Universität Stuttgart

Advanced Software Engineering

Übungsblatt 1

Maximilian Peresunchak (st152466@stud.uni-stuttgart.de)
Nico Reng (st188620@stud.uni-stuttgart.de)
Viorel Tsigos (st188085@stud.uni-stuttgart.de)
Philip Reimann (st182312@stud.uni-stuttgart.de)
Christian Keller (st166512@stud.uni-stuttgart.de)
Johannes Heugel (st183360@stud.uni-stuttgart.de)
Benedikt Wachmer (st177118@stud.uni-stuttgart.de)
Miles Holl (st180549@stud.uni-stuttgart.de)

Wintersemester 2025

2. November 2025

Aufgabe 1: Kritischer Pfad

a)

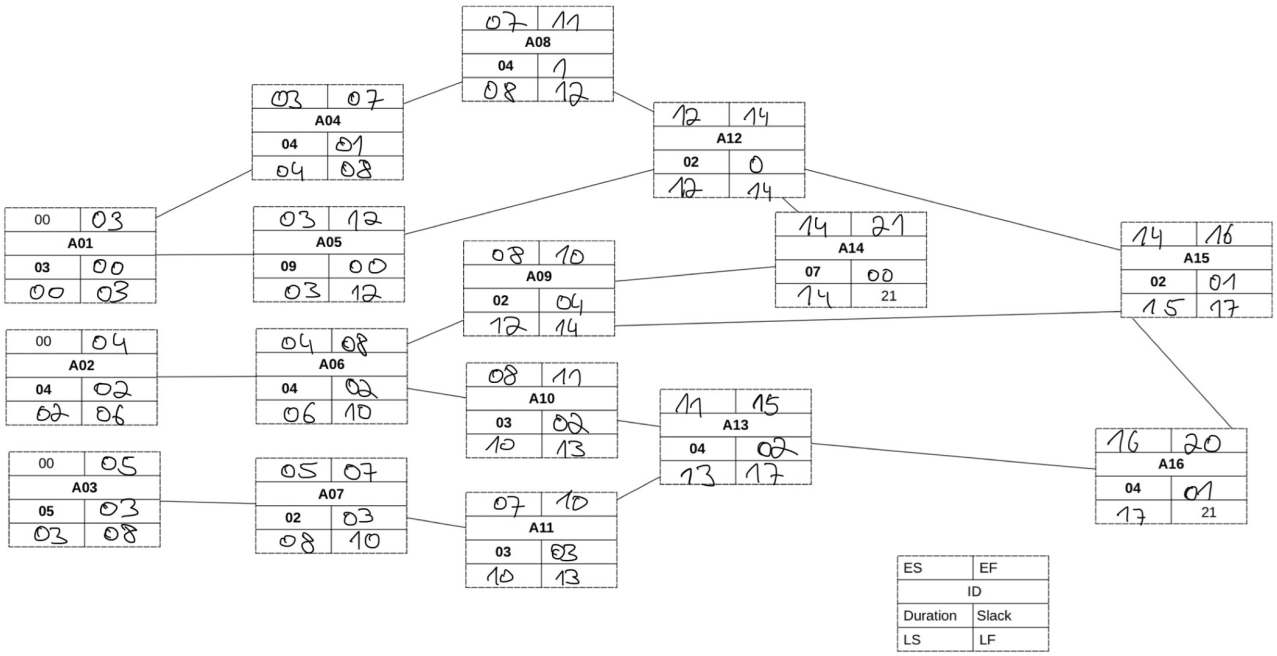

Abbildung 1: Instanz der CPM

c) Der verwendete Algorithmus besteht im westentlichen aus zwei Phasen. In der ersten Phase, der Forward-Propagation, werden zunächst die Startknoten identifiziert. Dies sind alle Knoten ohne Abhängigkeiten. Für diese wird dann EF als $ES + duration$ berechnet. Diese neue Informationen werden an alle Knoten mit direkten Abhängigkeiten weitergegeben. Dies wiederholt sich rekursiv, bis alle Knoten den Zeitpunkt EF erhalten haben. In der darauf folgenden zweiten Phase wird die Gesamtdauer des Projekt als Maximum aller EF berechnet und als LF aller Knoten ohne Nachfolger (Blattknoten) gesetzt. Daraus ergibt sich für alle Blattknoten der LS als $LF - duration$. Diese Informationen werden dann an die Eltern (alle Knoten von denen der aktuelle Knoten abhängt) rekursiv weitergegeben, bis schlussendlich alle Knoten ES, EF, LS, LF ermittelt haben. Zum Schluss wird für jeden Knoten der Slack als $LS - ES = LF - EF$ berechnet.
Um die Arbeitspakete und ihre Abhängigkeiten zu modellieren, haben wir uns für ein objektorientiertes Design entschieden. Dies hat für die Softwarequalität den Vorteil, dass die Kapselung von Daten und Methoden in einer Klasse eine hohe Kohäsion und geringe Kopplung fördert. Jede Instanz der Klasse `Package` modelliert dabei ein Arbeitspaket mit seinen Eigenschaften (Dauer, ES, EF, LS, LF, Slack) und Methoden zur Berechnung dieser Eigenschaften. Die Abhängigkeiten zwischen den Arbeitspaketen werden durch Referenzen auf IDs anderer `Package`-Objekte modelliert. Die gesamte Menge der Arbeitspakete wird in einer `CriticalPathFinder`-Klasse verwaltet, die die Logik für die Durchführung der Forward- und Backward-Propagation implementiert. Diese Struktur ermöglicht eine klare Trennung der Verantwortlichkeiten und erleichtert die Wartung und Erweiterung des Codes.

Aufgabe 2: Bowling Game Kata

Der Algorithmus zur Berechnung des Scores erfolgt über jeden Frame einzeln. Jeder Frame enthält dabei den geworfenen Score des Frames und welcher Frame der folgende Frame ist. Auf dieser Basis lässt sich dann der Score des Frames und die Bonuspunkte errechnen (bei Strike wird aus dem nächsten Frame der basis score entnommen, da dieser im Standard Spiel der Anzahl der regulären Würfe entspricht, falls im folgenden Frame ein Strike auftrat wird im darauf folgenden Frame (sofern vorhanden) der erste Wurf noch addiert. Bei einem Spare nur der erste Wurf im folgenden Frame sofern vorhanden).

Es wurde sich für einen Array an Frames entschieden da dieser schnelle Zugriffszeiten erlaubt, leicht indizierbar ist und es bei einem Bowlingspiel nicht wichtig ist, dass er wachsen kann, die Größe wird direkt beim Start festgelegt und wird während des Spiels nicht verändert.

Für einen einzelnen Roll wurde die Datenstruktur eines Records gewählt, ein Roll ist unveränderlich nachdem er geworfen wurde und daher eignet sich ein Record optimal.

Für die Rolls in einem Frame wurde eine ArrayList gewählt da diese es einfach erlaubt die länge abzufragen um daraufhin zu validieren, dass noch Würfe gemacht werden dürfen.

Um eine gute Wartbarkeit zu gewährleisten, wurden Konstanten eingeführt die bspw. die maximale Anzahl der Pins, der Frames oder der regulären Würfe beschreiben. Dies hat den Effekt, dass zum Beispiel leicht auf drei Rolls pro Frame oder ein kurzes Spiel über die maximale Anzahl der Frames gewechselt werden kann, so muss man nur eine einzige Stelle im Code ändern, um die Regeländerung zu implementieren.

Aufgabe 3: Game of Life Kata

Wir haben uns dazu entschieden, einen 2D-Array für das GameBoard zu verwenden, wobei hier die einzelnen Einträge aus einer GameCell bestehen, die mithilfe eines `bool` den Lebendigkeitsstatus der Zelle repräsentiert. Für die Nachbarzellen haben wir eine `coordinate offset mask` eingeführt, die uns die schnelle Berechnung aller Nachbarn für eine gegebene Zelle ermöglicht. Ein weiterer wichtiger Faktor war die Einführung eines GameBoard Buffers, sodass wir gesetzte Zellen vom Nutzer von denen innerhalb der Berechnung für die nächste Generation unterscheiden können. Wir beziehen uns hierbei immer auf den **Ist**-Zustand von unserem aktiven Board und berechnen die nächste Generation im Buffer, sodass keine kaskadierenden Effekte entstehen. Die Tests wurden so aufgebaut, dass wir die geltenden Regeln für Conways Game of Life in einzelnen Testmethoden festgelegt haben und diese dann mittels TDD implementiert haben.