SWO 3x **Übung zu Softwareentwicklung mit klassischen Sprachen u. Bibliotheken 3** **WS 2021/22, Übung 06**

Abgabe elektronisch bis Sa 8 Uhr in der KW 49

☐ ~~Gr. 1,~~ Winkler, BSc Msc **Name: Roman Kofler-Hofer** **Aufwand in h: 15**

☐ **Gr. 1,** Dr. Pitzer

Punkte: _____ **Kurzzeichen Tutor / Übungsleiter ____/____**

Beispiel	L Lösungsidee	I Implementierung	T Tests	S = L+I+T	Multiplikator	S*M
1a	3	4	3	10	6	60
1b	3	4	3	10	4	40
					Summe	**100**

1. DES Bibliothek (Teilaufgabe a)

1.1. Lösungsidee:

Es gibt zwei zentrale Klassen. 1) Die Simulation und 2) Events, welche einen Zeitpunkt haben, an dem sie auftreten. Ein Objekt der Simulationsklasse beinhält eine priority_queue, die die Events auf Basis des Eintrittszeitpunkts verwaltet.

Events benötigen Zugriff auf die Datenkomponenten, der Simulation zu der sie gehören. Daher enthält jedes Event auch eine Datenkomponente mit Zeiger auf die Simulation, die das Event verwaltet.

Ansonsten entspricht meine Herangehensweise der zweiten skizzierten Lösung. Die Klasse Event ist abstrakt und muss dann von konkreten Events überschrieben werden. Die Queue verwaltet daher Zeiger auf Events (für Polymorphismus). Aber nicht nur die Klasse "Event" ist abstrakt, sondern auch die Klasse Simulation. Es gibt nämlich die abstrakte Methode "hasEnded", über die man die Information erhält, ob das Endkriterium für eine Simulation schon eingetroffen ist oder nicht. Somit muss auch diese von einer konkreten Simulations-Klasse überschrieben werden (in meinem Fall heißt die abgeleitete Klasse "MarketSimulation".

Was bei mir nicht funktioniert hat, war der angegebene Vorschlag zur Deklaration der Queue bzw. der Vergleichsfunktion (den Vergleichsoperator für ein Event habe ich auch implementiert).

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

In dieser Implementierung habe ich beim Aufruf von "addEvent" ständig einen Error "bad_function_call" erhalten, den ich nicht auflösen konnte.

Nach einiger Recherche bin ich auf folgende Lösung mit einem Funktor gekommen.

```
class Simulation
{

    struct CustomComparator {
        bool operator()(Event* e1, Event* e2) {
            return e1->getEventTime() > e2->getEventTime();
        }
    };
```

```
    protected:
        std::priority_queue<Event*, std::vector<Event*>, CustomComparator>
            m_events;
        int m_currentTime{0};
};
```

Zu den geforderten Funktionalitäten der Simulations- & Event-Klasse:

addEvent:
einfaches Push eines Event-Pointers, wobei natürlich eine Funktion für den Vergleich der Events bereitgestellt werden musste.

Event kann neues Event anlegen und pushen:
Implementiert ist das dann erst in den konkreten, abgeleiteten Event-Klassen. Dort muss einfach ein neues Event mit "new" angelegt werden. Über den Zeiger, der die Besitzersimulation des Events referenziert, kann dann für das neue Event "addEvent" aufgerufen werden.

Auszug aus dem abgeleiteten Event "consumeEvent":

```
consumeEvent* delayedEvent = new consumeEvent(m_time + mySim->getWaitingTime(),
      m_ownerSimulation, remainingProducts);

m_ownerSimulation->addEvent(delayedEvent);
```

step:
Mit top hole ich das aktuellste Event und rufe für dafür "executeEvent" auf. Durch Polymorphismus wird die richtige executeEvent-Methode für das jeweilige Event aufgerufen. Mit pop entferne ich das Event aus der Queue (bei pop wird meines Wissens nach auch der Destruktor für das Objekt aufgerufen).

run:
In einer while-Schleife werden solange Events abgearbeitet, bis die Endbedingung der Simulation eintrifft. Alternativ kann meine Simulation enden, wenn n Durchgängen zu keiner Lösung gekommen sind oder wenn alle Events bereits abgearbeitet wurden.

Gute und verständliche Beschreibung!

1.2. Quellcode

```
// file: utils.h

#pragma once
#include <random>

//taken from Stack Overflow
inline int randomNumWithin(int min, int max) {
    std::random_device rd;                                  // obtain a random number from hardware
    std::mt19937 gen(rd());                                 // seed the generator
    std::uniform_int_distribution<> distr(min, max);        // define the range

   return distr(gen);                                       // generate numbers
}


//file: Simulation.h
#pragma once
#include <queue>
#include <vector>
#include <functional>
#include "Event.h"

class Event;

class Simulation
{

    struct CustomComparator {
        bool operator()(Event* e1, Event* e2) {
            return e1->getEventTime() > e2->getEventTime();
        }
    };

    protected:
        std::priority_queue<Event*, std::vector<Event*>, CustomComparator> m_events;

        int m_currentTime{0};
```

```
        int m_maxIterations;  Ich würde hier anstelle der Zahl 1000 ein Makro '#define' erstellen,
                              um bei späterer Anpassung alle Zahlen an einem zentralen Platz ändern zu können.
    public:
        Simulation(int maxIterations=1000);
        Simulation(const Simulation& mySim) = delete;
        Simulation& operator= (const Simulation& mySim) = delete;

        virtual ~Simulation();

        int getCurrentTime() {return m_currentTime;}
        int setCurrenttime(int t) {m_currentTime = t;}

        virtual void runSimulation();
        virtual bool hasEnded() = 0;
        void step();
        void addEvent (Event* e);

        virtual void printSimulation();

        //for debugging / testing
        void outputAndDestroyQueue();
};


//file Simulation.cpp
#pragma once
#include "Simulation.h"

Simulation::Simulation (int maxIterations) : m_currentTime{}, m_events{}, m_maxIterations(maxIterations) {}

Simulation::~Simulation ()
{
    while(!m_events.empty ()) {
        m_events.pop();
    }
}

void Simulation::step ()
{
    if(!m_events.empty ()) {
        Event* nextEvent = m_events.top();
```

```
        m_events.pop();
        m_currentTime = nextEvent->getEventTime();
        nextEvent->executeEvent();
    }
}

void Simulation::runSimulation ()
{
    int i = 0;
    while(!m_events.empty() && !hasEnded() && i < m_maxIterations) {

        std::cout << "Status of simulation at t=" << m_currentTime << std::endl;
        printSimulation();

        Event* currEvent = m_events.top();
        m_events.pop();
        m_currentTime = currEvent->getEventTime();
        currEvent->executeEvent();
        i++;
    }

    std::cout << "Result of simulation:" << std::endl;
    std::cout << " - Reason for end of simulation: ";
    if(m_events.empty ()) {
        std::cout << "no more events in queue" << std::endl;
    }
    else if(i >= m_maxIterations) {
        std::cout << "no result in less than " << m_maxIterations << " steps" << std::endl;
    }
    else {
        std::cout << "end condition fulfilled" << std::endl;
    }
    std::cout << " - Simulation steps: " << i << std::endl;
}

void Simulation::addEvent (Event* e)
{
    m_events.push(e);
}

void Simulation::printSimulation ()
```

```
{
    std::cout << "Current time: " << m_currentTime << std::endl;
    std::cout  << "Events queued: " << m_events.size() << std::endl;

    std::cout  << "Next event: ";
    if(!m_events.empty()) {
        Event* nextEvent = m_events.top();
        nextEvent->printEvent();
    }
    else {
        std::cout  << "no more events queued" << std::endl;
    }
}

void Simulation::outputAndDestroyQueue ()
{
    while(!m_events.empty ()) {
        Event* nextEvent = m_events.top();
        nextEvent->printEvent();
        m_events.pop();
    }
}
```

```cpp
//file: Event.h
#pragma once
#include <iostream>
#include "utils.h"

class Simulation;

class Event
{
    protected:
        int m_time;
        Simulation* m_ownerSimulation;

    public:
        explicit Event(int t, Simulation* ownerSimulation) : m_time(t), m_ownerSimulation(ownerSimulation) {};
        virtual void executeEvent() = 0;

        int getEventTime() {return m_time;}

        virtual void printEvent();
};
```

```cpp
//file: Event.cpp
#pragma once
#include "Event.h"

void Event::printEvent () {
    std::cout << "event at t= " << m_time;
}
```

Die eine Codezeile hätte ich ins Event.h gepackt, da hierfür nun extra eine neue Datei erstellt wurde. (Event.cpp -> Event.h)

Generell ist der Code ganz übersichtlich gestaltet, und kann gut nachverfolgt werden.
Ich würde aber innerhalb des .h-Files immer die einzelnen Methoden/Funktionen & Klassen mit Kommentaren versehen bzw. diese kurz beschreiben.

1.3. Tests:

Nachdem Simulation und Event beides abstrakte Klassen sind, sieht man die richtige Funktionsweise der Queue-Verwaltung etc. erst in den Testfällen der abgeleiteten Simulation (Käufer / Produzent). Somit habe ich mir erlaubt, die Testfälle im 2. Punkt zusammenzufassen.

2. Testszenario (Teilaufgabe b)

2.1. Lösungsidee:

In meiner Simulation gibt es zwei abgeleitete Events. Ein Event für das Produzieren von Waren und eines für den Verbrauch.

Von der Klasse Simulation wurde eine Klasse "MarketSimulation" abgeleitet in der ich zusätzlich Komponenten speichere:

- Puffer-Kapazität,
- Wartezeit, wenn Puffer leer / voll ist
- Endbedingung für Simulation (bestimmte Anzahl an konsumierten Stück)
- Bereits konsumierte Menge

Die Tabelle unterhalb soll die Zustände meiner Simulation skizzieren. Im Beispiel habe ich festgelegt, dass die **Puffer-Kapazität 50 Stück** beträgt und ein **Produzent/Konsument 10 Zeiteinheiten warten muss** wenn dieser voll bzw. leer ist (also ein neues Event in t+10 registriert wird).

Zustand der Queue zu den verschiedenen Zeitpunkten:

	next-Event	nE+1	nE+2		aktuelle Puffergröße
t0	produce(t=5, n=20)	consume(t=10, n=30)	produce(t=12, n=30)		0
t5	consume(t=10, n=30)	produce(t=12, n=30)			20
t10	produce(t=12, n=30)	consume(t=20, n=10)			0
t12	consume(t=20, n=10)				30
t20	no next event				20

Erklärung:

- t0: die Queue wurde mit 3 zufällig generierten Events initialisiert. Der Puffer beträgt zu dem Zeitpunkt natürlich noch 0. Als nächstes (erstes) Event sollen zum Zeitpunkt 5, 20 Stück produziert werden.
- t5: Zum Zeitpunkt 5 liegen nun also 20 Stück auf Lager. Das nächste anstehende Event ist eine Konsumation zum Zeitpunkt 10 mit 30 Stück.
- t10: Im Event hätten 30 Stück konsumiert werden sollen, es lagen aber nur 20 auf Lager. Ich habe es so implementiert, dass alle verfügbaren Stück genommen werden können. Somit hat der Puffer eine Größe von 0. Für die restlichen 10 Stück wurde aber ein neues Event in t+10 (also t=20) registriert (siehe rotes, neues Event).

Die MarketSimulation verwaltet also eine Priority_Queue, die entweder Zeiger auf Consumer- oder Produce-Events enthält. Durch die Abarbeitung der jeweiligen Events wird deren Aktion ausgeführt, welche einen Einfluss auf die Parameter der "besitzenden" Simulation haben. Sobald Konsumenten eine bestimmte Anzahl an Produkten konsumiert haben ist die Simulation beendet.

Gute Erklärung!

2.2. Quellcode:

```
//file: MarketSimulation.h

#pragma once
#include "Simulation.h"
#include "Event.h"
#include "produceEvent.h"
#include "consumeEvent.h"
#include <iostream>

class MarketSimulation: public Simulation
{

    private:
        int m_bufferCapacity;
        int m_curBufferSize{0};
        int m_consumedProducts{0};
        int m_endSimulationSum{100};
        int m_waitingTime;                  //time consumer/producer has to wait when buffer is empty/full

    public:
        MarketSimulation(int bufferCapacity, int waitingTime, int endSimulationSum=100, int maxIterations=1000);

        void randomInitSimulation(int nEvents, int maxEventTime=500, int maxEventNrPieces=50);

        int getBufferCapacity() {return m_bufferCapacity;}
        int getCurBufferSize() {return m_curBufferSize;}
        int getConsumedProducts() {return m_consumedProducts;}
        int getWaitingTime() {return m_waitingTime;}

        void setBufferSize(int v) {m_curBufferSize = v;}
        void setConsumedProducts(int v) {m_consumedProducts = v;}

        virtual bool hasEnded() override;
        virtual void printSimulation() override;
};
```

Gleich wie bei a) würde ich hier wieder die Zahlenwerte auslagern zwecks zentralen Platz für Änderungen.

```cpp
//file MarketSimulation.cpp

#pragma once
#include "MarketSimulation.h"

MarketSimulation::MarketSimulation(int bufferCapacity, int waitingTime, int endSimulationSum, int maxIterations) :
    Simulation(maxIterations), m_bufferCapacity(bufferCapacity),
                        m_waitingTime(waitingTime), m_endSimulationSum(endSimulationSum) {}

void MarketSimulation::randomInitSimulation (int nEvents, int maxEventTime, int maxEventNrPieces)
{
    std::cout << "~~~~Simulation randomly initialised with the following events~~~~" << std::endl;
        for(int i = 0; i < nEvents; i++) {
            int randEvent = randomNumWithin(1,2);
            int randTime = randomNumWithin(1, maxEventTime);
            int randAmount = randomNumWithin(1, maxEventNrPieces);

            if(randEvent == 1) {
                addEvent(new produceEvent(randTime, this, randAmount));
                std::cout << "Production event at time:  " << randTime << " | amount: " << randAmount << std::endl;
            }
            else {
                addEvent(new consumeEvent(randTime, this, randAmount));
                std::cout << "Consumption event at time:  " << randTime << " | amount: " << randAmount << std::endl;
            }
        }
}

bool MarketSimulation::hasEnded () {
    return m_consumedProducts >= m_endSimulationSum;
}

void MarketSimulation::printSimulation ()
{
    std::cout << "Current time: " << m_currentTime << std::endl;
    std::cout << "Current buffer size: " << m_curBufferSize << std::endl;
    std::cout << "Already consumed products: " << m_consumedProducts << std::endl;

    std::cout << "Events queued: " << m_events.size() << std::endl;

    std::cout << "Next event: ";
```

```
    if(!m_events.empty()) {
        Event* nextEvent = m_events.top();
        nextEvent->printEvent();
    }
    else {
        std::cout << "no more events queued" << std::endl;
    }
}


//file: produceEvent.h
#pragma once
#include "Event.h"
#include "MarketSimulation.h"


class produceEvent : public Event
{
    private:
        int m_numberProducts;

    public:
        produceEvent(int time, Simulation* ownerSimulation, int numberProducts);
        void executeEvent() override;
        void printEvent() override;
};


//file: produceEvent.cpp
#include "produceEvent.h"

produceEvent::produceEvent(int time, Simulation* ownerSimulation, int numberProducts) :
        Event(time, ownerSimulation), m_numberProducts(numberProducts) {}

void produceEvent::executeEvent ()
{
        MarketSimulation* mySim = dynamic_cast<MarketSimulation*>(m_ownerSimulation);

        std::cout<< std::endl << "## Processing next event ##" << std::endl;

        if((mySim->getCurBufferSize() + m_numberProducts) > mySim->getBufferCapacity ()) {
```

```
            //make buffer full
            int remainingProducts = m_numberProducts - (mySim->getBufferCapacity() - mySim->getCurBufferSize());
            std::cout << "Buffer too small. Only " << mySim->getBufferCapacity()-mySim->getCurBufferSize()
                      << " pieces will be added to buffer" << std::endl;

            mySim->setBufferSize(mySim->getBufferCapacity());
            std::cout << "For remaining " << remainingProducts << " pieces new production event in "
                      << m_time+mySim->getWaitingTime() << " is queued" << std::endl << std::endl;

            //new event for remaining production size
            produceEvent* delayedEvent = new produceEvent(m_time + mySim->getWaitingTime(), m_ownerSimulation, remaining-
Products);
            m_ownerSimulation->addEvent(delayedEvent);
        }
        else {
            std::cout << "Everything ok" << std::endl << std::endl;
            mySim->setBufferSize(m_numberProducts+mySim->getCurBufferSize());
        }

}

void produceEvent::printEvent ()
{
    std::cout << "Production event at t=" << m_time << " | producing: " << m_numberProducts << " pieces" << std::endl;
}


//file consumeEvent.h
#pragma once
#include "Event.h"
#include "MarketSimulation.h"

class consumeEvent: public Event
{
    private:
        int m_numberProducts;

    public:
        consumeEvent(int time, Simulation* ownerSimulation, int numberProducts);
        void executeEvent() override;
```

```
        void printEvent() override;
};

//file: consumeEvent.cpp
#include "consumeEvent.h"

consumeEvent::consumeEvent (int time, Simulation* ownerSimulation, int numberProducts) :
    Event(time, ownerSimulation), m_numberProducts(numberProducts) {}

void consumeEvent::executeEvent ()
{
    MarketSimulation* mySim = dynamic_cast<MarketSimulation*>(m_ownerSimulation);

    std::cout << std::endl << "## Processing next event ##" << std::endl;

    if(mySim->getCurBufferSize () - m_numberProducts > 0) {         //enough products in buffer to consume
        std::cout << "Everything ok" << std::endl << std::endl;
        mySim->setBufferSize(mySim->getCurBufferSize() - m_numberProducts);
        mySim->setConsumedProducts(mySim->getConsumedProducts()+m_numberProducts);
    }
    else {      //wait, register consumption event for later again

        //let consumer consum remaining products from buffer
        mySim->setConsumedProducts(mySim->getConsumedProducts()+mySim->getCurBufferSize());


        //create new event for remaining prodcuts from this event
        int remainingProducts = m_numberProducts - mySim->getCurBufferSize();
        std::cout << "Buffer too small. Only " << mySim->getCurBufferSize() << " pieces can be consumed" << std::endl;

        mySim->setBufferSize(0);
        std::cout << "For remaining " << remainingProducts << " pieces new consumption event in "
                          <<  mySim->getWaitingTime() + mySim->getCurrentTime() << " is queued" << std::endl <<
std::endl;

        consumeEvent* delayedEvent = new consumeEvent(m_time + mySim->getWaitingTime(), m_ownerSimulation, remainingPro-
ducts);
        m_ownerSimulation->addEvent(delayedEvent);
    }
}
```

```
void consumeEvent::printEvent ()
{
    std::cout << "Consume event at t=" << m_time << " | consuming: " << m_numberProducts << " pieces" << std::endl;
}
```

Auch hier war der Code wieder gut zu folgen.
Top!

2.3. Testfälle:

Initialisierung und push-Funktion:

Soweit ich ergoogelt habe gibt es für eine priority_queue keinen Iterator. Somit habe ich die Queue mit top und pop ausgegeben wobei die Queue jedoch zerstört wird.

Der obere Teil der Simulation zeigt die Events, die ich zufällig erstellt und gepusht habe. Der untere Teil zeigt, in welcher Reihenfolge die Events dann abgearbeitet wurden. Wie man sieht sind die Events aufsteigend nach der Zeit des Eintretens sortiert.

```
~~~~Simulation randomly initialised with the following events~~~~
Production event at time:  3 | amount: 8
Consumption event at time:  252 | amount: 13
Production event at time:  175 | amount: 36
Consumption event at time:  233 | amount: 47
Consumption event at time:  160 | amount: 14
Consumption event at time:  449 | amount: 27
Production event at time:  316 | amount: 43
Production event at time:  327 | amount: 41
Consumption event at time:  499 | amount: 38
Consumption event at time:  14 | amount: 42

################################

~~~~Structure of priority queue~~~~
Production event at t=3 | producing: 8 pieces
Consume event at t=14 | consuming: 42 pieces
Consume event at t=160 | consuming: 14 pieces
Production event at t=175 | producing: 36 pieces
Consume event at t=233 | consuming: 47 pieces
Consume event at t=252 | consuming: 13 pieces
Production event at t=316 | producing: 43 pieces
Production event at t=327 | producing: 41 pieces
Consume event at t=449 | consuming: 27 pieces
Consume event at t=499 | consuming: 38 pieces
```

Step-Methode

1. Initialisierung der Simulation mit 10 Events. Hinweis, das ist nicht die Reihenfolge, in der die Events in der Queue gespeichert sind (ohne Iterator hätte ich die Queue bei der Ausgabe zerstört)

```
Simulation parameters: BufferCapacity = 100, waiting time = 50
~~~~Simulation randomly initialised with the following events~~~~
Consumption event at time:  233 | amount: 24
Consumption event at time:  312 | amount: 37
Consumption event at time:  220 | amount: 7
Production event at time:  67 | amount: 34
Production event at time:  297 | amount: 1
Production event at time:  229 | amount: 45
Consumption event at time:  45 | amount: 2
Consumption event at time:  236 | amount: 45
Consumption event at time:  226 | amount: 10
Consumption event at time:  86 | amount: 47
```

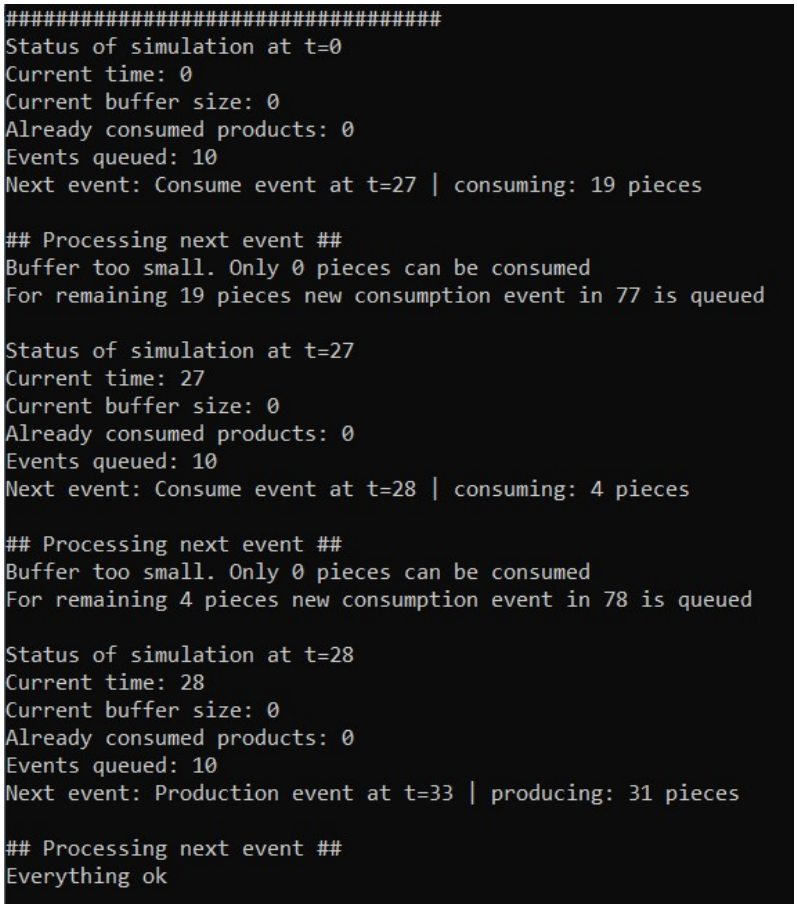
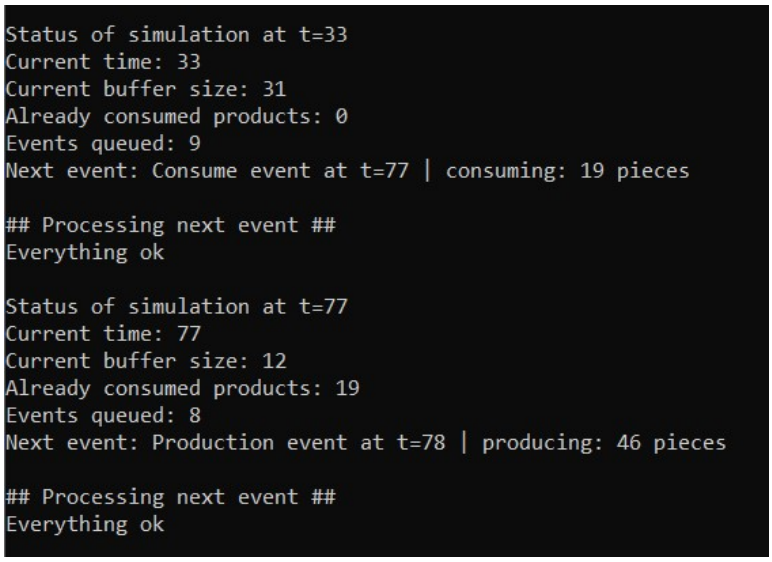
2. In einer for loop rufe ich step 10x auf (Ausschnitt zeigt nur 3 Aufrufe)

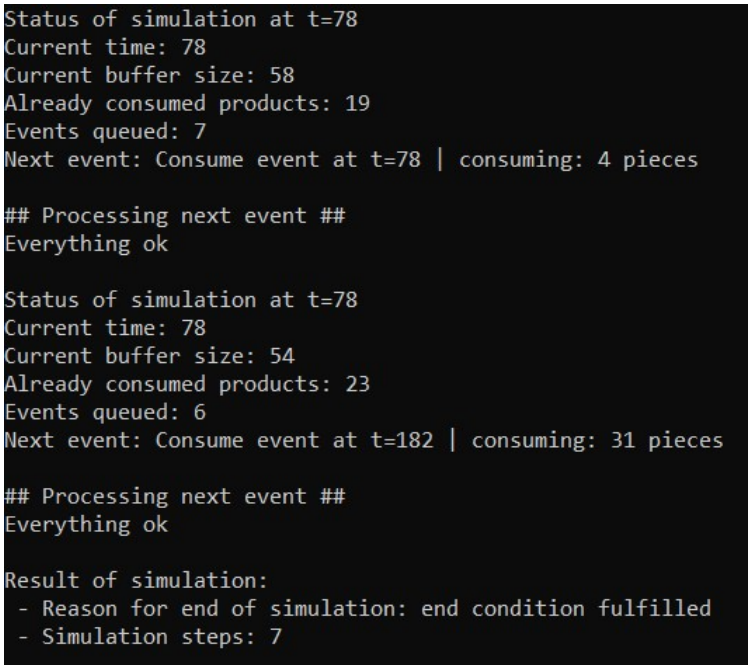
```
##################################
Status of simulation at t=0
Current time: 0
Current buffer size: 0
Already consumed products: 0
Events queued: 10
Next event: Consume event at t=45 | consuming: 2 pieces

## Processing next event ##
Buffer too small. Only 0 pieces can be consumed
For remaining 2 pieces new consumption event in 95 is queued

Status of simulation at t=45
Current time: 45
Current buffer size: 0
Already consumed products: 0
Events queued: 10
Next event: Production event at t=67 | producing: 34 pieces

## Processing next event ##
Everything ok

Status of simulation at t=67
Current time: 67
Current buffer size: 34
Already consumed products: 0
Events queued: 9
Next event: Consume event at t=86 | consuming: 47 pieces
```

RunSimulation-Methode

	Kommentar

`Simulation parameters:`
`- Buffer capacity = 100`
`- Waiting time = 50`
`- End condition = 50`
 	
`~~~~~Simulation randomly initialised with the following events~~~~~`
`Consumption event at time: 28 | amount: 4`
`Consumption event at time: 486 | amount: 16`
`Production event at time: 78 | amount: 46`
`Production event at time: 33 | amount: 31`
`Production event at time: 388 | amount: 44`
`Consumption event at time: 27 | amount: 19`
`Consumption event at time: 219 | amount: 18`
`Consumption event at time: 182 | amount: 31`
`Consumption event at time: 224 | amount: 45`
`Production event at time: 242 | amount: 10`	

	Zu t=0 ist der Puffer noch leer. Es sind 10 Events in der Queue. Das erste anstehende Event ist eine Konsumation. Nachdem der Puffer leer ist, wird das Event für t+50 (also 27+50 = 77) noch einmal eingeplant Nach dem ersten Event ist der Puffer noch immer auf 0, weil ja nur versucht wurde zu konsumieren. Das nächste Event steht für t=28 an. Aber leider wieder Konsumation. Puffer ist noch immer leer aber nun steht endlich eine Produktion an. Das Event wird normal verarbeitet
	Nachdem im letzten Event 31 Stück produziert wurden, liegen diese nun auf Lager. Hier taucht nun das allererste Event wieder auf, das für t+50 neu eingeplant wurde. Dieses mal sind genug Stück auf Lager. Das Event wird erfolgreich ausgeführt.



Bisher bereits 23 Stück konsumiert. Im nächsten Event sollen wieder 31 folgen. Der Puffer ist groß genug.

Nachdem nun die Abbruchbedingung von 50 konsumierten Stück erreich ist wird die Simulation beendet.

Gute Erklärung der Testfälle.
Im Gesamten habe ich nicht wirklich etwas gefunden,
was beim Anschauen deiner Dokumentation schon ins Auge sticht.

Das Programm zeigt beim Kompilieren auch keine Fehler an &
läuft einwandfrei bei mir.

Gute Ausarbeitung ;)

3. Testprogramm (main)

```
//file: main.cpp
#include <set>
#include <iterator>
#include <iostream>
#include "Simulation.h"
#include "produceEvent.h"
#include "consumeEvent.h"

using namespace std;

void test_simulation_creation () {
    MarketSimulation* mySimulation = new MarketSimulation(70,50,100); //parameter: bufferCap., waitingTime, endEcondition
    mySimulation->randomInitSimulation(10);
    cout << endl<< "####################################" << endl;

    //output of queue
    cout << endl << "~~~~Structure of priority queue~~~~" << endl;
    mySimulation->outputAndDestroyQueue();

    delete mySimulation;

}

void testStep () {
    MarketSimulation* mySimulation = new MarketSimulation(100,50,100);  //parameter: bufferCapacity, waitingTime, end con-
dition

    cout << "Simulation parameters: BufferCapacity = 100, waiting time = 50" << endl;
    mySimulation->randomInitSimulation(10);
    cout << endl<< "####################################" << endl;

    for(int i = 0; i < 10; i++) {
        cout << "Status of simulation at t=" << mySimulation->getCurrentTime() << endl;
        mySimulation->printSimulation(); cout << endl;
        mySimulation->step();
    }

    delete mySimulation;
}
```

```
void testSimulation () {
    int bufferCapacity = 100;           //max 100 pieces can be in the buffer
    int waitingTime = 50;               //if an event cannot be completed it is rescheduled for t+50
    int endCondition = 50;              //once consumers got 50 pieces simulation is ended

    cout << "Simulation parameters:" << endl;
    cout << " - Buffer capacity = " << bufferCapacity << endl;
    cout << " - Waiting time = " << waitingTime << endl;
    cout << " - End condition = " << endCondition << endl << endl;

    MarketSimulation* mySimulation = new MarketSimulation(bufferCapacity, waitingTime, endCondition);
    mySimulation->randomInitSimulation(10);

    cout << endl<< "####################################" << endl;

    mySimulation->runSimulation();
    delete mySimulation;
}

int main () {
    //test_simulation_creation();
    //testStep();
    testSimulation();

    return 0;
}
```