Objektorientierte Simulation und Analyse von Wertströmen in einem geschlossenen System bei Variation relevanter Parameter unter besonderer Berücksichtigung der Laufzeit

Gymnasium Mainz-Oberstadt

Ole Petersen

Betreuende Lehrkraft: Herr Schaede

Schulfach: Informatik

Jahrgangsstufe: 12

Schuljahr 2018/2019

Inhaltsverzeichnis

Sowohl für Unternehmen als auch für Wirtschaftspolitiker gibt es wohl kaum eine interessantere Frage als die nach der Zukunft. Man möchte voraussagen, wie sich gewisse Parameter entwickeln, wenn sich die Akteure nach bestimmten Regeln verhalten. So können Unternehmen beispielsweise möglichst effizient investieren und Politiker Steuern so wählen, dass die Wirtschaft in die gewünschte Richtung gelenkt wird, ohne jede Idee faktisch ausprobieren zu müssen, was gerade zu Ausbildungszwecken essentiell ist [4]. In dieser Arbeit wird eine eigene kleine Wirtschaftssimulation geschrieben, anhand derer grundsätzliche Prinzipien und Ansätze dazu getestet werden.

1 Vorstellung der simulierten Objekte

Um ein komplexes Problem wie das von Wertströmen zu simulieren, muss ein vereinfachtes Modell für die realen Vorgänge unserer Welt geschaffen werden. Dies wird durch eine Reduktion auf Privatpersonen, Unternehmen, Produktionsmitteln und Staaten erreicht. Die verwendete Währung soll größenordnungsmäßig dem Euro entsprechen und die Zeit in Jahren gemessen werden. Die Komponenten der Simulation werden nun näher beschrieben. Die genauen Vererbungsbeziehungen sind dem Klassendiagramm (*Abbildung 1*) zu entnehmen. Um dieses

übersichtlich zu gestalten, wurde die genaue Beschreibung der Attribute und Methoden der Klassen jedoch in die folgende Text- und Tabellenform ausgelagert.

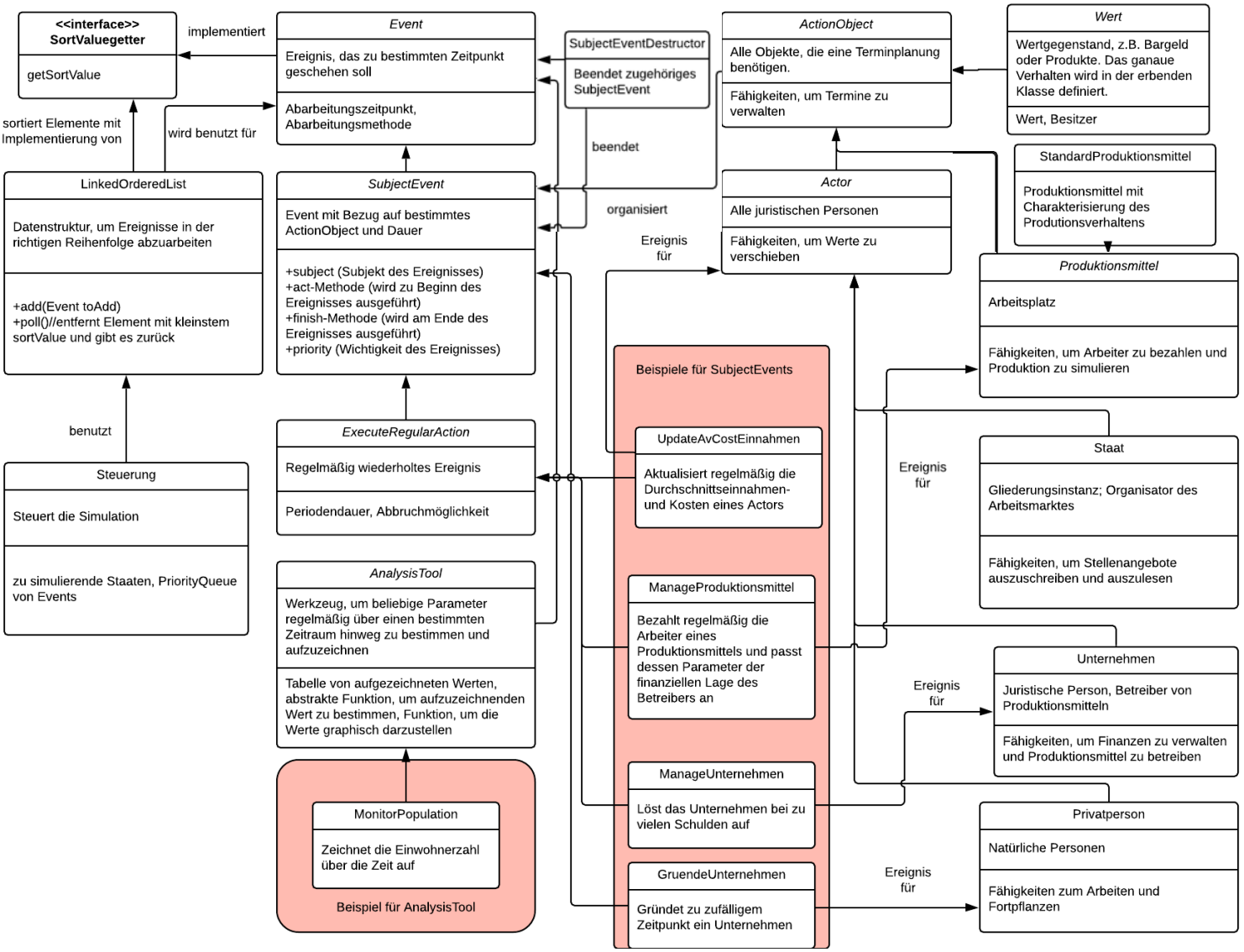


Abbildung 1: Vereinfachtes Klassendiagramm. Zugunsten der Übersichtlichkeit wird bei jeder Klasse im ersten Feld der Titel (kursiv bei abstrakten Klassen), im zweiten Feld eine nähere Beschreibung und ggf. im dritten Feld die Eigenschaften (Methoden und Attribute) der Klasse genannt, statt die technische UML - Notation zu verwenden.

1.1 Staaten

Jeder Staat hat eine Liste von seinen Einwohnern und Unternehmen. Außerdem wird über Staaten der Arbeitsmarkt geregelt. Zu jedem Staat gibt es eine Sammlung von ausgeschriebenen offenen Arbeitsplätzen (also eine Liste von Produktionsmitteln) namens *openJobs*. Unternehmen können dort offene Stellen eintragen und Privatpersonen können sich mittels der Methode

getOpenJobsForPerson eine Liste von Jobs liefern lassen, für die sie qualifiziert genug sind. Zudem werden alle Wertströme, deren genauer Sender oder Empfänger wegen des noch nicht vollständigen Modells nicht bekannt sind, über den Staat abgewickelt.

1.2 Actor

Instanzen der Klasse *Actor* (im Folgenden auch synonym als Akteure bezeichnet) sind juristische oder natürliche Personen. Sie haben folgende Eigenschaften:

Attribut	Datentyp	Bedeutung
location	*Location*	Aktuelle Position des *Actors*
besitz	Liste von Werten	Besitz des *Actors*
bargeld	*Bargeld*	Frei verfügbares Geld des *Actors*.
meinStaat	*Staat*	Staat des *Actors*
avEinnahmen	double	Durchschnittliche monatliche Einnahmen des *Actors*
avCost	double	Durchschnittliche monatliche Kosten des *Actors*
lastAvCostEinnahmenUpdate	double	Letzter Zeitpunkt, zu dem die durchschnittlichen Einnahmen und Kosten des *Actors* aktualisiert wurden.
cost	double	Kosten seit letzter Aktualisierung der Durchschnittskosten des *Actors*
einnahmen	double	Kosten seit letzter Aktualisierung der Durchschnittseinnahmen des *Actors*

Methoden:

1.2.1 giveMoney

Diese Methode gibt einem Akteur einen bestimmten Geldbetrag. Die Kosten des Senders sowie die Einnahmen des Empfängers werden dabei aktualisiert.

1.2.2 giveWert

Analog dazu kann auch ein Wertgegenstand an einen Akteur übertragen werden.

1.2.3 sellWert

Mit den letzten beiden Methoden kann ein Wert auch verkauft werden.

1.2.4 updateAvCostAndEinnahmen

Durch diese Methode soll in je einer Variablen eine Art Durchschnittswert der monatlichen Kosten und Einnahmen des Unternehmens aktualisiert werden. Allgemeiner ist das Problem Folgendes: Man erhält in bestimmten, nicht unbedingt regelmäßigen Zeitintervallen Werte eines Parameters *p* und möchte daraus einen von Fluktuationen möglichst bereinigten Wert *w* des Parameters

berechnen. Das arithmetische Mittel ist hierfür ungeeignet, weil es den gesamten Messzeitraum widerspiegelt anstatt eines aktuellen Wertes. Stattdessen wird deshalb in dieser Arbeit des Öfteren ein anderes Verfahren verwendet: Bei jeder neuen Messung wird der geglättete Wert des Parameters um einen kleinen Faktor, den Anpassungsfaktor *c*, mal den Abstand *d* des neuen Messwertes zum alten geglätteten Wert erhöht. Daraus ergibt sich die Formel für Aktualisierung des geglätteten Wertes $w_{neu}=w_{alt}+c*d=w_{alt}+c*(w_{alt}-p_{gemessen})$. So gleichen sich die Werte nur mit einer gewissen Verzögerung an. Dadurch werden kurze Schwankungen kaum berücksichtigt, weshalb das Verfahren von nun an als Glättungsverfahren bezeichnet wird.
Gegebenenfalls wird der „Angleichungswert“ *c*d* noch mit dem Zeitraum, für den die Messung steht, gewichtet.
Bei diesem Beispiel werden in nicht unbedingt regelmäßigen Zeitabschnitten die Kosten seit der letzten Aktualisierung der „Durchschnittswerte“ gemessen. Beim Aufruf der Methode werden zur Erstellung eines neuen Messpunktes für der beiden Parameter mittels des Wertes von *lastAvCostEinnahmenUpdate cost* und *einnahmen* auf einen Monat normiert. Schließlich werden die Parameter auf null gesetzt, da sie bereits in ihren jeweiligen „bereinigten Wert“ eingegangen sind und *lastAvCostEinnahmenUpdate* wird aktualisiert.

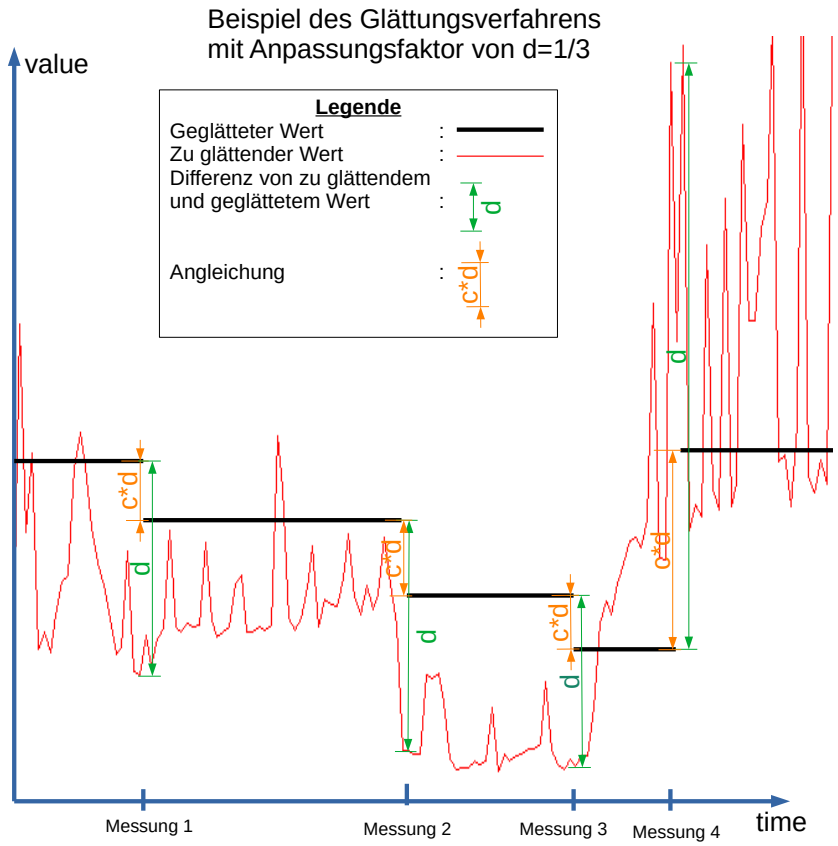


Abbildung 2: Visualisierung des Glättungsverfahrens

1.3 Privatpersonen

Diese Klasse (synonym verwendet: Mensch, Person, Arbeiter) beschreibt die Menschen der Simulation.

Attribut	Datentyp	Bedeutung
levelOfQualification	double	Gibt die Attraktivität der Person für Arbeitgeber an. Die Einheit hierfür ist der Geldbetrag, der bei einer durchschnittlichen Person nötig wäre, um die Qualifikation der Person zu erreichen.
skillFactor	double	Ein Maß für das Talent der Person. Es gibt an, wie sehr das *levelOfQualification* der Person pro Geldbetrag, der in ihre Bildung fließt, steigt. Dieser Parameter ist bei einer durchschnittlichen Person 1.
qualifications	Liste von *Produktionsmitteln*	Gibt an, an welchen *Produktionsmitteln* die Person berechtigt ist, zu arbeiten.
arbeitgeber	*Unternehmen*	Arbeitgeber der Person
arbeitsstelle	*Produktionsmittel*	Maschine, an der die Person aktuell arbeitet. Alle Personen, die am gleichen Produktionsmittel arbeiten, erhalten den gleichen Lohn.
home	*Location*	Wohnort der Person
worktime	double	Arbeitszeit in Jahren, die noch nicht vom Arbeitgeber bezahlt wurde
parent	*Privatperson*	Erzeuger der Person. Um das Problem der Partnerwahl zu umgehen, werden Kinder nur von einer Person erzeugt und Personen haben kein Geschlecht.
children	Liste von *Privatpersonen*	Von der Person erzeugte Personen, also ihre „Kinder“.

Methoden:

1.3.1 getQualification(Produktionsmittel toGet)

Qualifiziert die Person für das übergebene Produktionsmittel auf Kosten seines Betreibers. Dabei erhöht sich auch der *levelOfQualification* der Person um das Produkt aus *skillFactor* der Person und der Komplexität *toGets*.

1.3.2 double evaluateJob(Produktionsmittel angebot)

Gibt eine Bewertung eines Jobangebots zurück. Aktuell gehen nur Stundenlohn und Entfernung zum Wohnort der Person ein, die Methode lässt sich jedoch beliebig erweitern. Je höher die Bewertung ist, desto besser ist das Jobangebot.

1.4 Unternehmen

Ich verwende die Definition der „Bundeszentrale für politische Bildung“ von *Unternehmen* als „wirtschaftliches Gebilde, das nach einem von der Unternehmensleitung bestimmten Wirtschaftsplan durch Einsatz der Produktionsfaktoren Güter hervorbringt“ [1]. Sie fungieren also hauptsächlich als Betreiber von *Produktionsmitteln*.

Attribut	Datentyp	Bedeutung
angestellte	Liste von *Privatpersonen*	Angestellte des *Unternehmens*
produktionsmittel	Liste von *Produktionsmitteln*	*Produktionsmittel* des Unternehmens
products	Liste von *Werten*	Noch nicht verkaufte *Produkte* des *Unternehmens*
besitzer	Privatperson	Besitzer des *Unternehmens*

Methoden:

1.4.1 sellProducts

Die *Produkte* aller *Produktionsmittel* des Unternehmens werden an *meinStaat* verkauft.

1.5 Produktionsmittel

Anders als die „Bundeszentrale für politische Bildung“ definiere ich Produktionsmittel nicht als „alle bei der Produktion von Gütern erforderlichen Gegenstände wie Gebäude, Maschinen, Anlagen, Werkzeuge, Roh-, Hilfs- oder Betriebsstoffe“ [2]. Stattdessen verstehe ich darunter eine Menge von Werkzeugen, die zur Produktion von Gütern benötigt werden. Das kann beispielsweise eine Fabrik oder auch ein Büroraum mit Computern und der richtigen Software sein. Wichtig ist dabei, dass jeder Mensch an maximal einem Produktionsmittel arbeitet, weshalb beispielsweise ein Schraubenzieher kein Produktionsmittel ist.

Attribut	Datentyp	Bedeutung
stundenlohn	double	Stundenlohn aller Arbeiter am Produktionsmittel
angestellte	Liste von *Privatpersonen*	Angestellte an diesem Produktionsmittel. Nur Personen aus dieser Liste können am Produktionsmittel arbeiten.
workingWithMe	HashSet von *Privatpersonen*	Alle aktuell am Produktionsmittel arbeitende Arbeiter
betreiber	*Unternehmen*	*Unternehmen*, das das *Produktionsmittel* betreibt. Hierüber werden alle Kosten abgewickelt und die produzierten *Produkte* werden an dieses

Attribut	Datentyp	Bedeutung
		Unternehmen übergeben.
costPerMonth	double	Monatliche Fixkosten für das *Produktionsmittel*
minQualifikation	double	Mindestqualifikation für neu anzustellende Arbeiter an diesem Produktionsmittel. Sie dient dazu, die Kosten für die Umschulung neuer Arbeiter so gering wie möglich zu halten, indem eine gewisse Mindestausgangsqualifikation vorausgesetzt wird. Wenn zu wenige Bewerber vorhanden sind, kann beispielsweise die *minQualification* verringert werden.
lastChange	double	Der letzte Zeitpunkt, zu dem sich etwas an der Besetzung des *Produktionsmittels* geändert hat sowie der Zeitpunkt, bis zu dem bereits die vom Produktionsmittel produzierten *Produkte* beim Betreiber abgeliefert wurden.
averageAmountOfWorkersForMe	double	Durchschnittliche Anzahl der am *Produktionsmittel* arbeitenden Arbeiter
optimumAverageAmountOfWorkersForMe	double	Optimale durchschnittliche Anzahl der am *Produktionsmittel* arbeitenden Personen. Wenn diese größer als die tatsächliche Anzahl ist, sollten neue Arbeiter eingestellt werden.
komplexitaet	double	Maß für die Anforderungen an die Arbeiter für das *Produktionsmittel*.

Methoden:

1.5.1 Produzierte Produkte einfordern

claimProducts()

Diese abstrakte Methode bestimmt die von *lastChange* bis zum aktuellen Zeitpunkt produzierten *Produkte*. Die Idee ist, diese Methode immer aufzurufen, wenn sich etwas an der Besetzung des *Produktionsmittels* ändert. So wird die Produktion nämlich in Zeitabschnitte unterteilt, innerhalb derer das *Produktionsmittel* gleich effektiv gearbeitet hat. Über die Dauer eines solchen Abschnitts, die Parameter des *Produktionsmittels* und die Anzahl der am *Produktionsmittel* arbeitenden Kräfte (*workingWithMe.size()*) lassen sich deshalb recht genau die produzierten Güter charakterisieren.

getProducts()

Hier werden mittels der claimProducts-Methode die im letzten Zeitabschnitt produzierten Produkte ermittelt und diese an den Betreiber des *Produktionsmittels* gegeben. Zudem wird der Zeitpunkt der letzten Veränderung der Besetzung des *Produktionsmittels* auf den aktuellen Zeitpunkt gesetzt, da für den letzten Zeitraum die Produkte bereits eingefordert wurden, und *averageAmountOfWorkersForMe* wird mittels des Glättungsverfahrens mit der Anzahl der im letzten Zeitabschnitt am Produktionsmittel arbeitenden Personen aktualisiert.

1.5.2 Arbeit beginnen/beenden

boolean startWorking(Privatperson toStart)

Diese Methode startet das Arbeiten von *toStart* wenn möglich und gibt den Erfolg des Unterfangens zurück. Das Arbeiten ist nicht möglich, wenn *toStart* sich nicht beim *Produktionsmittel* befindet oder nicht für das *Produktionsmittel* qualifiziert ist.
Zum Starten der Arbeiten muss zunächst die getProducts - Methode ausgeführt werden, da nach dem Ereignis eine Person mehr am Produktionsmittel aktiv ist. Zudem wird *toStart workingWithMe* hinzugefügt.

boolean stopWorking(Privatperson toStop)

Diese Methode wird aufgerufen, wenn ein Arbeiter seinen Arbeitsplatz verlässt. Dabei müssen die produzierten Produkte abgerufen werden sowie *toStop* aus *workingWithMe* entfernt werden. Zurückgegeben wird, ob *toStop* davor am Produktionsmittel gearbeitet hat.

1.5.3 Personalmanagement

addAngestellte(Privatperson toAdd)

Stellt *toAdd* beim Produktionsmittel ein. Dazu muss *toAdd* die Qualifikation für seinen neue Arbeitsstelle erhalten. Ihm, seinem neuen Arbeitgeber sowie seiner neuen Arbeitsstelle (this) werden die neuen Arbeitsverhältnisse abgespeichert und der Stundenlohn *toAdds* wird aktualisiert.

removeAngestellte(Privatperson toRemove)

Kündigt *toRemove*. Dafür muss *toRemove* zunächst seine Arbeit beenden. Dann werden *toRemoves* Arbeitgeber und Arbeitsstelle auf null gesetzt und *toRemove* aus allen Angestelltenlisten entfernt.

double getCostForQualification(Privatperson privatperson)

Liefert die Kosten, um eine Person für das *Produktionsmittel* zu qualifizieren. Im Modell ist das die Differenz aus der Komplexität des *Produktionsmittels* und der Qualifikation der Person, mindestens jedoch 50.

1.6 StandardProduktionsmittel

Dies ist eine erste Version eines *Produktionsmittels*. Die Implementierung der Methode *claimProducts* beruht hierbei auf der Annahme, dass das Produktionsmittel einen maximalen Wert *maxProdValuePerYear* an jährlich produzierbaren Gütern, der dadurch erreicht würde, dass unendlich viele Personen an dem *Produktionsmittel* arbeiten. Wenn niemand am *Produktionsmittel* arbeitet, soll im Jahr ein Wert von null produziert wird. Es wird die Annahme getroffen, dass das *Produktionsmittel* ¾ des maximal von ihm produzierbaren Wertes produziert, wenn die optimale Anzahl an Arbeitern an ihm arbeitet. Um eine Asymptote des produzierten Werts über die Anzahl der am *Produktionsmittel* arbeitenden Menschen an *maxProdValuePerYear* zu erreichen, wird eine

Exponentialfunktion angesetzt (siehe *Abbildung 3*). Außerdem ist der produzierte Wert proportional zur Länge *deltaT* des betrachteten Zeitabschnitts. Das liefert folgende Formel für den produzierten Wert:

$$deltaT * maxProdValuePerYear * \left(1 - 4^{-\text{workingWithMe.size()} / optimumAverageAmountOfWorkersForMe}\right)$$

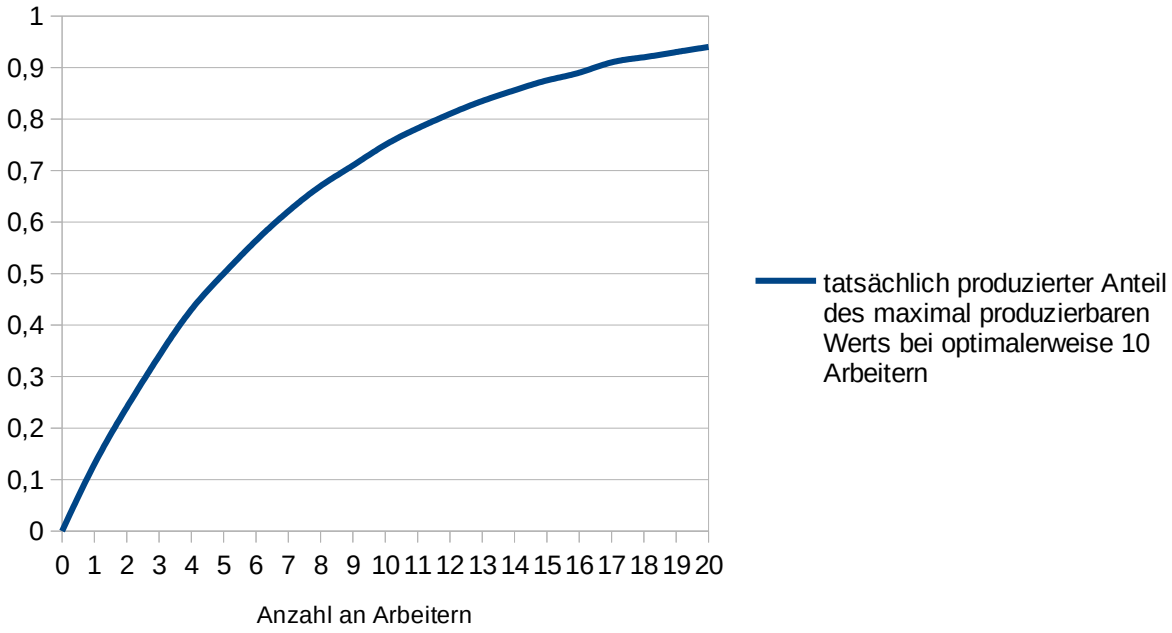


Abbildung 3: Produktionsverhalten eines Beispielproduktionsmittels. Hinweis: Die Annahme, dass bei der optimalen Anzahl an Arbeitern ¾ des maximalen Werts produziert werden, ist zwar aus der Luft gegriffen, allerdings lässt diese Anzahl sich nur berechnen, wenn man einen von dieser Anzahl unabhängigen Stundenlohn annimmt, was nicht der Fall ist.

2 Eventmanagement-System

Um eine lauffähige Simulation zu erstellen, müssen bei den Akteuren zur richtigen Zeit die richtigen Methoden aufgerufen werden. Dies geschieht durch die Verwendung von sogenannten *Events*:

2.1 Event

Events sind Ereignisse (wird im Folgenden synonym verwendet), die zu einem bestimmten Zeitpunkt *date* geschehen sollen. Das Abarbeiten eines Ereignisses erfolgt über den Aufruf der process - Methode desselben. Zudem wird zu jedem Ereignis in der Variable *sortedInList* abgespeichert, ob es bereits in die später erklärte Struktur zum Abarbeiten von Ereignissen einsortiert ist, damit dies nicht mehrmals geschieht, sowie ob es noch aktiviert ist. Zu betonen ist dabei, dass die *process* - Methode je nach Ereignis variiert, um verschiedene Funktionen ausführen zu können. Gemeinsam ist allen Ereignissen nur, dass die Methode vorhanden ist.

2.2 SubjectEvent

Ereignisse in direktem Bezug zu einem bestimmtes Objekt werden als *SubjectEvent* abgespeichert. Diese erweitern die Klasse *Event* durch das Abspeichern des Subjekts, also des Handelnden des Ereignisses und dadurch, dass die Ereignisse eine Dauer namens *length* haben können. Um die Ausdehnung der Ereignisse zu modellieren, wird die Methode *act* zu Beginn und die Methode *finish* am Ende des Ereignisses aufgerufen. Dazu wird ein weiteres Ereignis, der sogenannter *SubjectEventDestructor*, zu jedem *SubjectEvent* erstellt, sobald die process - Methode desselben aufgerufen wird. Der *SubjectEventDestructor* ist ein Ereignis, das lediglich zum richtigen Zeitpunkt die finish - Methode des zugehörigen *SubjectEvents* ausführt. Außerdem wird dabei abgespeichert, dass das *SubjectEvent* abgearbeitet ist und es wird aus der Liste von Ereignissen *events* seines Subjekts entfernt.
Die Klasse des Subjekts eines *SubjectEvents* ist generisch definiert, um sie für mehrere Klassen von Akteuren nutzen zu können.
Es gibt verschiedene Arten von Ereignissen: Solche, zu denen das Subjekt aktiv gebraucht wird (z.B. Arbeit) und andere, in denen passiv etwas passiert (z.B. die Aktualisierung des Lohns eines Arbeiters). Diese Eigenschaft wird in dem Attribut *activeEvent* abgespeichert. Zu passiven *SubjectEvents* muss kein *SubjectEventDestructor* erstellt werden, weil das Ereignis ohne Zeitverzögerung beendet werden kann.
Zu *SubjectEvents* wird außerdem in der Variable *priority* die entsprechende Wichtigkeit abgespeichert, um bei konfligierenden Terminen entscheiden zu können, welcher von ihnen Vorrang hat.
Die Methode *addingFailed* gibt an, wie vorgegangen werden soll, wenn ein *SubjectEvent* seinem Subjekt nicht hinzugefügt werden kann, z.B. weil zu dem Termin ein wichtigeres Ereignis stattfindet.

2.3 ActionObject

2.3.1 Nutzen und Aufbau

ActionObjects sind alle Objekte, die zu bestimmten Zeiten bestimmte *SubjectEvents* abarbeiten müssen. In der Klasse *ActionObject* ist ein Terminmanagementsystem implementiert. Dieses basiert auf einer nach dem Abarbeitungszeitpunkt geordneten Liste *events* von Ereignissen, die das Objekt betreffen. *Events* ist also eine Art Terminkalender für das Objekt.

2.3.2 boolean addEvent(SubjectEvent toAdd)

Diese Methode fügt *toAdd* dem Terminplan des *ActionObjects* hinzu, wenn es nicht mit einem wichtigeren Ereignis kollidiert, und sorgt dafür, dass es abgearbeitet wird.

1. Die Stelle in *events* finden, an der das hinzuzufügende Ereignis *toAdd* eingefügt werden müsste, wenn dies nicht durch eine Terminkollision verhindert wird
2. Nach Kollisionen mit anderen Ereignissen suchen, wenn das *ActionObject* für *toAdd* aktiv benötigt wird
    1. Bei allen Ereignissen, die vor *toAdd* beginnen, untersuchen, ob sie aktive *Events* sind und so lange andauern, dass sie erst aufhören, nachdem *toAdd* beginnen würde. Wenn sie zusätzlich noch wichtiger als *toAdd* sind, wird die Methode abgebrochen und zurückgegeben, dass das Ereignis nicht hinzugefügt werden kann sowie die *addingFailed* - Methode *toAdds* aufgerufen. Sonst wird durch das Abspeichern des *Events* in der Liste *eventsToRemove* vermerkt, dass es entfernt werden muss, wenn *toAdd* tatsächlich hinzugefügt wird.
    2. Bei allen Ereignissen, die nach *toAdd* und vor der Beendigung *toAdds* beginnen, untersuchen, ob sie aktive *Events* sind und *toAdd* so lange andauert, dass sie erst aufhören, nachdem das potentiell kollidierende Ereignis beginnen würde. Wenn sie zusätzlich noch wichtiger als *toAdd* sind, wird die Methode abgebrochen und zurückgegeben, dass das Ereignis nicht hinzugefügt werden kann sowie die *addingFailed*-Methode *toAdds* aufgerufen. Sonst wird durch das Abspeichern des *Events* in der Liste *eventsToRemove* vermerkt, dass es entfernt werden muss, wenn *toAdd* tatsächlich hinzugefügt wird.
3. Wenn nicht abgebrochen wurde, kann das Ereignis schließlich an der vorher bestimmten Stelle einsortiert werden.
4. Dann müssen noch die Ereignisse entfernt werden, die durch toAdd nicht mehr ausführbar sind. Zu ihnen wird abgespeichert, dass sie nicht mehr ausgeführt werden sollen und ihre *addingFailed* - Methode wird aufgerufen, um eventuell Alternativtermine zu finden.
5. *toAdd* wird ein die PriorityQueue einsortiert und dies auch in der Variable sortedInList in *toAdd* vermerkt.

2.3.3 Entfernen von Ereignissen

Manchmal kann es auch passieren, dass ein Ereignis nicht mehr benötigt wird, z.B. , wenn es abgearbeitet wurde. Der intuitive Ansatz wäre, das Ereignis aus der später erklärten *PriorityQueue* zu entfernen. Das ist aber rechenaufwändig und nicht nötig. Schließlich wird das entfernte Ereignis früher oder später in der *PriorityQueue* sehr effizient gefunden werden, nämlich wenn die *process* - Methode zum Zeitpunkt der eigentlichen Abarbeitung des entfernten Ereignisses aufgerufen wird. An der Einsortierung in der PriorityQueue wird also nichts verändert, stattdessen wird im Ereignis in der Variable *armed* vermerkt, dass es nicht mehr ausgeführt werden soll und es wird aus *events* entfernt. Wegen einer älteren Version des Programms, in der nur das aktuellste Ereignis zu einem

ActionObject in die *PriorityQueue* einsortiert war, um effizienter zu arbeiten, wird letztendlich noch das nun aktuellste Ereignis zu dem *ActionObject* der *PriorityQueue* hinzugefügt, falls dies noch nicht geschehen ist.

2.4 Steuerung

Dies ist die Klasse, die die gesamte Simulation steuert. In ihr steht die *main* - Methode, sie erzeugt neue *Staaten* und organisiert den (zeitlich) richtigen Ablauf der Ereignisse. Alle Ereignisse einer Simulation sind in der statischen *PriorityQueue events* abgespeichert. Der Routineablauf für das Abarbeiten der Ereignisse ist Folgender:

```
while (time < yearsToSimulate) {
        Event currentEvent = events.poll();//aktuellstes Ereignis aus events entfernen
        if (currentEvent.armed) {//wenn currentEvent nicht deaktiviert wurde
                time = currentEvent.getSortValue();//Zeit auf den Zeitpunkt des Events setzen
                currentEvent.process();//Ereignis abarbeiten
        }
}
```

2.5 PriorityQueue

Die Problemstellung ist jetzt, die Ereignisse in der korrekten Reihenfolge abzuarbeiten. Dabei stellt sich die Frage, in was für einer Datenstruktur die Ereignisse am besten abgespeichert werden. Zu jedem Element, das in der Datenstruktur abgespeichert werden soll, gibt es eine Gleitkommazahl *sortValue*, der mit der Methode *getSortValue* abgefragt werden kann. Dies ist in dem Fall dieses Projekts der Zeitpunkt, zu dem das Ereignis abgearbeitet werden soll. Die Datenstruktur muss dann folgende Operationen beherrschen:

- Bestimmen und entfernen des Elements mit dem kleinsten *sortValue*
- Hinzufügen eines Elements während der Laufzeit

Wegen dieser Anforderungen nennt man die Datenstruktur auch *PriorityQueue*. Nun soll zu Standardimplementierungen untersucht werden, mit welcher durchschnittlichen Laufzeitklasse sie die Operation ausführen.

Datenstruktur	Bestimmen des Elements mit dem kleinsten *sortValue*		Entfernen des Elements mit dem kleinsten *sortValue*		Hinzufügen eines Elements	
	Laufzeitklasse	Erklärung	Laufzeitklasse	Erklärung	Laufzeitklasse	Erklärung
Unsortiertes Array	O(n)	Es muss über das gesamte Array iteriert werden, um das Element mit	O(n) / O(1)	Wenn das Element wirklich entfernt werden soll, muss das gesamte Array kopiert werden, um	O(n) / O(1)	Ohne weitere Optimierungen muss das gesamte Array kopiert werden, um es um eins zu verlängern. Allerdings

		dem minimalen *sortValue* zu bestimmen.		es um eins zu verkleinern. Wird allerdings nur der sortValue eines Elements verändert, ist der Aufwand konstant, weil keine Umsortierung stattfinden muss. Dies wäre bei diesem Projekt meistens der Fall.		kann das Array immer länger gewählt werden, als es eigentlich notwendig wäre. Beispielsweise kann seine Länge immer verdoppelt werden, wenn die maximale Länge überschritten wird. Wenn diese Verdopplung k mal stattgefunden hat, fasst das Array 2^k Elemente. Der Aufwand dafür war 2^0+2^1+...+2^(k-1)=2^k-1. Die Laufzeitklasse für das Verlängern des Arrays ist also O((2^k-1)/2^k)=O(1). Dazu kommt noch ein Aufwand von O(1) für das Einfügen jedes einzelnen Elements, wenn das Array bereits eine ausreichende Länge hat.
Sortiertes Array	O(1)	Es muss nur das erste Element des Arrays zurückgegeben werden	O(n) / O(1)	Bei der Standardimplementierung muss das Array bei dieser Operation komplett kopiert werden. Allerdings kann auch abgespeichert werden, dass das Array erst ein Element später wirklich beginnt.	O(n)	Die Stelle, an der das hinzuzufügende Element einzufügen ist, lässt sich zwar mithilfe einer binären Suche effizient in einer worst-case-Laufzeit von O(log2(n)) bestimme, allerdings muss das Array im nächsten Schritt komplett kopiert werden, um es um ein Element zu erweitern.
Sortierte *LinkedList*	O(1)	getFirst() hat eine konstante Laufzeit	O(1)	Es muss lediglich eine einzige Referenz verändert werden.	O(n)	Um die Stelle zu bestimmen, an der das neue Element eingefügt werden soll, muss wegen des fehlenden direkten Zugriffs so lange über die Liste von Beginn an iteriert werden, bis die Stelle erreicht ist, an der das Element eingefügt

						werden soll. Dafür sind im Schnitt n/2 Iterationen nötig. Das Element an der gefundenen Stelle einzufügen geschieht jedoch in konstanter Laufzeit.

All diese Lösungen haben in einem Punkt eine Laufzeitklasse von O(n) . Weil die Anzahl der Aufrufe dieser Methoden proportional zur Anzahl der Akteure ist, wäre also die gesamte Laufzeitklasse mindestens O(n^2), was die Simulation bei einer größeren Anzahl von Akteuren unbrauchbar machen würde. Deshalb wurde eine eigene *PriorityQueue* entwickelt.

2.5.1 Idee der Datenstruktur

Grundsätzlich scheint an einer sortierten *LinkedList* sinnvoll, dass jedes Element eine Referenz auf das mit dem nächstgrößeren *sortValue* speichert, weil die Liste so sehr schnell von vorne nach hinten abgearbeitet werden kann, wenn die Elemente erst einmal richtig einsortiert sind. Allerdings muss die Geschwindigkeit für das Einsortieren der Elemente verbessert werden. Die erste Idee dafür war, sich „Shortcuts“ zu möglichst gut über den Wertebereich der *sortValues* verteilten Elementen in einem Array abzuspeichern. In dem Array würde dann mit einer binären Suche das Element mit dem größten *sortValue* bestimmt, der kleiner ist als der des einzufügenden Elements. Dazu ist lediglich eine Laufzeit von O(log2(n)) nötig. Von diesem Element aus wird dann in der *LinkedList* so lange weiter iteriert, bis die richtige Position zum Einfügen gefunden ist. Für den letzten Schritt sind durchschnittlich konstant viele Schritte nötig, weil sich ein konstanter Anteil der Elemente der Datenstruktur im Array befindet.
Praktisch gestaltet sich das Auswählen der Elemente für das Array allerdings komplizierter, als es zunächst wirkt. Die Anzahl dieser Elemente sollte optimaler Weise zu Beginn festgelegt werden. Immer, wenn eines von ihnen aus der Datenstruktur entfernt wird, muss es auch aus dem Array entfernt werden. Wird dann einfach das nächste in die *PriorityQueue* eingefügte Element in das Array aufgenommen, dürfte es zwar statistisch verteilt über den Wertebereich der *sortValues* ihrer Elemente verteilt sein, allerdings wird das sortierte Einfügen in das Array aufwändig, weil alle Elemente hinter der „Einfügeposition“ um eins verschoben werden müssen. Um dem zu begegnen, kann so lange mit dem Einfügen eines neuen Elements ins Array gewartet werden, bis ein Element in die Datenstruktur eingefügt wird, dessen *sortValue* größer als der größte *sortValue* eines Elements des Arrays ist. In der praktischen Umsetzung werden bei diesem Verfahren aber mehr Positionen im Array frei, als neue gefunden werden. Das macht es unbrauchbar.
Der nächste und letztendlich verwendete Ansatz ist, dass die Elemente der Datenstruktur nicht nur eine Referenz auf das Element mit dem nächstgrößeren *sortValue* abspeichern, sondern teilweise auch auf solche mit einem deutlich größeren sortValue. Die Referenzen sind in mehreren *Ebenen*

(synonym verwendet mit *Schicht*, *Layer*) organisiert. Die Referenzen in der nullten *Ebene* zeigen jeweils auf das Element mit dem nächstgrößeren *sortValue*. Die Elemente der restlichen *Schichten* haben folgende Anforderungen:

- Jede *Schicht* enthält nur Elemente der darunter liegenden *Schicht*.
- Alle Elemente einer *Schicht* haben eine Referenz auf das Element aus dieser *Schicht* mit dem nächstgrößeren *sortValue*.
- Es gibt zwei Elemente namens *firstSortable* und *lastSortable*, die an Anfang b.z.W. Ende aller *Schichten* stehen, sie sind deshalb auf allen Schichten einsortiert. *firstSortable* hat einen sortValue von -unendlich und *lastSortable* einen *sortValue* von unendlich. Sie erfüllen den Zweck, dass alle einsortierte Elemente einen *sortValue* haben, der zwischen den beiden liegt.

Nun soll das genaue Vorgehen bei den nötigen Operationen beschrieben werden:

2.5.2 Elemente hinzufügen –add(Object toAdd)

Die Idee dieser Datenstruktur ist, dass beim Einsortieren von Elementen zunächst auf höheren *Schichten* schnell viele Elemente überwunden werden können und so auf niedrigen *Schichten* nur noch wenige Schritte gegangen werden müssen. Ausgegangen wird von einem festen Startpunkt, dem *firstSortable, das* einen minimalen *sortValue* hat, weshalb *toAdd* mit Sicherheit dahinter einsortiert werden muss, und ist in alle vorhandenen *Schichten* einsortiert ist. Eine Visualisierung des Algorithmus‘ ist in *Abbildung 4* gegeben. Das genaue Vorgehen ist Folgendes:

1. **Bestimmung des maximalen *Levels*, auf dem *toAdd* einsortiert werden soll.** Dies ist ein stochastischer Vorgang. Dabei liefert die Methode getSteps(int level) , jedes wievielte Element, das auf dem *Level level* einsortiert wird, auch auf dem *Level level*+1 einsortiert werden soll. Um das zu erreichen wird vorgegangen wie beim Inkrementieren einer Zahl in einem Zahlensystem. Die Stellen dieser Zahl werden im Array *counter* abgespeichert. Das hinzuzufügende Element wird auf dem Level eingefügt, das der höchsten Stelle des counters entspricht, die sich bei seiner Inkrementierung verändert. Wäre der counter eine Zahl im Dezimalsystem, würde sich offenbar bei jeder Inkrementierung die 10^0-er-Stelle verändern und bei jeder zehnten Veränderung der n-ten Stelle würde sich auch die n+1-te Stelle verändern. Nun soll sich aber bei jeder *getSteps*(n)-ten Veränderung der *n*-ten Stelle des counters der Wert der n+1-ten Stelle des counters verändern. Dies wird offenbar dadurch erreicht, dass der *counter* in einem Zahlensystem steht, dessen 0-te Stelle den Wert eins und dessen n-te Stelle den Wert getSteps(n-1) hat. Das wird durch folgendes Vorgehen realisiert:
   I. Das Level level, auf dem das hinzuzufügende Element einsortiert werden soll, wird auf null gesetzt.
   II. Die nullte Stelle des Counters wird inkrementiert.

III. Wenn die *level*-te Stelle des counters größer als der maximal zulässige Wert *getSteps*(level) ist, wird die *level*-te Stelle auf null gesetzt, die *level*+1-te Stelle des counters um eins erhöht, falls sie vorhanden ist, und level inkrementiert. Dieser Schritt wird wiederholt, bis die Bedingungen zu Beginn dieses Schritts nicht mehr gelten.

2. Tatsächliches Einsortieren *toAdds* in die Datenstruktur

I. **Initialisierung der Startposition**: Die Variable *testSortable*, in der die „aktuelle Position in der Datenstruktur“ abgespeichert wird, wird auf den *firstSortable* initialisiert.

II. Die Variable *startLevel*, die abspeichert, von welcher Schicht aus begonnen werden soll, die Stelle zu suchen, an der das hinzuzufügende Element einsortiert werden soll, wird auf das größte vorhandene Level initialisiert.

III. **Abbrechen, wenn toAdd auf allen benötigten Schichten einsortiert ist:** Wenn *level* kleiner als null ist, wird abgebrochen.

IV. **Navigation zur Stelle in der Schicht *level,* an der *toAdd* einsortiert werden soll:** *testSortable* wird auf das Element der Datenstruktur gesetzt, das auf dem Level *level* einsortiert ist und dessen *sortValue* der größte der Elemente auf dem Level *level* ist, der kleiner als der des einzusortierenden Elements ist. Dies geschieht durch die Methode *goToPoint.* Bildlich gesprochen wird durch sie so lange auf jeder Schicht vorwärts gegangen, wie der *sortValue* der aktuellen Position kleiner gleich *toGet* ist. Dann wird die Schrittweite verkleinert, indem der Level, auf dem vorangeschritten wird, um eins verringert wird. Dies wird solange wiederholt, bis die Zielschicht erreicht ist. Konkret folgt daraus das Folgende Vorgehen:

i. Übergeben werden das einzusortierende Element *einzusortieren*, das Element, von dem aus der passende Punkt zum Einsortieren gesucht werden soll, *start*, die Schicht, auf der das Element einsortiert werden soll, *level* und die Schicht, von der aus mit der Einsortierung begonnen werden soll, *startLevel*.

ii. In der Variable *toGet* wird der sortValue des einzusortierenden Elements abgespeichert.

iii. **Abbrechen, wenn Zielschicht erreicht ist,** also wenn *startLevel* < *level* ist.

iv. **Auf aktueller Schicht so weit wie möglich gehen:** Es wird das Element aus der Schicht *startLevel* bestimmt, dessen *sortValue* der größte ist, der kleiner als *toGet* ist.

a. Die Variable *nextSortable* wird auf das nächste Element von *start* aus, das auf der Schicht *startLevel* einsortiert ist, initialisiert.

b. Wenn der sortValue von *nextSortable* größer als *toGet* ist, wird abgebrochen. Bei gleichem s*ortValue* werden stehen später einsortierte Elemente also weiter hinten.

c. Es wird „einen Schritt in der Schicht *level* weitergegangen“, indem *start* auf *nextSortable* und *nextSortable* auf auf den Nachfolger von *nextSortable* auf der Schicht *startLevel* gesetzt wird
d. Springe zu b.

v. In *start* steht jetzt das in iv gesuchte Element.
vi. **Eine Schicht nach unten gehen und Vorgehen wiederholen:** *startLevel* wird um eins verringert und zu iii. gesprungen.

V. Einsortieren *toAdd*s an der gefundenen Stelle *testSortable* auf der Schicht *level:*

i. Der Nachfolger von toAdd in der Schicht *level* wird zu dem von *testSortable*.
ii. Der Nachfolger von *testSortable* auf der Schicht *level* wird zu *toAdd*.

VI. **Eine Schicht nach unten springen:** *level* wird um eins verringert.

VII. *startLevel* wird auf *level* gesetzt, weil klar ist, dass auch im nächsten Schritt auf keiner höheren Schicht als level vorangegangen werden kann.

VIII. **Vorgehen auf neuer Schicht wiederholen:** Es wird zu III gesprungen.

Die getSteps-Methode

In dieser Methode ist die Regelung implementiert, welcher Anteil der Elemente auf welchen Ebenen einsortiert wird. Dafür wird betrachtet, wie viele Schritte auf dem nächst tieferen Level im Schnitt gemacht werden müssen, um das Element dort einzusortieren. Sind es zu viele Schritte, müssen mehr Elemente in die nächste Schicht einsortiert werden.

Bestimmung der durchschnittlich nötigen Schrittzahlen auf den Schichten

Sowohl zu Analysezwecken als auch zur Regelung mancher Parameter kann es nötig sein, zu wissen, wie viele Schritte beim Einsortieren eines Elements in die Datenstruktur auf einem bestimmten Level durchschnittlich nötig sind. Hierbei wird auf allen Schichten das Glättungsverfahren angewendet. Die Ergebnisse werden in der Liste *avStepsOnLevel* gesichert. Eine Aktualisierung der geglätteten Schrittwerte geschieht bei allen einzusortierenden Elementen, die auf einer Schicht größer gleich der Hälfte der vorhandenen Schichten einsortiert werden. So wird berücksichtigt, dass bei mehr einsortierten Elementen ein geringerer Anteil der hinzuzufügenden Elemente auf die Anzahl an benötigten Schritten untersucht werden muss, weil vermutlich weniger Fluktuationen der Schrittzahlen gegeben sind.

Hinzufügen von Schichten – die addLevel-Methode

Wenn in der obersten Schicht zu viele Schritte gemacht werden müssen, kann nicht einfach der Anteil der Elemente, die in die darüber liegende Schicht eingeordnet werden, erhöht werden, weil diese noch nicht existiert. Für diesen Fall existiert die *addLevel* - Methode. Sie fügt *firstSortable* eine Verknüpfung zu *lastSortable* auf einem neuen Level hinzu. Außerdem muss der *counter* um eins verlängert werden, weil auf einer weiteren Schicht mitgezählt werden muss, wann wieder ein

Element auf der nächsten Schicht einsortiert werden muss. Auch *avStepsOnLevel* muss um eins verlängert werden.

Laufzeitanalyse

Untersucht werden soll zunächst die average-case Laufzeitklasse.

- Wegen der Regelung des Anteils der Elemente, die auf bestimmten Schichten einsortiert werden, ist die benötigte Anzahl von Schritten auf einer bestimmten Schicht näherungsweise konstant.
- Die Anzahl *a* der Schichten hängt logarithmisch mit der Anzahl der Elemente in der Datenstruktur zusammen. Das ergibt sich dadurch, dass durch eine stochastische Verteilung der *sortValues* der einsortierten Elemente ein näherungsweise fester Anteil, der nun mit *c* bezeichnet sei, der Elemente der einer Schicht auch in der nächsten Schicht einsortiert werden müssen, um eine konstante Anzahl an Schritten auf der Schicht zu erreichen. Beispielsweise würde bei einer optimalen Verteilung der *sortValues* und einem Wert von *c*=1/2 auf jeder Schicht ein Vergleich zum Einsortieren nötig sein. Dabei gäbe es bei *n* einsortierten Elementen in der *a*-ten Schicht *n*(1/2)^a* Elemente. Es werden nur so viele Schichten erzeugt, dass in der letzten Schicht nicht leer ist. Da die Anzahl der Elemente einer Schicht ganzzahlig ist, gilt *n*(1/2)^a>=1 => a<=log2(n)*. Die Verteilung der Elemente ist zwar nicht so perfekt wie in diesem Beispiel, allerdings belegen Messwerte und Logik, dass die Elemente auf allen Ebenen „gleich gut verteilt“ sind; es sind also mehr als die *log2(n)* Schichten notwendig, aber der Zusammenhang ist trotzdem logarithmisch.

Demnach ist die average-case - Laufzeitklasse zum Einsortieren in die Datenstruktur O(k*log2(n))=O(log(n)).

Nun soll der worst-case analysiert werden. Dieser könnte beispielsweise folgendermaßen entstehen:

Durch einen unglücklichen Zufall hat jedes Element, das nur auf Level null einzusortieren ist, einen größeren *sortValue* als alle zuvor einsortierten Elemente und alle auf einem Level größer als null einzusortierenden Elemente einen kleineren *sortValue* als alle bisher einsortierten Elemente. Mit diesen Voraussetzungen werden *n* Elemente in die Datenstruktur eingefügt. Der Anteil der nur auf Level null einsortierten Elemente ist bei jeder Anzahl von einsortierten Elementen mindestens eine Konstante c. Der Aufwand für das Einsortieren des nächsten Elements ist dann mindestens: O(c*(n*c))=O(n). Es wird ein Anteil von c der Elemente ganz hinten einsortiert. Alleine für diese Elemente sind mindestens n*c Schritte zum Einsortieren nötig, weil alle Elemente, die nur auf der untersten Schicht einsortiert sind, überwunden werden müssen.

Andererseits kann die Laufzeitklasse für das Einsortieren von Elementen auch auf maximal O(n) eingegrenzt werden, weil maximal n Schritte nötig sind, da alle n Elemente in der Datenstruktur im nullten Level verknüpft sind.

Es sollte allerdings erwähnt werden, dass der worst-case extrem unwahrscheinlich ist. Das liegt daran, dass hierfür eine Abhängigkeit zwischen dem *sortValue* und der Schicht, in die ein Element einsortiert wird, bestehen müsste. Dies ist nicht einmal gezielt problemlos erreichbar, schließlich wird die „Frequenz“, in der Elemente auf bestimmten Schichten einsortiert werden, während der Laufzeit angepasst. Sich darauf abzustimmen ist nahezu unmöglich.
Wenn kein stochastischer Zusammenhang zwischen sortValue und der Zielschicht besteht, sind die Elemente aus höheren Schichten repräsentativ für alle Elemente der Datenstruktur verteilt. Somit sind dann durchschnittlich auf höheren Schichten entsprechend mehr Elemente überwindbar und die Argumentation für die logarithmische Laufzeit bei der Einsortierung greift wieder.

2.5.3 Das erste Element bestimmen und entfernen - die poll - Methode

Nun zahlt sich die komplizierte Einsortierung aus. Das Bestimmen und Entfernen des ersten Elements ist nämlich mit der geleisteten Vorarbeit sehr einfach und effizient:

Funktionsweise

1. Das erste Element der Datenstruktur ist offenbar der Nachfolger von *firstSortable* in der nullten Schicht und lässt sich damit direkt bestimmen.
2. Nun müssen noch einige Verknüpfungen verändert werden, um das Element zu entfernen: Die Verknüpfungen von allen Schichten, auf denen das zu entfernende Element einsortiert war, müssen nun von *firstSortable* übernommen werden.

Laufzeitanalyse

Der erste Schritt benötigt eine konstante Laufzeit.
Der zweite Schritt wird im average-case analysiert. Hierzu wird wieder angenommen, dass jedes *(1/c)^a*-te Element in der *a*-ten Schicht einsortiert ist. Es erfordert offenbar konstante Laufzeit *k*, ein Element aus einer Schicht zu entfernen. Das liefert für unendlich viele Schichten durch das Aufsummieren der Laufzeiten, die in den einzelnen Schichten im Mittel benötigt werden, die durchschnittliche Laufzeit:

$$\lim_{h\to\infty}\sum_{a=0}^{h} k*\left(\frac{1}{c}\right)^{a} = k*\lim_{h\to\infty}\sum_{a=0}^{h}\left(\frac{1}{c}\right)^{a} = k*\lim_{h\to\infty}\frac{\left(\frac{1}{c}\right)^{h+1}-1}{\frac{1}{c}-1} = k*\frac{-1}{\frac{1}{c}-1} = \frac{-k}{\frac{1-c}{c}} = \frac{k*c}{c-1}$$

Die Laufzeit erreicht also einen Grenzwert von $\frac{k*c}{c-1}$. Damit ist die Laufzeitklasse konstant.

Beim Beweis wurde sich einer Lemmas bedient, das nun bewiesen werden soll:

$$\sum_{i=0}^{h} b^i = \frac{b^{h+1}-1}{b-1} \Leftrightarrow \sum_{i=0}^{h} (b-1)*b^i = b^{h+1}-1 \Leftrightarrow \sum_{i=0}^{h} b^{i+1}-b^i = b^{h+1}-1 \Leftrightarrow \sum_{i=0}^{h} b^{i+1} - \sum_{i=0}^{h} b^i = b^{h+1}-1$$
$$\Leftrightarrow \sum_{i=1}^{h+1} b^i - \sum_{i=0}^{h} b^i = b^{h+1}-1 \Leftrightarrow b^{h+1}-1+\sum_{i=1}^{h} b^i - b^i = b^{h+1}-1$$

Es wurde dabei b=1/c eingesetzt.

Theoretisch kann die Laufzeitklasse im worst-case auf unendlich anwachsen, beispielsweise, wenn jedes Element auf jeder Schicht einsortiert ist und es so unendlich viele Schichten gibt (also im Fall, dass c=1 gilt). Dies wird jedoch von der Regelung des Einsortieralgorithmus‘ verhindert. In der Regel ist jedes zweite Element einer Schicht auch in der nächsten Schicht einsortiert.

2.5.4 Abkapselung und Implementierung

Im praktischen Gebrauch ist es unpraktikabel, die für die internen Vorgänge der PriorityQueue notwendigen Daten (z.B. die nächsten Elemente auf verschiedenen Ebenen) direkt in den einzusortierenden Elementen abzuspeichern. Zwar könnte man nur Elemente zulassen, die von einer Klasse erben, die diese Daten abspeichert, dann könnten die Elemente allerdings von keiner anderen Klasse mehr erben. Auch eine Implementierung als Interface löst das Problem nicht elegant, weil für die notwendigen Methoden Attribute notwendig sind, die bei jeder Klasse einzeln hinzugefügt werden müssten. Deshalb werden diese Eigenschaften auf ein separates Objekt, einen *SortableContainer* ausgelagert. Die einzusortierende Klasse muss lediglich ein Interface namens *SortValueGetter* implementieren, mit dem der *sortValue* eines einzusortierenden Objekts bestimmt werden kann. Beim Einsortieren eines Objekts in die Datenstruktur wird ein *SortableContainer* mit dem *sortValue* des Objekts erzeugt und dieser Container einsortiert. Jeder *SortableContainer* hat eine Referenz auf das zugehörige Objekt. So kann bei der *poll* – Methode der nächste *SortableContainer* bestimmte und das zugehörige Objekt ausgegeben werden.

2.5.5 Auswertung der faktischen Laufzeit im Vergleich mit der Implementierung von Java

In Java gibt es bereits eine Implementierung einer *PriorityQueue*. Ihre Laufzeit zum Einfügen wird in der offiziellen Dokumentation [3] mit O(log(n)) und zum Entfernen des ersten Elements mit O(1) angegeben. Diese Daten wurden in einem praktischen Test überprüft (siehe *Abbildung 3*). Dabei ist zu beachten, dass die Einwohnerzahl und damit auch die Anzahl der einsortierten Ereignisse näherungsweise exponentiell mit der Simulationszeit zusammenhängt. Bei der Java-Implementierung wächst die Laufzeit pro Person und Jahr näherungsweise linear mit der simulierten Zeit, also logarithmisch mit der Anzahl der einsortierten Elemente. Bei meiner *PriorityQueue* bleibt dieser Wert hingegen ab circa 300 Jahren konstant. In dieser Anwendung ist sie der Java – Variante also deutlich überlegen. Das könnte daran liegen, dass sie signifikant von Mustern profitiert, die sich in der Verteilung der s*ortValues* wiederfinden, weil durch den

stochastischen Prozess des Einsortierens mehr Verbindungen an Stellen mit häufig vorkommenden s*ortValues* gesetzt werden. So wird offenbar faktisch eine Laufzeit von toAdd von O(1) erreicht.

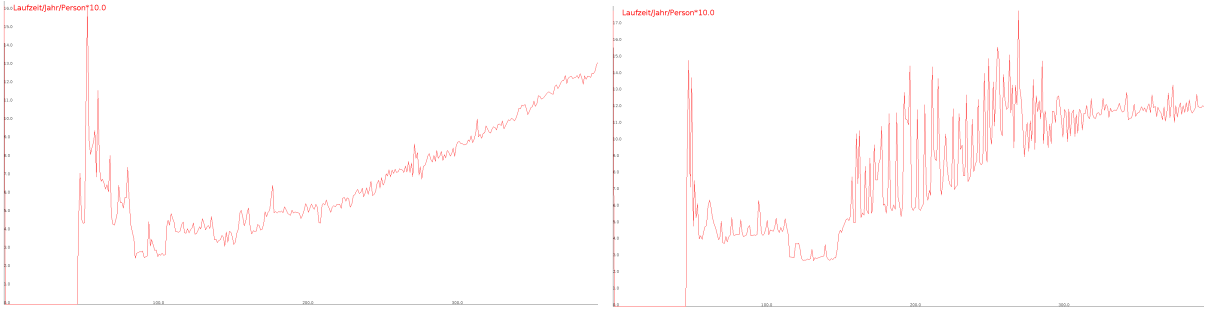


Abbildung 5: Laufzeit pro Einwohner pro simuliertem Jahr über die Simulationszeit aufgetragen bei der gleichen Simulation mit der Java - PriorityQueue (links) und meiner eigenen PriorityQueue (rechts). Auch bei mehrmaligen Simulationen sehen die Ergebnisse ähnlich aus.

3 Implementierung der Verhaltensregeln der Objekte

Bisher wurde die Struktur vorgestellt, mit der die vorhandenen Objekte ihre grundsätzliche Funktionalität und Parameter erhalten. Wann bzw. nach welchen Regeln diese Funktionen aufgerufen werden, soll nun geklärt werden.

3.1 Grundsätzliches Prinzip

Akteure implementieren ihre Verhaltensregeln durch die Ereignisse, die ihnen hinzufügt werden. Das hat den Vorteil einfacher Erweiterbarkeit und Modularität: Um neue Verhaltensmuster zu implementieren, muss dem zugehörigen Akteur bei seiner Erzeugung lediglich eine Instanz einer neuen Klasse von Ereignis in *events* hinzugefügt werden.

3.2 Wiederholte Ereignisse

Eine häufig auftretende Problemstellung ist, eine Handlung wiederholt auszuführen. Um dies nicht jedes Mal neu implementieren zu müssen, gibt es das abstrakte *SubjectEvent ExecuteRegularAction*. Die Idee ist, in der *finish* - Methode des Ereignisses den Zeitpunkt des Ereignisses um einen bestimmten Wert namens *period* nach hinten zu verschieben und es dem Subjekt direkt wieder hinzuzufügen. So wird das Ereignis in Intervallen von *period* ausgeführt, ohne es mehrmals abspeichern zu müssen. Dabei muss beachtet werden, dass ein Ereignis nicht häufiger ausgeführt werden kann, als seine Dauer erlaubt. Dies wird dadurch gelöst, dass *ActionObjects* zu einem Zeitpunkt *t* keine Ereignisse hinzugefügt werden können, die zu einem Zeitpunkt kleiner als t beginnen.
Um die Handlung abzubrechen, muss lediglich die Variable *stopAction* auf den Wert *true* gesetzt werden. Dann wird das Ereignis nicht erneut hinzugefügt.

3.3 Beispiele für Ereignisse

3.3.1 Actor

RegularPayment

Dies ist das erste Beispiel eines Ereignisses, das von *ExecuteRegularAction* erbt, anhand dessen die Funktionsweise regelmäßiger Handlungen mit Hilfe der abstrakten Klasse erläutert werden soll. Der Betrag *value* wird in Zeitabständen von *period* über eine Dauer von *duration* an die Zielperson *goal* gegeben. Durch das Aufrufen des super-Konstruktors wird die *act* - Methode von nun an zu den gewünschten Zeitpunkten aufgerufen. Der Vorteil dieses Prinzips ist eine Abkapselung und ein geringer Programmieraufwand für regelmäßige Ereignisse. Der benötigte Code für *RegularPayment* ist beispielsweise lediglich der Folgende:

```
public class RegularPayment extends ExecuteRegularAction<Actor> {
	Actor goal;
	double amount;
	double expirationDate;
	public RegularPayment(Actor subject, Actor goal, double amount, double period, double
duration) {
		super(subject,period);
		this.goal=goal;
		this.amount=amount;
		this.expirationDate=duration+Steuerung.getTime();
		this.date=Steuerung.getTime();
	}
	public void act() {
		subject.giveMoney(amount,goal);
		stopAction=date>expirationDate;
	}
}
```

UpdateAvCostEinnahmen

Bei jedem *Actor* soll jeden Monat die *updateAvCostAndEinnahmen* - Methode ausgeführt werden, wofür dieses Ereignis existiert. Jeder Actor hat ein Objekt der Klasse *UpdateAvCostEinnahmen* als Attribut, damit bei seiner Erstellung dessen Konstruktor aufgerufen wird, in welchem das Ereignis dem *Actor* hinzugefügt wird. So muss sich bei keinem *Actor* mehr darum gekümmert werden, die beiden Parameter zu aktualisieren.

3.3.2 Privatperson

Arbeit

Auch dieses Ereignis muss regelmäßig, nämlich jeden Tag, ausgeführt werden, weshalb es von *ExecuteRegularAction* erbt. Es wird von einer täglichen Arbeitszeit von acht Stunden ausgegangen, weshalb die Länge des Ereignisses dieser Zeitraum ist. Weil es aktiv ist, wird die *addingFailed* -

Methode benötigt. In diesem Fall wird das Datum um einen Tag erhöht und wieder versucht, das Ereignis hinzuzufügen.
Beim Beginn der Arbeit muss die Person zunächst zu seinem Arbeitsplatz gehen, um dann die *startWorking* - Methode ausführen zu können. Zum Beenden der Arbeit wird die *stopWorking* - Methode ausgeführt, die Zeitdauer, die das Subjekt gearbeitet hat, abgespeichert, und nach Hause gegangen.

SucheArbeitgeber

Dieses Ereignis erbt von *ExecuteRegularAction* mit einer Periodendauer von einem Jahr. Beim Abarbeiten des Ereignisses wird für das Subjekt von seinem Staat eine Liste mit Jobangeboten angefordert und diese in *options* abgespeichert. Dann wird mittels der *evaluateJob* - Methode des Subjekts das beste Angebot aus dieser Liste bestimmt. Wenn die Bewertung dieses Angebots mehr als eine Konstante besser ist als die des aktuellen Job der Person, nimmt die Person es an. Die Konstante dient dazu, den Aufwand eines Jobwechsels zu simulieren.

GetChildren

Das Datum dieses Ereignisses, der Geburt eines Kindes, wird auf einen zufälligen Wert zwischen 18 und 48 Jahren nach der Geburt des Subjekts gesetzt. Beim Abarbeiten wird eine neue Person mit *subject* als Elternteil erzeugt. Das Kind wird der Liste von Kindern des Subjekts und den Einwohner seines Staats hinzugefügt. Das Subjekt muss ab dann mittels des Ereignisses *RegularPayment* 18 Jahre lang ein monatliches Taschengeld an das Kind zahlen und hat zusätzliche Ausgaben für das neue Kind in der Höhe seines Einkommens/(3+Anzahl seiner Kinder).
Der *skillFactor* des Kindes wird zu einem zufälligen Anteil von seinem Erzeuger geerbt und der restliche Anteil wird ebenfalls zufällig bestimmt.
Jede Privatperson bekommt eine zufällige Anzahl zwischen null und drei Kindern, diese Anzahl wird beim Erzeugen der Person bestimmt.

GruendeUnternehmen

Mit einer Wahrscheinlichkeit von 0.1 wird einer neu erstellten Privatperson das Ereignis *GruendeUnternehmen* hinzugefügt. Sein Datum wird zufällig zwischen 18 und 78 Jahren nach der Geburt gewählt. Beim Abarbeiten des Ereignisses wird ein neues Unternehmen mit dem Subjekt als Besitzer erstellt. Beim Erzeugen eines Unternehmens gibt der Besitzer dem Unternehmen ein Startkapital von 1.000.000. Das neue Unternehmen kauft sich ein neues *StandardProduktionsmittel* und fügt sich eine neue Instanz der Ereignisklassen *SellProducts* und *ManageUnternehmen* hinzu.

3.3.3 Unternehmen und Produktionsmittel

ManageUnternehmen

Dieses Ereignis wird jedes Jahr ausgeführt. Wenn das Unternehmen mehr als einen bestimmten Wert an Schulden hat, wird es aufgelöst.

SellProducts

Jedes Unternehmen verkauft durch dieses Ereignis regelmäßig die von seinen Produktionsmitteln produzierten Produkte an den Staat. Durch das Verkaufen an den Staat statt an Konsumenten wird das Problem der Produktwahl umgangen, wofür allerdings die Annahme nötig ist, dass jedes Produkt verkauft wird. In späteren Versionen können noch Verfahren wie Angebot und Nachfrage implementiert werden.

ManageProduktionsmittel

Dieses Ereignis wird jeden Monat aufgerufen. Es sorgt für die Bezahlung der Angestellten und erfüllt außerdem den Zweck, diese so anzupassen, dass das betreibende Unternehmen keinen Verlust macht, aber auch genug Angestellte vorhanden sind. Um die Anzahl der Angestellten zu regeln, kann zudem die *minQualification* angepasst werden. Dazu wird die *minQualification* für jeden durchschnittlich fehlenden Arbeiter um 1% verringert und der Stundenlohn um 0.01% erhöht. Analog wird beim Stundenlohn mit der Bilanz den Geldreserven des Betreibers vorgegangen. All diese Parameter sind extrem sensibel und wurden nach dem Prinzip „trial and error“ angepasst. Wenn zu viele Angestellte vorhanden sind, wird den Überschüssigen gekündigt und das Produktionsmittel aus den offenen Stellen seines Staats gestrichen, wenn zu wenige vorhanden sind, wird das Produktionsmittel als offene Stelle ausgeschrieben.

4 Analyse

4.1 Analysewerkzeuge

4.1.1 Funktionsweise

Analysewerkzeuge sind als Ereignisse implementiert. Zugrunde liegt die abstrakte Klasse *AnalysisTool*. Jedes *AnalysisTool* zeichnet in einem bestimmten Zeitraum *length* in einem bestimmten zeitlichen Abstand *period* einen Datenpunkt bestehend aus einem Zeitpunkt und einem zugehörigen Datenwert, der mittels der abstrakten Methode *getValue()* bestimmt wird. Diese werden in zwei Arrays namens *xValues* und *yValues* abgespeichert.
Die *drawGraph* - Methode fügt den Graphen des Tools den zu zeichnenden Graphen der Klasse *DisplayGraphics* hinzu, welche auch für das graphische Ausgeben der Graphen verantwortlich ist. Die Implementierung sieht folgendermaßen aus:

```
public abstract class AnalysisTool extends Event {
	double period;
	int numOfFrames;
	double[] xValues;
	double[] yValues;
	/**
	 * Anzahl der bis jetzt aufgezeichneten Frames
	 */
	int frameCounter=0;
	String name;//Name des aufzuzeichnenden Wertes
	public AnalysisTool(double length, double period, String name) {
		this.numOfFrames=(int) (length/period);
		this.period=period;
		this.date=Steuerung.getTime();
		this.name=name;
		xValues=new double[numOfFrames];
		yValues=new double[numOfFrames];
		Steuerung.events.add(this);
	}
	public void process() {
		if(frameCounter<numOfFrames) {
			xValues[frameCounter]=Steuerung.getTime();
			yValues[frameCounter]=getValue();
			frameCounter++;
			date+=period;
			Steuerung.events.add(this);
		}
	}
	public abstract double getValue();
	public Graph getGraph() {
		return new Graph(xValues,yValues,name);
	}
	public void drawGraph() {
		DisplayGraphics.graphs.add(getGraph());
	}
}
```

Hinweis: In der neuesten Version wurde der Quellcode erweitert, um schon während der Laufzeit Daten anzeigen zu können, was allerdings nichts an der Logik der Klasse ändert.

Beispiel

Wenn die Einwohnerzahl des Staates aufgezeichnet werden soll, sieht das zugehörige *AnalysisTool* beispielsweise so aus:

```
public class MonitorPopulation extends AnalysisTool {
	public MonitorPopulation(int numOfFrames, double period, String name) {
		super(numOfFrames, period, name);
	}
	public double getValue() {
		return ((double)Steuerung.meinStaat.einwohner.size());
	}
}
```

Auf diese Weise lassen sich ohne großen Programmier- oder Rechenaufwand Daten wie die Anzahl der vorhandenen Unternehmen, das monatliche BIP, das durchschnittliche monatliche Einkommen der Bevölkerung, die Arbeitslosigkeit, der durchschnittliche monatliche Umsatz oder die durchschnittliche monatliche Bilanz der Unternehmen aufzeichnen und visualisieren.

4.2 Analyseergebnisse

Nun sollen drei Simulationen unter Variation relevanter Parameter durchgeführt werden. Gemeinsam haben alle Simulationen eine Startpopulation von 20 Personen, eine Simulationsdauer von 400 Jahren sowie eine zufällige Anzahl zwischen null und drei Kindern pro Person. Zudem arbeiten alle Personen ab einem Alter von 18 Jahren acht Stunden am Tag, sofern sie nicht arbeitslos sind.

Da im aktuellen Stand weder die Geburtenrate noch das Sterbealter von der finanziellen Lage der Einwohner abhängt, wächst die Einwohnerzahl in allen Simulationen in einer exponentiellen Kurve auf ca. 2500 an.

4.2.1 Simulation 1

Als erstes soll untersucht werden, welche Art von Unternehmen sich zu welchen Zeitpunkten durchsetzen. Dazu wird die Regel implementiert, dass neu gegründete Unternehmen Produktionsmittel mit einer zufälligen Produktionskapazität zwischen null und 5000000 und einer dazu ungefähr proportionalen Anzahl an Arbeitern, für die es ausgelegt ist, kauft. Mittels eines weiteren Analysewerkzeugs wird dann der Durchschnitt diese Arbeiterzahl bei erfolgreichen Unternehmen aufgezeichnet. Als „erfolgreich“ gelten dabei Unternehmen mit einem Alter von mindestens zehn Jahren. Wegen unterschiedlicher Unternehmensgrößen muss außerdem die Menge an Schulden, ab der ein Unternehmen pleite ist, von seinem Umsatz abhängig gemacht werden. Das Ergebnis der Simulation zeigt, dass sich zu Beginn der Population (d.h. die ersten 120 Jahre) kleinere Unternehmen mit circa zwölf Mitarbeitern durchsetzen, was durch einen Mangel an Arbeitern begründet sein könnte. Auf Dauer pendelt sich der Wert bei circa 20 Mitarbeitern ein. Das Bruttoinlandsprodukt hat, wenn auch mit starken Schwankungen, eine exponentiell steigende Tendenz, weshalb das monatliche Durchschnittseinkommen der Bevölkerung fast konstant bei gut 2000 verharrt. Auf Dauer schaffen es die Unternehmen auch, ihre durchschnittliche Bilanz ungefähr bei null zu halten, was allerdings auch an ihre wachsenden Anzahl liegen könnte.

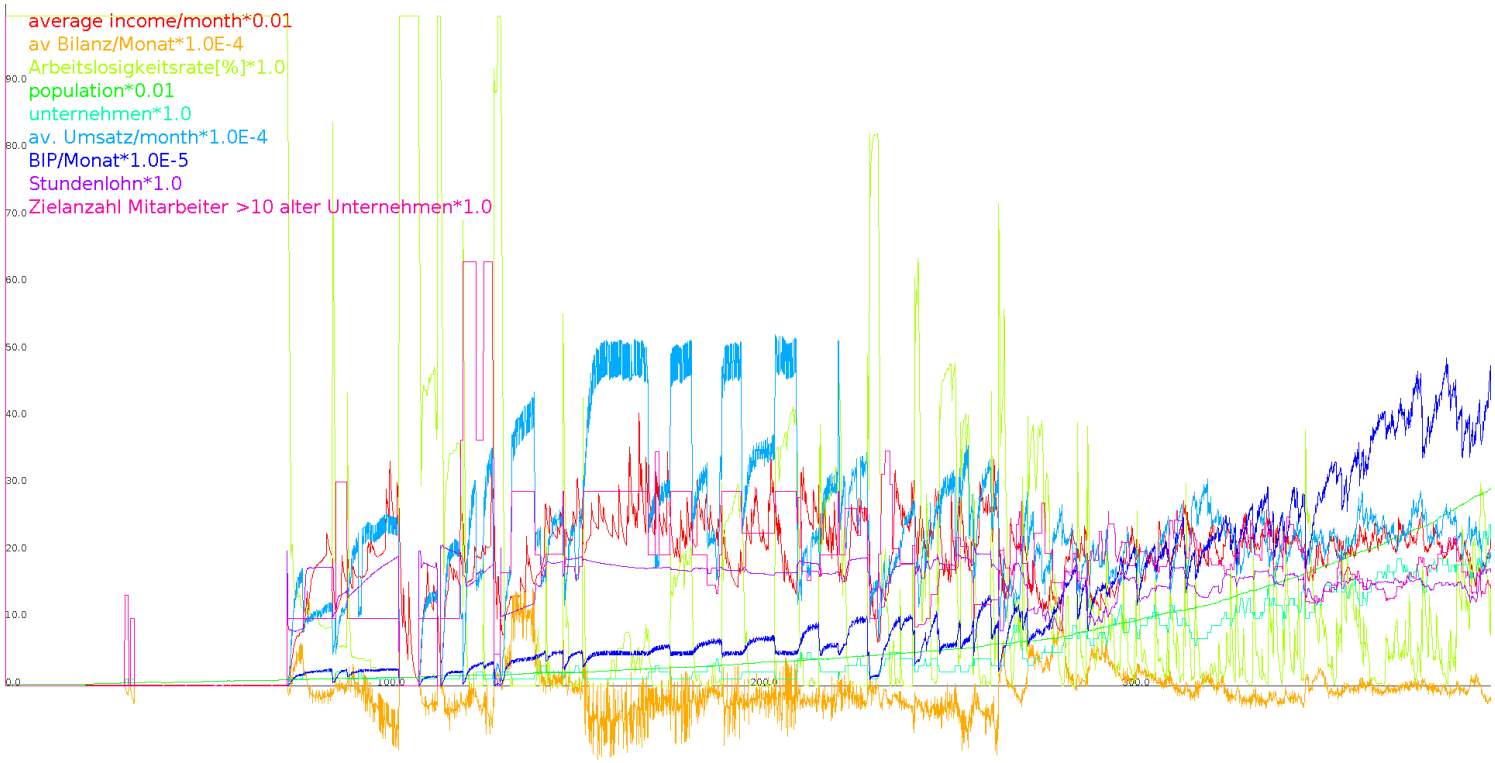


Abbildung 6: Visualisierung der Ergebnisse der ersten Simulation

4.2.2 Simulation 2

Nun stellt sich die Frage, was passiert, wenn Unternehmen deutlich höhere Kosten haben, beispielsweise durch Produktionsmittel mit mehr laufenden Kosten. Dazu werden diese von einem 200-tel des maximalen Produktionswert auf ein 80-tel angehoben.
In der Folge entstehen die ersten dauerhaften Unternehmen erst um die 20 Jahre später. Zudem liegt der durchschnittliche Stundenlohn bei recht konstant 10 statt 20. Die in der ersten Simulation am Ende fast vernachlässigbare Arbeitslosigkeit von circa 20% steigt hier auf fast 50% an. Das Durchschnittseinkommen halbiert sich beinahe mit sinkender Tendenz. Trotzdem bleibt die Bilanz der Unternehmen im Mittel ungefähr bei null.

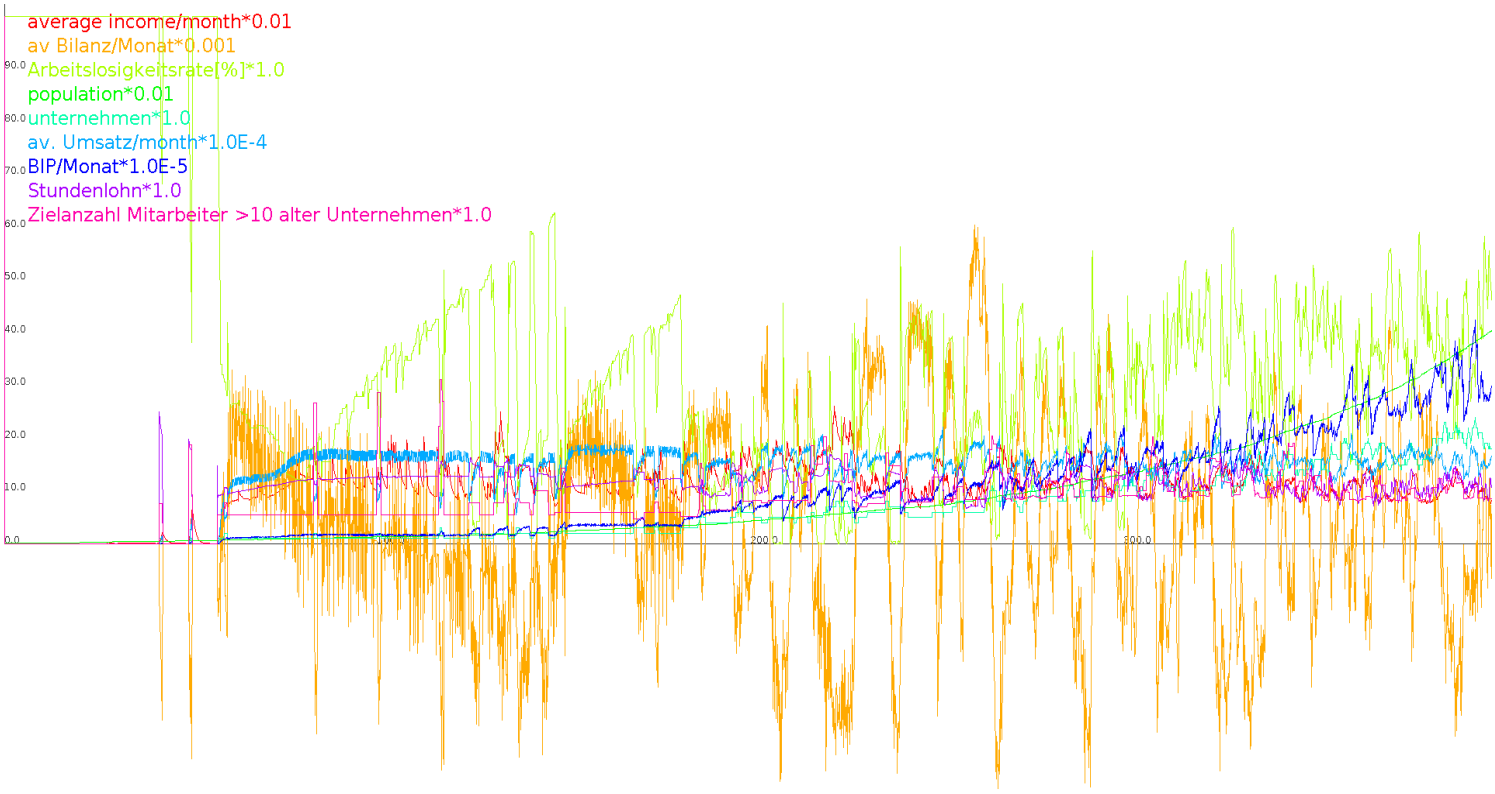


Abbildung 7: Visualisierung der Ergebnisse der zweiten Simulation

4.2.3 Simulation 3

Nun wird den Unternehmen nicht als Zielvorgabe gegeben, eine Bilanz von null, sondern einen Gewinn von 5000 zu erwirtschaften.

Sobald diese ihr Ziel erreichen (ab circa 200 Jahren) , ist der durchschnittliche Stundenlohn jedoch auf 10 und das Durchschnittseinkommen auf circa 1200 gefallen, die Werte halbieren sich also fast im Vergleich zu Simulation 1. Das Einkommen sinkt nicht ganz so stark wie der Lohn, weil die Arbeitslosigkeit auf circa 15% fällt. Dies könnte daran liegen, dass Unternehmen mit Gewinn unwahrscheinlicher pleite gehen und so mehr Arbeitsplätze vorhanden sind.

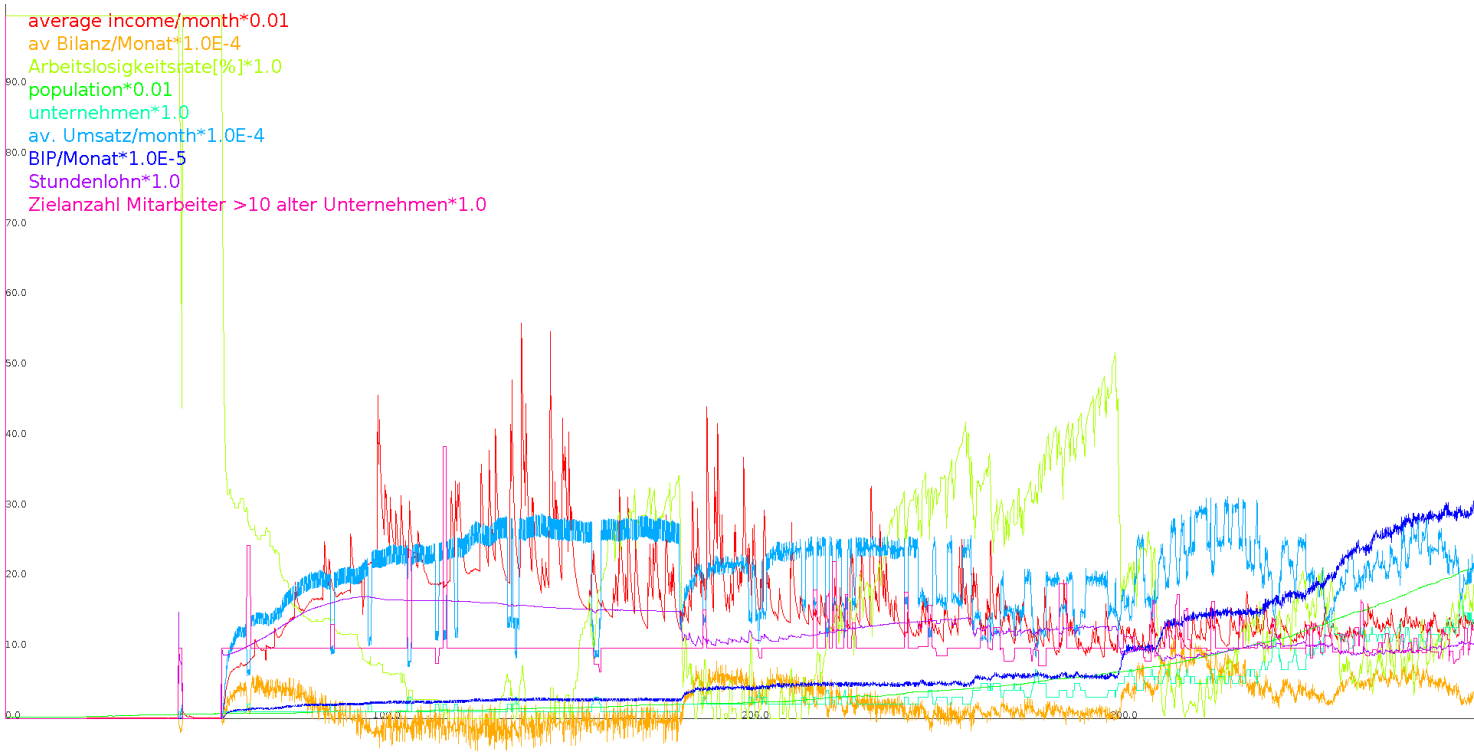


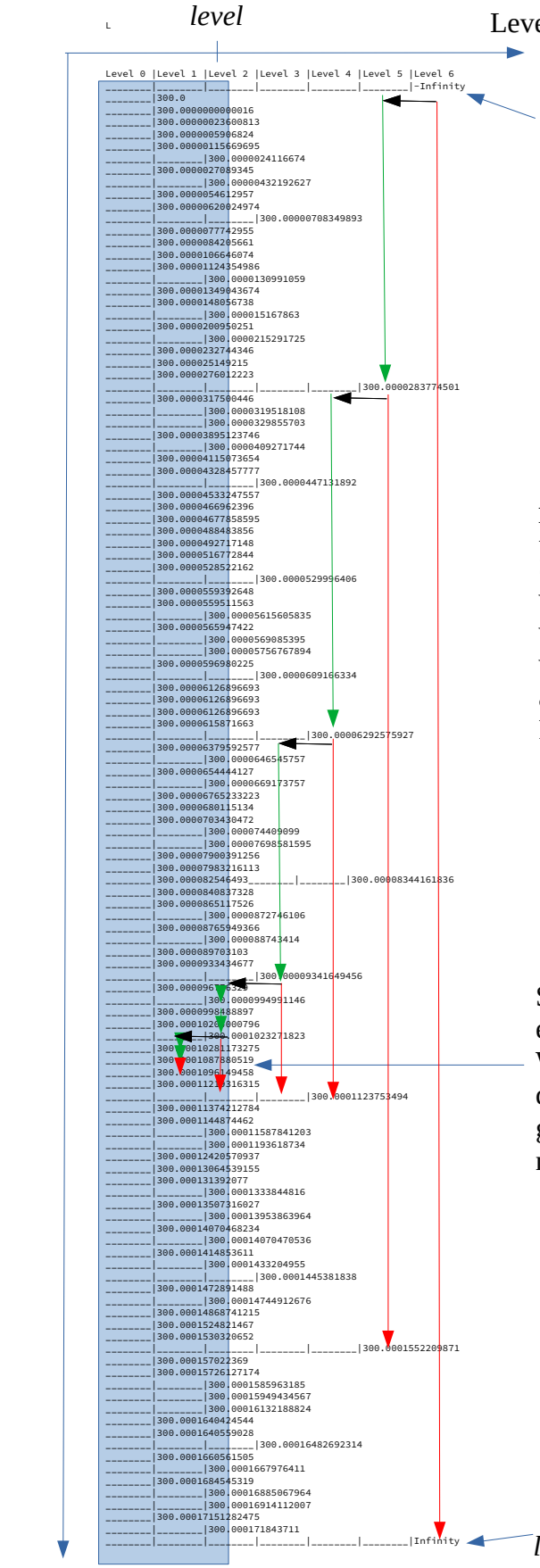
Abbildung 8: Visualisierung der Ergebnisse der dritten Simulation

5 Fazit und Ausblick

Auch wenn es unnötig komplex scheinen mag, zahlt sich ein solider Unterbau beim Schreiben von Simulationen aus. Durch die Ereignis-basierte Struktur mit einem Terminmanagementsystems lässt sich die Simulation einfach und effizient erweitern, was sich beispielsweise bei der Programmierung der Analysewerkzeuge gezeigt hat.
Auch die Implementierung erster Verhaltensregeln verläuft recht zielgerichtet und trotz ihrer Primitivität lassen sich schon einige interessante Effekte zeigen.
Um die Simulation tatsächlich zur Ausbildung von Managern oder zur besseren Planung von Unternehmen einsetzen zu können, müssen die Akteure jedoch noch deutlich genauer beschrieben werden. So sind bisher alle Parameter der Produktionsmittel aus der Luft gegriffen und beispielsweise die Struktur der Unternehmen ist kaum implementiert. Es werden noch keine Steuern simuliert und es besteht noch die Annahme, dass alle produzierten Produkte auch verkauft werden. Um praktisch nutzbare Ergebnisse zu erzielen, müssen diese Parameter in einer zukünftigen Version jedoch berücksichtigt werden.
Zudem besteht unsere reale Welt nicht aus tausenden, sondern aus Milliarden Akteuren. Dazu ist eine Steigerung der Effizienz beispielsweise durch Personen, die für mehrere Menschen stehen und eine starke Parallelisierung notwendig. Weil die Laufzeit proportional zur Anzahl der vorhandenen Akteure ist, scheint eine Realisierung der Erweiterungen durchaus praktikabel.
Mit einem solchen System würden sich dann ohne großen Aufwand oder erhebliches Risiko neue Strategien für Politik und Wirtschaft testen lassen. Allerdings muss stets beachtet werden, dass es sich bei solchen Simulationen stets nur um eine Vereinfachung der Realität handelt. Bevor ein Ergebnis verwendet wird, sollte es deshalb stets noch mit ausgereiften Analysewerkzeugen und gesundem Menschenverstand auf Plausibilität überprüft werden.

6 Quellen

- [1] https://www.bpb.de/nachschlagen/lexika/lexikon-der-wirtschaft/20918/unternehmen (20/06/2019)
- [2] https://www.bpb.de/nachschlagen/lexika/lexikon-der-wirtschaft/20365/produktionsmittel (20/06/2019)
- [3] https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html (17/06/2019)
- https://en.wikipedia.org/wiki/HSL_and_HSV (18/06/2019)
- [4] Abstract aus Schröder, Hans Peter & Ciucan-Rusu, Liviu. (2012): Business Simulation. https://www.researchgate.net/publication/280387309_Business_Simulation
- https://www.javatpoint.com/Graphics-in-swing (10/06/2019)



firstSortable (erster Wert von *testSortable)*

sortValue des einzusortierenden Elements *toAdd*: 300.0001088

Level *level*, auf dem toAdd einsortiert wird: 1

Alle roten Referenzen, die sich im blau hinterlegten Bereich befinden, müssen in Schritt 2.V auf *toAdd* umgelegt werden, weil *toAdd* auf diesen Schichten einsortiert wird, also „dazwischen geschoben“ werden muss. t*oAdds* Referenzen werden auf die ursprünglichen Ziele dieser Referenzen gesetzt.

Stelle, an der *toAdd* einsortiert werden soll. Wenn ein Schritt über diese Stelle hinaus ragt, geht er zu weit und darf nicht genommen werden

Legende:

Schritt, der zu weit führen würde (2.IV.iv.b)

Schritt, der gegangen wird (2.IV.iv.c)

Schritt ein Level nach unten (2.VI, wenn im blau markierten Bereich, sonst 2.IV.vi)

lastSortable

Abbildung 4: graphische Darstellung der add – Methode der Datenstruktur. Jede Zeile entspricht einem einsortierten Element mit dem angezeigten sortValue.

7 Erklärung über die selbstständige Anfertigung der Arbeit

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, alle aus anderen Werken wörtlich oder sinngemäß entnommenen Stellen und Abbildungen unter Angabe der Quelle als Entlehnung kenntlich gemacht und keine anderen Hilfsmittel als die angegebenen verwendet habe.

Mainz, den 21. Juni 2019

Ole Petersen