Projektdokumentation
Producer-Consumer-Problem

Aufgabe 2 des Praktikums des Modules Informatik 3 SoSe 24

Gruppe B

Hochschule Reutlingen Fakultät Informatik
Studiengang: Medien-Kommunikations-Informatik

Martin Lauterbach

Amina Anzorova

Sissi Wu

Abstract (Anzorova):
Das Projekt befasst sich mit der Implementierung des Erzeuger-Verbraucher-Problems (EVP) anhand eines realitätsnahen Szenarios in Java. Im Zentrum des Projekts steht ein Parkhaus, das als gemeinsamer Puffer dient. In dieses werden Fahrzeuge (repräsentiert durch Objekte der Klasse Car) von Produzenten (Producer) eingeparkt und von Konsumenten (Consumer) ausgeparkt. Das Projekt ist ein Lösungsvorschlag zum synchronischen Parkmechanismus. Die Implementierung erfordert eine Klassenstruktur, weshalb als Programmiersprache Java ausgewählt wurde. Ein weiterer Vorteil von Java ist die Möglichkeit, mit Threads zu arbeiten. Das Projekt zeigt insgesamt, wie Synchronisationsmechanismen effektiv eingesetzt werden können, um eine korrekte und effiziente Multithreading-Umgebung zu schaffen. Das Projekt verdeutlicht die Herausforderungen und Lösungen des Erzeuger-Verbraucher-Problems und bietet eine anschauliche und praxisnahe Anwendung dieses klassischen Synchronisationsproblems.

Link zur Hochschuloffenen Repository aller Praktikumsaufgaben:
`https://gitlab.reutlingen-university.de/inf3-theteam/aufgaben-repo-inf3-sose14`

Stand: 6. Juni 2024
SoSe 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Strukturelle Erklärung des Projekts (Lauterbach)

Die Lösung für das Producer-Consumer-Problem wurde durch die Nutzung eines Buffers (nicht-extended) (ParkingLot.java) ermöglicht. Bei Erstellung erhält er Argumente zu seinen Eigenschaften: Größe, Kapazität, Anzahl an Producer- und Consumer-Threads. Diese werden durch, durch das frontEnd aufrufbaren, Methoden in der Eltern-Klasse Main vor start-Aufruf festgelegt.

Das Programm öffnet sich durch die main-Method in Main.java, welches sich als erbbare quasi-Schnittstelle für Frontends aufbaut. Über Sie können ParkingLot-Objekte mit spezifischen Argumenten, die in Variablen gespeichert werden, erstellt werden, sowie Frontends den Datenhalter innerhalb des derzeitgen ParkingLot-Objekts abfragen.

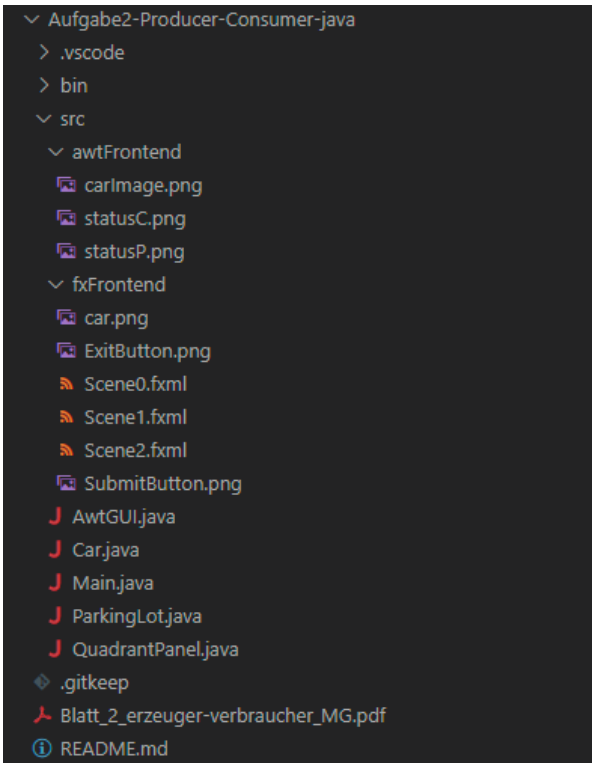

Abbildung 1: Projektstruktur, Screenshot aus Visual Studio Code, Lauterbach, 01.06.24

In einem ParkingLot-Objekt wird bei start() die bei Erstellung gegebene Anzahl an Producer- sowie Consumer-Threads erstellt. Ihr Ablauf wird durch pop() (Consumer-method) sowie push() (Producer-method) reguliert.

Die in ParkingLot.java deklarierte Thread-erweiternde Klasse Producer frägt nach Start sein ParkingLot, in einer endlosen Schleife, ob es ein Car-Objekt (deklariert in Car.jav) erstellen kann. Dies geschieht durch Aufruf von push(). Bei Erstellung wird dieses Car-Objekt in der ParkingLotQueue angehängt.

push() erlaubt dann die Erstellung eines Car-Objekts, wenn die Größe der Queue nicht größer oder gleich dem, bei Erstellung des ParkingLots genannten, Kapazitäts-

Argument ist.

Wenn durch push() ein Erstellen erlaubt wird, bewegt dieser Producer sein erstelltes Car je Schleifendurchlauf um eine Stelle innerhalb des ParkingLots. Es handelt sich hierbei nicht um eine physische Umsetzung um einen Index eines Arrays, sondern um eine Erhöhung der Ort-Variable (int) innerhalb des Car-Objekts durch die move()-Methode.

Der Producer bewegt das Auto so lange, bis der Ziel-Parkplatz seines Car-Objektes erreicht wurde. Der Producer setzt, wenn er ein Erreichen in einem Schleifendurchgang erkannt hat, die Statusvariable dieses Autos auf Nicht-In-Bewegung und Nicht-Von-Producer-kontrolliert.

Diese Variablen werden von pop() bei Aufruf durch einen Consumer in jedem im Queue befindlichen Car-Objekt kontrolliert. Wenn es ein Auto gibt, welches sich nicht bewegt und nicht von einem producer kontrolliert wird, übergibt pop() dieses Car-Objekt an den Consumer-Thread. Dieser bewegt je Schleifendurchgang sein Auto, bis es über das Ende der ParkingLot-Größe hinaus ist. Dann wird das Car-Objekt aus der Queue genommen.

Das awt-Frontend funktioniert, indem ein Frontend-Updater-Thread die Bausteine des Frontends durchgehend in einem Interval mit Pause verändert. Es erfrägt sich die ParkingLotQueue, geht durch die einzelnen Car-Objekte, frägt deren Variablen ab und setzt je QuadratPanel-Objekt, welches im Frontend die einzelnen Parkplätze repräsentieren, passende Bilder auf sichtbar.

AWT baut dann je revalidate() und repaint() Aufruf das Frontend mit den neuen Informationen neu.

2 UML-Diagram (Lauterbach)

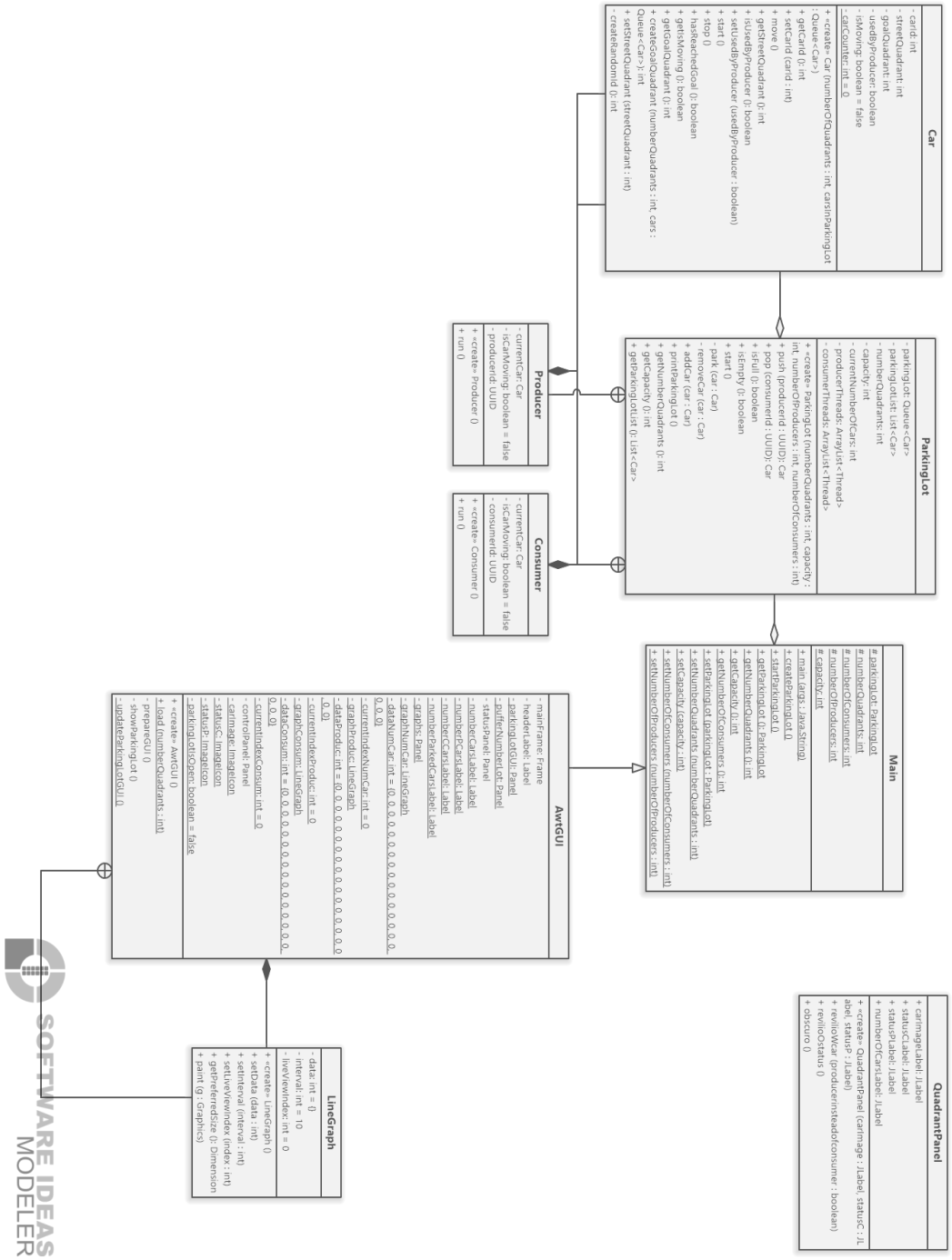

Abbildung 2: UML-Diagram, erstellt mit Software Ideas Modeler, Lauterbach, 01.06.24

3 Petri-Netz (Lauterbach)

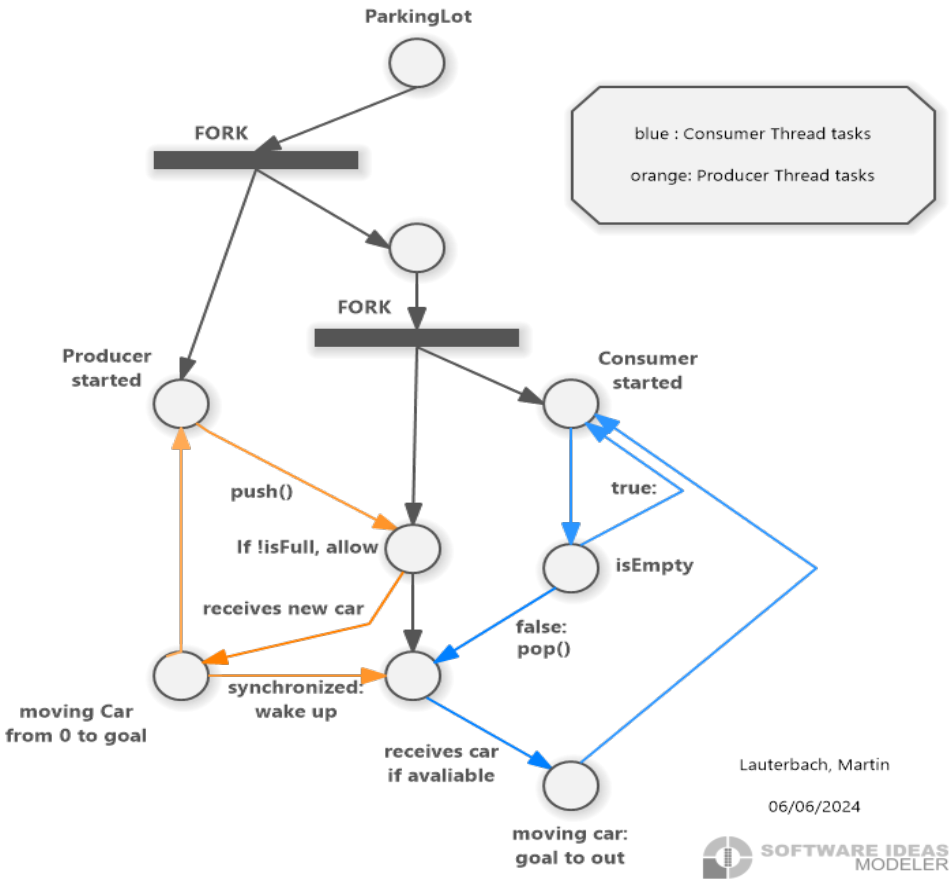

Abbildung 3: Petri-Netz der ParkingLot-Klasse, erstellt mit Software Ideas Modeler, Lauterbach, 06.06.24

4 Projektmetriken (Anzorova)

Tabelle 1: Code-Metriken pro Klasse und hervorgehobenen Methoden; Anzorova; 6. Juni 2024

Entity	Lines of Code	Lines of Comments
Klassen:		
Car	75	12
ParkinLot	202	60
Producer	42	9
Consumer	41	9
QuadrantPanel	45	3
Grundlegende Methoden:		
public Car push()	21	5
public Car pop()	24	7
Klasse Main:		
gesamt	47	82
main-method	4	15
Frontend:		
AwtGUI	251	58

5 Antworten zu Aufgabenstellungen (Wu)

1. Implementiert eine Klasse Car. Diese dient lediglich der Anschaulichkeit des Beispiels und kann beliebig gestaltet sein. (2p)

`Car.java`

- `Car_class` generiert eine eindeutige `carId`; initialisiert `streetQuadrant` und `usedByProducer`, die generiert das `Zielquadrant` basierend auf verfügbaren Quadranten.
- `Car_Attribute` stellt Standard-Getter- und Setter-Methoden bereit. Die Klasse beinhaltet auch die Methoden `move()`, `start()`, `stop()` und `hasReachedGoal()`.

2. Implementiert einen generischen Buffer. Dieser soll mindestens die folgenden Operationen unterstützen:

`ParkingLot.java`

- `push(e)`: fügt `e` in den Puffer ein. Ein Element einzufügen, obwohl der Puffer voll ist, löst einen Fehler aus! (1p)
  - `Class Car_push()`, Zeile 110-134 bei `ParkingLot`
  - Fügt ein Auto dem Puffer (Parkplatz) hinzu. Wenn der Puffer voll ist, wird der `producerThread` warten. Nachdem das Auto hinzugefügt wurde, benachrichtigt es alle wartenden `consumerThreads`.
- `pop()`: nimmt das nächste Element aus dem Puffer heraus und gibt es zurück. Ein Element zu entnehmen, obwohl der Puffer leer ist, löst einen Fehler aus! (1p)
  - `Class Car_pop()`, Zeile 183-209 bei `ParkingLot`.
  - Entfernt ein Auto aus dem Puffer. Wenn der Puffer leer ist, wird der `consumerThread` warten. Nachdem das Auto entfernt wurde, benachrichtigt es alle wartenden `producerThreads`.
- `full()`: prüft, ob der Puffer voll ist. (1p)

  - `boolean isFull()` in `ParkingLot.java`, Zeile 213
- `empty()`: prüft, ob der Puffer leer ist. (1p)

  - `boolean isEmpty()` in `ParkingLot.java`, Zeile 218
- Jeder Puffer hat außerdem eine feste Größe `size` und ein `Mutex`, mithilfe dessen man vorübergehend exklusiven Zugriff auf den Puffer erhält. (4p)
  - `capacity_attribute`, Zeile 18; bei `car_push()` und `car_pop()` mit `synchronized (Mutex)`, Zeile 112-116 und Zeile 186-203.

3. Implementiert einen Consumer, der Autos aus einem Buffer `<Car>` nimmt. Dazu soll er in zufälligen Abständen prüfen, ob noch ein Auto im Parkhaus steht. Falls ja, „parkt" er dieses aus, indem er es aus dem Buffer entfernt. Dazu muss während des Prüfens und während des Entnehmens exklusiver Zugriff auf den Puffer erlangt werden! Ist das Parkhaus leer, schläft der Consumer stattdessen. War der Puffer vor der Entnahme komplett voll, weckt er alle schlafenden Producer auf. (6p)

`Consumer_class` bei `ParkingLot.java`, Zeile 202-246. Die `Consumer_Klasse` ist für das Ausparken der Autos aus dem Puffer verantwortlich und wird ebenfalls als innerer Thread der `ParkingLot_Klasse` definiert. **Ausparken eines Autos**: Der `ConsumerThread` entfernt ein Auto aus dem Puffer. Falls der Puffer leer ist, wartet der Consumer. Das Auto wird in regelmäßigen Abständen bewegt, bis es den Parkplatz verlässt.

4. Implementiert einen Producer, der Autos in denselben Buffer `<Car>` schreibt. Dieser prüft in zufälligen Abständen, ob noch Platz im Parkhaus ist. Falls ja, „parkt" er dort ein Auto ein. Sollte der Puffer vor dem Einparken leer gewesen sein, weckt er alle schlafenden Consumer auf. (6p)

`Producer_class` bei `ParkingLot.java`, Zeile 156-200. Die `Producer_Klasse` ist für das Einparken der Autos in den Puffer verantwortlich und wird als innerer Thread der `ParkingLot_Klasse` definiert. **Einparken eines Autos**: Der `ProducerThread` fügt ein neues Auto in den Puffer ein. Falls der Puffer voll ist, wartet der Producer. Das Auto wird gestartet und in regelmäßigen Abständen bis zum Zielquadranten bewegt.

5. Implementiert eine Main-Methode, die die Anzahl an Producern und Consumern in der Form zweier Programmparameter entgegennimmt. Erstellt für jeden Producer und jeden Consumer einen eigenen Thread und lasst diese dann endlos arbeiten. Implementiert eine geeignete (Konsolen-) Ausgabe, um darzustellen, dass sowohl die Producer als auch die Consumer die Regeln des EVPs in 1.3 einhalten. (4p)

`main.java`

- `void_main()`, Zeile 13-18, `void_createParkingLot()`, Zeile 48-51, `void_startParkingLot()`, Zeile 56-60. Initialisiere `parkingLot` Parameter durch die `main_method`, erstelle `ParkingLot_method`, rufe die `startParkingLot_method` auf.
- In `ParkingLot.java`, `void_start()`, Zeile 213-227. Starte Threads in der `start_method` der `ParkingLot_class`, starte alle `producer` und `consumer` Threads.

(Lauterbach:) Um sicherzustellen, dass die Regeln des EVPs eingehalten werden, haben beide Thread-Objekte sowie pop und push print statements, welche mit einem UUID je Thread aus dem gleichnamigen Java-package zuordenbar sind. Sie sind jeweils in Entry, Critical und wo anwendbar Exit und Remainer aufgeteilt.