.cpp header.h //Objektdatei definieren
12     $(CC) $(CFLAGS) -c Klasse2.cpp
13
14 clean: //Befehlt zum löschen von Projektdaten
15     $(RM) Demo *.o *~
```

Listing 6.1: makefile

```
1  Project(new) //Projektnamen definieren
2  cmake_minimum_required(VERSION 3.0) //Mindest CMake Version
3
4  set(src main.cpp Klasse2.cpp)//Quelldatei Variable
5
6  find_package(Boost REQUIRED COMPONENTS date_time)
     ↪ //Abhängigkeit suchen
7
8  //Prüfen, ob Abhängigkeit gefunden.
9  //Wenn ja, dann Projekt mit Abhängigkeit bauen.
10 //Wenn Abhängigkeit nicht gefunden, Projekt nicht bauen
11 if(Boost_FOUND)
12   include_directories(${Boost_INCLUDE_DIRS})
13   add_executable(demo ${src})
14   target_link_libraries(demo ${Boost_LIBRARIES})
15 endif()
```

Listing 6.2: CMakeLists.txt

```
1 project('Showprogramm', 'cpp') //Projekt definieren
2 src = ['main.cpp','Klasse2.cpp'] //Quelldatei Variable
3
4 //Abhängigkeit suchen
5 boost_dep = dependency('boost', modules : ['date_time'])
6
7 //Ziel erstellen und Abhängigkeit hinzufügen
8 exe = executable('Showprogramm', src , dependencies :
    ↪ boost_dep)
```

Listing 6.3: meson.build

Quellen

[1] GNU Make, 28.2.2018
https://www.gnu.org/software/make/

[2] GNU Make Dokumentation, 28.2.2018
https://www.gnu.org/software/make/manual/make.html

[3] Github Projekt CMake, 28.2.2018
https://github.com/Kitware/CMake

[4] CMake Dokumentation, 28.2.2018
https://cmake.org/documentation/

[5] CMake Tutorial, 28.2.2018
https://cmake.org/cmake-tutorial/

[6] Meson Dokumentation, 28.2.2018
http://mesonbuild.com/index.html

[7] Meson Stellung zu Make, 28.2.2018
http://mesonbuild.com/FAQ.html#why-is-there-not-a-make-backend

[8] Ninja Design, 28.2.2018
https://ninja-build.org/manual.html#_design_goals

[9] Meson Stellung zu eigenen Funktionen, 28.2.2018
https://github.com/glslang/meson/commit/eefcec28c818c324d4ed4d576cee193ea5a4689

[10] Meson zu Python, 28.2.2018
http://mesonbuild.com/FAQ.html#why-is-meson-not-just-a-python-module-so-i-could