W3WI_SE303.2 - Moderne Programmierkonzepte

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B

Version: 1.0 (24SEA)

Ausgewählte Kerninhalte gem. MHB

Wir werden uns insbesondere auf die folgenden Themen konzentrieren:

- Funktionale Programmierung: Grundlagen und praktische Umsetzung an Beispielen, Funktionen als Datenwerte.
- Streaming-Konzepte für Datenstrukturen: Grundlagen und praktische Umsetzung an Beispielen, Parallelisierungsaspekte.

Vorkenntnisse ?

- Programmiersprachen (Java, Python, ...)

  Konzepte:

  ? Algorithmen und Datenstrukturen

  ? Objekt-orientierte Programmierung

  ? Generische Programmierung

  ? Nebenläufige Programmierung

  ? Funktionale Programmierung

  ? (embedded) *Domain Specific Languages*

  ? RESTful Web-Services
- Versionsverwaltung (GIT, ...)
- Entwicklungsumgebungen (IDEs, Build-Tools, ...)
- Softwarequalitätssicherung (Testen, Metriken, ...)

Inhalte

Wir werden drei verschiedene Programmiersprachen und Konzepte kennenlernen anhand ausgewählter Aufgabenstellungen (Go, Rust und Scala):

- Entwicklung eines einfachen, *arbitrary precision* Taschenrechners in Reverse Polish Notation (RPN).
- Implementierung einfacher generischer Datenstrukturen (über 3 Termine hinweg)

  (Algorithmen stehen hier nicht im Vordergrund, sondern die konkrete Implementierung der Datenstrukturen sowie die Bereitstellung einer fortgeschrittenen API.)
- Implementierung einer einfachen eDSL (embedded Domain Specific Language)
- Entwicklung eines einfachen parallelisierten RESTful Web-Services
- Durchführung von Übungen zu den Programmiersprachen

Prüfungsleistung - Portfolio

Hintergrund

- das Modul Software Engineering I hat 60 VL (5 ECTS)
- die Lehreinheit *Moderne Programmierkonzepte* hat 30VL

In dieser LV sind max. 60 Punkte zu erreichen (diese werden ggf. umgerechnet).

3 Bestandteile

01	regelmäßige Vorträge - max. 30 Punkte (Jeder muss vortragen; jeder Vortrag ergibt eine persönliche Note und eine Teilnote für die Gruppenleistung bei den Vorträgen)
02	Programmierübung (Gruppenleistung) - max. 15 Punkte
03	finale Abgabe (Gruppenleistung) - max. 15 Punkte

Vorträge - Rahmenbedingungen

- Jedes Thema wird von genau einer Person vorgetragen; alle Studierenden müssen einmal vortragen.
- Die Vorträge sind 25 Minuten lang.
- Die Vortragsnote ergibt sich zu ca. $2/3$ aus der persönlichen Note und zu ca. $1/3$ aus der Gruppenleistung, die aus den bewerteten Vorträgen V_i besteht; mit $i \in [1, G]$ wobei G die Anzahl der Gruppenmitglieder ist.

  D. h. für einen Vortrag werden bis zu 30 Punkte vergeben, die sich wie folgt ergeben:

  max. 10 Punkte: persönliches Auftreten (V_i^P)

  max. 20 Punkte: Ausgestaltung des Vortrags (V_i^A) mit folgender Verteilung: 50% persönlicher Anteil und 50% Gruppenanteil/-beitrag. Am Ende ergibt sich somit die persönliche Note für den Vortrag(enden) V_i wie folgt:

  $$\text{Pkt. pers. Auftreten} + \frac{1}{2} \cdot \text{Pkt. Ausgestaltung} + \sum_{i=1}^{G} \frac{(V_i - V_i^P) \cdot \frac{1}{2}}{G}$$

  Die Note des Vortrags ergibt sich auch aus der Eleganz und Korrektheit der Lösung und am Ende des Vortrags sind immer auch kurz die Tests zu zeigen.

Achtung!

Die Vorträge sind immer am Abend vor dem Vortrag in Moodle hochzuladen. Sollte der Vortrag nicht hochgeladen sein, erfolgt ein Malus von 5 Punkten auf die Vortragsnote im Bereich persönliches Auftreten.

Allgemeine Kriterien

Struktur des Vortrags:
War die Struktur einleuchtend und unter den gegebenen Umständen (Publikum, etc.) angemessen? War „jederzeit" klar wie der Vortrag strukturiert ist und in welchem Abschnitt man sich gerade befindet?

Logischer Aufbau: Haben die Folien logisch aufeinander aufgebaut oder gab es „Vorwärtsverweise", bzw. wurden inhaltliche Fragen, die für ein Verständnis des Vortrags wichtig gewesen wären, aufgeworfen und nicht beantwortet?"

Aussagekraft: Hatte jede Folie eine wohldefinierte Botschaft? War für jede Folie klar welchen Beitrag diese Folie leistet bzw. welchen Beitrag die Inhalte auf der Folie in Hinblick auf die Gesamtpräsentation leisten?

Präsentation des Inhalts:
Wurden die geplanten Inhalte verständlich und ohne zusätzliche Fragen aufzuwerfen dargestellt. Hat die Präsentation ein „rundes Bild" ergeben oder wurden (mit Hinblick auf das Kernthema) irrelevante Inhalte vermittelt?

Verständlichkeit des Inhalts der Präsentation:
War die Präsentation (jederzeit) für das Zielpublikum verständlich, d.h. wurden keine unnötigen Fachbegriffe verwendet, wurden Begriffe / relevante Konzepte hinreichend eingeführt?

Visualisierungen / Grafiken":
Wurden aussagekräftige, dem Verständnis hilfreiche Visualisierungen verwendet?

Foliendesign: Wurden Animationen und ähnliche Effekte „sinnvoll" eingesetzt? Wurden Fonts und Farben vernünftig verwendet.

Sorgfalt: War die Präsentation frei von Tippfehlern und waren Grafiken, Quelltext, etc. konsistent formatiert?

Zusammenfassung:
Gab es eine und hat diese kurz und prägnant die wichtigsten Aussagen dargestellt?

Relevante Literatur/Quellen:
Wurde auf die verwendete / relevante / weiterführende Literatur hingewiesen?

Kriterien bzgl. des persönlichen Auftretens

(Aus-)Sprache: Gab es keine „Ähms", kein Räuspern? War die Sprechgeschwindigkeit angemessen?

Redezeit: Wurde die vorgegebene Redezeit eingehalten bzw. musste die Präsentation abgebrochen werden? (+/- 10% ist OK - danach Abzug)

Vortragsstil: Wurde der Vortrag flüssig vorgetragen oder kam der / die Vortragende ins Stocken (d.h. er / sie kannte die Folien nicht)? Wurden Grafiken vollumfänglich und auch verständlich erklärt oder wurden Teile einfach unerklärt gelassen?

Auftreten: Kontakt zum Publikation hergestellt (nicht auf das Notebook geschaut, nicht auf die Wand geschaut)?

Interaktion: War die Interaktion mit dem Fragenden freundlich und zuvorkommend - wurde auf den Fragenden eingegangen. Wurden Fragen inhaltlich korrekt und umfassend beantwortet, oder wurden „andere" - d.h. nicht gestellte - Fragen beantwortet.

Vertrautheit mit der Präsentation:
Wurden alle Folien in der Vortragszeit hinreichend dargestellt oder mussten Folien (z.B. aufgrund von Zeitmangel) übersprungen werden?

Programmierübung

- Jede Gruppe erstellt eine kleine Programmierübung zu der von Ihr vorgestellten Programmiersprache.
- Die Übung richtet sich an alle Studierenden der anderen Gruppen; die Übung wird von der Person vorgestellt und begleitet, die nicht an der Präsentation beteiligt war.
- Die Übung hat benotungstechnisch sowohl einen Anteil bzgl. der Vortragsnote als auch einen reinen Gruppenanteil; der sich aus der Qualität der Übungsaufgabe und der zur Verfügung gestellten Umgebung ergibt.
- Die zu bearbeitende Aufgabe sollte sich von den Studierenden innerhalb von ca. 45 Minuten bearbeiten lassen und in mehrere Schritte gegliedert sein.
- Bei der Ausgestaltung sind sie frei. Es steht Ihnen dabei frei ob Sie zum Beispiel einen vorgefertigten Rahmen zur Verfügung stellen oder etwas "auf der grünen Wiese" entwickeln lassen. Auf jeden Fall sollte die Übung allen Teilnehmern einen allerersten Einblick in die Programmiersprache geben. Es ist also ggf. notwendig umfangreiche Hilfestellungen zu geben.
- Die Übung sollte nur Kenntnisse von den in der Vorträgen vorgestellten Konzepten voraussetzen.

Einführung - 18. Feb 2025

- Aufteilung der Gruppen (3 Gruppen mit je 8 Studierenden) und Präsentation der Aufgabenstellungen
- kurze Einführung in GIT und Testabdeckung

1. Termin - 25. Feb 2025

Pro Programmiersprache/Team:

- [ ] [1 Stud. - 25min] Präsentation der grundlegenden Konzepte der Programmiersprachen anhand des Beispiels RPN (Reverse Polish Notation) Taschenrechner.
- [ ] [1 Stud. - 25min] Präsentation des Ecosystems (Build-Prozess, Central Repository, und Testen)

2. Termin - 11. Mar 2025

Pro Programmiersprache/Team:

- [ ] [1 Stud. - 25min] Präsentation der Unterstützung für generische Programmierung und der Wiederverwendbarkeit von Funktionalität anhand der Implementierung einfacher Datenstrukturen. D. h. erklären des Typsystems. Darstellen der Tests und der Testabdeckung und des konkreten Build-Prozesses.

3. Termin - 18. Mar 2025

Pro Programmiersprache/Team:

- [ ] [1 Stud. - 25min] Präsentation der grundlegenden Konzepte der funktionalen Programmierung anhand der Implementierung einer einfachen Datenstruktur sowie der Verarbeitung von Datenströmen. Themen, die explizit diskutiert und präsentiert werden sollten, sind Scoping, Closures, Tail Recursion sowie Eager und Lazy Evaluation.

4. Termin - 25. Mar 2025

Pro Programmiersprache/Team:

- [ ] [1 Stud. - 25min] Präsentation der grundlegenden Konzepte der nebenläufigen Programmierung anhand der Parallelisierung von gängigen Funktionen höherer Ordnung.

5. Termin - 1. Apr 2025

Pro Programmiersprache/Team:

- [ ] [1 Stud. - 25min] Präsentation einer einfachen eDSL (embedded Domain Specific Language)
- [ ] [60 min] Übung zu Scala

6. Termin - 8. Apr 2025

Pro Programmiersprache/Team:

- [ ] [1 Stud. - 25min] Präsentation eines einfachen parallelisierten RESTful Web-Services
- [ ] [60 min] Übung zu Rust

Abschluss - 22. Apr 2025

Pro Programmiersprache/Team:

- [ ] [60 min] Übung zu Go
- [ ] Lehrveranstaltungsabschluss

Finale Abgabe (pro Gruppe)

- ein von allen Gruppenbeteiligten unterschriebenes Dokument, in dem genau angegeben wurde welche Hilfsmittel verwendet wurden und das die Arbeit eigenständig erstellt wurde. Das Dokument muss folgenden Text enthalten:

  > *Hiermit erklären wir ehrenwörtlich, dass wir die vorliegende Portfolio-Arbeit zur Vorlesung „Moderne Programmierkonzepte" bestehend aus Vorträgen sowie Programmcode selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.*
  >
  > ***—Datum, Unterschriften***

- *alle Vorträge* (inkl. der Übungsaufgabe) sind als PDF-Datei abzugeben.
- *ein aufgeräumtes GIT Repository* (oder ggf. ein Link auf ein öffentliches Repository), in dem alle Quelltexte zu finden sind, inkl. einer README.md, die die Struktur des Repositories erläutert und auch genau erklärt wie die Beispiele gebaut werden können.

  Als Teil des Build-Prozesses müssen auch die Tests ausgeführt werden. Diese müssen eine sehr gute Abdeckung aufweisen. Sie müssen die von Ihnen erreichte Testabdeckung dokumentieren und begründen, warum diese Abdeckung ausreichend ist und wo sie ggf. noch *sinnvolles* Verbesserungspotential sehen, dieses aber nicht umgesetzt haben.

  Die Versionen aller verwendeten Bibliotheken müssen im Build-script fixiert sein.
- *die virtuelle Maschine* mit der vorkonfigurierten Übungsumgebung

1. Gruppenaufteilung

1. Aufgabe: RPN Calculator

Entwickeln Sie einen einfachen interaktiven Taschenrechner, der beliebig große Zahlen verarbeiten kann und der die Reverse Polish Notation (RPN) verwendet.

Bei RPN werden die Operanden auf den Stack gelegt und die Operatoren arbeiten auf den entsprechenden obersten Elementen des Stacks. D. h. die Eingabe erfolgt zum Beispiel in der Form *1 2 + 3 ** und das Ergebnis ist 9.

Zu unterstützende Operationen des Taschenrechners

- \+ Addition, - Subtraktion, * Multiplikation, */* Division, ^ Potenzierung, *sqrt* Wurzel, *log* Logarithmus (zur Basis 10), *!* Fakultät, *abs* Betrag, ++ alle Zahlen auf dem Stack addieren, ** alle Zahlen auf dem Stack multiplizieren.
- Ausgabe in Infix-Notation (d. h. mit Klammerung); Beispiel

  $1\ 2 + 3\ *$ ist in Infix-Notation: $(1 + 2) * 3$
- Ausgaben als Latex-Formel; Beispiel:

  RPN: $1\ 2\ /\ 2\ \mathrm{sqrt}\ 3 + *$

  Latex: `\[\frac{1}{2} \cdot \sqrt{2} + 3\]`

  Latex (rendered): $\frac{1}{2} \cdot \sqrt{2} + 3$

Benutzung

- die Eingabe wird von der Kommandozeile gelesen und soll auf der Kommandozeile ausgegeben werden.
- immer wenn ein Ergebnis berechnet werden kann („*eager evaluation*“), soll dieses ausgegeben werden.

  Für die Darstellung des Gesamtausdrucks in Infix-Notation/Latex müssen Sie sich die Historie speichern.

Sonstiges

- entwickeln Sie hinreichende Tests

2. Aufgabe: Einfache Datenstrukturen

Entwickeln Sie folgende drei einfachen Datenstrukturen, um komplexe Werte wie zum Beispiel Zeichenketten (ugs. `String`s) oder allgemeine `Record`s zu speichern.

- Liste
- Stack (*LIFO* ≙ Last in, First out) (🇩🇪 *Stapel*)
- Queue (*FIFO* ≙ First in, First out) (🇩🇪 *Warteschlange*)

Bemerkung

Die konkrete Implementierungsstrategie ist nicht vorgegeben. Sie sind an dieser Stelle frei in der Wahl der Implementierung.

- **Liste**

  get(pos): Gibt das Element an der gegebenen Position zurück.

  add(elems): Fügt alle Elemente der Liste hinzu.

  insert(elem, pos):
  Einfügen eines Elements an der gegeben Position.

  remove(elem):
  Entfernt das erste Vorkommen des Elements aus einer nicht leeren Liste.

  removeAt(pos):
  Entfernt das Element an der gegebenen Position.

  replace(elem, pos):
  Ersetzt ein Element an der gegeben Position durch das neue Element.

  size(): Gibt die Anzahl der Elemente in der Liste zurück.

  isEmpty(): `true` wenn die Liste leer ist sonst `false`

  isFull(): `false` wenn die Liste leer ist sonst `true`

- **Stack**

  push(elem): Fügt ein neues Element dem Stapel hinzu.

  pushAll(elems):
  Fügt alle Elemente dem Stapel hinzu.

  Wenn möglich, dann nutzen Sie eine variadische Methode (*Varargs* in Java).

  pop(): Entfernt das zuletzt hinzugefügte Element.

  peek(): Gibt das zuletzt hinzugefügte Element zurück, ohne es zu entfernen.

  size(): Gibt die Anzahl der Elemente des Stacks zurück.

  isEmpty(): `true` wenn der Stack leer ist sonst `false`

  isFull(): `false` wenn der Stack leer ist sonst `true`

- **Queue**

  enqueue(elem):
  Fügt ein Element am Ende der Warteschlange ein.

  dequeue(): Entfernt das erste/älteste Element der Warteschlange und gibt es zurück.

  peek(): Gibt das erste/älteste Element der Warteschlange zurück, ohne es zu entfernen.

  size(): Gibt die Anzahl der Elemente der Warteschlange zurück.

  isEmpty(): `true` wenn die Warteschlange leer ist sonst `false`

  isFull(): `false` wenn die Warteschlange leer ist sonst `true`

- **Alle**

  Im Folgenden gilt, dass die Namen der Methoden ggf. durch idiomatische Namen ersetzt werden können.

  equals(other):
  Gibt `true` zurück, wenn zwei Datenstrukturen des gleichen Typs die gleichen Werte (in gleicher Reihenfolge) enthalten; sonst `false`.

  toString(): eine Repräsentation als String haben (toString), die den Inhalt der Datenstruktur darstellt und die die programmatische Rekonstruktion der Datenstruktur ermöglicht bzw. erleichtert.

allgemeine Anforderungen

- Die Datenstrukturen sollen (insbesondere) das Speichern von komplexen Objekten erlauben.
- Versuchen Sie Code-Duplikation zu vermeiden; wägen Sie dabei - in Abhängigkeit von der Möglichkeiten der Sprache - das Entwurfsprinzip: *Composition over Inheritance* mit möglichen Codeeinsparungen ab.
- Entwickeln Sie hinreichende Tests für die geforderten Operationen.
- Die Datenstrukturen sollen so entwickelt werden, dass eine Nutzung in anderen Projekten möglich ist.
- Versuchen Sie eine möglichst typsichere Implementierung zu realisieren.
- Ist es möglich - und wenn ja zu welchen Kosten - primitive Datentypen in Ihren Datenstrukturen zu speichern?

  (Stichwort: Zero-Cost Abstractions.)
- *Fehlerbehandlung*: Fehler sollten (sprach-)angemessen behandelt werden.

3. Aufgabe: Funktionale Programmierung und verwandte Konzepte

Setzten Sie die Entwicklung der drei einfachen Datenstrukturen (Liste, Stack und Queue) fort mit dem Ziel, die grundlegenden Konzepte der funktionalen Programmierung zu demonstrieren. Diesbezüglich gilt, dass die Originaldatenstruktur unverändert bleibt und die Methoden ggf. neue Datenstrukturen anlegen.

Setzen Sie dabei für alle drei Datenstrukturen folgende Methoden um.

- `forEach(f)`: `f` wird auf alle Elemente angewendet.
- `filter(f)`: eine neue Datenstruktur wird angelegt, die alle Elemente enthält für die `f` den Wert `true` zurückgibt.

  Der Nutzer soll optional eine Zieldatenstruktur angeben können.
- `map(f)`: wendet die Funktion `f` auf alle Elemente der Datenstruktur an und gibt eine neue Datenstruktur zurück. Dabei ist zu beachten, dass die ursprüngliche Datenstruktur unverändert bleibt und `f` auch den Typ der Elemente ändern kann.

  Der Nutzer soll optional eine Zieldatenstruktur angeben können.
- `reduce(f)`: nutzt die binäre Funktion `f`, um die Elemente der Datenstruktur auf einen einzigen Wert zu reduzieren.

  Ein Beispiel für die Verwendung von `reduce` wäre eine Liste von Zahlen und die Funktion `f` ist die Addition, um die Summe zu berechnen.

  - Die Reihenfolge der Anwendung von `f` ist nicht spezifiziert.
  - Der Zieldatentyp kann unterschiedlich vom Elementtyp sein.
  - Ist die Datenstruktur leer, dann soll eine aussagekräftige Ausnahme geworfen werden.
- `reduceLeft(f)`: nutzt die binäre Funktion `f`, um die Elemente der Datenstruktur auf einen einzigen Wert zu reduzieren.

  Ein Beispiel für die Verwendung von `reduceLeft` wäre eine Liste von Zahlen und eine Funktion `f` die einen String erzeugt, der die natürliche Reihenfolge der Element wiederspiegeln soll.

  - Die Reihenfolge der Anwendung von `f` erfolgt gemäß der natürlichen Reihenfolge der Datenstruktur.
  - Der Zieldatentyp kann unterschiedlich vom Elementtyp sein.
  - Ist die Datenstruktur leer, dann soll eine aussagekräftige Ausnahme geworfen werden.

Anforderungen an die Datenstrukturen

- Alle Operationen sollen typsicher sein.
- Das Ziel ist es die Methoden mit möglichst wenig Code zu implementieren; d. h. versuchen Sie — soweit es Ihnen und in der Programmiersprache möglich ist — die Methoden nur einmal zu implementieren und dann für alle Datenstrukturen zu verwenden. Sollte dies signifikante Kosten (zur Laufzeit/beim Codeverständnis) verursachen, dann implementieren Sie die Methoden für jede Datenstruktur einzeln und dokumentieren/präsentieren Sie die Gründe für diese Entscheidung.
- Soweit erforderlich erklären Sie wie folgenden Konzepte umgesetzt sind:
  a. Closures (Funktionen, die auf Variablen aus ihrer Umgebung zugreifen und deren Werte speichert)
  b. Tail Recursion
- Für die Funktionen `map` und `filter` soll es sowohl *lazy* als auch *eager* Versionen der Operationen geben und auch Verkettung (`.map(...).filter(...).map(...)`) ermöglicht werden. Insbesondere für die *lazy* Varianten gilt, dass diese erst dann zur Evaluation der Operationen führen sollen, wenn das Ergebnis benötigt wird. (Zum Beispiel aufgrund eines Aufrufs von `reduce`. Sie können ggf. auch gerne eine Methode (`to`) hinzufügen, die die Datenstruktur explizit reifiziert.)

Hinweise zur Präsentation

- Starten Sie mit einer Demonstration der Funktionalität der Methoden anhand von Beispielen.
- Präsentieren Sie die Implementierung der Methoden und die Tests danach.
- Diskutieren Sie die Vor- und Nachteile der Implementierung.

4. Aufgabe: Nebenläufige Programmierung

Setzen Sie die Entwicklung der drei einfachen Datenstrukturen (Liste, Stack und Queue) fort mit dem Ziel, die grundlegenden Konzepte der nebenläufigen Programmierung zu demonstrieren.

Setzen Sie dabei für alle drei Datenstrukturen folgende Methoden um:

- `parallelMap(f)`: wendet die seiteneffektfreie Funktion `f` auf alle Elemente der Datenstruktur an und gibt eine neue Datenstruktur zurück. Dabei ist zu beachten, dass die ursprüngliche Datenstruktur unverändert bleibt und `f` auch den Typ der Elemente ändern kann.
  - Die Reihenfolge der Anwendung von `f` ist nicht spezifiziert.
  - Der Zieldatentyp kann unterschiedlich vom Elementtyp sein.
- `parallelReduce(f)`: nutzt die binäre, seiteneffektfreie Funktion `f`, um die Elemente der Datenstruktur auf einen einzigen Wert zu reduzieren.

  Ein Beispiel für die Verwendung von `parallelReduce` wäre eine Liste von Zahlen und die Funktion `f` ist die Addition, um die Summe zu berechnen.
  - Die Reihenfolge der Anwendung von `f` ist nicht spezifiziert.
  - Der Zieldatentyp kann unterschiedlich vom Elementtyp sein.
  - Ist die Datenstruktur leer, dann soll eine aussagekräftige Ausnahme geworfen werden.

Anforderungen an die Datenstrukturen

- Alle Operationen sollen typsicher sein.
- Das Ziel ist es die Methoden mit möglichst wenig Code zu implementieren; d. h. versuchen Sie — soweit es Ihnen und in der Programmiersprache möglich ist — die Methoden nur einmal zu implementieren und dann für alle Datenstrukturen zu verwenden. Sollte dies signifikante Kosten (zur Laufzeit/beim Codeverständnis) verursachen, dann implementieren Sie die Methoden für jede Datenstruktur einzeln und dokumentieren/präsentieren Sie die Gründe für diese Entscheidung.

Hinweise zur Präsentation

- Führen Sie erst kurz in die Konzepte der nebenläufigen Programmierung in der entsprechenden Sprache ein. D. h. stellen Sie die Konzepte der nebenläufigen Programmierung in der Sprache vor. Auf eine Einführung in die theoretischen Konzepte kann verzichtet werden; es geht um die konkrete Umsetzung in der Sprache.
- Starten Sie mit einer Demonstration der Funktionalität der Methoden anhand von Beispielen.
- Präsentieren Sie die Implementierung der Methoden und die Tests danach. Stellen Sie sicher, dass erkenntlich wird, wie Sie auf die Korrektheit der Implementierung - insbesondere in Hinblick auf die Parallelisierung - getestet haben.
- Diskutieren Sie die Vor- und Nachteile der Implementierung.