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

Abgabetermin: SA in der KW 11

[redacted]1, Winkler, BSc MSc
Gr. 2, Dr. Pitzer

Name: Zlatan Dindic

Aufwand in h: 9

Punkte _____ Kurzzeichen Tutor / Übungsleiter _____ / _____

Feedback: Roman Kofler-Hofer

Beispiel	L Lösungsidee	I Implementierung	T Tests	S = L+I+T	M Multiplikator	S*M
1a	0..3	0..4	0..3		6	42
1b	0..3	0..4	0..3		4	40
				Summe (Erfüllungsgrad) **Bitte online ausfüllen!**		**82**

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.

Aufgabe 1)

Lösungsidee:

Zur Auswahl der Datenstruktur bzgl. des Simulators entschied ich mich (Dank des Hinweises) wenig überraschend für die Priority-Queue aus der Standard-Bibliothek. Die Queue selbst besteht aus einem STD-Vektor von Events. Mit Hilfe des mitgegebenen Funktionszeiger wird die Queue auch nach Priorität dementsprechend sortiert. Zur Sortierung entschied ich mich für eine einfache Compare-Funktion.

Quellcode:

Event.cpp

```
#pragma once

class Event {
private:
    int timestamp;
    void (*memberfunc)();

public:
    Event(int t, void(*f)())
        : timestamp(t), memberfunc(f) {
        //nothing else to do
    }

    int Get_Time_Stamp() const {
        return timestamp;
    }

    void Execute_Event() {
        memberfunc();
    }

};
```

jetzt wo ich es sehe ist dieser Ansatz leichter. Ich hab den anderen gewählt (eine generische Klasse Event und davon abgeleitet konkrete Events)

EventQueue.cpp

```
#pragma once

#include <queue>
#include "Event.h"

struct Compare : public std::binary_function<Event*, Event*, bool> {
    bool operator()(const Event* e1, const Event* e2) const {
        return e1->Get_Time_Stamp() > e2->Get_Time_Stamp();
    }
};
```

```
class EventQueue {
private:
    std::priority_queue < Event*, std::vector<Event*>, Compare> eq;
public:
    EventQueue() {
        //nothing to do
    }
    void Add_Event(Event *e) {
        this->eq.push(e);
    }

    void Step() {
        if (!this->eq.empty()) {
            Event* temp = this->eq.top();
            temp->Execute_Event();
            this->eq.pop();
        }
    }
    void Run() {
        while (!this->eq.empty()) {
            this->Step();
        }
    }
    int Get_Queue_Size() {
        return this->eq.size();
    }
};
```

Aufgabe 2)

Lösungsidee:

Bzgl. der Datenstruktur des Buffers entschied ich mich für eine einfache Queue aus der Standard-Bibliothek. Grund dafür war das FIFO-Prinzip.
Darauf aufbauend war die Produce bzw. Consume-Funktion trivial.
Beim Produce push ich einen Wert in den Buffer, beim pop entnehme ich den Wert. Der Wert selbst wird von einem Zufallsgenerator hergeleitet.
In der Main habe ich eine äußere Abweisschleife, die solange läuft, bis ich eine gewisse Anzahl von Produkten konsumiert habe. Hier habe ich einfach eine Konstante gesetzt.
Im inneren Bereich produziere ich Produkte solange der Buffer (um genauer zu sein wieder eine Konstante) voll ist. Ist das der Fall ist der Konsument dran (bis der Puffer eben wieder leer ist) usw.

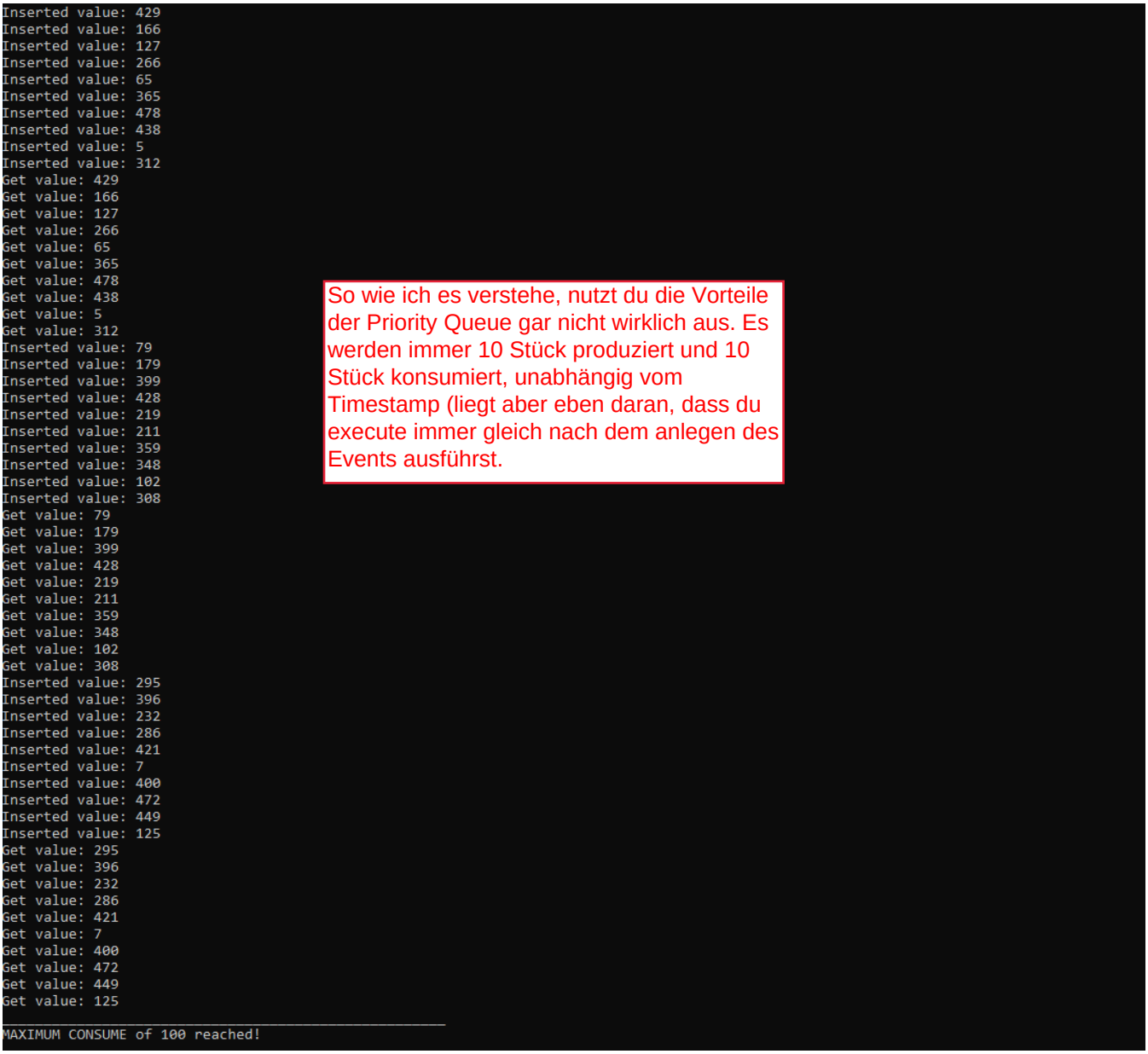
Quellcode:

```
#include <iostream>
#include <vector>

#include "Event.h"
#include "EventQueue.h"
using std::cout;
using std::endl;

std::queue<int> buffer;
const int BUFFER_CAPACITY = 10;
const int MAX_CONSUMED = 100;
int amount_produced = 0;

int Generate_Random_Number() {
  int random_number = rand() % 500;
  return random_number;
}

void Produce() {
    int value = Generate_Random_Number();
    buffer.push(value);
    cout << "Inserted value: " << value << endl;
    amount_produced++;
}

void Consume() {
    cout << "Get value: " << buffer.front() << endl;
    buffer.pop();
}

int main() {
  EventQueue q;
  int c_time = Generate_Random_Number();
  int p_time = Generate_Random_Number();
  int counter_consumed = 0;
  srand(time(0));
  while (counter_consumed < MAX_CONSUMED) {
    while (buffer.size() < BUFFER_CAPACITY) {
      Event* e = new Event(p_time, &Produce);
      q.Add_Event(e);
      e->Execute_Event();
      p_time = Generate_Random_Number();
    }
    while (!buffer.empty()) {
      Event* e = new Event(c_time, &Consume);
      q.Add_Event(e);
      e->Execute_Event();
      counter_consumed++;
    }
  }
  cout << "_____" << endl;
  cout << "MAXIMUM CONSUME of " << counter_consumed << " reached!" << endl;
}
```

ich glaub das wäre von der Idee her anders geplant/gefordert gewesen.

Du erstellst ein Event und rufst gleich danach für dieses Event execute auf. Ich glaub man hätte es erstellen und in die Queue einsortieren sollen. Je nach Zeitstempel wäre es dann erst irgendwann in der Zukunft drangekommen... und in der Zwischenzeit hätten sich auch noch andere Events davor reinschieben können.

Test



So wie ich es verstehe, nutzt du die Vorteile der Priority Queue gar nicht wirklich aus. Es werden immer 10 Stück produziert und 10 Stück konsumiert, unabhängig vom Timestamp (liegt aber eben daran, dass du execute immer gleich nach dem anlegen des Events ausführst.

Event & Queue super implementiert. Sehr schöner, leserlicher Code. Das mit der konkreten Simulation (konsumieren / produzieren) war glaub ich etwas anders gedacht. Da hätte man die Zeitpunkte der Events berücksichtigen müssen und sie entsprechend abarbeiten (execute).