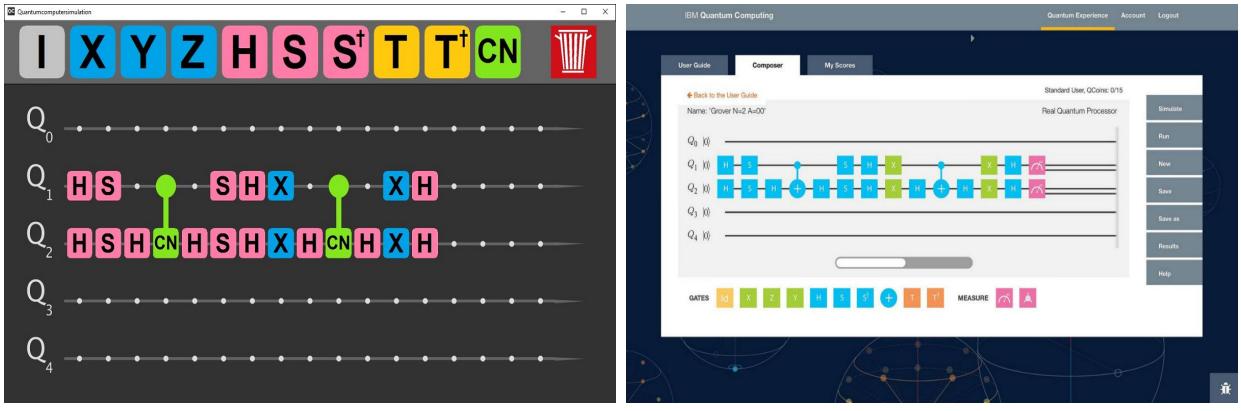
Kapitel 1

Quantencomputersimulation

1.1 Grundidee

Da jede bisherige Auseinandersetzung mit dem Konzept des Quantencomputers sehr theoretisch verlief, wurde die Simulation als praktischer Überblick über gegenwärtige und Ausblick auf zukünftige Fortschritte der Quantencomputerforschung verwendet. Die Inspiration für die Umsetzung dieses Zieles entstand durch das „Quantum-Experience“ Projekt des Unternehmens IBM.[1] Als eines der gegenwärtig führenden IT-Unternehmen investiert und forscht es in jede Richtung der Computertechnologie und ist unter anderem durch die Watson KI bekannt. 2016 stellte IBM ein weiteres Forschungsprojekt online: einen Quantencomputer. Jeder Nutzer kann über einen sogenannten Composer Gatter auf einen Gesamtqubitzustand anwenden. Der Composer ist hierbei die Schnittstelle zwischen dem Nutzer und dem Algorithmus und damit ein Teil der GUI. Man besitzt hierbei zwei Möglichkeiten zur Interaktion. Entweder man arbeitet mit einem echten Gesamtqubitzustand, welcher bei ca. 0.015 Kelvin parallel im „Thomas J. Watson Research Center“ erzeugt wird, oder man entscheidet sich für die Arbeit mit einer rein durch Software definierten Simulation.



(a) Simulationsprogramm dieser Seminarfacharbeit (b) IBM Quantum Experience Composer[1]

Abbildung 1.1: Graphische Benutzeroberflächen beider Quantencomputersimulationen

[1] IBM Q Experience, Mai 2016

Der Computer ist je in der Größe des Projektes, der Kommunikation zwischen Qubits und der Wiedergabe eines vollständig richtigen Ergebnisses aufgrund von Ungenauigkeiten und Grenzen der physikalischen Welt eingeschränkt, während die Simulation mathematisch richtige – und damit rein theoretische – Ergebnisse liefert und IBM's Quantencomputer nicht benötigt. Es war ein Ziel unserer Arbeit, eine eigene Simulation zu entwerfen und zu implementieren sowie die GUI in den Hauptpunkten wie Gatterdarstellung und allgemeiner Verwendung möglichst nah an IBM's „Quantum-Experience" anzulehnen. Dabei wurde kein Quelltext übernommen, denn der benutzte Code und die verwendeten Bilder des eigenen Programmes sind alle selbst erstellt. Die starken Ähnlichkeiten der Layouts sind auch aufgrund von Notationskonventionen nötig, da besondere Gatter spezielle Namen und Abkürzungen besitzen, welche benutzt werden sollten, um die GUI allgemein verständlich zu halten. Ausserdem ist die Darstellung von Gattern auf einem sogenannten „Wirediagramm" allgemein üblich. Ein Wirediagramm teilt die möglichen Gatterplatzierungen auf N Qubits und M Zeitabschnitte auf. Die Gatter im Wirediagramm werden dann nacheinander auf die korrespondierenden Qubits angewendet. In den obigen Bildern wirken also zuerst zwei Hadamard-Gatter auf Qubit eins und zwei, dann zwei Phase-S-Gatter und so weiter. Für das Programmieren der Quantencomputersimulators wurde aufgrund des umfassenden Vorwissens durch den Schullehrplan die Programmierbasissprache Java[2] verwendet. Processing[3], eine quelloffene (open source) Entwicklungsumgebung und Programmiersprache mit Schwerpunkten auf Animation und Grafik, wurde zusätzlich für einen einfacheren Visualisierungscode eingebunden.

1.2 Programmstruktur und Zustandsberechnung

Java ist eine objektorientierte Programmiersprache, weswegen Objektorientierung zur Übersichtlichkeit und Funktionstüchtigkeit von in Java geschriebenen Programmen ab gewissen Größen von elementarer Bedeutung ist. Objektorientierte Programmierung baut ein Programm als Zusammenspiel einzelner Objekte auf, welche eigene Daten(Attribute) und ausführbaren Code(Methoden) besitzen und durch Klassen mithilfe spezieller Ausgangsvariablen instanziiert werden. Nimmt man Qubits als Beispiel, dann stellt der Vorgang der Erstellung eines neuen Qubits die abstrakte Klasse dar. Möchte man nun ein neues Qubit, also ein Objekt der Klasse, erzeugen, dann gibt man als Ausgangsvariablen den Startzustand des Qubits an, damit durch den Erstellungsvorgang ein neues Qubit erzeugt werden kann. Der Vorgang ist dadurch jedoch nicht verbraucht, denn durch neue Ausgangsvariablen können weiterhin neue Qubits erzeugt werden und parallel existieren.
Betrachtet man nun die gesamten Ausgangsqubits, dann besitzen sie eine festgelegte Anzahl und einen festgelegten Zustand und können mit Gattern interagieren. Dies sind die Attribute und Methoden des Objektes „Qubits". Außerdem besteht es aus den einzelnen Qubits, welche wiederum als Objekte mit eigenen Attributen und Methoden fungieren, die als einzelne Attribute der Qubitgesamtheit gelten und darin gebunden sind.
Die Simulation funktioniert auf diese Weise mit 7 teilweise ineinander verschachtelten Klassen, welche durch das unten aufgeführte Klassendiagramm modelliert werden.

[2]Oracle, Java SE Development Kit 8, März 2014
[3]Ben Fry, Casey Reas, Processing 3.0, September 2015

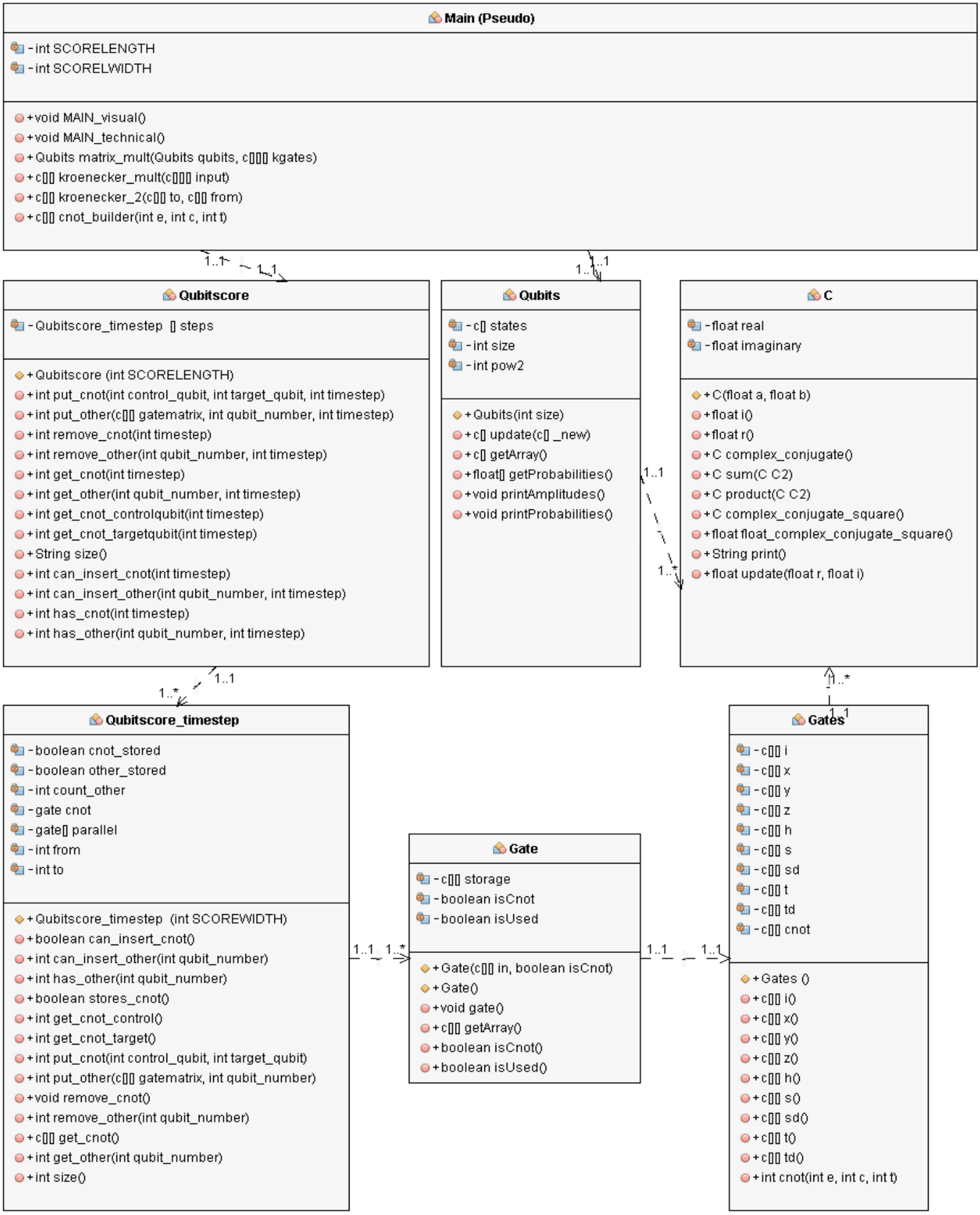


Abbildung 1.2: Klassendiagramm der Quantencomputersimulation

1.2.1 C

Die „C“-Klasse ist als Darstellung von komplexen Zahlen die Basis der Quantencomputersimulation, da Qubits universell nur durch 2 komplexe Zahlen modelliert werden können. Diese sind in Java nicht standardmäßig implementiert, weswegen selbst eine Klasse programmiert werden musste. Jedes Objekt der Klasse besitzt als Attribute je eine Fließkommazahl zur Beschreibung des Realteils r und des Imaginärteils i, um eine komplexe Zahl der Form $r + i\sqrt{-1}$ zu simulieren. Berechnungen können durch bekannte mathematische Gleichungen im komplexen Raum durchgeführt werden[4].

$$\Re(r + i\sqrt{-1}) = r \qquad \textit{Realteil} \qquad (1.1)$$

$$\Im(r + i\sqrt{-1}) = i \qquad \textit{Imaginärteil} \qquad (1.2)$$

$$(a + b\sqrt{-1}) + (r + i\sqrt{-1}) = ((a + r) + (b + i)\sqrt{-1}) \qquad \textit{Summe} \qquad (1.3)$$

$$(a + b\sqrt{-1}) \cdot (r + i\sqrt{-1}) = ((ar - bi) + (ai + br)\sqrt{-1}) \qquad \textit{Produkt} \qquad (1.4)$$

$$\overline{r + i\sqrt{-1}} = r - i\sqrt{-1} \qquad \textit{komplexes Konjugat} \qquad (1.5)$$

$$|r + i\sqrt{-1}|^2 = (r^2 + i^2) + 0\sqrt{-1} = r^2 + i^2 \qquad \textit{Absolutes Quadrat} \qquad (1.6)$$

$$r := a,\ i := b,\ r + i\sqrt{-1} = a + b\sqrt{-1} \qquad \textit{Zuweisung} \qquad (1.7)$$

Hierbei ist $\sqrt{-1}$ als imaginäre Einheit für die Implementation nicht nötig, da keine komplexen Zahlen verwendet, sondern nur via Neuzuweisungen oder Ausgaben von Attributen des „C“-Objektes durch Methoden simuliert werden.

$$(r, i).r := r \qquad (r, i).i := i \qquad (1.8)$$

$$(r, i).c\_conjugate := (r, -i) \qquad (r, i).update(a, b) := (a, b) \qquad (1.9)$$

$$(r, i).sum(a, b) := (a + r, b + i) \qquad (r, i).product(a, b) := (ar - bi, ai + br) \qquad (1.10)$$

$$(r, i).cc\_square := (r^2 + i^2, 0) \qquad (r, i).float\_cc\_square := r^2 + i^2 \qquad (1.11)$$

1.2.2 Gates

Die „Gates“-Klasse ist als Sammel-Speicher der einzelnen Gatter die Basis der „Gate“-Klasse, da sie die einzelnen Gatter frei zur Verfügung stellt. Hierbei benutzt sie „C“-Objekte, da jedes Gatter durch komplexe Zahlen beschrieben wird[5], und speichert jedes dieser Gatter als zweidimensionales Array.

$$\text{Bsp: } \mathbf{S} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \begin{bmatrix} 1 + 0i & 0 + 0i \\ 0 + 0i & 0 + i \end{bmatrix} \Longrightarrow \left[\Big[(1, 0), (0, 0) \Big], \Big[(0, 0), (0, 1) \Big] \right] \qquad (1.12)$$

Das C-Not Gatter nimmt wieder eine Sonderrolle ein, da es nicht vorher definierbar ist, sondern durch 3 Parameter bestimmt wird, welche sich bei jedem C-Not Gatter erst während der Platzierung ergeben. Deswegen wird es durch eine eigene Methode neu generiert, welche wegen ihrer Komplexität in die Main ausgelagert wurde.

[4] Eric W. Weisstein, 2003, S.1

[5] Kapitel 2.3 Quantengatter

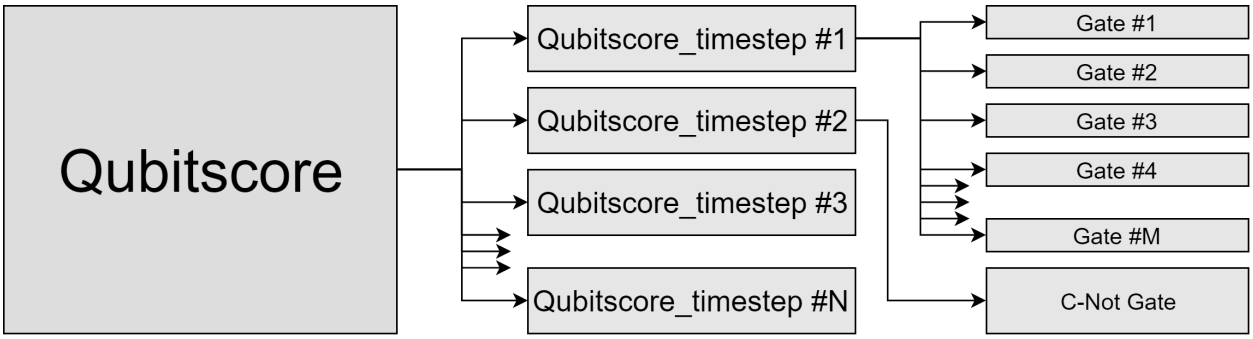


Abbildung 1.3: Hierarchie des Qubitscores

1.2.3 Gate

Die „Gate“-Klasse speichert genau ein Gatter aus der „Gates“-Klasse und kann dieses Gatter ausgeben. Es unterscheidet zusätzlich noch zwischen 2 Kategorien von Gattern, dem C-Not Gatter und allen anderen Gattern. Dies geschieht, da diese beiden Arten von Gattern unterschiedliche Größen aufweisen. Das C-Not Gatter besteht aus Arrays der Länge 2^N, während alle anderen Gatter aus Arrays der Länge 2 aufgebaut sind. Man benötigt also N normale Gatter, um ein Gatter derselben Größe zu erreichen, weswegen das C-Not Gatter extra betrachtet werden muss.

1.2.4 Qubitscore_timestep & Qubitscore

Jedes Objekt der „Qubitscore_timestep“-Klasse stellt einen Zeitabschnitt im Wirediagramm dar. Es unterteilt sich in N Gatter, welche jeweils mit einem Qubit korrespondieren. Während der Ausgabeberechnung führt es eine Multiplikation mit dem Objekt der „Qubits“-Klasse aus. Um dies umzusetzen, benötigt es eine Matrix P der Größe 2^N und muss deswegen entweder ein C-Not Gatter Q_{C-Not} oder bis zu N andere Gatter Q_x enthalten, um nach einer Kroneckermultiplikation die richtige Größe aufzuweisen.

$$P = \begin{cases} Q_{C-Not} & \\ \prod_{n=1}^{N} Q_n & \Big| \; Q_n = \begin{cases} Q_n & \textit{for P.has\_other(n)==true} \\ \mathbf{I} & \textit{for P.has\_other(n)==false} \end{cases} \\ \text{Kronecker} & \end{cases} \quad \textit{for P.cnot\_stored()==true} \tag{1.13}$$

Das Objekt der „Qubitscore_timestep“-Klasse stellt das gesamte Wirediagramm dar, ist hauptsächlich zum Zusammenfassen der einzelnen Qubitscore_timesteps gedacht und leitet „Getter“ und „Setter“[6] des Codes zum Erstellen der visuellen Repräsentation auf diese weiter.

1.2.5 Qubits

Die „Qubits“-Klasse verwendet 2^N komplexe Objekte, um die N benötigten Qubits mathematisch korrekt darzustellen. Diese Liste stellt das Kroneckerprodukt aus allen einzel-

[6]Methoden zum Entgegennehmen und Verändern von Objektattributen

nen Qubitmatrizen dar und enthält je einen Eintrag für das Produkt der Elemente jeder möglichen Permutation von N der $2N$ Elemente α_1 bis α_N und β_1 bis β_N.[7] Beginnt die Berechnung des vom Nutzer erstellten Programms, dann wird nacheinander jedes der M Gesamtgatter mit dem Kroneckerprodukt der Qubits, also dem Gesamtqubitzustand multipliziert, um am Ende die Ausgabedaten zu erhalten. Beide Arten der Matrixmultiplikation wurden allerdings aufgrund ihrer Komplexität nicht objektintern geregelt, sondern wieder in die Main als Methoden ausgelagert. Das „Qubits"-Objekt ist die passive Eingabe, auf die andere Objekte einwirken, wodurch es schließlich zur Ausgabe wird. Danach werden die einzelnen Wahrscheinlichkeiten als absolutes Quadrat der einzelnen Teilsuperpositionszustände an die Main übergeben und optional sowohl die Amplituden als auch die Wahrscheinlichkeiten ebendieser auf der Konsole ausgegeben.

1.2.6 Main

Ein Teil der „Main"-Klasse stellt die Übersicht über alle anderen Objekte jeder Klasse dar, regelt den groben Ablauf des Programmes und sorgt für die Berechnung des Ausgabesuperpositionszustandes. Der andere Teil beherbergt die ausgelagerten Methoden.

```
float[] state_calculator() {
  // Dem Qubitscore werden alle Gatterpositionen und Gatterarten
     uebergeben.
  Qubitscore QS = Gatterspeicher.getData();
  // Das Kroneckerprodukt der einzelnen Qubitscore_timesteps wird
     berechnet.
  c[][][] after_kronecker = new c[M[(int) pow(2, N)][(int) pow(2, N)];
  for (int i=0; i<M; i++) {
        // Spezialfall: C-Not Gatter
        // In diesem Fall kann man einfach kopieren.
    if (QS.has_cnot(i) == true) {
      after_kronecker[i] = QS.get_cnot(i);
    }
    // Normalfall: gewöhnliche Gatter
    // In diesem Fall muss man das Kroneckerprodukt berechen.
    else {
      c[][][] normal_gates = new c[N][][];
      for (int j=0; j<normal_gates.length; j++) {
        normal_gates[j] = QS.get_other(j, i);
      }
      after_kronecker[i] = kronecker_mult(normal_gates);
    }
  }
  // Das Matrixprodukt aus Qubits und Qubitscore_timesteps wird
     berechnet.
  Qubits after_product = new Qubits(SCOREWIDTH);
  after_product = matrix_mult(qubits, after_kronecker);
  // Die Endwahrscheinlichkeiten werden zurückgegeben.
  return after_product.getProbabilities();
}
```

[7] Kapitel 2.2.1 Mathematische Grundlagen

Kroneckerprodukt

Die Methode zum Berechnen des Kroneckerprodukts ist für ihre Aufgabe zweigeteilt. Der erste Teil empfängt eine Liste von Matrizen, teilt diese auf und übergibt sie nacheinander immer wieder dem zweiten Teil. Dieser iteriert dann über beide Arrays und schreibt die Produkte der einzelnen Arrayelemente in das Ausgabearray. Das Ausgabearray wird durch den ersten Teil mit einer weiteren Matrix neu übergeben, bis alle vorhandenen Matrizen genutzt wurden.

```
// Kroneckerprodukt von 2 Matrizen
c[][] kronecker_2(c[][] to, c[][] from) {
  // Instanziierung des Ausgabearrays
  c[][] output = new c[to.length*from.length]
      [to[0].length*from[0].length];
  // Iteration über Array "to"
  for (int i=0; i<to.length; i++){
  for (int j=0; j<to[0].length; j++){
    // Iteration über Array "from"
    for (int k=0; k<from.length; k++){
    for (int l=0; l<from[0].length; l++){
      // Berechnung der Koordinaten des
          einzelnen Ergebnisses im
          Ausgabearray dortiges Eintragen
      int y = i*from.length + k;
      int x = j*from[0].length + l;
      out[y][x]=to[i][j].product(from[k][l]);
  }}}}
  // Ausgabe des Ausgabearrays
  return out;
}
```

```
// Kroneckerprodukt von n Matrizen
c[][] kronecker_mult(c[][][] input) {
  // Sonderfall: 2 Matrizen
  if (input.length==2) {
    // Matrizenübergabe an kronecker_2
        Methode und Rückgabe des Ergebnisses
    return kronecker_2(input[0], input[1]);
  // Normalfall: mehrere Matrizen
  } else {
    // Sequenzielle Matrizenübergabe an
        kronecker_2 Methode und
        Wiederverwendung der Rückgabe
    c[][] first = input[0];
    c[][][] rest = new c[input.length-1][][];
    for (int i=1; i<input.length; i++) {
      rest[i-1] = input[i];
    }
    // Ausgabe des Endergebnisses
    return kronecker_2(first, kronecker_mult(
        rest));
}}
```

Matrixprodukt

Die Methode zum Berechnen des Matrixproduktes funktioniert analog zu der des Kroneckerproduktes. Sie erstellt das Ausgabearray jedoch basierend auf den Größen der Eingabearrays und bildet für jedes Element die Summe aus den einzelnen benötigten Produkten, um sie in das Ausgabearray zu schreiben.

```
//Matrixprodukt von 2 Matrizen
Qubits matrix_mult_2(Qubits qb, c[][] kgate) {
  // Instanziierung der zu multiplizierenden Matrizen
  c[] q = qb.getArray();
  c[] out = new c[q.length];
  // Iteration über das Ausgabearray und Summierung der
       einzelnen Produkte
  for (int m=0; m<kgate.length; m++) {
    c partSum = new c(0, 0);
    for (int n=0; n<kgate[0].length; n++) {
      partSum = partSum.sum(q[n].product(kgate[m][n]));
    }
    // Eintragen der jeweiligen Produktsummen
    out[m] = partSum;
  }
  // Ausgabe des upgedateten "Qubits" Objekts
  qb.update(out);
  return qb;
}
```

```
// Matrixprodukt von n Matrizen
Qubits matrix_mult
    (Qubits qb, c[][][] kgates) {
  // Sequenzielle Matrizenübergabe
      an matrix_mult_2 Methode und
      Wiederverwendung der Rückgabe
  for(int i=0;i<kgates.length;i++){
    qb=matrix_mult_2(qb,kgates[i]);
  }
  // Ausgabe des "Qubits" Objektes
      nach Matrixmultiplikation mit
      jedem Kroneckergatter
  return qb;
}
```

C-Not Gatterberechnung

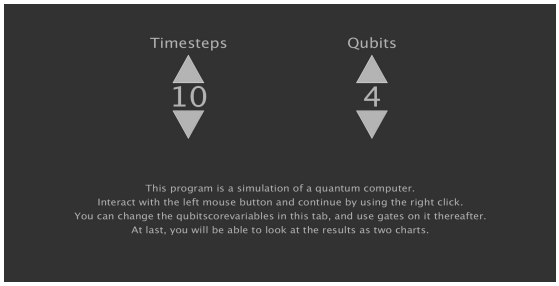
Die Methode zum Erstellen des C-Not Gatters kann durch einen Vergleichsalgorithmus von einer normalen Qubitzustandsliste und dieser nach Anwendung der Veränderung durch Kontrollqubit und Zielqubit das gesuchte Gatter bestimmen, indem sie untersucht, welcher Zustand wohin übergeht.

```
// Pseudocode
c[][] cnot_builder(N, Kontrollqubit, Zielqubit) {
  // Erstellen zweier Arrays mit Zustaenden abhaengig vom C-Not Gatter
  int[][] davor = new int[2^N][N];      // Zustaende davor
  int[][] danach = new int[2^N][N];     // Zustaende danach
  // Mit dem Stellenwertsystem und den binaeren Zahlen lassen sich Qubitzustandsarten
      darstellen: |0110> => 0110 => 6
  // Durch diese Vorgehensweise werden alle Qubitzustandsarten im "davor" Array
      aufgelistet
  for (zaehler = 0 bis 2^N) {
    int binaer_zaehler = binary(zaehler);
    for (j = 0 bis 2^N) {
      davor[i][j] = binaer_zaehler%10;
      binaer_zaehler = binaer_zaehler/10;
  }}
  // "danach" analog zu "davor", aber falls der Zustand des Kontrollqubits 1 ist, wird
      dar Zielqubitzustand invertiert
  for (i = 0 bis 2^N) {
    danach[i] = davor[i];
    if (davor[i][Kontrollqubit]==1) {
      if      (danach[i][Zielqubit]==1) danach[i][Zielqubit]=0;
      else if (danach[i][Zielqubit]==0) danach[i][Zielqubit]=1;
  }}
  // Vergleichsarray, welches parallel die Position jeder Qubitzustandsart im "davor"
      und "danach" Array speichert
  int[][] aenderung = new int[2^N][2];
  for (position_1,position_2 = 0 bis 2^N) {
      if (davor[position1] = danach[position_2]) {
        aenderung[position_1][0]=position_1;
        aenderung[position_1][1]=position_2;
  }}
  // Erstellung des Basisarrays für das C-Not Gatter der Größe 2^N
  c[][] output = new c[2^N][2^N];
  for (i,j = 0 bis 2^N) {
      output[i][j] = new c(0, 0);
  }
  // Einfügen der Einsen an den richtigen Stellen mithilfe des "aenderung" Arrays
  for (i = 0 bis 2^N) {
    output[ aenderung[i][0] ][ aenderung[i][1] ] = 1;
  }
  // Ausgabe des fertigen C-Not Gatters
  return output;
}
```

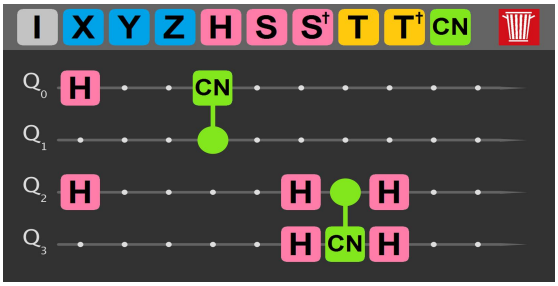
1.3 GUI

Die GUI stellt die Nutzeroberfläche der Quantencomputersimulation dar und sollte möglichst intuitiv zu bedienen sein. Aus diesem Grund wurde eine rein auf der Maus als Eingabegerät basierende Oberfläche implementiert, mit welcher man über Click oder Click & Drop interagieren kann. Das Interagieren erfolgt mit der linken Maustaste, das Bestätigen mit der rechten Maustaste und das Beenden über die Escape-Taste. Die GUI ist in zwei Teile geteilt: vor der Berechnung des Endsuperpositionszustandes mit Fokus auf Eingabe und Manipulation des Qubitscores und danach mit Fokus auf klarer Informationsdarstellung.

1.3.1 Eingabe



(a) Auswahl der Qubit- und Zeitabschnittsanzahl

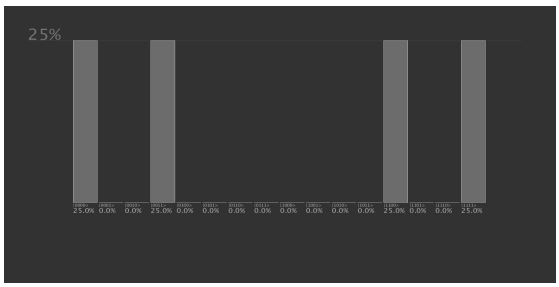


(b) Quantumscore Nutzeroberfläche

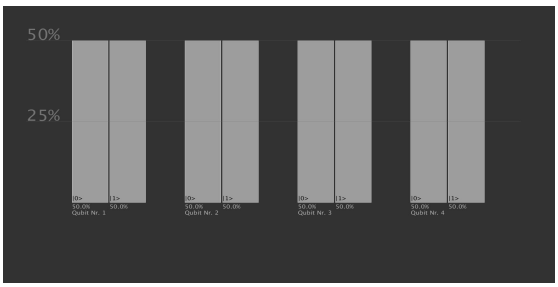
Abbildung 1.4: Graphische Benutzeroberflächen des Eingabeteils

Der Eingabeteil ist wiederum in zwei Unterkategorien gegliedert. Das erste Element ist der Auswahlbildschirm, auf dem man 1 bis 20 Zeitabschnitten und 2 bis 5 parallele Qubits einstellen kann. Wurde dies gemacht, dann wird das zweite Element angezeigt, ein Qubitscore mit der gewählten Anzahl an Zeitabschnitten und Qubits. Auf diesem kann man nun seine Gatter frei platzieren. Die einzige Einschränkung stellt die Regel dar, dass nur ein C-Not Gatter oder N andere Gatter gleichzeitig einen Zeitabschnitt belegen dürfen. Bestätigt man nun seine Eingabe, dann wird der Endsuperpositionszustand berechnet und man gelangt in den Ausgabeteil.

1.3.2 Ausgabe



(a) Wahrscheinlichkeit jedes Qubitzustandes



(b) Zustandswahrscheinlichkeit von jedem Qubit

Abbildung 1.5: Graphische Benutzeroberflächen des Ausgabeteils

Auch der nach der Berechnung angezeigte Ausgabeteil besteht aus zwei Unterkategorien: einem Diagramm zur Verdeutlichung der Wahrscheinlichkeit von jedem möglichen erreichbaren Qubitzustand und einem Diagramm zur Verdeutlichung der Zustandswahrscheinlichkeit von jedem Qubit. Wendet man also die oben gezeigten Gatter auf die Qubits an, so sieht man, dass nur $|0000\rangle, |0011\rangle, |1100\rangle, |1111\rangle$ als Zustände mit den gleichen Wahrscheinlichkeiten von 25% auftreten können. Außerdem hat jedes Qubit eine Wahrscheinlichkeit von 50%, selbst im Zustand $|0\rangle$ oder $|1\rangle$ aufzutreten.