SWE 4x Übung zu Softwareentwicklung mit modernen Plattformen 4 — SS 2022, Übung 1

Abgabetermin: SA in der KW 11

☐ **Gr. 1, Winkler, BSc MSc** **Name** _____ **Aufwand in h** _____

☐ **Gr. 2, Dr. Pitzer**

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ **/** _____

Beispiel	L Lösungsidee	I Implementierung	T Tests	S = L+I+T	M Multiplikator	S*M
1a	0..3	0..4	0..3		6	
1b	0..3	0..4	0..3		4	
				Summe (Erfüllungsgrad) **Bitte online ausfüllen!**		

Bitte erstellen Sie für diese Übung ein ZIP-Archiv in dem im Unterverzeichnis „`doc`“, die gesamte Dokumentation (inkl. Projektbeschreibung, Lösungsidee und formatiertem Quelltext) als PDF vorliegt, sowie der Quelltexte in den Angegebenen Unterverzeichnissen des Verzeichnis „`src`“.

Denken Sie bitte auch an die Angabe des Aufwandes auf dem Deckblatt und online!

Discrete Event Simulation (DES)

Implementieren Sie in C++ einen Simulator, der die Interaktion von Objekten nachstellen kann. Als Methode soll die diskrete Ereignissimulation eingesetzt werden. Bei dieser Methode wird nicht in Realzeit simuliert, es werden nur jene Zeitpunkte betrachtet, zu denen Änderungen am System eintreten. Die Zeit zwischen diesen Ereignissen kann vernachlässigt werden. Das kann die benötigte Simulationszeit drastisch reduzieren, da nur interessante Zeitpunkte betrachtet werden. Herzstück eines solchen Simulators ist daher eine Liste mit zukünftigen Ereignissen. Da jederzeit neue Ereignisse hinzukommen können und die Anzahl der geplanten Ereignisse mitunter hoch ausfallen kann, ist die Wahl einer geeigneten Datenstruktur essenziell. Hier bietet sich die C++-Standardbibliothek mit ihrer großen Anzahl an sehr effizient implementierten Datenstrukturen an.

a) Recherchieren Sie mögliche Designansätze und passende Datenstrukturen für diese Aufgabenstellung und implementieren Sie eine möglichst allgemein einsetzbare Simulationsbibliothek in Form von C++-Klassen. Folgende Funktionalität muss mindestens verfügbar sein:

   - Zukünftige Ereignisse müssen für einen bestimmten Zeitpunkt geplant werden können und in einer effizienten Datenstruktur gehalten werden.
   - Ereignisse müssen andere Ereignisse (in derselben Simulation) erzeugen können. Überlegen Sie hier, welche API Sie verwenden, um diesen Aspekt, der in der Praxis häufig gebraucht wird, möglichst komfortabel gestalten zu können.
   - Die Simulation muss schrittweise ausgeführt werden können. Es muss also z.B. eine Methode `step()` geben, die einfach nur das nächste Ereignis abarbeitet.
   - Die Simulation muss bis zu einem bestimmten Kriterium ausgeführt werden können, nachdem sie entweder pausiert oder stoppt. Es muss also z.B. eine Methode `run()` geben die durch Ereignisse in der Simulation beendet werden kann.

b) Demonstrieren Sie die korrekte Funktionsweise Ihrer Bibliothek anhand von folgendem Test-Szenario: Ein Produzent erzeugt Produkte, z. B. einfach ganze Zahlen, die in einem Puffer abgelegt

werden. Falls der Puffer voll ist, muss der Produzent warten. Zusätzlich gibt es einen Konsumenten, der die erzeugten Produkte im Puffer wieder verbraucht, falls der Puffer leer ist, muss der Konsument warten. Dabei soll die Zeit, die Produzent und Konsument für ihre Arbeit brauchen nicht konstant sein sondern zufällig schwanken. Sobald der Konsument eine bestimmte Anzahl von Produkten verbraucht hat soll die Simulation beendet werden.

Hinweise:

Das Design des Simulators ist Ihnen überlassen. im Folgenden finden Sie einige Hinweise, die dazu dienen sollen, Ihnen das Leben zu erleichtern:

- Die STL-Datenstruktur `std::priority_queue` ist möglicherweise gut geeignet für die zentrale Ereigniswarteschlange.
- Der Einsatz von Funktoren, Lambda-Ausdrücken oder Funktionszeigern bietet einen eleganten Weg, um vorher nicht bekannte Funktionalität zu kapseln. Ein Ereignis könnte als z.B. einfach eine Klasse `event` mit einem Zeitpunkt (`int`) und einer Aktion (`std::function<void()>`) sein.
- Alternativ dazu könnte auch eine eigne abstrakte Klasse mit einer bestimmten Schnittstelle dienen, z.B. mit einer abstrakten Methode (`virtual void execute() = 0`) die dann von konkreten Ereignissen überschrieben werden muss. Dabei müssen die Ereignisse dann polymorph sein und die Priority Queue muss Zeiger von Ereignissen nach deren Zeit sortieren:

  ```
  std::priority_queue<event*, std::vector<event*>, std::function<bool(event*, event*)>>
    m_events {
      [](event *a, event *b) { return *a > *b; }
    };
  ```

- Das Design der API hat möglicherweise weitreichende Auswirkungen auf das Test-Szenario. Spielen Sie einige Möglichkeiten im Test-Szenario nur mit der API durch, bevor Sie mit der Implementierung beginnen.