Modulare Programmierung in C

Einführung in die Programmierung

Michael Felderer (QE)

Institut für Informatik, Universität Innsbruck

Entwicklung eines Programms

- Softwareprojekte werden in mehrere voneinander abgegrenzte Phasen unterteilt.
- Man erhält einen Softwarelebenszyklus, einen so genannten **Software Life Cycle**.
- Es gibt zwar einen eindeutigen Ablauf eines Software Life Cycles, allerdings aber keine einheitliche Form der Darstellung.
- Typische Phasen (im historischen Wasserfallmodell)
  - Problemanalyse
  - Systementwurf
  - Programmentwurf
  - Implementierung und Test
  - Betrieb und Wartung

Phasen (1)

- **Problemanalyse**, auch Anforderungs- oder Systemanalyse genannt
  - Diese Analyse wird in Zusammenarbeit mit dem Auftraggeber durchgeführt.
  - Das Ergebnis ist eine Anforderungsbeschreibung (auch Pflichtenheft).
- **Systementwurf**
  - Hier werden die zu lösenden Aufgaben in so genannte **Module** aufgeteilt.
    - Ein Modul ist eine abgeschlossene funktionale Einheit einer Software, bestehend aus einer Folge von Verarbeitungsschritten und Datenstrukturen.
    - Erhöht die Übersichtlichkeit und verbessert die Korrektheit und Zuverlässigkeit.
  - Das Ergebnis ist eine Systemspezifikation (Grundlage für die Implementierung).
- **Programmentwurf**
  - In dieser Phase werden die einzelnen Module weiter verfeinert, indem die Datenstrukturen festgelegt und Algorithmen entwickelt werden.
  - Das Ergebnis besteht aus mehreren Programmspezifikationen.

Phasen (2)

- **Implementierung** und **Test**
  - Hier werden die Module programmiert und anhand ihrer jeweiligen Spezifikation getestet (verifiziert).
  - Durch das Zusammensetzen der einzelnen Module erhält man das Programm.
- **Betrieb** und **Wartung**
  - Diese Phase umfasst die Pflege der Software, in der Erweiterungen und Änderungen eingebracht oder entdeckte Fehler behoben werden.
  - Unter Umständen führt dies wieder zu einer vorherigen Phase oder überhaupt zur Problemanalyse zurück, wodurch ein Zyklus entsteht.

Das Wasserfallmodell wird heute kaum mehr verwendet!

Etliche Erweiterungen bzw. Veränderungen werden mittlerweile in der Praxis verwendet (z.B. V-Modell XT, Agile Softwareentwicklung)

Implementierung und Test

- Mit dieser Phase haben wir uns **teilweise** in dieser LV beschäftigt!
- Bisherige Programme waren recht klein und überschaubar.
  - Wenige Funktionen
  - Eine Datei
- Reale Programme bestehen aus hunderttausenden Zeilen Programmcode.
- Wie organisiert man diesen Code?
  - Eine Datei wird dabei nicht mehr ausreichend sein!
  - Größere Programme werden in mehrere Dateien aufgeteilt.

C und Software Engineering

- C unterstützt das sogenannte **Structured Design**
  - Dabei wird das Programm in Unterprogramme zerlegt, die zu anderen Unterprogrammen möglichst wenig Querbeziehungen haben.
- In C kann man solche Unterprogramme mit **Funktionen** realisieren.
  - Diese Funktionen sollten nur eine minimale Schnittstelle zu Ihrer Umgebung haben.
    - Wenige Übergabeparameter.
    - Möglichst keine globalen Variablen!
  - Durch den Aufruf von Funktionen entstehen Beziehungen und Abhängigkeiten.
    - Je kleiner die Schnittstelle, desto leichter lässt sich eine Implementierung austauschen.

Modulares Design

- Die Funktionen selbst können wiederum in größere Einheiten zusammengefasst werden.
- Solche Einheiten bezeichnet man als Module.
- Ein Programm besteht dann in der Regel aus mehreren Modulen.

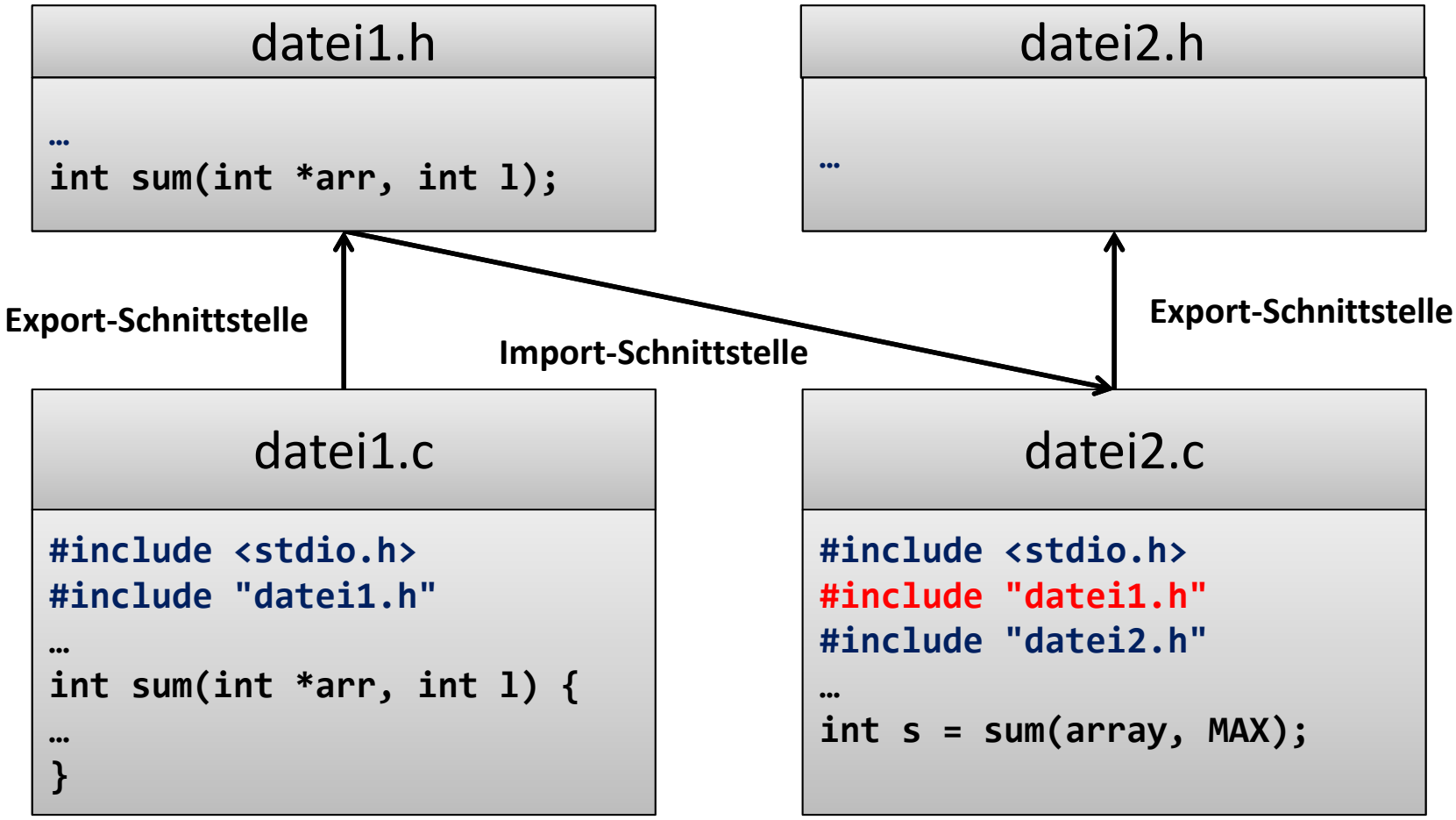
Modul – Kapselung und Information Hiding

- Kapselung
  - Kapselung der Daten und Funktionen als Einheiten.
  - Definition der Schnittstellen zwischen den einzelnen Einheiten.
- **Export-Schnittstelle**
  - Über diese Schnittstelle stellt ein Modul Ressourcen für andere Module zur Verfügung.
  - Alle anderen Interna sind verborgen (Information Hiding).
  - Entwickler kann bestimmen, was von einem Modul benutzt werden kann (Export-Schnittstelle) und was verborgen bleibt (Information Hiding).
- **Import-Schnittstelle**
  - Zur Implementierung eines Moduls kann man andere Module benutzen, die man in der Import-Schnittstelle auflistet.
    - Modul ist ersetzbar durch ein Modul gleicher Export-Schnittstelle.
    - Export- und Import-Schnittstelle können überprüft werden.

Modulares Design in C

- Modul = Datei
- Information Hiding mit `static`
  - Globale Variablen und Funktionen, die im Modul verborgen bleiben sollen (nur im Modul kann man darauf zugreifen), werden mit dem Schlüsselwort `static` versehen.
  - Funktionen aus anderen Dateien können dann nicht mehr darauf zugreifen.
- Trennung von Schnittstelle und Implementierung mit **Header-Datei**
  - Die Header-Datei enthält die Funktionsprototypen und entspricht damit der Schnittstelle.
    - Daneben kann eine Header-Datei noch globale Variablen, Makros etc. enthalten.
    - Enthält aber immer nur Deklarationen (keine Definitionen).
  - Die C-Datei, die die Funktionen einer Header-Datei realisiert, stellt die Implementierung dar.
  - C- und Header-Datei bilden eine logische Einheit.
    - Sollte sich auch im Dateinamen ausdrücken.
    - Gleicher Name, unterschiedliche Dateinamenserweiterung.

Modulares Design in C (schematisches Beispiel)



Datei **datei2.c** benutzt Teile (zum Beispiel Funktionen) aus **datei1.c**. Mit Hilfe von **datei1.h** werden die Funktionsprototypen der **datei1.c** in **datei2.c** bekannt gemacht.

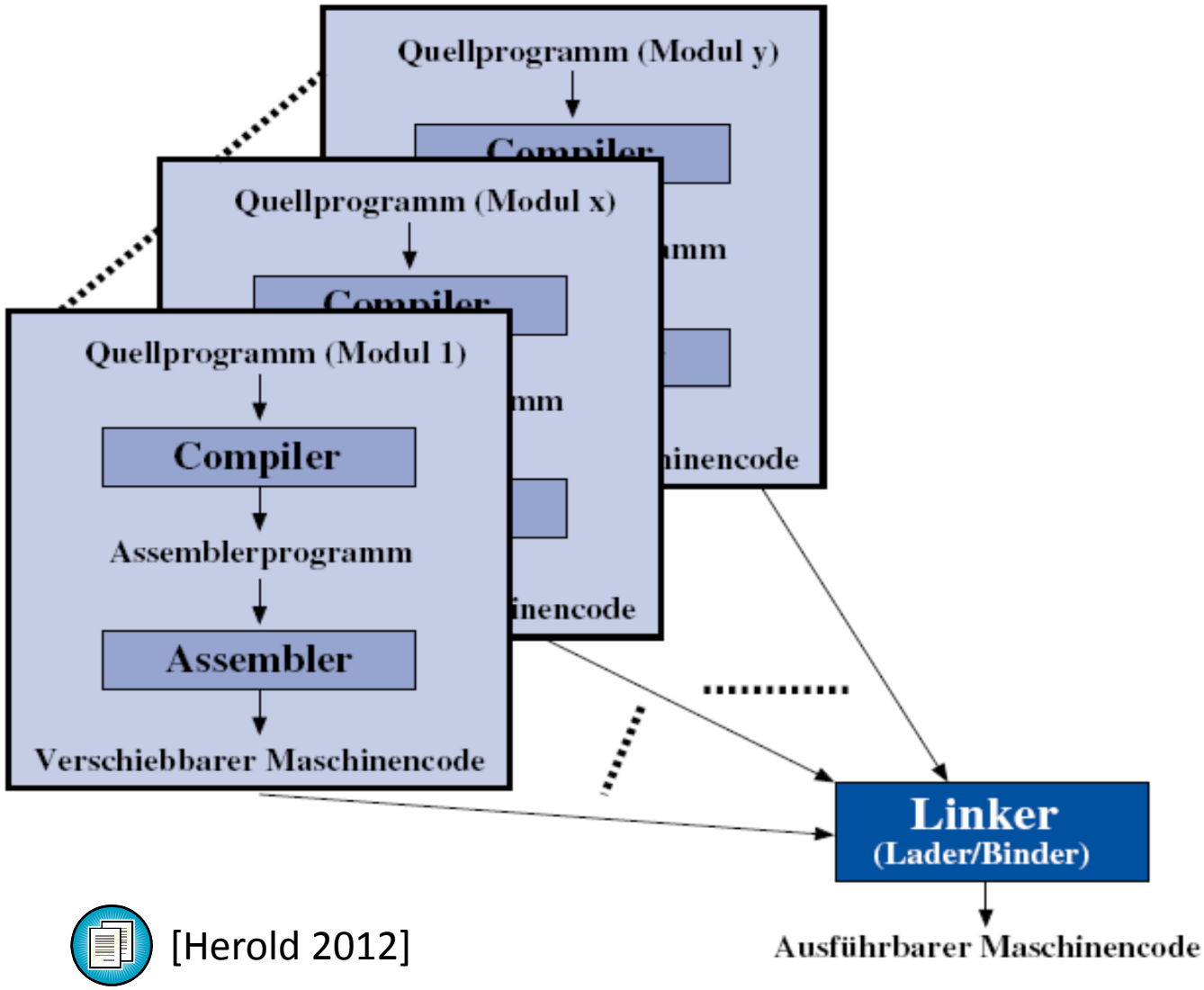
Einfache Regeln für Header-Dateien

- Jede Funktion, die **außerhalb** der Datei benutzt werden soll, in der sie **definiert** ist, erhält einen **Prototypen** in einer **Header-Datei.**
- Jede Datei, die einen **Aufruf einer Funktion** (nicht in der Datei definiert) enthält**, inkludiert** die entsprechende Header-Datei.
- Die Quelldatei, die die Definition einer Funktion enthält, **soll** die Header-Datei ebenfalls inkludieren.

Programme aus mehreren Dateien in C

- C unterstützt eine getrennte Übersetzung
- Jedes Modul (Datei) kann für sich getrennt übersetzt werden.
- Unterschiedliche Module können zu unterschiedlichen Zeitpunkten von verschiedenen SW-Entwicklern unabhängig voneinander übersetzt werden.
- Später können dann die aus den Übersetzungsvorgängen resultierenden Objektdateien mit dem **Linker** zu einem ablauffähigen Programm zusammengebunden werden.

Linker (1)

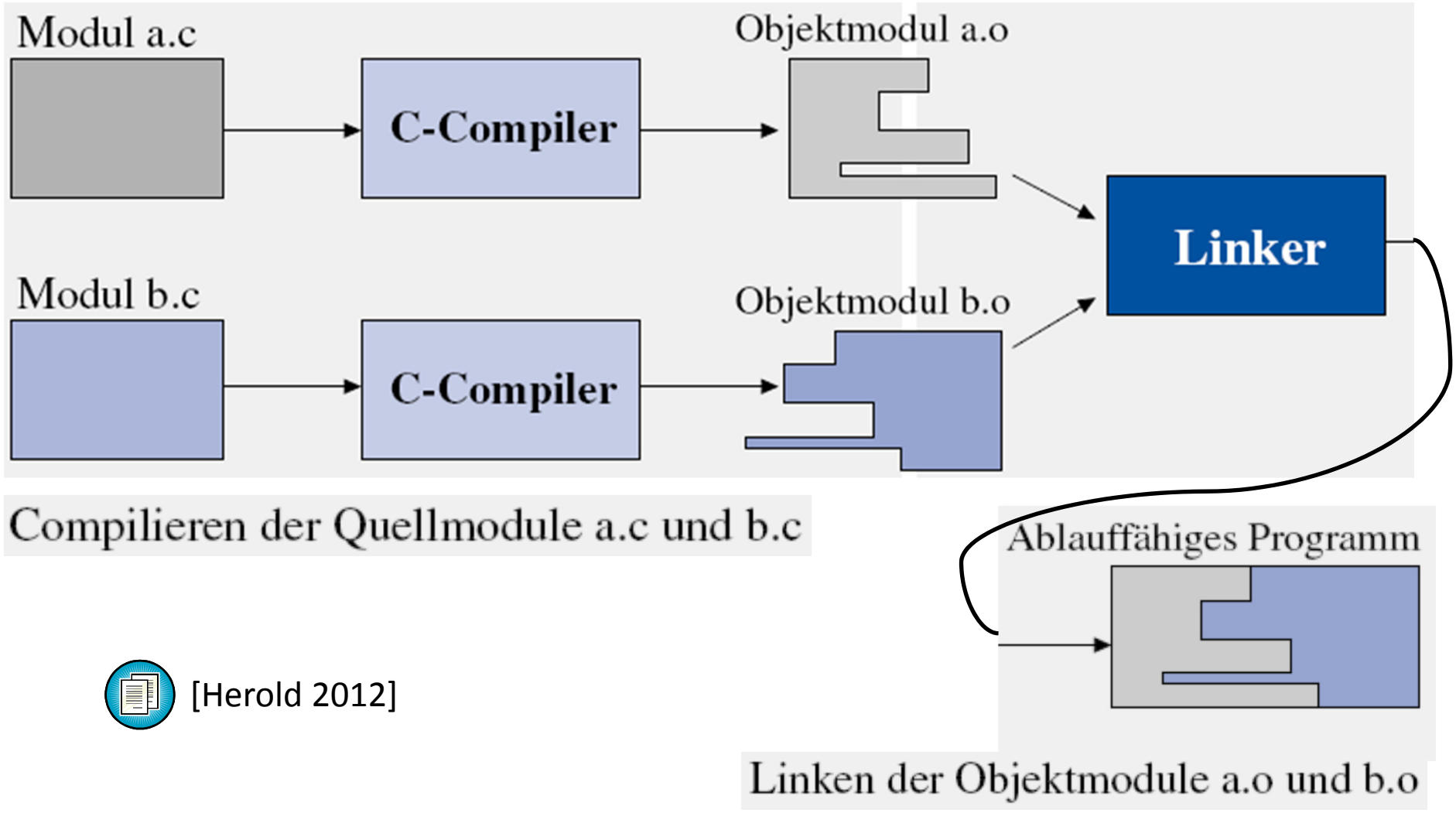


[Herold 2012]

Linker (2)

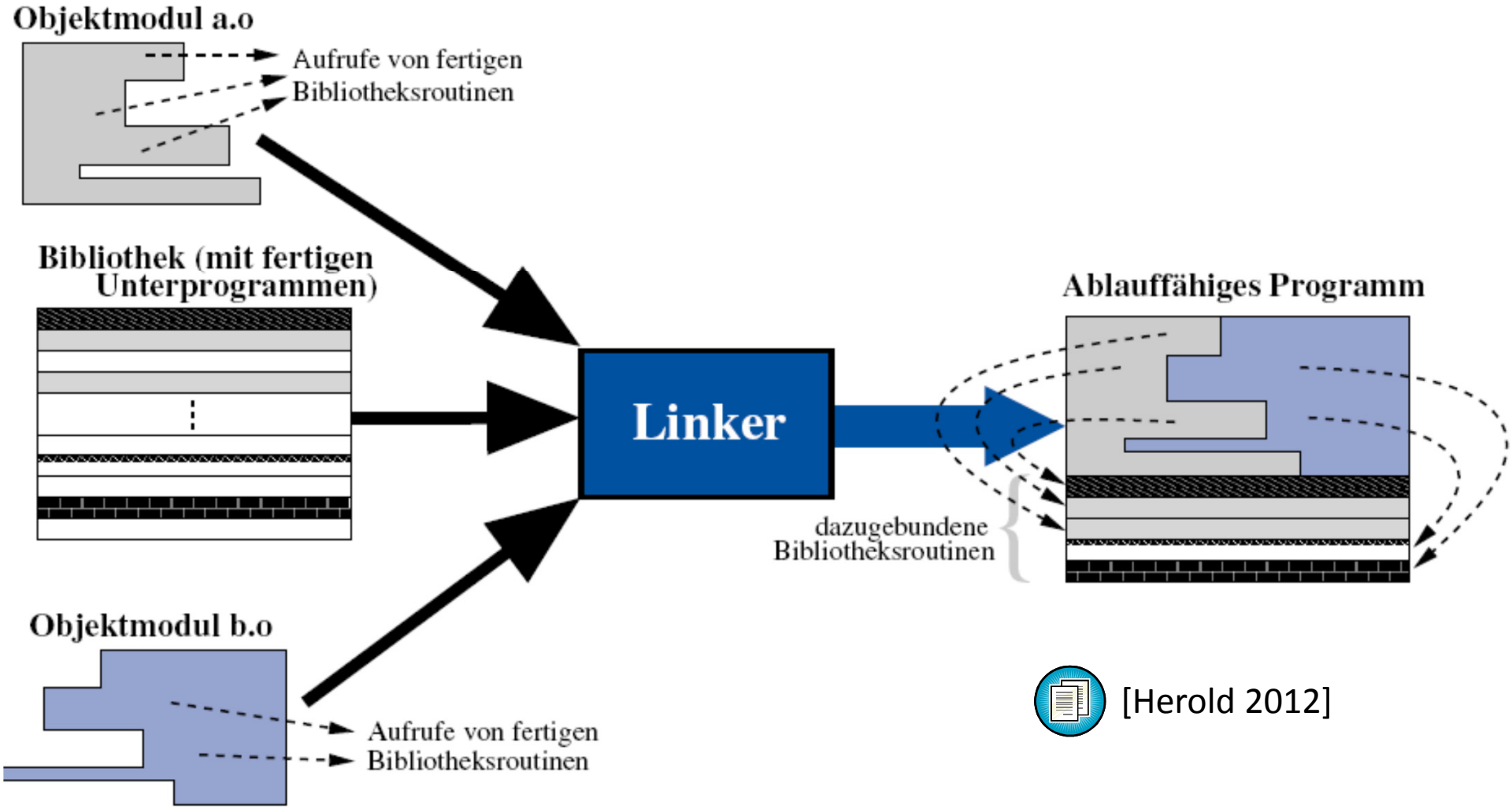
- Der Linker ist normalerweise ein eigenes Programm, das unabhängig vom Compiler ist und die vom Compiler übersetzten Objektmodule erhält.
- Diese erkennt man an der Endung .o oder .obj.
- Module kommunizieren häufig miteinander über Schnittstellen.
  - Schnittstellen sind üblicherweise Funktionen eines Moduls, die von einem anderen Modul aufgerufen werden.
  - Legen wir dem Linker Objektmodule vor, so generiert er hieraus ein ablauffähiges Programm.
- Sollten sich in den Modulen Aufrufe zu fertigen Bibliotheksroutinen befinden (z.B. `printf`), so muss der Linker auch nach diesen in den entsprechenden Bibliotheken suchen und diese zum Programm hinzubinden.

Linker (3)



[Herold 2012]

Linker (4)



[Herold 2012]

Beispiel (stack.h)

```
#ifndef _STACK_H_
#define _STACK_H_

enum stack_errors {
      STACK_OK,
      STACK_FULL,
      STACK_EMPTY
};

extern enum stack_errors stack_push(const int);
extern enum stack_errors stack_pop(int * const);

#endif
```

Bedingte Übersetzung!
Beim ersten Einbinden ist
_STACK_H_ noch nicht vorhanden.

Beispiel (stack.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"
#define MAX 5
static int array[MAX];
static int topOfStack = 0;

static void print_error(const enum stack_errors errNr) {
	static char *errorStrings[] = { "OK", "Stack is full!", "Stack is empty!" };
	char *errString =  (errNr < 0 || errNr > STACK_EMPTY) ? "Undefined error!" : errorStrings[errNr];
	fprintf(stderr, "Error nr. %d: %s\n", errNr, errString);
}

enum stack_errors stack_push(const int number) {
	if (topOfStack == MAX) {
		print_error(STACK_FULL);
		return STACK_FULL;
	} else {
		array[topOfStack++] = number;
		return STACK_OK;
	}
}

enum stack_errors stack_pop(int * const number) {
	if (topOfStack == 0) {
		print_error(STACK_EMPTY);
		return STACK_EMPTY;
	} else {
		*number = array[--topOfStack];
		return STACK_OK;
	}
}
```

Beispiel (stacktest.c – benutzt die Stack-Implementierung)

```c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

int main(void) {
     int count, run, number;
     enum stack_errors status = STACK_OK;
     printf("How many numbers should be read?: ");
     scanf("%d", &count);
     for (run = 0; run < count && status == STACK_OK; run++) {
          printf("Please insert a number: ");
          scanf("%d", &number);
          status = stack_push(number);
     }
     if (status == STACK_OK) {
          printf("Printing all numbers:\n");
          for (; run > 0 && status == STACK_OK; run--) {
               status = stack_pop(&number);
               printf("Number %d was %d\n", run, number);
          }
     }
     return EXIT_SUCCESS;
}
```

Literatur

- [Herold 2012] H. Herold, B. Lurz, J. Wohlrab, **Grundlagen der Informatik**, Pearson, 2. Auflage, 2012